12. ePOST

Alan Mislove, Andreas Haeberlen, Ansley Post, Peter Druschel
(Rice University & Max Planck Institute for Software Systems)

ePOST is a Peer-to-Peer email system that provides the same functionality as existing, server-based email systems while providing better availability, scalability, fault tolerance, and security. The ePOST system has been in production use within the Computer Science department at Rice University since early 2004 and is being adopted by an increasing number of outside users.

Traditional email and news services, along with 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 mostly implemented in the client-server model, where messages are 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 all the users it supports. Also, dedicated infrastructure and a substantial maintenance and administration effort are required to provide services to large numbers of users.

A decentralized, cooperative approach, i.e., a Peer-to-Peer (P2P) based solution like ePOST, seems like a natural fit for collaborative applications. Rather than requiring dedicated server infrastructure, ePOST scales organically with the number of participating users, since each participant contributes resources to offset the additional demand he places on the system. Also, ePOST removes all single points of failure by distributing the services across all member nodes, thus providing the potential for a more highly available system. Lastly, the self-organizing properties of ePOST promise to reduce the cost of maintaining email services.

To use ePOST, users download and run a Pastry [527] node on their desktop, which connects to the ePOST overlay network. Folder information and email messages are stored in the network using the PAST [526] distributed hash table, and the Scribe [109] multicast system is used to efficiently communicate among users. To allow for users to view and send mail, each ePOST node acts as a IMAP, POP3, and SMTP server. Thus, each user has their own private mail server on their desktop, which he accesses using any standard mail client program.

ePOST is built upon the POST distributed messaging system. POST offers a resilient, decentralized infrastructure by providing three basic, efficient services to applications: *(i)* secure, durable storage, *(ii)* metadata based on single-writer logs, and *(iii)* event notification. While ePOST is currently the only deployed application built upon POST, a wide range of collaborative applications, such as instant messaging and shared calendars, can be constructed on top of POST using just these services.

The architecture of ePOST is shown in Figure 12.1. In this chapter, we start by describing scoped overlays, which provide autonomy and locality for organizations using ePOST. Next, we describe the POST system, which provides support for secure, reliable messaging and data storage. We then describe how ePOST is built using POST. Lastly, we detail the Glacier data durability system, which ePOST and POST use to ensure that data is durable even in the event of a large-scale correlated failure.

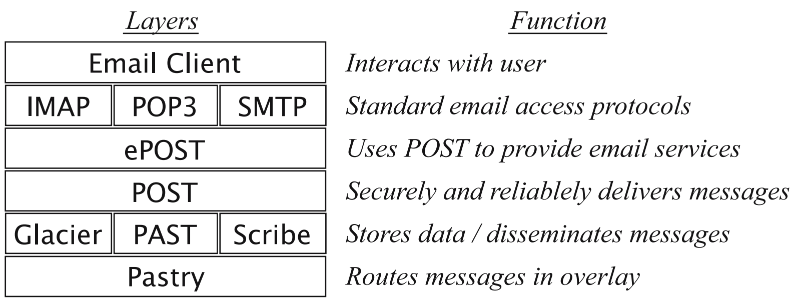
Layers
Function
Email Client
Interacts with user
IMAP
POP3
SMTP
Standard email access protocols
ePOST
Uses POST to provide email services
POST
Securely and reliablely delivers messages
Glacier
PAST
Scribe
Stores data / disseminates messages
Pastry
Routes messages in overlay

Fig. 12.1: ePOST Stack

12.1 Scoped Overlays

In most structured 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 [279]. This is particularly problematic when deploying highly reliable services, as organizations may require that internal data and message traffic remain local. For instance, organizations using ePOST will probably want all intra-organizational email data to stay within the organization, even in encrypted form.

Moreover, participants in a conventional overlay must agree on a set of protocols and parameter settings such as the routing base, the size of the neighbor set, failure detection intervals, and the replication strategy. Optimal settings for these parameters depend on factors like the expected churn rate, node failure probabilities, and failure correlation. These factors may not be uniform across different organizations and may be difficult to assess or estimate in an Internet-wide system. The choice of parameters also depends on the required availability and durability of data, which may differ between participating organizations. Yet, conventional overlays require global agreement on protocols and parameter settings among all participants. For example, Company A may use mostly of desktop PCs with a high churn rate, while Company B may use mostly dedicated servers. In a traditional overlay, these two companies are required to use the same replication and fault tolerance parameters, even though they may be inappropriate for both.

