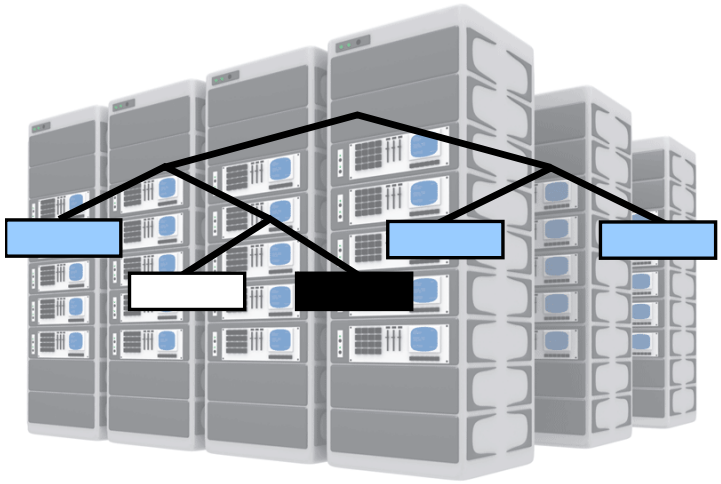
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

TripletMail: Replicated E-Mail Storage



Gabor Cselle
gabor@student.ethz.ch

August 19, 2005

Supervising Professor: Prof. Dr. Roger Wattenhofer
Advisor: Keno Albrecht

Abstract

We describe TripletMail, a system for outsourcing e-mail backups. The typical corporate mail server that handles receiving, sending and storing e-mails is replaced by two components: A mail server for SMTP/IMAP and a storage system which maintains several encrypted copies of each e-mail in racks of inexpensive servers.

In this thesis, we focus on two problems of the storage system: In inexpensive racks, servers can break down often. Therefore, the storage system needs to deal with server failures gracefully and re-replicate copies of e-mails that were lost. Also, storage load should be distributed evenly across all servers. For this, we introduce a scheme that uses consistent hashing to create a key for each e-mail message. A tree then maps parts of the keyspace to triplets of servers on which replicas should be stored. We then describe an approach for re-replication and evaluate this scheme against simpler ones.

A Java implementation is included.

1 Introduction

1.1 Motivation

E-mail service in corporate environments is typically provided by a single SMTP/IMAP server. This server quickly becomes a single point of failure: There is usually no fail-over mechanism in place and data backups are seldom up-to-date. Corporate e-mail often contains valuable and confidential information, so there is reluctance to outsource these services. However, if backup services were available that have lower adminstration costs than current solutions, and fulfill the requirements below, they could be successful.

1.2 Requirements

The goal is to create a storage system for e-mails that fits these requirements:

Confidentiality Through encryption, the storage provider never sees the contents of e-mails.

High availability Users are able to retrieve their e-mail at all times.

No data loss Maintaining multiple copies of each e-mail message lowers the likelihood of losing data due to hardware failures.

Economies of scale Minimize costs per stored e-mail message by pooling the data of multiple companies and using large numbers of servers.

1.3 Outline

Figure 1 illustrates the architecture of TripletMail. We will examine each of the components in bottom-up sequence.

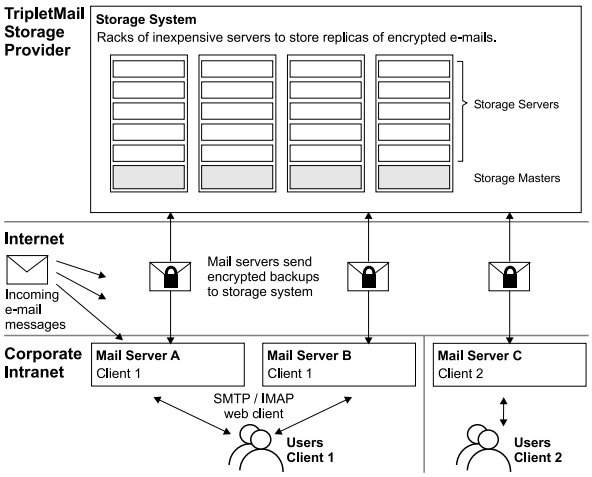


Figure 1: TripletMail architecture with remote, encrypted backups.

The thesis is structured as follows:

Section 2 starts by listing the assumptions upon which the system was built.

Section 3 examines the functionality offered by the mail servers, including the encryption and signature schemes used for communication with the storage system.

