P2P E-Mail

Jussi Kangasharju Keith W. Ross David A. Turner

December 7, 2002

Abstract

1 Introduction

E-mail is a mission-critical communication function for virtually all institutions, including corporations, universities, militaries, and families. Given the paramount importance of e-mail in modern society, it is disturbing how vulnerable the e-mail user can be to attacks and failures. Indeed, modern e-mail employs a server-centric design, in the user is critically dependent on its mail server, which receives, stores, and provides access to the user's inbox. If this mail server is down – due to, for example, inadvertant faults, disasters, physical attacks or cyber attacks – the user can neither receive nor access its messages.

Many mail service providers - including Hotmail, YahooMail, and Critical Path - have patched the dependability problem by creating mail-server clusters. The cluster is not only responsible for receiving, storing and delivering many users' messages, but it also replicates messages across the different servers in the cluster. Thus, if one of the mail servers becomes unavailable, the cluster can continue to receive, store, and deliver messages. However, this patch only provides a marginal improvement in dependability, as it does not address scenarios in which the entire cluster is taken out, such as when the cluster is behind an access link that is severed for flooded with denial-of-service traffic, or when the building housing the cluster is physically bombed.

In addition to providing insufficient dependability, today's server-centric approach suffers from a number other problems [**?**] [**?**], including storage stress due to attachments with multiple recipients; and server processing stress, requiring mail server providers to deploy hundreds of machines in their clusters.

In this paper we describe an architecture, and a corresponding prototype that we have built, for serverless P2P e-mail. Our P2P e-mail application runs directly over a distributed hash table (DHT) substrate, such as CAN [1], Chord [2], Pastry [3], or Tapestry [4]. Although our architecture requires a DHT substrate, it can use any DHT substrate that provides a key-to-node mapping. Furthermore, our architecture does not use an intermediate persistent file-system layer such as CFS [5], PAST [6] or OceanStore [7]. The architecture is resilient to faults, disasters, and attacks, can diminish server storage and processing stress, and provides better security and privacy than ordinary e-mail.

This paper is organized as follows. Section **??** presents an overview of the problem we are tackling. In Section **??** we present an overview of our architecture and how users can send and read email with it. In Section **??** we describe our prototype implementation of P2P email. In Section **??** we present some extensions to our basic architecture. Section **??** discusses the future of P2P email and applications and Section 5 concludes the paper.

2 Architectural Preliminaries

We have designed a simple store and forward e-mail delivery system. In our design, we leave the preservation and management of retrieved e-mail messages to the user agent (UA), whether this be done in local storage or in a distributed file system. We have chosen this design to increase the ease of adoption by individual e-mail users and by e-mail service providers.

2.1 Basic Components

Our system requires the external services of a DHT substrate. A DHT substrate consists of a large number of computers (hundreds to billions), called **nodes** or **peers**. Each node has a nodeID in the **DHT space**,

Table 1: Objects and their identifiers

Object	**Identifier**
Inbox	e-mail address
e-mail address certificate	e-mail address and ”-certificate”
Message body	RFC 822 Message ID

which is the set of all binary strings of some fixed length along with a metric. We assume that the DHT substrate provides applications the following **lookup service**: The application supplies an arbitrary key (an element in the DHT space) and a variable k, and the lookup service returns to the application the k active nodes in the DHT that are the closest to the key.

The system is comprised of nodes and UAs. The nodes are the computers in a DHT substrate. The role of the nodes is to provide persistence for messages that are in transit from sender to recipient. The UAs are the mail reader programs that are run by the users. The UAs access the e-mail system through the system nodes. A UA may or may not be running on a system node (that is a node in the DHT substrate); in the case when it is not, the user agent must have the IP address of at least one of the system nodes to access system functionality.

Each user, such as Alice, has an e-mail address, which Alice can publish and which other users can send e-mail to. Our system also requires the external services of a certificate authority, which creates **e-mail address certificates** that bind e-mail addresses to public keys.

Alice has an incoming **inbox**, which is associated with her e-mail address. As we shall discuss subsequently, Alice’s inbox only stores notifications of unread messages; the e-mail message bodies themselves are stored separately under their own unique identifiers.

