Secure and Resilient Peer-to-Peer E-Mail: Design and Implementation

Jussi Kangasharju
Telecooperation Group
Dept. of Computer Science
TU Darmstadt
Darmstadt, Germany

Keith W. Ross
Dept. of Computer and
Information Science
Polytechnic University
Brooklyn, NY

David A. Turner
Dept. of Computer Science
CSU San Bernardino
San Bernardino, CA

Abstract

E-mail is a mission-critical communication function for virtually all institutions. Modern e-mail employs a server-centric design, in which the user is critically dependent on her mail server. In this paper we present a peer-to-peer (P2P) email architecture that eliminates the need to rely on a single server and boosts the resilience of email against any kinds of attacks. Our architecture also provides confidential communications for all users. We present how the basic mechanisms of sending and reading email are implemented in our architecture. We also consider additional schemes to improve anonymity in our architecture. We present our prototype implementation and discuss the future of P2P communication architectures.

1. Introduction

E-mail is a mission-critical communication function for virtually all institutions, including corporations, universities, armies, 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 which the user is critically dependent on her mail server, which receives, stores, and provides access to the user's inbox. If this mail server is down – due to, for example, inadvertent faults, disasters, physical or cyber attacks – the user can neither receive nor access her messages.

Many mail service providers – including Hotmail, Yahoo, 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 resilience, 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 or flooded with denial-of-service traffic, or when the building housing the cluster is physically destroyed. Although it is possible to distribute the cluster (even across continents), operating such a cluster is also extremely expensive. Our architecture does not incur any additional costs since it uses resources (disk space, processing capacity) that already exist on the computers of the users.

In addition to providing insufficient resilience, today's server-centric approach suffers from a number other problems [9, 10], 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. Even on a smaller scale, such as in a university, the mail servers have become a severe bottleneck because the mail servers are responsible for filtering out spam and checking messages for viruses.

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 [5], Chord [8], Pastry [6], or Tapestry [11]. 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 [1], PAST [7] or OceanStore [3]. 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 2 presents the architectural components of our system. Section 3 presents the details of P2P e-mail delivery and Section 4presents a reliability analysis. In Section 5 we consider interoperability with standard email protocols. Section 6 presents our prototype implementation of the P2P email architecture. Section 7 discusses the future of P2P email systems. Finally, Section 8 concludes the paper.

2. Architectural Preliminaries

As a first step, 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.

In order to study the feasibility and requirements of a P2P email system, we have kept the features of our architecture to a bare minimum. This architecture is not meant to replace the current server-based email architecture; instead our goal is to identify the key requirements of such an architecture.

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 millions), called **nodes** or **peers**. Each node has a $nodeID$ in the **DHT space**, 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[1].

The system is comprised of system nodes (also called nodes) and UAs. The system nodes are the computers in a DHT substrate. The role of the system 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 UA 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. We also require the external services of a certificate authority, which creates **e-mail address certificates** that bind e-mail addresses to public keys. Alice's public key is contained in the certificate and we assume that Alice has access to her private key on her local computer in a trusted manner.

Alice has an incoming **inbox**, which is associated with her e-mail address. Alice's inbox only stores notifications of unread messages; the e-mail message bodies themselves are stored separately under their own unique identifiers. Figure 1 shows an overview of the inbox and messages. Our initial architecture is similar to the traditional POP-based email architecture, that is, Alice keeps track of the messages she has read on her computer. The same computer also stores information about all folders (except inbox). When Alice reads her new messages she downloads them to her computer and deletes to message bodies on the peers. (It is possible to leave them on the peers, but as we present later, our architecture does not guarantee their persistence.)

[1] Pastry and Tapestry provide this functionality; Chord and CAN would need to be extended to get k closest nodes.

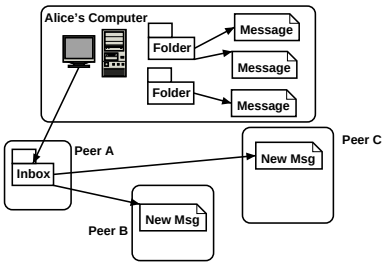


Figure 1. Overview of architecture. Peer A stores Alice's inbox and peers B and C store new message bodies.

Object	Identifier
Inbox	E-mail address
E-mail address certificate	E-mail address and "-certificate"
Message body	Message ID header

Table 1. Objects and their 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 1 lists these objects along with their respective identifiers. All UAs and system nodes use the same hash function 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: store, fetch, delete, append-inbox and read-inbox. User agents call these service primitives to perform the higher-level system functions of sending, retrieving, and deleting e-mails. 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. We assume that the IP address is returned by the lookup service when the UA queries it.