Section 4 explains the storage system and the two kinds of roles - storage master and storage server - assigned to the servers.

Section 5 describes how the system determines on which storage servers e-mail message replicas should be kept.

Section 6 closes with a performance evaluation.

1.4 Related Work

There already exist solutions for storing multiple copies of files on inexpensive servers. A good example is the Google File System [1], which stores very large files of several GB to keep management overhead low. Still, we are aiming at storing e-mail messages, whose sizes rarely exceed 100 kB. The tree mapping approach was inspired by work on peer-to-peer lookup protocols such as Kademlia [2] or Willow [3]. The focus of these protocols is more on file-sharing systems or group communication, and instances where nodes leave and join often. Also, individual nodes cannot exercise control over others. For TripletMail, things are simpler, as all servers are owned and fully controlled by the operator of the storage service.

On the other end of the spectrum, we find commercial solutions for storing large amounts of data on custom hardware, such as those offered by NetApp [4].

2 Assumptions

Instead of using high-cost storage solutions, we aim at using inexpensive machines for storing the e-mail replicas. Of course, low cost implies lower reliability, so we expect about 3-5% of all machines to be failed at any time. To build a reliable system, we make up for high individual failure rates through software logic: When a machine fails, others will have to take over its load automatically.

The network layout is influenced by the fact that the servers are typically kept in racks by the dozen. These racks are connected to other racks via a 1 GBit Ethernet switch. Inside the racks, all machines are often connected with only 100 MBit Ethernet to reduce networking cost. Finally, the servers themselves consist of just barely more than hard drives, a mainboard, and a network interface.

While unlikely, it's possible that an entire rack fails at the same time, as all contained servers share a common power and network connection.

Each rack contains two different kinds of units: a number of storage servers and a single storage master, as described in Section 4. Storage servers are responsible for storing replicas of large quantities of e-mail. We assume that the cost for such a machine with 500 GB of hard drive storage to be 1000 USD. Storage masters manage the storage servers. We estimate their cost at 1500 USD.

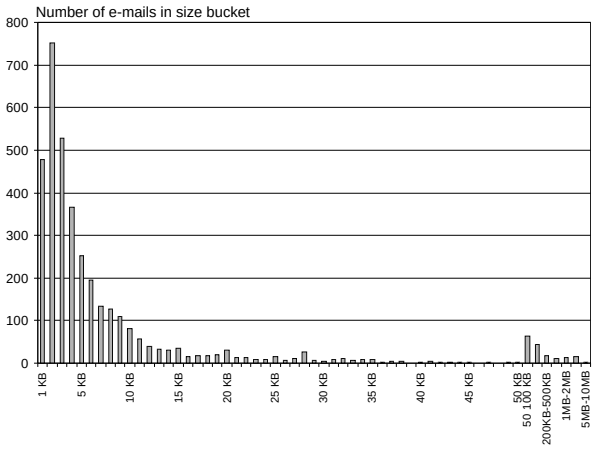


Figure 2: Approximate size distribution of e-mails. Measured in inbox of the author. $\mu = 34.6$ kB, $\sigma = 273$ kB

The smallest chunk of information the system handles is one e-mail message. We assume that each user receives and sends around 5000 messages per year. For estimating storage requirements and measuring performance, we assume that the average e-mail size is around 35 kB, as demonstrated in Figure 2. Clearly, this number is just a rough approximation: widely different quantities are possible depending on the user.

We focus on IMAP-like schemes of retrieving e-mail. It does not make sense to introduce backup schemes for e-mail which is quickly wiped from the server via POP3. IMAP is already in wide use in corporations, in part due to the legal benefits of having an archive of all employees' e-mails.

Finally, during implementation, we assumed that a replication factor of 3 is enough; this not only explains the TripletMail name, but also why this factor is used throughout this thesis. The number seemed safe enough to guard against data loss and is in line with the common practice of having a "primary" and a "secondary" backup of important data.

2.1 Definitions

Throughout this thesis, we will use the following terms: A TripletMail storage system will store e-mails from multiple *clients* - a company that has outsourced its e-mail backups to a storage provider. Each client will have at least one mail server, and several employees using e-mail - we call them *users*. Each user will have a number of *folders* which contain the actual e-mail messages.

3 Mail Servers

We now start explaining the system's architecture in detail. We start with the servers that receive and serve e-mail; the next section will explain the storage system.