Thus, our P2P e-mail system uses the DHT substrate to store three types of objects: e-mail address certificates, e-mail message bodies, and inboxes. Table **??** lists these objects along with their respective identifiers. A fixed hash function is used by all UAs and nodes to map an object’s identifier to a key, which is an element in the DHT space.

2.2 Service Primitives

Our P2P e-mail system has five service primitives that can be invoked on individual nodes, which are: store, fetch, delete, append-inbox and flush-inbox. User agents call these service primitives to perform the higher-level system functions of sending an e-mail, retrieving an e-mail, and deleting an e-mail. UAs invoke these services on individual nodes in the DHT substrate; thus, to invoke the service on a particular node, the UA must know the IP address of the node.

In order to avoid complex synchronization procedures, we require no coordination between the storage nodes - the UAs are responsible for replicating data across the DHT nodes. We now describe the five service primitives.

Store service primitive

The store function is used to store e-mail message bodies and e-mail address certificates. The store function takes as arguments an object, the object's identifier, and a set of e-mail address certificates for the users who have permission to delete the object. In the case of when the object is an e-mail message body, the object identifier is the RFC 822 message ID, and the certificate set contains the certificates of each recipient. In the case of when the object is an e-mail address certificate, the object identifier is an e-mail address appended with "-certificate," and the certificate list will contain the certificate itself to enable the owner of a certificate to remove it from storage.

As we will describe more fully in a subsequent section, a UA uses the store primitive to store an e-mail message body in the k nodes responsible for the message body. When a UA creates a message, it hashes the message ID to obtain a key. The UA then uses the DHT lookup service to obtain the IP addresses of k nodes that are closest to the key. The UA then sends a copy of the e-mail message body to each of the k nodes, asking each one to use its store service primitive, and providing each one with the message body, the message ID, and the set e-mail address certificates.

Delete service primitive

The delete function is used by a requester to reclaim storage space in an individual node by removing

unneeded objects from its storage. The delete function takes an object identifier and a requester's e-mail address as its arguments. When the receiving node gets a request to delete an object, it locates the requester's certificate in the object's certificate list, and uses it to authenticate the requester. After authentication, the receiving node removes the requester's certificate from the object's certificate set. If the resulting certificate set is empty, the object is discarded. If the certificate set is not empty, the receiving node maintains the object. This procedure enables e-mails with multiple recipients to consume the same amount of storage resources as e-mails with a single recipient.

It should be noted that the UA's list of k nodes closest to the key for a message body may include some nodes that do not contain the message body. This will be the case when new nodes join the network with node IDs that are close to the key. The UA may attempt to reach older nodes by widening its search to include more than k nodes. Otherwise, garbage collection procedures performed by individual nodes will eventually remove the certificate from storage.

Fetch service primitive

The fetch function is used to retrieve stored objects. The fetch operation takes the object identifier. The operation returns the object, be it an encrypted message body, or an email address certificate.

Append-Inbox service primitive

The append-inbox function is used by a sender to append email message headers to a recipient inbox, which is a container of message notifications for the recipient. Because multiple nodes are used to maintain instances of this inbox, the inboxes will not necessarily be consistent, because of unsynchronized message delivery from multiple senders to a single recipient, along with the possibility of one or more inbox nodes arriving or departing the DHT substrate. Thus, a user's inboxes are not required to be consistent in the same manner as the persistent objects created through the store operation. The append function takes as arguments an e-mail address and the encrypted email message headers. The e-mail address identifies the recipient of the email, and the email message headers are encrypted with the recipient's public key.

Read-Inbox service primitive

The user agent calls the read-inbox function on a node when it wants to retrieve message notifications placed into its user's inbox. The function takes an e-mail address, and returns (and deletes) the list of message notifications stored on the node. The node will permit this operation only to user agents that can authenticate themselves as the owner of the email address.

2.3 Garbage collection

A node may perform garbage collection in the normal course of events, or in response to a storage request for an object of size that exceeds currently available resources. The node maintains a list of its stored objects sorted by key. Starting from either end of the list, the node checks to see if it is still one of the k closest nodes to the object. If it isn't, then this object can be removed from its store. This procedure is continued until there is enough available storage for the new object. If the procedure terminates without reclaiming sufficient additional storage, it reports this failure to the requesting UA.