In order to avoid complex synchronization procedures, we require no coordination between the system nodes – the UAs are responsible for replicating data across the system nodes. We stress that our architecture does *not* require any

consistency mechanisms for the stored objects, since we do not need the ability to modify the objects. 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 that the object is an e-mail message body, the object identifier is the RFC 2822 message ID, and the certificate set contains the certificates of each recipient. In the case that 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 of 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 system 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.

We note 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 $nodeID$s 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 message 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. Our architecture does not require that the inboxes remain consistent; instead we will combine the (possibly) different inboxes while reading them (see Section 3.2 for details). 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

A UA 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 message notifications stored in the node. The node will permit this operation only to UAs that can authenticate themselves as the owner of the email address.

Note that although the append-inbox and read-inbox primitives cause changes to the stored objects, they do not require any coordination between system nodes. Each system node which receives one of these primitives needs only to enforce local consistency, i.e., each system node can decide its own order of competing append-inbox and read-inbox calls. The UA will later resolve the (possible) differences between different system nodes.

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. For each item on the list, the node checks to see if it is still one of the k closest nodes to the object on the end of the list. 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.

If we order the list of stored objects according to their access times, we can use the Least Recently Used (LRU) replacement policy. This avoids removing objects which are still needed since it would first target objects which have not been accessed recently. Note that even with the LRU replacement policy, the node would still verify whether it is part of the k closest nodes to the object. If that is the case,

then the object would not be subject to removal.

It is, in principle, possible that the system runs out of storage space. This is unlikely, though, since it would require that users send enough mails to fill the system nodes, but that they never read their mails (which normally triggers deletion).

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 system nodes. In the following sections, we explain how reliability and privacy are 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 service to obtain the list of k nodes closest to the key, and fetches Bob's certificate from one of these nodes. 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 include the session key, the message ID header, and a digest of the message), 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.

3.2. Reading and flushing the inbox

When Bob wants to read his new messages, his user agent 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. (As Figure 1 shows, it is possible to let the messages remain on the peers, but unless they are accessed regularly, they are likely to be evicted through garbage collection. This discourages using the email storage as a distributed storage system for files.)

3.4. Virus Checks and Spam Filtering

One important feature of our architecture is that it allows for complex spam processing and checking without overloading a mail server. Because the messages are encrypted, any filtering or virus checking can only take place when the message is decrypted at the UA. Hence, we propose to include these functions as an integral part of the UA.

The UA periodically updates its virus definitions from a known source, such as the web site of the manufacturer of the anti-virus software. Before showing the message to the user, the UA will check it for viruses and if any are found, the message is quarantined.

Spam filtering can function the same way, by downloading a list of known spammers from some well-known source. Alternatively, a user can require all messages to be signed by the sender and allowing only authorized senders to send messages. (This would require some additional mechanism for people who are not on the whitelist to be able to send email, such as a confirmation message.)

Currently filtering and virus checks are typically performed by the mail server of the user, in order to guarantee the same service for all users. This additional processing can, however, put a considerable strain on the mail server. Our architecture offloads this processing on the peers. If this functionality is provided as an integral part of the mailreader and it automatically updates itself as presented above, we believe that the users will enjoy the same level of filtering and virus checking.

4. Persistence of Data

Stored objects may become unavailable because of nodes going up and down or joining the network, or through garbage collection. In this section, we will investigate the effects of node dynamics on the persistence of objects.

Suppose the whole network comprises of I nodes, and that a message (or inbox or email address certificate) is replicated on k nodes. Suppose that a node may be down at any given time with probability p.[2] Furthermore, suppose that N nodes join the network between the creation of the message and the time the user reads it. We also assume that nodes are uniformly distributed in the DHT space.

The message may be lost if one of three things happen. First, all the k nodes with the message may be down when the user reads the message. Second, k nodes that were down when the message was created were actually the k closest nodes and they are up when the message is read. Third, k of the N new nodes are closer to the ID of the message than any of the k nodes (i.e., the new nodes become the new k closest nodes). Note that second and third cases can be solved by extending the search beyond k nodes. The probability of the first case happening is equivalent to

$$p_{l_1} = (1-p)^k. \tag{1}$$

At any given time, $(1-p)I$ nodes on average are down. For simplicity, we assume this to be an integer and then the probability of the second case happening is

$$p_{l_2} = \sum_{i=k}^{(1-p)I} \binom{(1-p)I}{i} \left(\frac{p(1-p)}{I}\right)^i \tag{2}$$

[2]For simplicity, we have assumed nodes to be homogeneous. In the real world node up probabilities would be heterogeneous and our analysis would give results for the average case.

