Scalability and fault-tolerance in the groupware domain

Master Thesis, January 12^{th}, 2014, Final Version

Simon Bernardus Kok

Scalability and fault-tolerance in the groupware domain

MASTER THESIS

submitted in partial fulfilment of
the

requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

by

Simon Bernardus Kok
born in Voorburg, The Netherlands

Web Information Systems Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, The Netherlands
http://eemcs.tudelft.nl

Scalability and fault-tolerance in the groupware domain

Author:	Simon Bernardus Kok	
Student id:	1550616	
Email:	siem.kok@gmail.com	

Committee:	Prof. dr. ir. G.J.P.M. Houben	(Delft University of Technology)
	Dr. ir. A.J.H. Hidders	(Delft University of Technology)
	Dr. ir. A. Iosup	(Delft University of Technology)
	Ir. B.R. Joseph	(Zarafa)
	Ing. I. Timmermans	(Zarafa)

Abstract

In most businesses, email and collaboration services are essential to the performance of the company. Delivering a communication platform that scales well with the growth of the company, and provides the services anytime, anywhere, even in the event of failures is hard to achieve at low costs. Literature has proven that simplified email storage is scalable. Interactive collaboration services on the MAPI protocol, however, are limited in their scalability due to the data complexity. With this thesis, I analysed the groupware use case and data structure for possible solutions to this problem. Based on the service requirements and storage layers available, a proposed key-value data structure is presented. While out dated, the literature presented benchmark results for this database category with small 1 KB values. In this thesis, I benchmarked MySQL Cluster, Cassandra, Riak, Voldemort, and HBase using 10 KB values while focussing on the I/O subsystem throughput and failure tolerance of these databases, simulating email characteristics. The proposed solution, utilizes Riak with ZooKeeper to provide a single point of entry, scalable, and fault-tolerant communication service. I developed a prototype service and load simulator to demonstrate its scalability and failure tolerance through an extensive load simulation of 32 thousand users. The results show how failures are dealt with, and how the cluster expands, all without disrupting the user interaction on the service.

Preface

This report is the result of my master thesis project, written as the conclusion to my study Computer Science at the Delft University of Technology. The research has been conducted at Zarafa in Delft. At Zarafa, I was introduced with the scalability problem of groupware data and the business side of open source development. To conduct a thorough research, Zarafa supported me to conduct a broad literature survey on possible data storage solutions. Furthermore, Zarafa enabled me to conduct an independent load simulation of several storage architectures.

Through Zarafa's network, I was introduced with several key players in the open source groupware community. I would like to thank Zarafa for their support and for showing me the beauty of open source development. More specifically, I would like to thank Steve Hardy, Michael Kromer, John van der Kamp, and Mark Sartor for their technical insights, support, and brainstorm sessions. Additionally, I would like to thank Brian Joseph and Ivo Timmermans for their support and guidance as my direct supervisors. I would like to thank Joeri Smit, Heleen van Beek, Dominique Debyttere, Ivo van Geel, and Paul Boot for keeping me motivated throughout the graduation project.

I would like to thank my colleagues and friends at FeedbackFruits, without their support I would not have been able to reach my goals. Carlos Toro-Bermudez, Sander Geursen, Felix Akkermans, Jakob Buis, and Niels Doekemeijer, thanks for discussing all the technical difficulties I faced. I would like to thank Amber van Hauwermeieren, Ewoud de Kok, Bart Kaas, and Daan Eigenraam for their extensive support, scrum meetings, and advice on getting it done.

Furthermore, I would like to thank my supervisors at the Delft University of Technology for their support and advice on this matter. Especially I would like to thank Jan Hidders and my fellow graduation students, who were present at the colloquium meetings, for their interest and advice.

Last but not least, I would like to thank my family for their endless support throughout my study, without you I would not have been able to achieve this.

Simon Bernardus Kok

Delft, The Netherlands
January 12^{th}, 2014

Table of Contents

Appendices

List of Figures

List of Tables

List of Algorithms

1 Introduction

The research of this thesis has been conducted at Zarafa. Zarafa is the developer of its like-named open-source groupware solution. The groupware solution offered by Zarafa allows its client to set-up a groupware environment that operates similar to Microsoft Exchange. However, since the product of Zarafa is open source, the community around the product helps to improve and extend it. The customer demand has grown very rapidly in the last couple of years, as large corporations start to inquire for their solutions. The current server architecture of the groupware product, scales perfectly for small to middle-sized corporations. However, it has its limitations in terms of scalability. With the research presented in this thesis, Zarafa wants to acquire insights how to organise data stores, to deal with the scalability demands for its groupware services.

1.1 Problem statement and motivation

In order to achieve scalability, large customers need to set-up multiple servers, where the user base is split into many smaller groups that are spread across these servers. The set-up is failure resistant by adding slave servers that mirror the data set completely. However, the current implementation of the Zarafa server does not support automatic load balancing, and in terms of scalability it has several limitations as well.