The ePOST system is organized as a hierarchy of overlay instances with separate identifier spaces. This 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. Scoped overlays generalize existing structured overlay protocols with a single ID space, thus leveraging prior work on all aspects of structured Peer-to-Peer overlays, including secure routing [108].

12.1.1 Design

A *multi-ring* protocol interfaces between organizational 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 [148].

Figure 12.2 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; an instance of the gateway node appears in each separate ring. The 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. Thus, in the example discussed above, company A and company B can run separate rings with different protocols and parameters while maintaining connectivity between them.

12.1.2 Ring Structure

The system forms a two-level tree of rings, consisting of a *global ring* at the root and several *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

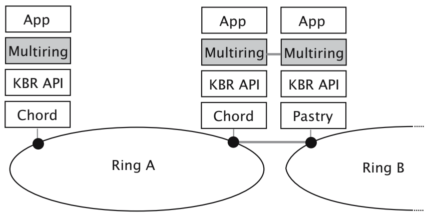

Fig. 12.2: Diagram of application layers. Note that rings may be running different protocols, as in this example.

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 an organizational 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 of which it is a member.

An example configuration is shown in Figure 12.3. Nodes shown in gray are instances of the same node in multiple rings and nodes in black are only in a single ring because they are behind 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 mostly behind a firewall.

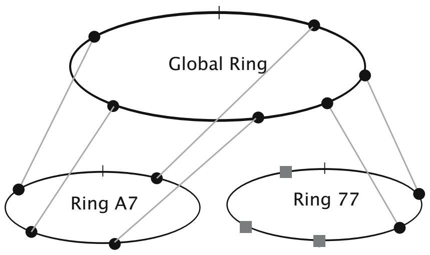

Fig. 12.3: Example of a ring structure.

The global ring is used primarily to route inter-organizational queries and to enable the global lookup of keys, while application objects 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 include nodes that are connected to the Internet through a firewall or NAT box.

12.1.3 Gateway Nodes

Recall that 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 subsection. In Figure 12.3 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 multicast group in each of the rings. The group identifiers of these groups are the ringIds of the associated rings. In Figure 12.3 for instance, a node 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*

12.1.4 Routing

Recall that each node knows the ringIds of all rings in which it is a member. We assume that each message carries, in addition to a target key, the ringId of the ring in which the key is stored. Gateways forward messages as follows. 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 in the target ring, which is accomplished via a Scribe 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 destination's ringId. The message will be delivered by Scribe 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 sender 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 local Scribe 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 identities 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.

12.1.5 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 may need to look up a key without knowledge of where the object 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 even when the ring in which it resides is not known to the requester. When a key is inserted into an organizational ring and that key should be visible at globally, a special indirection record is inserted into the global ring that associates the key with the ringIds of the organizational rings where replicas of the key are stored. The ringIds of a key can now be looked up in the global ring. Note that indirection records are the only data that need to be stored in the global ring. To prevent space-filling attacks, only legitimate indirection records are accepted by members of the global ring

12.2 POST Design

ePOST uses the POST messaging system to provide email services. At a high level, POST provides three generic services: *(i)* a shared, secure, durable 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, such as email, news, instant messaging, shared calendars, and whiteboards.

In a typical pattern of use, users create messages (such as emails in the case of ePOST) 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, such as a private mail folder in ePOST.

POST assumes the existence of a certificate authority. This authority signs identity certificates binding a user's unique name (e.g., his email address) to his public key. The same authority issues the nodeId certificates required for secure routing in Pastry [108]. 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 material.

Though participating nodes may suffer from Byzantine failures, POST also assumes that a large majority (> 75%) of nodes in the system behave correctly, and that at least one node from each PAST replica set has not been compromised. If these assumptions are violated, POST's services may not be available, though the durability of stored data is still ensured thanks to Glacier, an archival storage layer that is described in Section 12.4.2. Additionally, POST makes the common assumption that breaking the cryptographic ciphers and signatures is computationally infeasible.

Table 12.1 shows pseudocode detailing the POST API that is presented to applications. The `store` and `fetch` methods comprise the single-copy message store. Similarly, the `readMostRecentEntry`, `readPreviousEntry`, and `appendEntry` methods provide the metadata service, and the `notify` method represents the event notification service.