2.4 Persistence of data

We assume the following conditions: (1) k is sufficiently large, (2) the growth rate in terms of number of nodes in the DHT substrate is sufficiently slow, (3) inboxes are read with high enough frequency, and (4) message bodies are fetched sufficiently soon. These conditions ensure that messages are not lost due to garbage collection before they are delivered. However, we do not assume this is the case for email address certificates. For this reason, we specify a mechanism whereby persistence of e-mail address certificates are ensured. Whenever a user agent requests a certificate from a node that is one of the k closest nodes to the certificate, but that node returns a failure response, the user agent performs a store operation on that node for that certificate (which it fetches from another node). Additionally, the user agent of the certificate's owner can periodically check to see that the k closest peers each has a copy of her certificate. If not, the user agent invokes a store operation on that node to place the certificate into its storage.

3 P2P E-mail Mechanics

The e-mail system is built on top of an overlay network comprised of semi-reliable nodes. For this reason, data is replicated across a sufficient number of nodes to guarantee persistence. The user agents handle replication of data through the service primitives that are exposed by the nodes. In the following sections, we explain how reliability and privacy is accomplished.

3.1 E-mail message creation

To understand how the system delivers email, we describe the sequence of events that occur when Alice sends Bob an email message. We refer to Alice's user agent as A.

1. A appends Bob's email address with "-certificate," and maps this to a key. A uses the lookup server to obtain the list of k nodes closest to the key, and invokes the fetch operation on one of these nodes for Bob's certificate. A authenticates the certificate, and extracts Bob's public key.

2. A generates a session key, and uses it to encrypt the e-mail message body.

3. A generates an RFC 822 message ID that will be used to identify the message body.

4. A maps the message ID to a key. A uses the lookup service to obtain the k nodes closest to the key, and invokes the store operation on each of them, using the message ID as identifier, and the encrypted message body as object.

5. A constructs the e-mail message headers (which includes the session key used to encrypt the message body and the message ID to locate the body), and encrypts them with Bob's public key. A maps Bob's e-mail address to a key. A obtains from the lookup service the k nodes closest to the key, and invokes the append-inbox function on each of them with Bob's e-mail address and the encrypted message headers.

6. If A fails to complete an append operation with any of the k nodes, because of node failure or

network failure, it requests a fresh set of k nodes, and tries to complete the operation with the new nodes it receives. The same procedure is used if A fails to complete an initial attempt at a store operation for the message body.

3.2 Reading and flushing the inbox

When Bob wants to read his new messages, he instructs his user agent B to retrieve them. B obtains the k nodes closest to Bob's e-mail address, which are the nodes to which senders are appending message notifications. Because there is a chance that the inboxes stored across these nodes are inconsistent, B invokes the read-inbox operation on all k nodes, and forms the *superset* of message notifications returned from all k nodes. Each node will delete the message notifications once it has sent them to the user agent, and so the user agent becomes wholly responsible for maintaining the persistence of these e-mail message notifications. B decrypts the message headers using Bob's private key.

3.3 Retrieving a message

When Bob wants to read a particular e-mail, the user agent obtains the k nodes closest to the key of the message ID. The user agent invokes the fetch operation on one of the nodes, and verifies that the message body is valid by comparing a digest of the retrieved object with a digest that the sender placed in the message headers. If the message body is not on the node, or if the message digest is not valid, the user agent invokes the fetch operation on one of the other nodes, and repeats until it obtains a satisfactory result.

Once the message body object has been obtained, the user agent decrypts the object with the session key that the sender placed in the message headers. The user agent invokes the delete operation on all k peers to remove its certificate from the list of certificates attached to the message body. If Bob's certificate is the last on the list, the node will remove the object from storage.

Because the message headers have left peer storage, there is little motivation for user agents to leave message bodies in peer storage, other than to avoid the consumption of local storage. For this reason,

user agents are expected to perform message deletion immediately following message retrieval.