$p_{l_3} \approx$

I		
10^2	10^3	10^4
10^{-10}	10^{-15}	10^{-20}

p	$p_{l_1} \approx$
0.99	10^{-10}
0.9	10^{-8}
0.5	0.03
0.3	0.17

$p_{l_2} \approx$ (for given I and p)

0	10^{-15}	10^{-15}
10^{-8}	10^{-8}	10^{-8}
10^{-4}	10^{-4}	10^{-4}
10^{-3}	10^{-3}	10^{-3}

Table 2. Probability of loss. Orders of magnitude for p_{l_1} is on the left and for p_{l_3} at the top. The inner cells give the order of magnitude of p_{l_2}

In other words, at least k nodes were down and when they came up, they turned out to be closer than any of the old nodes. The probability of the third case happening is

$$p_{l_3} = \sum_{i=k}^{N} \binom{N}{i} \frac{1}{(I+1)^i} \tag{3}$$

Table 2 shows approximate values of (1), (2), and (3) for $k = 5$ and different values of p and I. As we can see, p_{l_1} typically dominates over the other two cases. The loss could also happen because of a combination of all the three causes, but the probability of this would still be mainly determined by p_{l_1}. In the following, we will only consider the possibility of the k nodes being down.

Given the above definitions, when a user reads her message, the probability of success is

$$p_s = 1 - p_{l_1} = 1 - (1-p)^k. \tag{4}$$

From a user's point of view, a successful action requires several operations to succeed. For example, to read new messages, the user needs to be able to retrieve her inbox *and* also the new message. Hence, we need two successful operations in a row to complete the user's action. Likewise for sending a message, where the sender needs to retrieve the recipient's email address certificate and append the message headers to the inbox. Deleting does not require any successful operations, since the garbage collection will eventually remove the message from storage. In the following, we consider that we require two successful operations to complete a user's action successfully. Note that a failure here only means a *temporary failure*; the UA can repeat the operations later, until they succeed.

We define p_t as the probability of success we want our email application to provide to users. That is, we want that any action taken by any user succeeds with probability p_t. By varying p_t we can provide different levels of quality of service to the users.

Let p_t be our target level of quality of service and let n be the number of operations we need to complete an action. Then

$$p_t = p_s^n = (1 - (1 - p)^k)^n \quad (5)$$

Solving for k we get

$$k = \frac{\log(1 - \sqrt[n]{p_t})}{\log(1 - p)} \quad (6)$$

This is the number of copies we should create to provide the desired level of quality of service (p_t) given the up probability of the nodes (p). In Figure 2 we shows what value of k gives us the desired p_t for three values of $n = 2$, 100, and 10000. We show values of p_t from 99% up to 99.999%. Although an overall target probability of 99% is likely to be too low for important applications (1 action in 100 cannot be immediately satisfied), it gives us insight how much we need to increase k to get an "extra nine" in p_t.

Figure 2(a) shows the basic case of completing one action. As we can see, when p is relatively high, $p > 0.75$, we need roughly 1 or 2 more copies to get an extra nine, and when nodes are mostly down, $p = 0.3$, then we need roughly 5–7 additional copies for the same performance increase.

Figures 2(b) and 2(c) show the behavior of (6) for higher values of n. When n is 100, this is equivalent to sending an email to 50 recipients and $n = 10000$ corresponds to 5000 recipients (e.g., a mailing list). As we can see, for higher values of p, the required increase in k is quite small, only a few copies more are needed to complete 5000 actions successfully. For lower values of p, the additional number of copies is higher, as is to be expected.

Interestingly, the number of additional copies to get an extra nine is roughly constant, even when we need to complete a large number of actions.

In the real world, a node would not know the value of p, hence it must adaptively decide on a correct value of k. A node (or UA) can determine this by storing small files in the system and later trying to retrieve them. Depending on the level of success, the node can determine what the average p is and, consequently, what value of k is suitable.

5. Interoperability Issues

In this section we will consider issues which arise when our architecture is used in combination with the legacy email architecture and its protocols.

Consider a university which has decided to replace its mail system in favor of our architecture. First, they will need to install our P2P email application on the computers they want to use for this system. Note that the number of participating computers does not need to be large and can be grown dynamically, even while the system is running.