The most interesting 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 `readMostRecentEntry`, 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
Key store(Object)
Object fetch(Key)

// and these methods provide the metadata service
LogEntry readMostRecentEntry(LogName)
LogEntry readPreviousEntry(LogEntry)
void appendEntry(LogName, LogEntry)

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

Table 12.1: POST API

12.2.1 Data Types

POST uses the PAST distributed hash table to store three types of data: *content-hash blocks*, *certificate blocks*, and *public-key blocks*.

Content-Hash Blocks

Content-hash blocks, which store immutable data objects such as email data, are stored using the cryptographic hash of the block's contents as the key. Content-hash blocks can be authenticated by obtaining a single replica and verifying that its contents match the key; because they are immutable, any corruption of the content can be easily detected.

Certificate Blocks

Certificate blocks are signed by the certificate authority and bind a name (e.g. an email address) to a public key. Certificate blocks are stored using the cryptographic hash of the name as the key and are also immutable after creation. Certificate blocks can be authenticated based on their digital signature, since all users are assumed to know the certificate authority's public key.

Public-Key Blocks

Public-key blocks contain timestamps, are signed with a private key, and are stored using a secure hash of the corresponding public key as the key. The signature attached to the block allows for block mutation after creation. First, the nodes storing replicas of the block must verify that the signature on the update matches the already-known public key. To prevent an attacker from trying to roll the block back to an earlier valid state, the storage nodes verify that the timestamps are increasing monotonically. Finally, the object requester must obtain all live replicas, verify their signatures, and discard any with older timestamps.

12.2.2 User Accounts

Each user in POST possesses an account, which is associated with an identity certificate. This certificate is stored as a certificate block, using the secure hash of the user's name as the key. Also associated with each account is a user identity block, which contains a description of the user and the contact address of the user's current trusted node. 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 the certificate authority to bind names to keys, while giving users the ability to subsequently change their personal contact data later without having to interact with the certificate authority.

The Scribe group provides a rendez-vous point for nodes waiting for news from the associated user, or anybody wishing to notify the user that new data is available. For example, users waiting for another user A to come online can subscribe to A's group. Once A is online again, he publishes to his group, informing others of his presence.

12.2.3 Single-Copy Store

While POST stores potentially sensitive user data on nodes throughout the network, the system seeks to provide a level of privacy, integrity and durability comparable to maintaining data on a trusted server. A technique called convergent encryption [176] is used. This allows a message to be disclosed to selected recipients, while ensuring that copies of a given plain-text message inserted by different users or applications map to the same cipher-text, thus ensuring that only a single copy of the message is stored.

To store a message X, POST first computes the cryptographic $\mathit{Hash}(X)$, uses this hash as a key to encrypt X with an efficient symmetric cipher, and then stores the resulting ciphertext with the key

$$\mathit{Hash}\left(\mathit{Encrypt}_{\mathit{Hash}(X)}\,(X)\right)$$

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

Convergent encryption reduces the storage requirements when multiple copies of the same content are inserted into the store independently. This happens, for example, when a popular document is sent as an email attachment or posted on bulletin boards by many different users.

In certain scenarios, it may be undesirable to use convergent encryption, such as when the plain-text can easily be guessed. In these cases, the POST store can be configured to use conventional symmetric encryption with randomly generated keys.

12.2.4 Event Notification

The notification service is used to alert users and groups of users to certain events, such as the availability of a new email 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 a newsgroup posting, the intended recipient(s) must be alerted to the availability of the message and be provided with the appropriate decryption key. Commonly, this type of notification involves obtaining the contact address from the recipient's identity block. Then, a notification message is sent to the recipient's trusted node, containing the message's decryption key, and is 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 delegates 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 with B's public key and signed by A. 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 associated with B.

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. Upon receipt of this message, the subscribers deliver the notification by sending it to the contact address. As long as not all of the replica nodes fail at the same time, the notification is guaranteed to be delivered. POST relies on Scribe only for timely delivery – if Scribe messages are occasionally lost due to failures, the notification will still be delivered since users periodically publish to the their group.

12.2.5 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 such a log consisting of insert, update, and delete records to keep track of the state of the folder or newsgroup.

The log head is stored as a public-key block and contains the location of the most recent log record. Keys 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 key of the next recent record in the log. Applications can optionally encrypt the contents of log records depending on the intended set of readers.

To allow for more efficient log traversal, POST aggregates clusters of M consecutive log records in a single PAST object. Partially filled clusters are buffered in the log head object, and are added to the log as a separate cluster entry once they are full. This reduces the number of keys associated with log entries by a factor of M and increases the speed of log traversals accordingly.