4 Anonymous Senders

When anonymity is important to the sender, there is a procedure to accomplish this. The procedure is based on the ability to establish an anonymous communication channel with a random node. We start by explaining how this works.

4.1 Anonymous communication channels

The following procedure is used for a node A to establish an anonymous communication channel with a random node Z. A generates a random key (keyZ), and desires to establish an anonymous communication channel with the node that is closest to keyZ. A cannot route a message in the P2P network to find Z, because Z may get the message and therefore know A. To find Z, A does the following: it generates another random key (keyX), and it finds the node X closest to keyX. A asks X to find the node closest to keyZ. X finds Z, and returns Z's certificate to A.

It is possible that X is hostile, in which case, Z will be bogus. To guard against this, A generates another random key (keyY) and finds the closest node to it (call it Y). It asks Y to locate the node closest to keyZ. If the certificate that Y returns matches what it got from X, then A is assured that Z is valid (that is, not under the control of X).

Assuming A determines that it has a valid certificate for Z, A can now encrypt its messages to Z by encrypting them with Z's public key. However, Z can't use A's public key, otherwise A loses anonymity. Therefore, A generates a session key to give to Z. A encrypts the session key with Z's public key, and asks the go-between node X to deliver this to Z. Now, whenever Z wants to say something to A, it encrypts the message with the session key, so that the go-between node can't read it.

4.2 Anonymous Mail delivery

Suppose Alice sends email to Bob, and that Alice's user agent is on peer A. A must do two things: it must store the message body on the k nodes closest to the key of the message ID, and it must append the encrypted message headers to Bob's inbox on the k nodes closest to the key of Bob's email address. The following two steps show how these functions can be accomplished while keeping A anonymous to everyone except Bob.

Step 1: storing the message body

A establishes an anonymous communication channel with Z, and asks Z to fetch the certificates of the k nodes closest to the key of the message ID. A establishes an anonymous communication channel with another random node W, and requests the same information. A then compares the node list and certificates returned by Z and W; if they match, A is assured that it has valid certificates.

A encrypts the email message body with Bob's public key, and asks Z to store it on the k peers closest to the key of the message ID. Each of these peers signs a digest of the encrypted message, which Z returns to A as proof that the requested storage operation succeeded.

Step 2: appending the message headers to the inbox

The procedure is identical to step 1, except that the requested operation is to append rather than store.

The peers storing the inbox can count the number of messages sent to Bob, but they do not know the senders. The random peers involved as go-betweens know only that Alice sent at least one message. The random peers involved as anonymizers know that Bob received at least one message.

The protection scheme will fail if the random keys keyX and keyZ resolve to two cooperating hostile peers. In this case, the hostile peers will know that Alice sent at least one message to Bob.

5 Conclusion

In this paper we have presented an architecture for implementing an email service on a DHT-based P2P network. Our architecture eliminates the single-point of failure of modern mail servers, reduces stress

on the mail servers, and it handles mailing lists in a scalable manner. This email architecture is meant as a first step towards understanding how complex applications can be built on top of unreliable P2P networks. We have also presented a prototype implementation of our email architecture.

References

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *SIGCOMM*, San Diego, CA, Aug. 2001.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, San Diego, CA, Aug. 2001.

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001.

[4] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB//CSD-01-1141, UCB, Apr. 2000.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SOSP*, Banff, Canada, Oct. 2001.

[6] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, per sistent peer-to-peer storage utility," in *ACM SOSP*, Banff, Canada, Oct. 2001.

[7] J. Kubiatowicz et al. , "OceanStore: An architecture for global-scale persistent storage," in *ASPLOS*, Boston, MA, Nov. 2000.

[8] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, June 1999.

[9] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," in *Proc. 1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA, 2002.

[10] P. V. Mockapetris, *RFC 1035: Domain names — implementation and specification*, Nov. 1987.

[11] R. Cox, A. Muthitacharoen, and R. Morris, "Serving DNS using a peer-to-peer lookup service," in *Proc. 1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA, 2002.

[12] J. Klensin, *RFC 2821: Simple Mail Transfer Protocol*, Apr. 2001.