In order to communicate with the outside world, both for incoming and outgoing mail, we designate a gateway. Note that *any peer* in the system could be the gateway and there could be any number of such gateways. The only requirements for a gateway are that it can be contacted by both the peers inside the campus as well as anyone outside the campus. For outgoing mail, the sending peer can act as gateway. From the point of view of users outside the campus, the gateway is identical to the old mail server.

Furthermore, we need a mechanism for users on-campus to determine which addresses are local and which are not. Mail to local addresses should be delivered with the P2P email system and mail to remote addresses is to be sent to the gateway. This could be determined, for example, from the domain-part of the destination address.

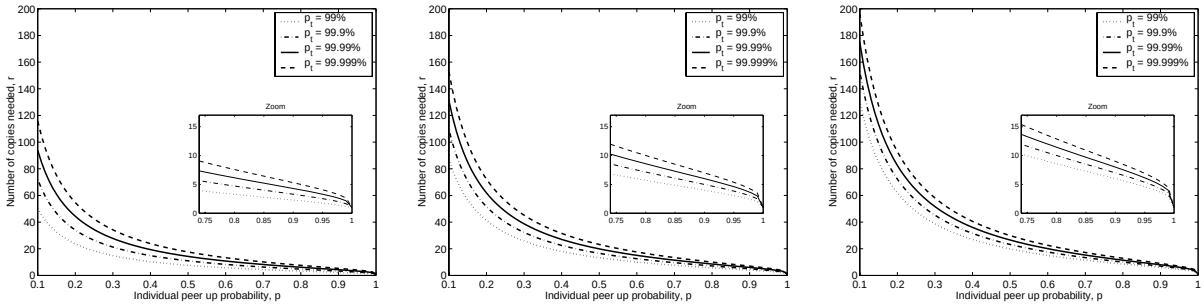
For incoming mail, off-campus users need a way of determining the address of the gateway. Currently this is achieved through the MX-records in the Domain Name System (DNS) [4]. If the university maintains its current mail server as the gateway, then no changes are needed. Because the mail server would only act as gateway, i.e., only take mail from outside senders and distribute it in the P2P email system, the load on this server would be low. Alternatively, the university could specify several peers from campus as its MX-hosts. For interoperability with the legacy mail system, it is vital that outside users can get the address of a gateway by performing an MX-query in the DNS.

Hence, for SMTP interoperability, the only requirement is that the gateway implements SMTP, both for incoming and outgoing mails, and that the address of the gateway can be obtained with an MX-query.

Our P2P email application uses its own HTTP-based commands for reading mails (see Section 6) which is not compatible with either POP or IMAP. Using a gateway which translates these protocols into our protocol, it would be possible to use any mailreader for reading email in our architecture. However, because we encrypt the inboxes, it may be difficult to use a standard mailreader, since they cannot handle this; encrypted messages are typically not a problem for most modern mailreaders. We do not expect this to be a problem initially, since a user is required to use the same computer for reading mails; this computer will have the P2P email application installed. Interoperability with POP and IMAP is mostly important for mobile users and we discuss this issue in Section 7.

6. Implementation

In this section we will describe our prototype implementation of our P2P email architecture. Our implementation is programmed in Java and it currently implements all the



(a) $n = 2$ (b) $n = 100$ (c) $n = 10000$

Figure 2. Number of copies needed

basic functionality described above. The application is divided into two parts, UA and system node.

The UA application provides an interface for the user to read and write mails. When the user retrieves new mail (initiated by clicking on a button), the UA application determines the k peers responsible for the user's inbox (using a simple P2P substrate), downloads the inbox from one or more of the peers (see below for details on how many peers we contact), clears the inboxes, and finally builds the superset of all new messages locally. Note that the downloading and clearing are performed atomically; the atomicity is enforced by the peers holding the copies of the inbox by queuing any new notifications that arrive during that time.

Each peer running the system node application listens for incoming requests from the UAs. The system nodes also store objects on their local storage and return these objects when UAs request them. Currently the system nodes do not implement any garbage collection.

When retrieving or storing objects, we have two additional parameters that control the extent of the replication and search, namely `maxOps` and `maxConnections`. The parameter `maxOps` determines how many times the operation will be performed and `maxConnections` determines how many peers may, at maximum, be contacted. For example, if we are reading the inbox and `maxOps` is 2 and `maxConnections` is 5, then we will read the inbox from 2 peers and we will attempt to contact at maximum 5 peers for retrieving the 2 required copies of inbox. We are investigating the effects of these two parameters to determine how a future P2P filesystem should be tuned in order to provide our email application the best possible service.