Other optimizations are used to reduce the overhead of log traversals, including caching of log records at clients and the use of snapshots. POST applications periodically insert snapshots of their metadata into PAST. Thus, log traversals can be terminated at the most recent snapshot.

12.2.6 Garbage Collection

In order to make the PAST DHT practical for use in applications such as ePOST, we found it necessary to introduce a mechanism for removing objects from the DHT.

Disk space is not necessarily a problem, since the rapid growth in hard disk capacity would probably make it possible to store all inserted data ad infinitum. However, the network bandwidth required to repair failed replicas would become unwieldy over time. Such maintenance is necessary to ensure that there always are at least k live replicas of each stored object, and re-replicating each object as necessary.

The obvious solution is to add a `delete` operation to PAST that removes the object associated with the given key. However, a delete method is unsafe, because a single compromised node could use it to delete data at will. Moreover, safe deletion of shared objects requires a secure reference-counting scheme, which is difficult to implement in a system with frequent node failures and the possibility of Byzantine faults.

As an alternative solution, we added leases to objects stored in PAST. Each object inserted into the DHT is given a expiration date by the inserting node. Once the expiration date for a given object has passed, the storage nodes are free to delete the object. Clients must periodically extend the leases on all data they are interested in. The modified PAST API is shown in Table 12.2.

```
void put(Key, Object, Expiration)
Object get(Key)
void refresh(Key, NewExpiration)
```

Table 12.2: Modified PAST API

Adding leases to PAST required other slight modifications. Specifically, the replication protocol must now exchange tuples (*key, expiration*). When a node is told to refresh a key that it already stored with a different lease, it simply extends the expiration date of the stored key if the new lease is longer.

We cannot assume that the clocks on different storage nodes are perfectly synchronized. 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.

12.2.7 POST Security

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

Threat Model

Our threat model for POST includes of attacks from both within and outside of POST. Internal attacks can be broken down into two classes: free riding and malicious behavior. Free riding, discussed below, 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.

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 be able to determine whether any given node has requested that particular message. This is a particular concern for short messages, messages that are highly structured, or generally any messages with low entropy. To address these concerns, POST uses traditional cryptographic techniques (AES encryption with a random key) to encrypt such messages, and to protect data that is not meant to be shared, such as the logs and other per-user metadata maintained by the system.

Data Integrity

Due to the single-writer property and the content-hash chaining [408] of the logs, it is 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 to chain 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 timestamps. When reading a public-key block (e.g., a log-head) from the store, nodes read all replicas and use the authentic replica with the most recent timestamp. When reading content-hash blocks or certificate blocks, they can use any authentic replica.

Denial of Service

A variety of denial of service (DoS) attacks may be mounted against Peer-to-Peer 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 such DoS attacks, both from within and outside the overlay [108]. When secure routing is used, an attacker would need to control over 25% of the overlay nodes to mount an effective DoS attack.

Another type of DoS attack is space-filling, where a malicious node simply tries to insert as much junk data as possible into the DHT. While this attack is not unique to ePOST, the organizational scoping of rings in ePOST helps to mitigate this attack. Since all nodes in a given ring are in a single administrative domain, space-filling attacks can be detected and the faulty node shut down or punished by the local administrator.

Free Riding

Nodes within the network may try to consume much 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 scarce resources, 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 [448, 135]. Such mechanisms can guarantee that it is rational for nodes to behave correctly.

POST, in its present form, does not yet include any explicit incentives mechanisms [448, 135]. The reason is that within an administrative domain, members generally have external incentives to cooperate. If abuses do occur, they can be localized to an organizational ring, and the offending users can be reprimanded within the organization.

12.3 ePOST Design

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 an 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 material. No other participating nodes in the system are assumed to be trusted by the user.

12.3.1 Email Storage

When ePOST receives messages from a client program, it parsers them into MIME components (message body and any attachments) and these are stored as separate objects in POST's secure store. Recall that 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.

12.3.2 Email Delivery

The delivery of new email is accomplished using POST's notification service. The 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 asymmetric public key cryptography with a fast symmetric cipher.

If the recipient of the 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 the fact that a recipient typically has greater confidence in his own organizational ring. Therefore, ePOST replicates all incoming mail in the recipient's local ring by default.