Mail servers, called "*MServers*" are placed in the clients' offices. They act as SMTP servers to the outside: They receive and send out all e-mail to and from the client. Users can retrieve their e-mail via a web client (IMAP access should be a possibility, but is not currently implemented).

A client may have multiple MServers for higher availability. Also, MServers cache recently seen e-mail from the storage system, to provide for the case that the outside Internet connection is lost.

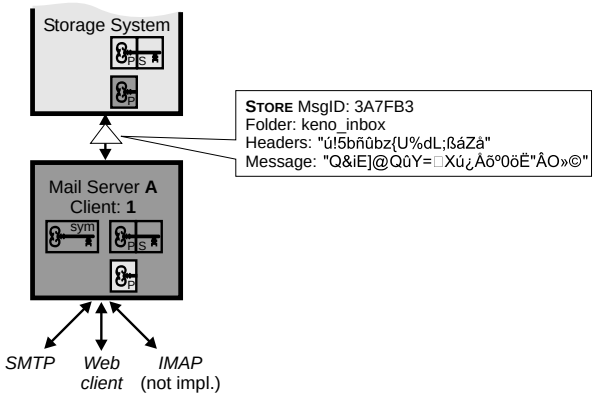


Figure 3: Mail Server functionality

Figure 3 shows different aspects of the mail server's functionality: On the bottom, it accepts requests from users. On the top, the MServer communicates directly with a storage master, the management component of the storage system (see Section 4). When the connection to a storage master is lost, the MServer starts trying to connect to any of the other masters, until a connection can be made.

On this connection, there are four main types of messages that can be sent: STORE, DELETE, RETRIEVE, and LISTFOLDER.

Of these commands, LISTFOLDER is the most interesting. It asks the storage system to return a list of e-mails in a folder, along with their most important headers, such as `From`, `Subject`, and `Date`. These correspond to the columns commonly displayed in the GUI of the user's e-mail client. LISTFOLDER is a very common request, as it is the one called whenever the user checks his or her inbox.

3.1 Ensuring Confidentiality

When designing TripletMail, one of our goals was ensuring that the client's data was kept secure: neither the storage provider nor the outside world should be able to find out the contents of e-mails. In addition, we wanted to make certain that no attacker can impersonate the storage system, causing messages to bypass storage by going to the wrong destination.

Therefore, we implemented an encryption and signature system. As shown in Figure 3, each client has a symmetric mail encryption key and a private / public key pair for signing messages. Only the client knows the symmetric key: without it, the storage provider cannot know the e-mails' contents.

Newly incoming e-mails are packed into a STORE message with encrypted body, encrypted headers, and a time stamp. A 256-bit *MsgID* is generated by running a one-way hash function such as SHA-1 over the contents and headers. Also, to avoid known-cleartext attacks on the symmetric key, a few bytes of random content are added. This message is then signed and sent to the storage system. The storage master processes the message only if the signature was correct. Similarly, the storage system has its own key pair so the mail server can check its messages for authenticity.

4 Storage System

Machines in the storage system racks take on two different roles: storage master ("*SMaster*") or storage server ("*SServer*"). They are assigned identi-

fiers as seen in Figure 4: SMasters get ascending single letters, the SServers in the same rack share this letter and a number.

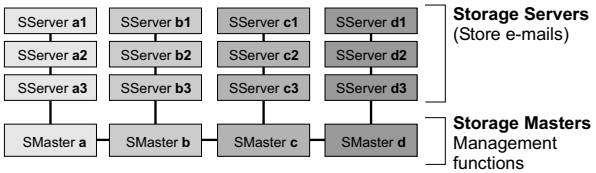


Figure 4: Storage system roles.

Each SMaster keeps a TCP connection open to every other SMaster and every SServer. Periodically, a ping message is sent to each machine. This allows the SMasters to quickly notice the failure of a machine in the system.

Three copies of each e-mail reside on three different storage servers. Triplets are chosen so storage load is balanced across servers.

4.1 Folder Information

To facilitate a quick response to ListFolder calls, all folder information is kept on the SMasters.

Folder information consists of a list of *MsgID*s for each folder. Every *MsgID* has an associated bytestring, which happens to be the encrypted headers sent by the mail server with the Store command.

Since SMasters can fail, we must assure that the same folder information is available at every one. For this, we have implemented a simple method: Each SMaster pushes information originally received at the Store call (the new *MsgID* and encrypted headers) to the next one, until all have heard the news.