The message format for the requests and replies (retrieving or storing objects) is HTTP/1.1 [2]. We chose to use HTTP as the protocol because of the availability of existing tools for speeding up the development. Table 3 shows the mapping between the service primitives and the corresponding HTTP requests and their parameters.

We are currently testing our prototype in various situations to determine the best values for the different parameters. We are also looking into implementing garbage collection in the system nodes and improving the persistence of the stored objects (see Section 4). In the future we hope to make the email application available for download.

Primitive	**HTTP request**
Store	PUT
Delete	DELETE
Fetch	GET
Append-Inbox	PUT and Pragma: append-inbox
Read-Inbox	GET and Pragma: read-inbox

Table 3. Service primitives and corresponding HTTP requests

7. Future of P2P Email

In this section we will discuss further work in the area of P2P email and discuss the future of such systems in general.

Our architecture is only meant as a first step to building a more complete P2P email architecture. Our goal is to identify key features and requirements of a P2P email architecture. In our basic architecture, these requirements are, perhaps surprisingly, low. We only need support from a DHT substrate and the possibility to store objects on the peers. However, as far as the storage is concerned, we do not require any strong consistency features, since we do not need to modify the stored objects. Any message that is stored on the peers does not change until the recipient reads it and deletes the stored object. Through local garbage collection we can eliminate objects that are lingering around in the storage (for example because the node was down when the message was deleted). Likewise for inboxes, we only append new message headers and clear the inboxes fully when messages are read. If there are any inconsistencies in the contents of the inboxes, these are resolved by creating the superset of the downloaded inboxes. Should the user desire

a more permanent storage, she can have her UA make sure that the objects (most notably her certificate) is available from the k closest nodes, as described in Section 4.

In contrast to a traditional server-centric email architecture, our initial P2P architecture has two shortcomings. It does not provide persistent message storage and it constrains the user to access her mails from the same computer. The first one can be remedied by having the UA enforce a replication policy which guarantees that any stored message would always be available from the k closest nodes; this would be relatively simple to achieve. The second one, providing access on the move, is harder to resolve. Naturally, we can store the information about folders, read messages, and mail filters in a file which is stored on the peers and enforce a replication policy to guarantee its availability. This would be trivial to implement and as such, providing access on the move is a non issue. However, because a user needs access to her private key very often (for decrypting emails sent to her, for example), she will need to be able to access it on the move in a trusted manner. This requires either some trusted hardware (e.g., a smartcard) or requires that the user always absolutely trust the computer she is using. We are currently looking into different possibilities for providing this trusted access.

Because the storage provided by the peers is highly unreliable, we will need to create several copies of each stored object, on the order of 10 to 20 copies at least. In a centralized server-based architecture, we would need much less storage space to provide the same level of service. However, our architecture does not require any additional resources, since is uses the disk space that is already installed on the peers. This means that our email architecture incurs no extra cost as opposed to setting up a costly mail server.

Although we have concentrated on email in this paper, our architecture could just as easily be used for any kind of user-to-user communication, such as instant messaging. A P2P architecture could be an ideal vehicle for bringing these different forms of communication together.

Another interesting aspect of P2P communication architectures is their resilience against attacks. A single mail server makes for an easy target, but if the mail is delivered through a P2P architecture, comprising of millions of peers, communications can be guaranteed in any conditions. A P2P email architecture could also serve as a backup system for the standard mail server and would provide continuity of service. Even the simple architecture presented in this paper would provide sufficient service to ensure business continuity should the primary mail server fail.

8. 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 and reduces stress on the mail servers. It can also guarantee the anonymity of the senders and recipients. 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] 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.

[2] 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.

[3] J. Kubiatowicz *et al.* OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS*, Boston, MA, Nov. 2000.

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

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*, San Diego, CA, Aug. 27–31, 2001.

[6] 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*, Heidelberg, Germany, Nov. 2001.

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

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*, San Diego, CA, Aug. 27–31, 2001.

[9] D. A. Turner and K. W. Ross. Continuous media e-mail on the internet: Infrastructure inadequacies and a sender-side solution. *IEEE Network*, 14(4):30–37, July/Aug. 2000.

[10] D. A. Turner and K. W. Ross. A comprehensive architecture for continuous media email. *IEEE Multimedia*, 8(2):88–98, Apr./June 2001.

[11] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California, Berkeley, Apr. 2000.