12.3.3 Email Folders

Each email 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. Since the log can only be written by its owner and its contents are encrypted, ePOST preserves or exceeds the level of privacy and integrity provided by conventional email systems with storage on trusted servers. A diagram of the logs used in ePOST is shown in Figure 12.4.

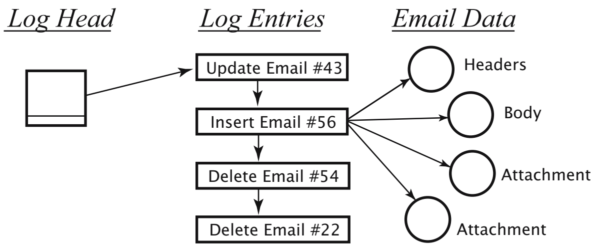

Fig. 12.4: Log structure used in ePOST. Each box or circle represents a separate object in the DHT.

Next, we describe a log record representing an insertion of an email message into a user's folder, such as his inbox. Other types of log records are analogous. An email insertion record contains the content of the message's MIME header, the message's key, and its decryption key, protected by a signature and 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 in Figure 12.5.

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

Fig. 12.5: Messages transmitted sending an email.

12.3.4 Incremental Deployment

To allow an organization to adopt ePOST as its email service, 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 how it interoperates with conventional email services in the general Internet.

For outgoing mail, if an email recipient is not reachable within ePOST, then the sender's ePOST proxy contacts the recipient's mail server using SMTP. For inbound email, the organization's DNS server provides MX records referring to a set of trusted ePOST 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.

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. Administrators of conventional email services own root passwords that allow them to access incoming email anyway. Thus, ePOST provides the same privacy for incoming email from non-POST senders as existing systems, and provides stronger security for email transmitted within ePOST.

12.3.5 Management

If ePOST is to replace existing 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.

Software

The first management task incurred with ePOST is maintaining the proxy software. This software needs to be kept running and up-to-date as bugs are fixed and features are added. In our deployment, the ePOST proxy is configured as a service that is restarted automatically if it fails. Software upgrades are handled by signing updated code and having users' proxies periodically check and download authorized updates.

To allow administrators to efficiently monitor the ePOST system, we have built a graphical administrative monitoring interface. This application allows administrator to monitor an entire ePOST ring at a glance and to track down any problems. Administrators are automatically alerted to error conditions or unusual behavior.

Storage

In a distributed storage system such as ePOST, a certain level of administration is necessary to monitor the storage pool. For example, administrators need to ensure that space-filling attacks are not taking place and that nodes that are running out of disk space are promptly serviced. Such monitoring can be done using the tool described in the previous section. The administrator is alerted to nodes that are close to their disk space limit, and can then take appropriate actions.

Access Control

Controlling access to ePOST can be broken down into two related tasks: trust and naming. Trust is based on certificates, which users must obtain from their organization 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. For example, in our experimental deployment, we provide a web page where users can sign up and download certificates. In practice, the process may require various forms of authentication before the new certificate is produced.

Naming in ePOST is managed in a manner similar to current systems. Organizations ensure that email addresses are unique and associated with only one public key. This is easy to accomplish, since each user must obtain a certificate from his organization.

ePOST has the potential for requiring substantially lower administrative overhead than conventional email systems, 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 by the overlay has the potential to significantly reduce the overhead associated with scaling an existing email service to more users.

12.4 Correlated Failures

ePOST relies on *cooperative storage* to store email messages. Each node is required to contribute a small fraction of its local disk space; the system then aggregates this storage and provides the abstraction of a giant single store. As mentioned earlier, this approach is well suited for serverless applications like POST because it is highly scalable - not only in terms of overhead, but also because as the size of the system increases, the storage supply increases also. This allows the system to support organic growth.

Since the system is built out of unreliable components, it must be prepared to handle occasional node failures. Cooperative storage systems like PAST often assume that the node population is highly diverse, i.e., that the nodes are running different operating systems, use different hardware platforms, are located in different countries, etc. Under these conditions, node failures can be approximated as independent and identically distributed. To ensure data durability, it is thus sufficient to store a small number of replicas for each object, and to create new replicas when a node failure is detected.

Unfortunately, most real distributed systems exhibit high diversity only in some aspects, but not in others. For example, the fraction of nodes running Microsoft Windows can be as high as 60% or more in many environments. In such a system, failures are not independent. For example, if the Windows machines share a common vulnerability, a worm that exploits this vulnerability may cause a *large-scale correlated failure* that can affect a majority of the nodes. Moreover, if the worm can obtain administrator privileges on the machines it infects, the failures can even be Byzantine.

The reactive replication strategy in PAST is clearly not sufficient to handle failures of this type. Even if the failure is not Byzantine, there may simply not be enough time to create a sufficient number of additional replicas. As a consequence, early deployments of ePOST sometimes suffered data loss during correlated failures. Since this is not acceptable for critical data like email, the system needed another mechanism to ensure data durability.

12.4.1 Failure Models

If the system must sustain fast-spreading correlated failures such as power outages or Warhol worms [572], a reactive defense that detects and repairs failures as they occur is not enough. Instead, the system must b e *proactive* and prepared for the failure in advance.

An ideal proactive system would foresee which nodes are going to be affected by the next failure and then store the data on the remaining nodes. This method has zero overhead but is infeasible, so practical systems must use an approximation. A common technique, which is used in systems like Phoenix [329], is to use *introspection* to collect information about each node, which is then used to predict correlations between the nodes. The data is then stored on a set of nodes that are expected to fail with low correlation.

Introspective systems are still very storage efficient but crucially depend on the correctness of their failure model. Even small inaccuracies may lead to incorrect placement decisions and thus to data loss in a correlated failure. Moreover, the participants in an introspective system actually have an incentive to report incorrect data, e.g. to reduce their load by making their node

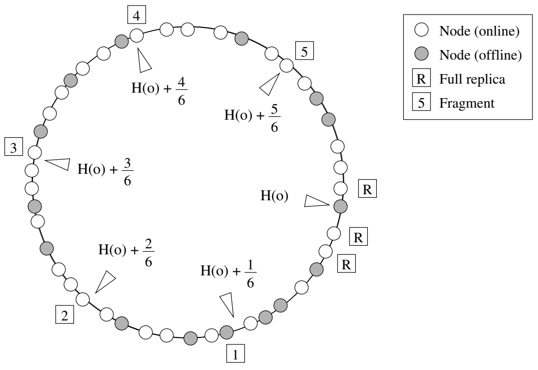

Fig. 12.6: Fragment placement in Glacier.

appear heavily correlated with others. Finally, it is very difficult to identify *all* potential sources of correlation in a realistic system.

12.4.2 Glacier

The durability layer in POST, which is called Glacier, takes a different approach [269]. Instead of relying on a sophisticated failure model, it makes a very simple assumption, namely that the correlated failure does not affect more than a fraction f_{max} of the nodes; all failure scenarios up to that fraction are assumed to be equally likely. In order to tolerate such a wide range of failures, Glacier must sacrifice some capacity in the cooperative store for additional redundancy; thus, it trades abundance for increased reliability.

When a new object is inserted, Glacier applies an *erasure code* to transform it into a large number of *fragments*. Together, the fragments are much larger than the object itself, but a small number of them is sufficient to restore the entire object. For example, Glacier may be configured to create 48 fragments, each of which is 20% the size of the object. This corresponds to a storage overhead of 9.6, but the object can be restored as long as any five fragments survive.

Glacier then attaches to each fragment a so-called *manifest* which, among other things, contains hashes of all the other fragments. This is used to authenticate fragments. Finally, Glacier spreads the fragments across the overlay, calculating the key of fragment i as

$$k_i = K + \frac{i}{n+1}$$

where K is the key of the object and n is the total number of fragments. This ensures that the fragments are easy to find without extra bookkeeping (which may be lost in a failure). Also, if the overlay is large enough, each fragment is stored on a different node, which ensures that fragment losses are not correlated.

For security reasons, Glacier does not allow fragments to be overwritten or deleted remotely. If these operations were permitted, a compromised node could use them to delete its own data on other nodes. However, objects may be associated with a lease, and their storage is reclaimed when the lease expires. Also, Glacier supports a per-object version number to implement mutable objects.

12.4.3 Maintenance in Glacier

Since some fragments are continually lost due to individual node failures and departures, Glacier implements a maintenance mechanism to reconstruct missing fragments. However, because of the high redundancy, Glacier can afford high latencies between the loss of a fragment and its recovery; thus, the maintenance mechanism need not be tightly coupled.

Because of the way fragments are placed in the ring, each Glacier node knows that its peers at a distance $k \cdot \frac{1}{n+1}$ $(k = 1..n)$ in ring space store a set of fragments that is very similar to its own. Thus, each node periodically (say, once every few hours) asks one of its peers for a list of fragments it stores, and compares that list to the fragments in its own local store. If it finds a key for which it does not currently have a fragment, it calculates the positions of all corresponding fragments and checks whether any of them fall into its local key range. If so, it asks its peers for a sufficient number of fragments to restore the object, computes its own fragment, and stores it locally.

Glacier also takes advantage of the fact that nodes often depart the overlay for a certain amount of time (e.g. because of a scheduled downtime) but return afterwards with their store intact. Therefore, Glacier nodes do not immediately take over the ring space of a failed neighbor, but wait for a certain grace period T. If the node returns during that time, it only needs to reconstruct the fragments that were inserted while it was absent; the vast majority of its fragments remains unmodified.

The loosely coupled maintenance mechanism greatly reduces the bandwidth required for fragment maintenance. In the actual deployment which has moderate churn and the configuration mentioned earlier, Glacier uses less bandwidth than PAST, even though it manages over three times more storage.

12.4.4 Recovery After Failures

A large-scale correlated failure has two main effects on a Glacier deployment: First, a large fraction of nodes may lose the fragments they store locally, and second, the overlay may be shattered, and communication with other nodes may become impossible. Both effects may be aggravated by Byzantine failures: Malicious nodes may corrupt their local fragments in order to complicate recovery, and they may mount attacks on the overlay (e.g. an Eclipse attack) to interrupt communication. However, even a malicious node cannot change its certified nodeId and take over a portion of the ring space that is occupied by an unaffected node. Therefore, the fragments on the surviving nodes remain safe as long as their leases do not expire.

While the failure lasts, we cannot assume that the unaffected nodes can make any progress towards recovery, since this would require communication with other nodes. Therefore, these nodes simply 'weather the storm' and do nothing. Eventually, the administrators of the failed nodes will notice the problem and repair their nodes. After that, the maintenance mechanism will gradually recover the lost fragments and thus restore full redundancy. To prevent congestion collapse, the amount of bandwidth each node is allowed to spend on maintenance is limited, so full recovery may take several hours to complete; however, even though the data is not fully durable during that time, it still remains available and can be retrieved on demand.

12.4.5 Object Aggregation

In ePOST, the storage load mainly consists of small objects (email texts and headers). This causes more overhead in Glacier because the number of keys is higher, and thus more storage space and bandwidth is required for per-key metadata such as the fixed-size manifests. To reduce this overhead, ePOST aggregates objects before inserting them into Glacier.

The main challenge in object aggregation is how to do it securely in an environment with large-scale Byzantine failures. Even though there are considerable advantages in performing aggregation on the storage nodes, Glacier cannot allow this because these nodes cannot be trusted. Therefore, each node is required to create and maintain its own aggregates. This includes keeping a mapping from object keys to aggregate keys (which is required to locate objects), extending the leases of aggregates whose objects are still in use, and merging old aggregates whose objects have mostly expired.

The mapping from object keys to aggregates requires special attention because it is crucial during recovery. Without it, the application may be unable to find its objects after a failure without searching Glacier's entire store, which is infeasible. For this reason, the system adds to each aggregate a few pointers to other aggregates, thus forming a directed acyclic graph (DAG). During recovery, an ePOST node traverses its DAG and is thus able to locate all non-expired objects it has inserted. Moreover, the DAG contains a hash tree, which is used to authenticate all aggregates. The only additional requirement for ePOST is to maintain a pointer to the top-level aggregate; this pointer is kept in an object with a well-known key that is directly inserted into Glacier.

12.5 Preliminary Experience

We implemented a version of POST and ePOST on top of FreePastry, an open-source implementation of Pastry, PAST and Scribe, and the POST and ePOST code was released alongside FreePastry 1.4 [233]. 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. Many of our users rely on ePOST as their primary email system, no longer using their conventional accounts. For more information on our deployment, please see `http://www.epostmail.org`.

The current ePOST deployment has two separate ePOST rings: a ring at Rice University limited to members of Rice only, and a ring based on PlanetLab [486], which is open to the public. We currently have approximately 20 registered users in the Rice ring and 73 registered in the PlanetLab ring. We have found the storage and bandwidth requirements to be relatively modest: the average storage requirement on the Rice nodes after one year of use was approximately 500 MB, and the average bandwidth usage was 500 bytes per second per node.