5 Assigning Storage Locations

We have shown how the SMaster answers ListFolder requests using local information. However, for requests like Store, Delete, Retrieve, it must know where e-mail replicas reside. To accomplish this, storage locations are assigned using a mapping tree with a triplet of storage server IDs at its leaves.

5.1 Mapping Tree

Figure 5 shows a small example mapping tree, as used in TripletMail. The *MsgID* of each e-mail is mapped to a leaf in the tree, which in turn lists a triplet of SServers. To determine the storage locations, the *MsgID* is walked in a bitwise manner. If a bit is 0, the left child node is taken; if the bit is 1, the right node is taken. This is done until a leaf node with an assigned triplet is reached. For example, all e-mails whose *MsgID*s start with 01 would be stored on the storage servers $a2$, $b2$, $c2$.

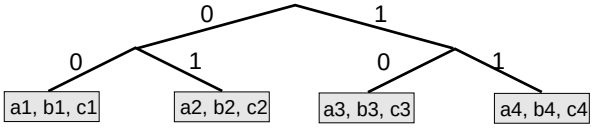


Figure 5: An example mapping tree.

5.2 Triplets Selection Scheme

The triplets at the leaves of the mapping tree control the load distribution across storage servers. We now present a heuristic to generate these triplets so load is distributed evenly.

We divide all storage servers into 3 groups of equal size. For example, in the case of Figure 4, the groups would be: $G1 = \{a1, a2, a3, b1\}$, $G2 = \{b2, b3, c1, c2\}$, $G3 = \{c3, d1, d2, d3\}$. The first entry in the triplet may only come from the first group, the second from the second group, and so on. With this restriction, we avoid that all replicas of an e-mail message are stored in the same rack.

Triplet Properties The following properties should hold for all triplets:

1. The tuple elements should not all be contained in the same rack.
2. The same triplet occurs twice only once all possible triplets have been exhausted.
3. If $load(s)$ is the number of occurrences of a storage server s in all triplets, then the following must hold: $\max_s load(s) \leq \min_s load(s)+1$

Property 1 ensures that we do not lose all replicas when a rack loses its power or network connection.

Property 2 and 3 ensure that load is spread evenly over the machines: By requiring all permutations of group elements to occur, we make every server from every group share some load with every server from another group. Property 3 applies more to states where the possible space of triplets has not yet been exhausted: It makes sure we don't first try all triplets containing $a1$, then $a2$, and so on.

We now define some load counters for use in the triplet generation algorithm.

$load(s)$	How many triplets storage server s already occurs in.
$conn_{G1,G2}(a,b)$	number of times SServer $a \in G1$ appears in a triplet with SServer $b \in G2$
$conn_{G1,G3}(a,c)$	number of times SServer $a \in G1$ appears in a triplet with SServer $c \in G3$
$conn_{G2,G3}(b,c)$	number of times SServer $b \in G2$ appears in a triplet with SServer $c \in G3$
$conn_{G1,G2,G3}(a,b,c)$	number of times SServers a, b, c appear in a triplet together

Given a set of already existing triplets in the tree, we may find the next set using the Generate-New-Triplet algorithm:

1. From G1, pick the element t_1 with the lowest $load(t_1)$
2. From G2, pick element t_2 with the lowest $(load(t_2)+1) \cdot (conn_{G1,G2}(t_2)+1)$
3. From G3, pick element t_3 with the lowest $(load(t_3)+1) \cdot (conn_{G1,G3}(t_1,t_3)+1) \cdot (conn_{G2,G3}(t_2,t_3)+1) \cdot (conn_{G1,G2,G3}(t_1,t_2,t_3)+1)$

For systems with more than 3 SServers in more than one rack, property 1 holds because of our grouping scheme.

The algorithm generates all permutations of the three groups, in a sequence that spreads the load evenly at all times, thereby keeping to properties 2 and 3. Figure 6 shows example results.

0.	a1b1c1	**22.**	a3b4c2	**44.**	a1b4c4
1.	a2b2c2	**23.**	a4b3c1	**45.**	a2b3c3
2.	a3b3c3	**24.**	a1b3c3	**46.**	a3b2c2
3.	a4b4c4	**25.**	a2b4c4	**47.**	a4b1c1
4.	a1b2c3	**26.**	a3b1c1	**48.**	a1b1c4
5.	a2b1c4	**27.**	a4b2c2	**49.**	a2b2c3
6.	a3b4c1	**28.**	a1b4c1	**50.**	a3b3c2
7.	a4b3c2	**29.**	a2b3c2	**51.**	a4b4c1
8.	a1b3c4	**30.**	a3b2c3	**52.**	a1b2c2
9.	a2b4c3	**31.**	a4b1c4	**53.**	a2b1c1
10.	a3b1c2	**32.**	a1b1c3	**54.**	a3b4c4
11.	a4b2c1	**33.**	a2b2c4	**55.**	a4b3c3
12.	a1b4c2	**34.**	a3b3c1	**56.**	a1b3c1
13.	a2b3c1	**35.**	a4b4c2	**57.**	a2b4c2
14.	a3b2c4	**36.**	a1b2c1	**58.**	a3b1c3
15.	a4b1c3	**37.**	a2b1c2	**59.**	a4b2c4
16.	a1b1c2	**38.**	a3b4c3	**60.**	a1b4c3
17.	a2b2c1	**39.**	a4b3c4	**61.**	a2b3c4
18.	a3b3c4	**40.**	a1b3c2	**62.**	a3b2c1
19.	a4b4c3	**41.**	a2b4c1	**63.**	a4b1c2
20.	a1b2c4	**42.**	a3b1c4	**64.**	a1b1c1
21.	a2b1c3	**43.**	a4b2c3		...

Figure 6: Generate-New-Triplet algorithm: Resulting triplet sequence for 3 racks with 4 SServers each.

5.3 Tree Leaf Splitting

Leaf splitting is an important part of TripletMail. As we will see, it facilitates quick re-replication in case of failures.

Each leaf represents a up to certain number of e-mails. Once this amount of data is exceeded, the leaf node must be split.

This leads us to a trade-off: For example, if we set the limit to 1000 e-mails per leaf, leaf splits will occur twice as often as with a limit of 2000. However, the amount of data to copy during the split is around 500, not 1000, so copying will take less time. Finally, with smaller limits, load will be spread more evenly across servers, because the load granularity is smaller; on the other hand, negotiating each leaf split takes some effort.

How does a leaf split take place? The storage master that first needs to store an e-mail which breaks the leaf node storage limit is responsible for splitting the node. A new triplet is determined using the Generate-New-Triplet algorithm. The storage master then splits the tree leaf and assigns the new triplet to the new 1 branch. Figure 7 illustrates this.

But before we can initiate the process of copying

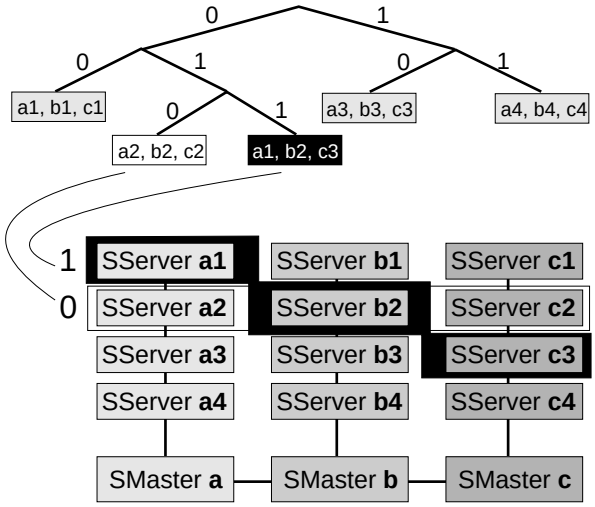


Figure 7: Result of splitting the mapping tree at leaf 01.

over information from the old to the new storage servers, we must agree with other servers on this change. Another storage master might have noticed the very same problem at about the same time: A new message was added to a leaf node, which raised the requirement for a leaf to be split. However, only one storage master may be responsible for the split, as only a single SMaster may send out the commands to copy the affected messages to their new location and then delete them from the old one.

Therefore, TripletMail uses a variant of 2-Phase-Commit [5] to make sure only one storage master modifies the tree at a time, and all on-line storage masters agree to this change.

The limitation to on-line storage masters allows us to split leaves even when some storage masters have failed. However, it adds a disadvantage: When communication between storage masters fails, they can be partitioned into two groups that independently modify the tree. The likelihood of a network partition, however, is minimal, since all storage masters are directly connected to the same network switch.

5.4 Re-Replication

When servers go down, a replica of each e-mail message on the server is lost. Since one of our design principles is that each message should at all times have three replicas, we must re-replicate it to other servers.

Since SMasters keep TCP connections open to each SServer and ping them periodically, the loss of a server is noticed quickly. After a grace period, re-replication is started. The length of the grace period should be long enough so that minor errors are ignored: For example, it can be set to the time it takes a machine to reboot.

Once again, the first storage master to notice the loss of a storage server jumps into action.

Initially, it adjusts the groups G1, G2, and G3 used in triplet generation: The SServers which are still alive are again divided into three equal-size groups. Now, all triplet occurrences of the failed server in the mapping tree must be replaced with an SServer from the same group. Each time, the SServer s with the lowest $load(s)$ is chosen.

Then, the 2PC protocol is run to make sure no conflicting changes are being made to the tree. When the 2PC is complete, the newly chosen SServers are advised to copy the e-mail messages affected from the remaining copies, thereby restoring the original replication factor.

5.5 Extending Storage Capacity

The re-replication mechanism is also used for expansion: When more storage servers or whole racks are added to the storage system, the load will reach them quickly: After the groups have been adjusted, new triplets generated by the Generate-New-Triplet algorithm will always contain the new servers. Similarly, failed servers will be replaced by the new ones in the affected triplets. Therefore, the triplet-based approach also allows for expansions for the original storage system to add more capacity.

5.6 Comparison: Top-Down Filling

We now compare our triplet-based approach to other approaches to spread several replicas across servers. An important measure will be the time it takes to re-replicate failed servers. For simplicity, we will assume that all messages are the same size, and it takes one time unit to copy each message.

Clearly, the easiest approach for distributing replicas would be that of *top-down filling*: each server on the first rack gets assigned two fixed

peers on the other racks on which replicas are stored. Once the first server is filled, we start filling the next three servers. For example, in Figure 7, SServer $a1, b1, c1$ would first be filled completely, then $a2, b2, c2$, and so on.

With the top-down filling approach, it would take a long time to re-replicate: a fail-over server would have to copy all contents from its two peers. If there were m messages on the failed server, it will take $\frac{m}{2}$ time units to re-replicate the lost server.

Also, with this approach, each group of three servers would have exactly the same load. As experience teaches, three hardware components manufactured in the same batch and exposed to the same load tend to fail at the same time. In TripletMail, each SServer is exposed to a different load, making accidents where all three copies are lost at the same time less likely.

In an *ideal case*, all messages would be spread evenly over n servers. If one failed, then the total amount of time necessary to re-replicate all messages would be $\frac{m}{n-1}$: The load of copying m messages would be evenly distributed over the $n-1$ remaining servers. Implementing a system, however, would not be easy: while re-replication would be quicker, one needs a method to ensure that two replicas never end up on the same server.

Our *triplet-based* approach comes close: For a sufficiently large number of triplets in the mapping tree, a failed storage server with m messages is re-replicated in $3 \cdot \frac{m}{n-1}$ time units.

Because of our re-replication scheme introduced in Section 5.4, the failed server will be replaced by $\frac{n-1}{3}$ storage servers, who will evenly share the load of copying the m messages.

5.7 Future Work

Several ideas are not yet implemented. Clearly, focussing on 3 replicas is somewhat rigid: This work needs to be generalized to more than a fixed replica count. Also, e-mails can currently be accessed by web client only, as IMAP support is not yet available. The caching strategies for the MServers could be optimized. Finally, e-mails often have the same attachments: These could be identified through content hashes and saved only once.

6 Performance Evaluation

To estimate system performance, we subjected the system to some tests. Parameters were chosen as follows: All messages were of size 35 kB, as that corresponds to the average size of e-mails we measured earlier. The limit on the number of messages per tree leaf was set to 1000. The hardware we had available were homogenous machines with P4 processors running at 3 GHz, and 1 GB RAM. We simulated 3 racks with 3 SMasters, and 6 SServers per rack. Also, we ran 6 MServers simulating 2 clients. Each client had 100 users, with 1 folder each. Messages were generated for random users, with random content and headers. Performance measurements were taken at SMasters.

These tests were run in a student lab. Therefore, we could not replicate the network layout of a server farm organized around racks.

Two measurements were made: Firstly, we measured the write performance by having the MServers saturate the system with STORE requests. Figure 8 shows that system could easily handle storing around 1000 e-mails per minute.

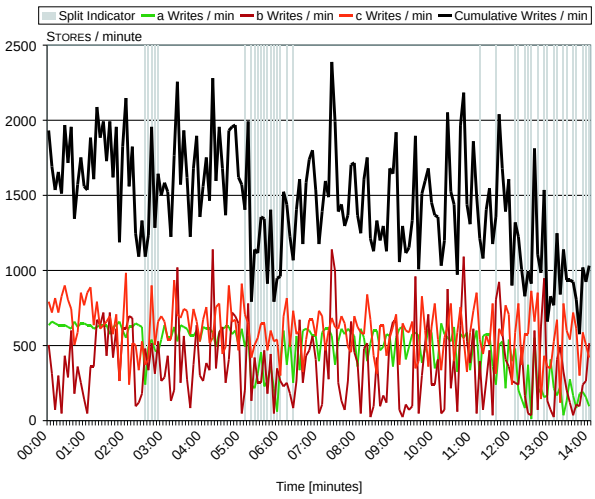


Figure 8: Performance for number of messages stored per minute.

Still, the system will usually be faced with a different load: Users will receive, check, and read e-mail. For this, we had the MServers performing cycles of storing a new message, listing a random folder, and retrieving a random message from

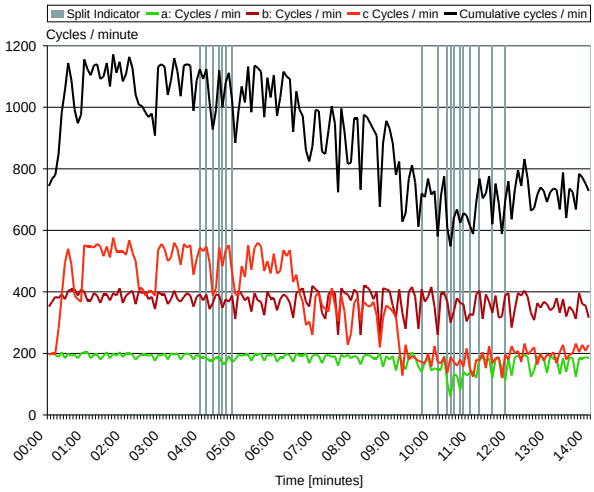


Figure 9: Performance for mixed use: STORE / LISTFOLDER / RETRIEVE cycles.

that folder. The results are shown in Figure 9 and demonstrate that the system should be able to sustain around 400 cycles / minute.

In effect, the performance displayed in here also limits the use of the system: Since 5000 e-mail messages per user per year could be interpreted as 5000 such cycles per year during work hours, this system could support about 8000 users. At a hardware cost of around 3 USD per user for our evaluation system, this seems acceptable. However, the software is implemented in Java and not optimized.

7 Conclusion

We implemented a storage system for outsourcing e-mail storage. The system is optimized for inexpensive servers stored in racks. E-mails are stored in an encrypted manner so the storage provider cannot see their contents. For assigning storage locations, we use consistent hashing to generate keys for each message. Using a tree structure, parts of the keyspace are then mapped to servers on which replicas should be stored. This approach fits well with small-size data chunks of our e-mails. We demonstrated that this system spreads load evenly across the servers used for storage. When servers fail, we can quickly re-replicate the lost data because of these well-spread copies.

With the current implementation, TripletMail could be made into a viable service, since performance requirements are met.

Acknowledgements

The author wishes to thank his advisor, Keno Albrecht, and Prof. Roger Wattenhofer for their help. Also, thanks go to Florian Walpen for fruitful discussions.

References

[1] S. GHEMAWAT, H. GOBIOFF, S.-T. LEUNG, The Google File System, *SOSP 2003*, Bolton Landing, NY, USA, Oct 2003

[2] P. MAYMOUNKOV, D. MAZIERES, Kademlia: A peer-to-peer information system based on the XOR metric, *IPTPS 2002*, Cambridge, MA, USA, Mar. 2002

[3] R. VAN RENESSE, A. BOZDOG, Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol, *IPTPS 2004*, San Diego, CA, USA, Feb. 2004

[4] NetApp: http://www.netapp.com/

[5] BERNSTEIN, P.A. ET AL: *Concurrency Control and Recovery in Database Systems*: Chapter 7, Addison Wesley, 1987