Since the current set-up forces the user base to be divided into multiple smaller groups, the end-users cannot share their data directly across these groups. The customer, however, would prefer to work on a single environment, where a scalable cluster of servers operates as if it were a single extremely robust server. The data inside this preferred set-up should be stored redundantly, where the clients would be load balanced between the servers, such that the server capacity is utilized across the complete user group effectively. In a preferred setup, data loss due to server outage is highly unlikely and the end-users have a reliable service to operate on. In short, the customer is looking for a system that is easy to manage, and scales well with the growth of the organisation.

A problematic aspect of scaling groupware services is the amount of shared data across the user base. Part of the data is only accessible to its own end-user, or shared with a small group of users. Whereas some data needs to be accessed by large groups of users. Additionally, since Zarafa allows its end-users to access their data from multiple clients concurrently, the data needs to be passed to these different clients in a small period of time. These clients include mobile devices, tablets, and personal computers. Keeping this data in sync across these devices is important, as the end-user needs to be able to trust the data to be up-to-date in order for the product to have a good customer satisfaction ratio.

1.2 Research relevance

Groupware need to store a lot of data, of which a huge proportion is accessed. This data, however, cannot be stored in memory of the servers providing a groupware service, as the costs would exceed the reasonable limit. Therefore, groupware requires a very well performing Input/Output (IO) sub layer to retrieve data from disk. The research covers the limitations and opportunities of data redundancy in IO intensive applications that require scalability. Researchers and companies that are looking into the horizontal scalability of IO intensive applications might find the findings of this research relevant.

The research in terms of scalability covers several related aspects of hosting a fault-tolerant and scalable service. The research on this matter will be relevant for developers that need to develop a service requiring the same levels of scalability and redundancy.

The research on fault-tolerant databases that provide near linearly-scalable storage of data is limited to small data objects, allowed the database to cache a lot of the data internally, and used out-dated versions of the databases that are available today. This research analyses MySQL Cluster, Cassandra, HBase, Riak, and Voldemort through extensive tests where the performance of the database was measured using object sizes of 10 KB. The storage servers had limited memory available, such that it would be possible measure the IO performance. Furthermore, the benchmark tests included failure tests as well, disconnecting one of the servers temporarily, and making one of the nodes fail completely with a recovery process afterwards. The benchmark results are interesting for any party that needs fast and reliable performance of their scalable database. Additionally, the shared insights on the usability of the databases might be of interest to the developers of the databases and companies that want to incorporate a new scalable database into their platform.

1.3 Research objectives and scope

The objective of this research is to achieve virtually unlimited scalability of a groupware service. With this research, Zarafa wants to acquire insights how they should effectively organise user data, such that a cluster of servers provides access to the groupware service, allowing it to grow with user demand, while being failure resilient.

The scope of the research is limited to the area of groupware services, in which the users share calendars, have their own mailboxes, and access shared mailboxes. The research will focus on achieving a scalable groupware service, hosted on a single location; the geo-graphical distribution of data is outside the scope of this research. Out of scope are search operations, the security related to the storage of user data, and archiving of old mails on different storage clusters. Furthermore, the user authentication is outside the scope as well. Throughout the

research, it is expected that a scalable and fully fault-tolerant user-authentication service exists.

1.4 Research questions

Through an explorative research, the following research questions will be answered to find an appropriate solution offering data redundancy and load balancing for scalable groupware solutions. The main question for the research of this thesis is:

> What would be an optimal solution to store groupware data, across multiple servers that together behave like one robust service provider, allowing scalability in the order of $n + 1$?

This main research question has been broken down into multiple sub questions that together give a broader idea why this is optimal and how this can be applied best in the field of groupware solutions.

Q1. What is required by the customers of groupware solutions in the area of data redundancy?

Q2. What data replication techniques could be implemented to achieve $n + 1$ scalability?

Q3. How could a failure resilient single-point of entry be defined to ease set-up for end-users of the cluster?

Q4. How should the data be split across multiple servers? On which data level: per user, store, or even entity?

Q5. How could a cluster of servers operate fully autonomously, achieving scalability, failover, and load balancing?

Q6. How could data consistency be realized, if data is redundantly stored and users access the stores using different clients?

With these sub questions, data replication techniques used in other fields will be analysed, an analysis on combining load-balancing techniques with redundancy techniques will be given, and methods to keep all the data consistent on all servers within a reasonable amount of time are discussed.

1.5 Research strategy

In order to research how to design one robust service provider for groupware data, the research is structured in several phases, these phases are depicted in Figure 1. At the start, data and usage statistics were acquired of several groupware servers and research reports. Along with an in-depth analysis of the data architecture of the Zarafa implementation, this showed where

the bottlenecks in groupware services are located. The findings were compared to competing groupware solutions and research projects that focus on scaling Input/Output (IO) intensive services.

Before this research started, I conducted an extensive literature survey on the topic of scalable online, and fail-safe data technologies [37]. The insights acquired in this literature research are the fundament of this research. The literature survey, along with additional literature on scaling IO intensive services, brought forward several storage solutions that theoretically would be up to the job to scale data in linear terms, while providing fault-tolerant, high-available data access. These storage solutions are benchmarked on their performance in terms of reading, updating, and inserting data, as well as their ability to scale and their ability to operate while facing failures. Based on the results of these benchmarks, and the conceptual design that was formed based on the storage criteria, literature, and data analysis, the benchmark results showed the architectures that are up to the job. Aside of the database layer, other fundamental architectural decisions are also presented, resulting in the proposal of a scalable, fault-tolerant groupware service architecture. The proposal is validated using a load simulator that is developed during the research, the simulator will test whether the system is able to withstand node failures, and performs acceptably. Furthermore, the validation will show whether the system is able to operate as if it were a single robust server, while actually running on a shared-nothing cluster of commodity hardware.

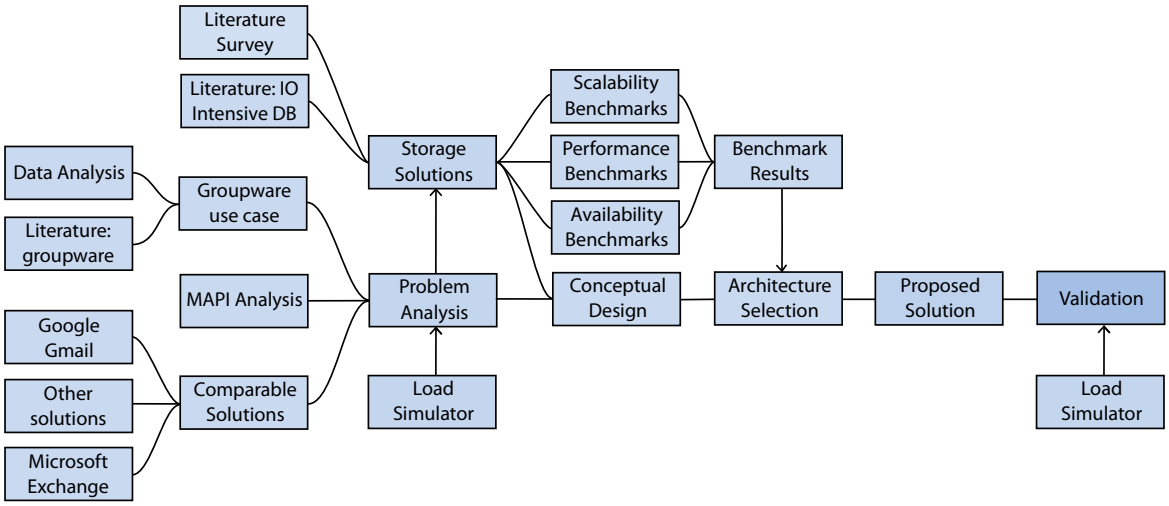

Figure 1: Research Strategy

1.6 Thesis overview

The background of groupware and its characteristics, in terms of data and usage, are discussed in Section 2.1. As this research focuses on the scalability and availability problems of Zarafa's groupware solution, the internals of MAPI are discussed in Section 2.2 to describe

the requirements and locate bottlenecks. Section 2.3 describes how other groupware solutions have been designed to tackle the scalability and availability issues. The requirements in terms of performance, availability, and manageability are discussed in Section 3. Followed by Section 4, where the service bottlenecks are analysed in detail. To work around these bottlenecks, the data and possible strategies to deal with the data are discussed in Section 5. Several storage solutions that theoretically offer linear scalability, while being fault-tolerant, and well performing, are presented in Section 6. These storage engines are benchmarked on several aspects in Section 7. In Section 8, the architecture of the proposed solution is presented. Followed by a validation of the proposed architecture in Section 9. Lastly, the conclusions and areas for future work are discussed in Section 10.

2 Background

The background of this thesis is discussed through analysis of the use case of groupware. As the MAPI protocol is required by enterprise groupware products, the analysis of this protocol is discussed in the section after that. Related work in terms of research and product development is covered last. These sections require knowledge on MVCC, BASE, ACID, and CAP, these are described in the literature survey [37].

2.1 Groupware use case

Groupware services have to deal with a lot of data with very diverse usage patterns, as was shortly introduced in Section 1.1. Part of the data is shared among groups of users, while other data is only accessible to a single user. Groupware entails enormous amounts of data, which all need to be accessible, but of which only the most recent proportion is operated on a lot. This later characteristics is exactly what makes archiving email so important. By setting up an archiving server, data that is not used frequently can be moved to slower storage servers for archiving, allowing the high-end servers to work on a smaller data set. Figure 2 depicts the access rates of data over time.

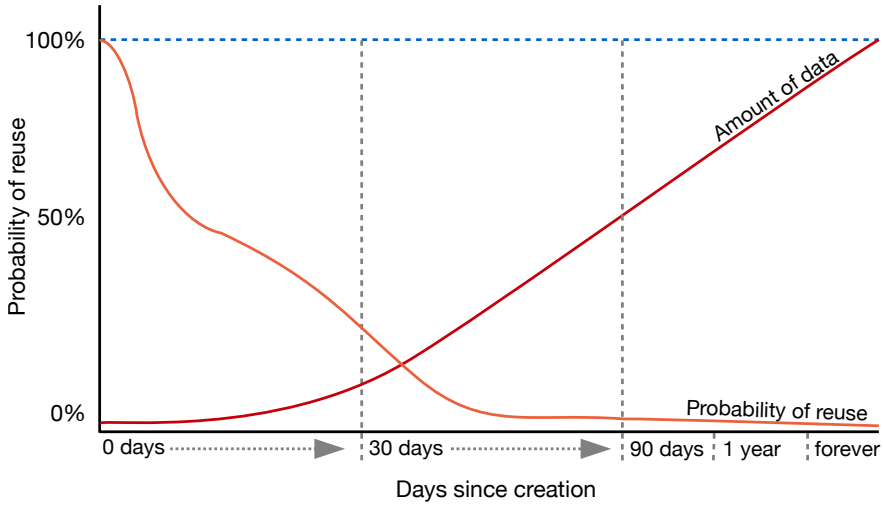

Figure 2: Email usage versus the cost of storing over time. Modelled based on data acquired by Horizon [48]

Aside of being a fully extended Messaging API (MAPI) compliant email server, the Zarafa server is designed to offer its end-users capabilities such as calendar sharing, meeting request support, mail delegations, integration with Spreed, Alfresco, and other third party software solutions. All of these data types are stored using the same base type as used by email messages in the database, therefore they are all located in the same database table and perform like they are no different. However, as the main data of users is formed by mail

messages, the performance and storage requirements of users in groupware systems is mainly determined through their email usage.

Focussed on the business market, there are 850 million email accounts world-wide, sending about 89 billion email messages a day, as was reported by the Radicati Group in 2012 [34]. Limited to this segment, the amount of mails sent per day is expected to increase with an average of 13 per cent per year. In 2012, this adds up to about 105 emails per business account per day, heading towards 125 emails per day in 2016 [34]. In 2012, the Microsoft Security Intelligence team [44] published that besides all these related emails, about 75 per cent of the messages that gets delivered to the server are spam and should be filtered.

According to Radicati [56], an average email with attachments takes 500 KB, while emails without attachments take 26 KB including their body. Per attachment email, on average the attachments together would require about 474 KB of storage. Radicati states that a total of 24.2 per cent of all messages sent have attachments. Thus, attachments would take about 81.5 per cent of the total storage; calculated using Radicati's email statistics report [56].

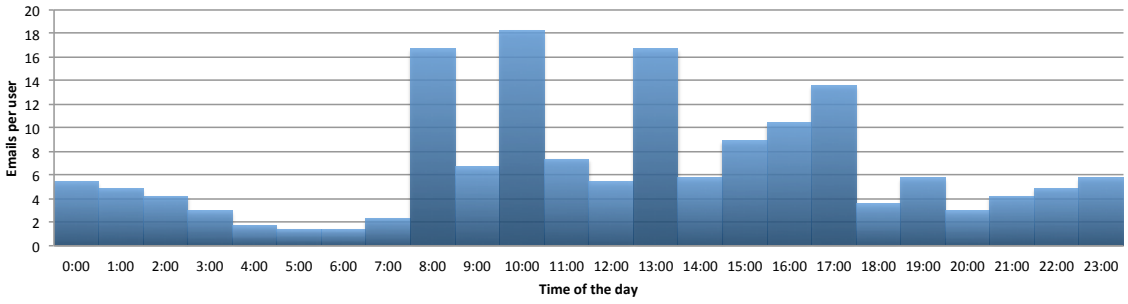

Figure 3: Emails sent and received in an IT company on a typical workday.

Figure 3, shows a two-week average of the emails sent and received on a workday in an IT company. For each industry this pattern might differ, for example a transportation company might send and receive most emails around the start and end of the day. These differences between industries and companies are very useful, according to Harms and Yamartino [33], as the differences flatten out the spikes that would otherwise stress the service. Therefore, the more different the companies that use a groupware service are, the steadier the resources will be utilized. This allows a better provisioning of servers to operate the service.

When dealing with emails, actions like marking a message as read, flagging it, moving it, or acting with the message all update some fields in the headers of the message. Using the access, move, reply, and forward action statistics of the Microsoft Email benchmark configuration [43], each message will be opened, and 70 per cent of those messags will face another update. Whereas the body of the message is rarely touched after it has been created, the body only gets updated in case a draft is edited, or the body of a calendar item gets

changed. The attachments of an email are only added or deleted, facing no updates at all.

Unfortunately, the powerful MAPI interface is not compatible with generic solutions to scale email and other common Internet services. MAPI adds a lot of metadata to the email headers, making it a complex object. The clients use these headers to query the data. Therefore, the metadata needs to be available to index the email; this is a feature that generic scalable mail solutions cannot perform.

In terms of types of users, there are two important characteristics to group the users by. The first characteristic is how they organize their emails. All users have at least one folder where the messages are initially delivered to, their inbox. However, whether a user organizes email in folders and when these users do so differs. Boardman and Sasse [65], categorized the users into four types based on their organizing strategy. Bälter [9] analysed the categories further, depicting four types of users, how they are related and showed how a user could move from one category to another due to time-pressure or an email overload, this is depicted in Figure 4. There are some users that organize their messages continuously, organizing the mail threads in folders, these types of users are called the frequent filers [65]. Another type of user organizes their inbox every couple of weeks, these are referred to as Spring Cleaners [9], moving messages to folders, and deleting those that are not relevant to keep. The folderless cleaners, keeps important messages in their inbox, utilizing the search feature of their mail client to look-up messages later on. Fisher et al. [28] introduced a fourth category that covers users that keep their inbox and folders small, using multiple strategies to achieve this. Figure 2 shows that the likelihood of a message being used later on decreases quickly over time [48].

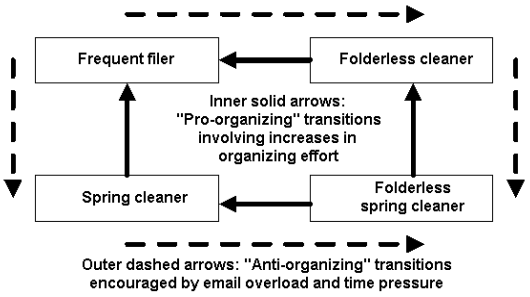

Figure 4: Email management strategies, model of changes [9].

The second characteristic is their level of interaction with the groupware service. So called Power Users are connected to their groupware account throughout the day, make heavy use of the search feature, have larger inboxes, receive a lot of emails and invitations throughout the day, and make use of the mobile support extensively [29]. The Information Workers have larger inboxes, use a mail client on their pc to access their mailboxes throughout the day [29]. The last user group are the Occasional Users, these mainly use a web client to access their mailbox, have limited storage available, and periodically check their mailboxes [29].

2.2 MAPI structure

All emails, calendar events, meeting request, and other related data types are stored as messages on the server. These messages can embed other messages. Although one could store all messages as raw data inside a mail journal if all messages were only made available as a time-ordered list. The MAPI servers need to be capable of sorting the messages on different properties as well, more specifically, the API states that any property should be available as a sort-key. This requirement forces the server to index all the data on these properties. The message structure, how all of these properties are indexed and stored efficiently on a single server, as well as the storage of messages in folders are described in this section.

2.2.1 Message structure

Figure 5 shows the message structure diagram, each message can have multiple recipients and attachments. The message itself, each recipient, and each attachment are entities themselves. Each entity has its properties stored inside the properties table.

Each message is able to hold a reference to multiple recipients and attachments, in which an attachment can have another message embedded. Every message, recipient, and attachment has its own unique Hierarchy ID. These recipients and attachments exist as a collection of properties, and are stored in a special properties table, just like any message.

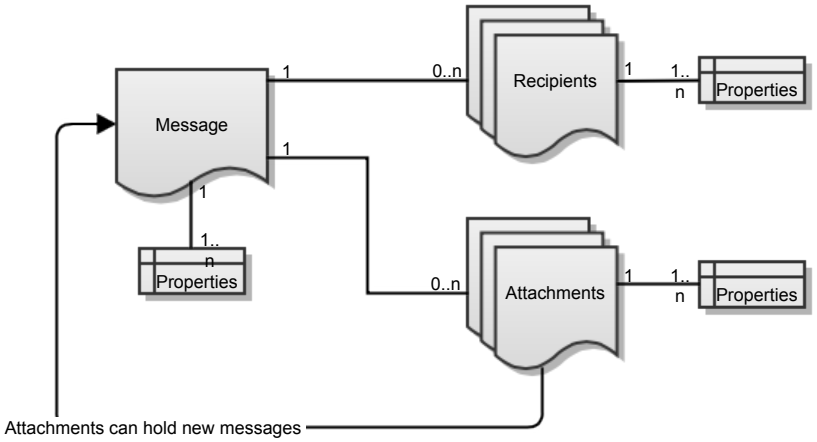

Figure 5: Message entity structure, showing how each message comprises several entities.

The hierarchy table is used to link these entities together; it stores the parent of each entity, enabling quick lookup of child entities, as well as parent entities, given a Hierarchy ID. In general, an email consists of about 10 hierarchy entities.

2.2.2 Entity properties

By the design of MAPI, each entity type has different properties, and between two entities of the same type there can be a difference in the properties that are available. To enable this flexible design in data layer, Zarafa implemented a centralized property table that would be able to hold all properties of all entities using a limited number of columns.

Each property is stored in the property table, as shown in Table 1, using the Hierarchy ID and the type attribute of the property that is stored. The Hierarchy ID and Type columns define the primary key on the property table. This allows the Zarafa server to lookup properties of each message, recipient or attachment very quickly; as the InnoDB engine will write these near to each other. How this results in improved performance is further discussed in Section 4.1.

HID	Type	Value
1	5	...
1	6	...
1	7	...
1	8	...
1	9	...
1	10	...
1	13	...
2	2	...
2	3	...
2	4	...
2	5	...

Table 1: Property table, storing all properties (Type) related to an object (HID).

Each entity in the property table is written by a single thread, connected to a single database, therefore all of the written properties will be located near each other on disk. Unless, of course, the user added new properties to the entity, in this case the InnoDB engine might need to move elements around to make them fit in the same page files as the other related properties. This operation can be quite expensive in terms of disk Input/Output operations (IOPS).

In order to speed up the sorting process, Zarafa introduced a transposed properties table, referred to as the tproperties table. This table enables quick lookup of all entities in a certain folder, sorted on the type of the properties, for example on the subject of the message. This table structure enables the user to sort on any property very quickly. However, inserting new entities to this table is quite expensive, as all the properties of an entity are placed on different places, forcing the InnoDB engine to use as many IOPS as the number of properties of an email message. On average this results in about 50 IOPS per message written.

The transposed property table as shown in Table 2, is structured to store all properties of type x in folder y. This allows the Zarafa Server to lookup all properties of the objects in a certain folder. The primary key is formed using the Folder ID, Type, and Hierarchy ID columns.

FID	T	HID	Value (truncated)
11	5	1	...
11	5	2	...
11	5	4	...
11	5	5	...
11	5	7	...
11	6	9	...
11	6	10	...
11	6	11	...
11	8	13	...
12	5	2	...
12	5	3	...

Table 2: Transposed property table, storing all properties of type x (T) of all the objects (HID) in folder y (FID).

To speed up the writing process of new emails, the Zarafa server waits until it collected 20 new messages in that single folder, before it will commit the data to the tproperties table. This reduces the amount of IOPS by a factor of 20, as for each property the InnoDB engine is able to write 20 rows at once. All new messages that have been written to the properties table are put into the deferred updates table. This allows the server to keep track of the amount of records that still need to be inserted into the tproperties table. This table is also used to add these entities on the fly until they have been inserted. Although this means that the server still needs to retrieve a maximum of 19 records for each folder to construct the sort list. However, since these messages have recently been created they are most likely still available in the cache of the database, allowing them to be retrieved instantly.

FID	HID
11	5
11	6
12	2

Table 3: Deferred updates table, stores all the Hierarchy IDs (HIDs) that have been updated in a folder (FID) recently.

In the deferred updates table, of which an example is shown in Table 3, the table stores which Hierarchy IDs have been updated and need to be merged into the transposed properties table. The Folder ID and Hierarchy ID columns together form the primary key of this table. If an object gets moved to another folder before it is merged, only the reference to this object will be changed in the deferred updates table.

Constructing a list view for the user requires the server to sort the elements on the user-specified property within that folder; using the tproperties table it retrieves all the objects with that property quickly. The constructed view is stored in the server cache to speed up reloads, all new and updated messages end up in the deferred updates table which are quickly inserted before the list is sent to the user. This requires the server to retrieve the new and

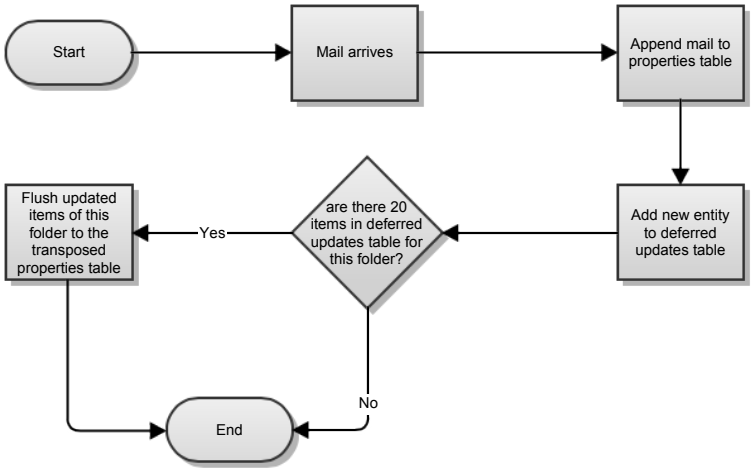

Figure 6: Logic behind deferred updates.

updated messages from the properties table instead of the tproperties table.

2.2.3 MAPI data view

The Zarafa server offers methods to access the data; the most commonly used protocol to access it is MAPI, as most enterprise email clients and mobile devices use this protocol. Figure 7 depicts the MAPI view on a table of messages. The MAPI view is constructed using a quick look-up of all the entries in that folder, merging it with the deferred update records retrieved from the properties table.

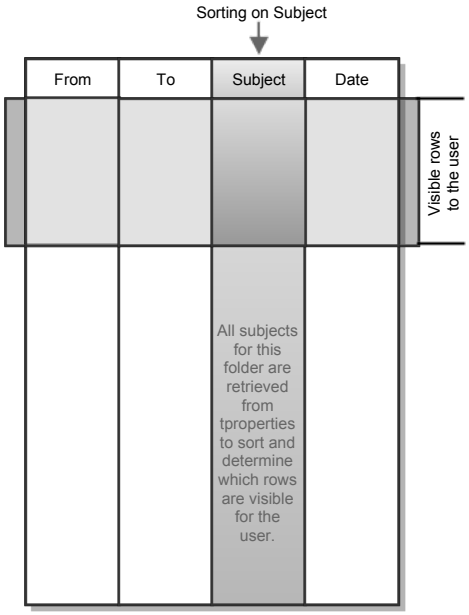

Figure 7: MAPI view on a table of messages.

The primary key in the tproperties table is defined as the Folder ID, Type of property, and Hierarchy ID columns together. If a user looks up all messages in a certain folder, the InnoDB engine can retrieve all the records with the given sort property with a single IOP, this drastically improved the sorting of messages in a folder, compared to previous sort implementations that walked through the property table to define the order, resulting in as many IOPS as there are messages in the folder.

Field	Bytes
Primary key (FID, T, HID)	12
Folder ID	4
Type	4
Hierarchy ID	4
Date	8
Transaction and rollback	6
InnoDB overhead	7
Total	**33**

Table 4: Row storage calculation, for each entry in the tproperties table.

In the tproperties table, each record holds about 33 bytes of data including InnoDB overhead. With a page size of 16 KB, each page file is able to hold about 496 records. If you would have a single folder with 100 thousand messages, this would result in 202 page files. In order to retrieve all of the records, the InnoDB engine would have to execute 202 IOPS. Compared to 100 thousand IOPS this is a really good improvement.

Increasing the page size to 64 KB, would reduce the number of IOPS from the theoretical 202 to about 51. In other words, increasing the page size lets us linearly scale the number of IOPS. However, although this theory sounds great, increasing the page size would also incur a lot of extra overhead costs, as for each new page that is created more data needs to be reserved.

2.2.4 Microsoft Exchange 2013

Microsoft Exchange uses an almost similar approach to store each entity in a dedicated properties table, as the Zarafa architecture described in Section 2.2. The differences in the MAPI view and scalability are discussed next. After which the cloud offering of Microsoft Exchange is discussed.

MAPI views In order to speed up the sorting process, Microsoft Exchange constructs a dedicated sorted view for the last eleven views that have been used. Figure 8 shows a drawing of these concurrently stored views. This is designed with the mindset that an Exchange server faces far more read operations than write operations, as writing a single message to each of these views requires 22 IOPS.

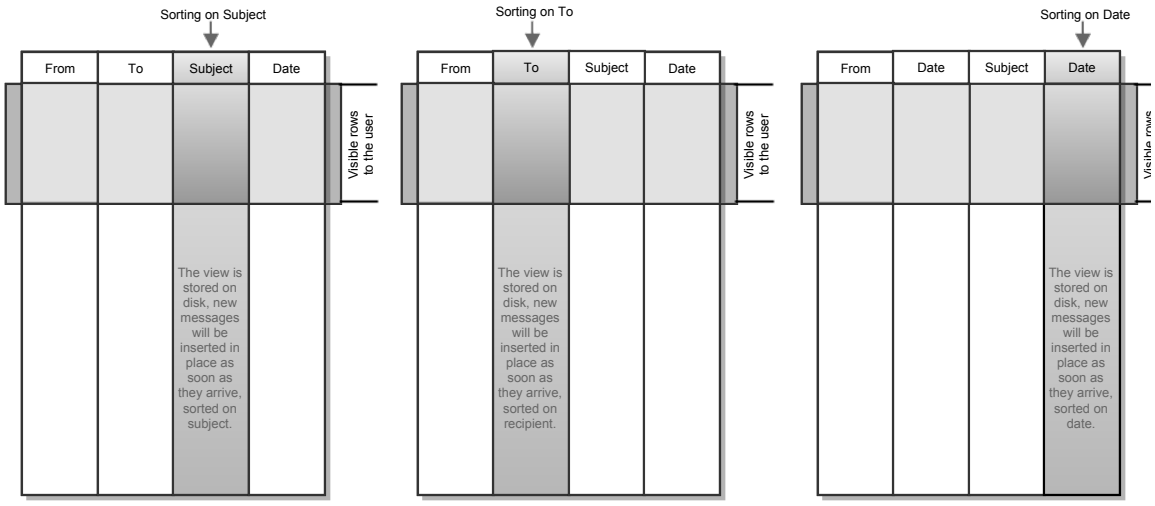

Figure 8: MAPI view used by Microsoft Exchange, storing multiple MAPI views concurrently.

Scalability The complete domain of Exchange users is split up into several segments, chunks, in which the data of a group of users is stored completely. In other words, the smallest data segment in one such a chunk is all the data of one single user. On average, the number of users that together form such a chunk is about 100. Each of these chunks have one primary server, which hosts the primary database, and one or more secondary servers (its replicas). In case the primary server goes down, one of the secondary servers becomes the primary for that chunk of data.

Exchange in the cloud Microsoft offers access to their latest Exchange 2013 servers through its Office 365 SaaS platform. Behind the scene, the service uses multiple clusters of exchange servers to provide a single interface. Each company is assigned to a specific cluster, such that the employees within that company can access each other their agenda's and share accounts as well. With the earlier 2010 introduction of the cloud service, the size of each company was limited to a maximum of thirty thousand accounts [47].

2.3 Related Work

Related to the research of this thesis is research related to setting up email as a service, as well as load simulations, and I/O performance papers. Related research on these topics is covered in that order in this Section.

2.3.1 Email as a Service

Saito, Bershad, and Levy introduced Porcupine in 2000 [60]. Porcupine is a scalable cluster mail server that operates on commodity hardware. Their service is able to distribute the load, where the nodes in the cluster are all equal to each other. In the case of failures, the cluster gracefully degrades, keeping the service up and running as long as the replication nodes are still operational. The cluster uses standardized SMTP, IMAP, and POP to deliver email to the users [59]. This is quite different from MAPI mail data, as these services are limited in their write intensity and offer limited support for sorting and other more advanced operations.

At the start of the 21^{st} century, Gribble et al. [32] described the Ninja project, to research possible innovations to distribute Internet services at scale, creating a robust service. Within this project, several applications were analysed. One of these services is generic email, hosted through several worker replicas that are able to perform the task. The project covered standard mail interfaces, including SMTP, IMAP, and POP. How it manages to keep the data up to date across the workers was not covered by the research.

In 2012, research conducted by Bloomberg [15] concluded that email as a service might face difficulties to integrate fully with the identity and access management of corporations. Forrester [30, 29] stated that email as a service brings many advantages for the corporate market, most importantly allowing its staff to focus on corporate matters, scalability of the service, and the price at which it is offered. In an interview, Raghu Ramakrishnan [14] stated that "people are more comfortable with not owning and operating critical computing services than they were (say in the early 2000s. . .)".

Cáceres et al. [17] discuss the generic ability of the cloud to provide scalability at the infrastructure level. This discussion includes showing how compute instances can be scaled vertically, by adding more resources to the machine. As well as scaling a cluster horizontally, by adding more instances to the cluster. Cáceres et al. show how generic applications would be able to take advantage of these scalabilities. The paper, however, focuses mainly on the infrastructure level, leaving out the difficulties of scaling the data to support such a horizontally scaled cluster.

On behalf of Microsoft, Harms and Yamartino [33] conducted a research to investigate whether cloud services could be an advantage in several markets, including email as a service. The email service provided by Microsoft uses a similar set-up of their on-site Exchange software. Each user is located in a distributed access group (DAG) that is served by a group of servers that provide failover internally. Since this DAG setup uses a relational database underneath, only a small group of servers can be used to provide the service, this limits the scalability of the platform to the maximum number of users that can be located in a DAG

cluster setup.

Where Microsoft moved its mail server product to the service market, Google developed the service from scratch to become scalable and fault-tolerant. In order to achieve this, Google developed Megastore [8] on top of its Bigtable database engine. Megastore is ACID compliant, offering fast and reliable read operations, whereas writing required the database to solve write conflicts and stale data. Megastore is used in many of Google's interactive applications, including Gmail, to make it perform and fault-tolerant at large scale. A June 2012 press release stated that Gmail has over 425 million monthly active users [54].

2.3.2 Load simulations

The email information flow inside enterprises was analysed by Karagiannis and Vojnovic [36], showing the characterized workload of an email service and benefits of sharing resources to put an optimal load on the server hardware.

As a service to its Exchange mail server customers, Microsoft developed the LoadSim [43] tool such that the customer could compare the performance of their mail server setup with other customers. The 2003 version of the load simulator used a fully MAPI compliant interface to perform benchmarks on the server, allowing it to simulate the load on non-Microsoft email servers as well. With their newer releases, however, the tool used an internal protocol, which makes integration impossible. However, in their benchmark tool, Microsoft described the exact workload that is performed to simulate the users, called MMB3. This workload describes how many email are sent, flagged, moved, etc. on a normal workday, simulating the load of thousands of users on the target email cluster.

On the I/O performance of databases, the research by Cooper et al. [22] on the Yahoo! Cloud Serving Benchmark (YCSB) has become the benchmark tool to compare the performance and behaviour of the different scalable databases. The results of the tests performed by Cooper et al. are based on 1 KB records, each record consists of 10 columns with 100 bytes. The benchmarks focussed on the throughput of the databases at a fixed scale of heavy-duty servers, with multiple thread configurations, and analysed the performance if the database scaled up in terms of number of servers.

2.3.3 I/O Performance

Luo and Yokota [42] compared the Hadoop Distributed File System (HDFS) with their own Fat-Btree, their performance results showed that HDFS is designed to operate on large files, resulting in a network performance bottleneck when a lot of small files are stored. Their solution introduced a more efficient storage, up to 400 times faster when using a lot of small

files.

Expósito et al. [26] performed benchmarks on the I/O performance of the different types of compute solutions provided by Amazon. In their analysis, they looked at the different levels of the I/O subsystem to evaluate the cloud storage devices thoroughly, from ephemeral disks to Elastic Block Store volumes. Through parallel I/O benchmarks, the performance and cost metrics of these solutions have been tested as well. After the extensive tests, the conclusion was drawn that Amazon's latest High I/O instance type (HI1), provides the best performance through the ephemeral SSD storage. However, considering the costs of this instance type, it might not be the best solution to opt for.

In High Performance Computing (HPC), the computational power increases at the rate described by Moore's Law. However, the I/O subsystems do not scale at the same rate, therefore, they have become the bottleneck for data-intensive scientific applications. Ali et al. [2] proposed an I/O forwarding paradigm where all the I/O operations are forwarded to a dedicated I/O cluster. The I/O cluster performs the operation on behalf of the compute nodes, enabling caches, rescheduling, and aggregating techniques to process the requests more efficiently. Similarly, Lang et al. [39], described the IO bottlenecks they faced in the design of the Blue Gene/P system, concluding that a dedicated storage cluster would have a lot of performance benefits with related read and write operations. Logan and Dickens [41] introduced an interval based IO file system to deal with the parallel bottlenecks to process scientific data, as they think the access requirements do not fit with a traditional file structure. Providing a scalable and fault-tolerant groupware service, requires a powerful I/O subsystem to deal with the enormous load of I/O operations and store the data while ensuring fault-tolerant access to it. