

POST: A Secure, Resilient, Cooperative Messaging System

Alan Mislove¹ Ansley Post¹ Charles Reis¹ Paul Willmann¹ Peter Druschel¹
Dan S. Wallach¹ Xavier Bonnaire² Pierre Sens² Jean-Michel Busca²
Luciana Arantes-Bezerra²

¹Rice University, Houston, TX, USA
²LIP6, Université Paris VI, Paris, France

Abstract

POST is a cooperative, decentralized messaging system that supports traditional services like electronic mail (email), news, instant messaging, as well as collaborative applications such as shared calendars and whiteboards. Unlike existing implementations of such services, POST is highly resilient, secure, scalable and does not rely on dedicated servers. POST is built upon a peer-to-peer (p2p) overlay network, consisting of participants' desktop computers. We sketch POST's basic messaging infrastructure, which provides shared, secure, single-copy message storage, user-specific metadata, and notification. As an example application, we sketch how POST can be used to construct a cooperative, secure email service called ePOST.

1 Introduction

Messaging systems like traditional email and news, as well as instant messaging, shared calendars and bulletin boards, are among the most successful and widely used distributed applications. Currently, these services are implemented in the client-server model. Messages are stored on and routed through dedicated servers, each hosting a set of user accounts. This partial centralization requires substantial infrastructure costs when the system is scaled to large numbers of users. The client-server model also limits reliability, as servers present a single point of failure or attack on the system for the users they support. Additionally, maintenance and administration costs can become significant for large organizations.

POST is a cooperative infrastructure that leverages the resources of users' desktop workstations to provide messaging services. POST provides three fundamental services to applications: (1) persistent single-copy message storage, (2) per-user metadata, and (3) notification. A wide range of messaging applications can be constructed on top of POST using these services.

POST itself is built upon a structured p2p overlay network, providing 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 uti-

lize its services. POST assumes that participating nodes can suffer byzantine failures. Stronger failure assumptions may be unrealistic, even in scenarios where participating hosts belong to a single organization, because a single compromised node may be able to disrupt critical messaging services or disclose confidential messages.

In this paper, we sketch the design of the POST infrastructure, and then describe how a cooperative, secure email system can be built on top of POST. Unlike conventional SMTP-based email services, our *ePOST* system provides secure email services by default and requires no dedicated servers. Furthermore, due to its strong sender authentication, ePOST makes efficient spam defense easier. We chose email as the initial application for POST because it is well understood, and because its high availability, reliability and security demands make it a challenging driver for POST and p2p systems in general.

The remainder of this paper is organized as follows. Section 2 provides background information on Pastry, PAST, and Scribe, which are used as building blocks for POST. Section 3 sketches the design of the POST infrastructure. In Section 4, we sketch the design of a cooperative email system as an example POST application. Section 5 discusses integrating POST with existing messaging systems. Section 6 outlines related work, Section 7 reports the status of the project, and Section 8 concludes.

2 Background

In this section, we briefly describe Pastry, PAST and Scribe, which are used as building blocks in POST.

Pastry [12] 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 chosen from a large id space, referred to as a *nodeId* and *key*, respectively. Given a message and a key, Pastry can efficiently route the message to the node whose *nodeId* is numerically closest to the key.

PAST [13] is a storage system built on top of Pastry and can be viewed as a distributed hash table. 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* nodes whose

nodeIds are the numerically closest to the object's handle. PAST also 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 [2] 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 paper, we assume that at most $k - 1$ nodes are faulty in any replica set.

A variant of PAST is used in POST to store three types of data: *content-hash blocks*, *public-key blocks*, and *certificate blocks*. Content-hash blocks are stored using the cryptographic hash of the block's contents as the handle. Public-key blocks contain monotonically increasing timestamps, are signed with a private key, and are stored using the cryptographic hash of the corresponding public key as the handle. Certificate blocks are signed by a trusted third party and bind a public key to a name (e.g., an email address). The block is stored using the cryptographic hash of the name as the handle.

Content-hash blocks can be authenticated by obtaining a single replica and verifying that its contents match the handle. Unlike content-hash blocks, public key blocks are mutable. To prevent rollback attacks by malicious storage nodes, it is necessary to obtain k replicas and choose the authentic block with the most recent timestamp. Certificate blocks require a signature verification using the well-known public key of a trusted third party.

Scribe [3] is a scalable multicast 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*.

3 POST Architecture

As a generic messaging system, POST provides three fundamental services: a shared, secure single-copy message store, per-user metadata, and notification. These services can be combined to implement a variety of collaborative applications, like email, news, instant messaging, shared calendars and whiteboards.

A typical pattern is that users create messages, which are inserted in encrypted form into the secure store. To send the message to another user or group, the 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 metadata to incorporate the message into their view (e.g., into a private mail folder).

POST assumes the existence of a certificate authority. This authority signs 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 [2]. Furthermore, the authority may require that each user also owns a nodeId bound to a live IP address, thus forcing the user to contribute a node to the system. Users can access the system from any node, but it is assumed that the user

trusts her local node, hereafter referred to as the trusted node, with her private key.

Throughout the design of POST, we assume that objects stored in PAST cannot be deleted. Thus, we assume that the amount of available disk space in the system is always increasing and greater than the total storage requirements, which is reasonable to expect in a p2p environment where each participant is required to contribute a portion of her desktop's local disk [10].

3.1 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 in PAST, using the secure hash of the user's name as the handle. Also associated with each account is a user identity block, which contains an XML 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 in PAST, and signed with the user's private key. Finally, each user account has an associated Scribe group used for notification, with a *groupId* equal to the cryptographic hash of the user's public key.

3.2 Secure Message Storage

POST provides a shared, secure message storage facility. Application-provided message data is encrypted using a technique known as convergent encryption [6]. Convergent encryption allows a message to be disclosed to selected recipients, while ensuring that copies of a given cleartext message inserted by different users map to the same ciphertext, thus requiring only a single copy of the ciphertext to be stored.

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

$$H(\{X\}_{H(X)})$$

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

3.3 Notification

The purpose of the notification is to alert a user to the availability of a message and to provide her with the appropriate decryption key. In the common case, a 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, which contains the secure hash of the message's ciphertext and its decryption key, and is encrypted with the recipient's public key and signed by the sender.

In practice, 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 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.

3.4 Per-User Metadata

POST provides a facility that allows applications to maintain per-user metadata that refers to stored messages of interest to the user. The facility provides single-writer logs that can be used by applications to represent changes to application metadata. For instance, an email application can use a log of insert and delete records to keep track of the state of a user's mail folder. 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 in PAST.

The log head is stored as a public-key block in PAST 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 in PAST 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 a straightforward implementation, the log head and each log record are stored at a different set of PAST nodes. To allow for more efficient log traversal, POST stores clusters of M consecutive log records on the same PAST node, under the handle of the least recent of the M records. To deal with partially filled clusters, the log head contains an additional handle, referring to the least recent record in a partially filled cluster. This handle identifies the cluster in PAST.

Other optimization are possible to reduce the overhead of log traversals, including caching of log records at clients and the use of snapshots. POST applications may periodically insert snapshots of their metadata into PAST making log traversal only necessary up to the most recent snapshot. Similar

ideas were used in Ivy [9] in order to represent file metadata in a p2p file system.

3.5 Discussion

POST provides single-writer logs as the only mechanism to maintain mutable state in the system. Thus, it avoids the cost and complexity of a general byzantine fault-tolerant replicated state machine. We are confident that POST's restricted mechanism for mutable state is flexible and efficient enough for email and instant messaging, especially given the snapshot and clustering optimizations that POST uses. We conjecture that it is also sufficient for other applications like shared calendars.

However, we believe that cooperative applications exist that may require a more flexible mechanism for maintaining mutable state. To support such applications, the authors at LIP6 are currently investigating additional, byzantine fault-tolerant mechanisms for maintaining multi-writer, mutable state. These mechanisms target applications supporting a large number of writers and allowing disconnected operations.

4 Example: Electronic Mail

In this section, we sketch the design of a serverless email system, ePOST, on top of the POST infrastructure. The goal is to leverage POST to build a secure, scalable and highly resilient email system, while leveraging 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.

4.1 Overview

Each ePOST user is expected to run a daemon program on his desktop computer that implements the Pastry, PAST, Scribe and POST protocols, and contributes some CPU, network bandwidth and disk storage to the system. The daemon 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.

4.2 Message 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 messages in POST. 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. If messages are forwarded or sent by different users, the original message data does not need to be stored again; the original 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 clear-text attacks. To avoid a loss of confidentiality, small message bodies are padded by ePOST with a number of random bits. This measure defeats the single-copy storage, but this is not a concern given the small size of the affected messages.

Due to the necessary data replication in PAST, the storage overhead per message is higher in POST compared to a conventional server-based email system. However, this effect is partly offset by POST's single-copy store, which eliminates large amounts of duplication due to large, widely circulated email attachments. Moreover, exploiting the typically under-utilized disk space on desktop computers should more than compensate for this overhead [1].

4.3 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, the subject, 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. It is noteworthy to mention that ePOST extends recipient control beyond current systems by allowing the recipient to append the message to his mailbox or to simply ignore the notification, perhaps based on a spam filter.

4.4 Metadata

Each mail folder is represented by a 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 can be encrypted, ePOST preserves the expected semantics of current email systems.

Next, we describe a log record representing an insertion of a email message into a user's Inbox folder. 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 from the sender, all of which are encrypted with the recipient's public key.

Thus, the recipient can verify that the message was actually sent by the stated sender, and both parties have the confidence that only the intended recipient will be able to read the message. As an example, if user A sent a message to user B with subject S and message text X at time T , the insertion record in B 's Inbox will be

$$\{A, B, S, T, H(\{X\}_{H(X)}), H(X), sig_A\}_B$$

4.5 Discussion

By default, ePOST provides strong confidentiality, authentication and message integrity. Moreover, with very high probability, the system is able to tolerate up to a given fraction of faulty or malicious nodes without loss of data or service. It relies on Pastry's secure routing facilities [2], data replication, and cryptographic techniques to achieve robustness under a wide range of attacks, including denial-of-service. 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. Results of a prior study on p2p filesystems in corporate environments indicate that modest levels of replication can yield very high availability [1].

In the simplest configuration, all participating POST nodes form a single Pastry ring, and a message inserted into POST is replicated on a set of random nodes throughout the system. In practice, it would be desirable to instead replicate a message inserted by a given user among the nodes within the user's organization. This *scoped* insertion leads to better network locality, and provides an effective defense against denial-of-service attacks aimed at exhausting the system's storage. It also permits better node failure rate estimations and fault assumptions within a given organization and may thus permit less conservative replication levels. Moreover, a scoped insertion approach fits more naturally firewalled intranets and it arises naturally when ePOST is deployed incrementally. Scoped insertion can be supported with minor extensions to POST and PAST, but we omit the details due to lack of space.

It is interesting to note that ePOST provides better spam prevention than current email systems. In ePOST, all notification messages are signed by the sender, which makes it possible to build effective spam block lists. These block lists could be compiled on a per-user basis, and possibly shared among users. Additionally, ePOST could limit the rate of sending messages by requiring senders to solve small cryptographic puzzles [5] before being allowed to send notification messages. This would not have much of an effect on normal ePOST users but would slow down bulk emailers.

Mailing lists can be easily supported 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 Incremental Deployment

In this section, we discuss integration issues in the context of ePOST. The approaches could be generalized to other POST applications. 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 sketch here how ePOST could be deployed in a single organization and interoperate with email services in the general Internet.

To send email messages to the outside world, the ePOST proxies use standard SMTP to contact the recipient's email server, whenever a recipient is outside the local organization. For inbound email, the organization's DNS server delivers MX records referring to a random proxy in the ePOST system, which accepts the message using SMTP, and delivers it locally to the intended recipients. Of course, ePOST's built-in authentication and privacy mechanisms are not available when email is exchanged with a party that does not use ePOST. Incoming messages are tagged with a MIME header indicating that the message's origin and integrity could not be verified.

ePOST currently assumes that all participating hosts can communicate with each other, without intervening firewalls. ePOST systems separated by firewalls can interoperate via SMTP, at the cost of losing the security aspects and shared message storage. Allowing ePOST systems separated by firewalls to be integrated more tightly is the subject of ongoing work.

6 Related Work

Current email protocols, including SMTP [11], POP3 [8], and IMAP [4], are tailored towards an infrastructure based on dedicated servers. Minimal security is provided in these protocols, and the email service does not provide confidentiality, verifiability, or data integrity. Extensions like PGP [15] provide secure email, but are not widely used.

Lotus Notes and Microsoft Exchange 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.

There has been much work to allow email services to scale more effectively through the use of cluster-based servers, such as the Porcupine System [14] as well as Hotmail and Yahoo's mail services. ePOST instead utilizes a completely decentralized, self-scaling architecture, thus eliminating the need for dedicated, powerful mail servers.

7 Status

We have implemented a prototype of POST and ePOST; an implementation of an instant messaging service based on POST, called *imPOST*, is currently underway. We plan to start using ePOST shortly, initially within our research groups, and hope to expand the user base within Rice and

LIP6 and beyond, as we gain experience and confidence in the system. Given users' dependence on email services, we view this as a proof of concept for mission-critical p2p systems, and as a vehicle to gain practical experience and workload trace data from such a system. A full description and evaluation of POST and ePOST will be provided in an upcoming full paper.

8 Conclusions

POST is a p2p, collaborative 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. The fundamental services provided by POST can be used to support a variety of messaging applications. In this paper, we have sketched how POST can be used to construct ePOST, a cooperative, secure email system.

References

- [1] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. SIGMETRICS'2000*, Santa Clara, CA, 2000.
- [2] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proc. of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
- [3] M. Castro, P. Druschel, A-M. Kermaec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), October 2002.
- [4] M. Crispin. RFC 2060: Internet message access protocol version 4rev1, December 1996.
- [5] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proc. 10th USENIX Security Symposium*, Washington, D.C., August 2001.
- [6] J. Douceur, A. Adya, W. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS 2002)*, Vienna, Austria, July 2002.
- [7] Jussi Kangasharju, Keith W. Ross, David A. Turner, Juha Syrjala, and David-Steve Digeon. Peer-to-peer e-mail, November 2002. Submitted for publication.
- [8] J. Meyers and M. Rose. RFC 1939: Post office protocol version 3, May 1996.
- [9] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
- [10] T. Ngan, P. Druschel, and D. S. Wallach. Enforcing fair sharing of peer-to-peer resources. In *Proceedings for the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February 2003.
- [11] J. Postel. RFC 821: Simple mail transfer protocol, August 1982.
- [12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.
- [13] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP'01*, Banff, Canada, October 2001.
- [14] Y. Saito and B. Bershad H. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proc. ACM SOSP'99*, Charleston, South Carolina, December 1999.
- [15] P. Zimmerman. PGP user's guide, December 1992.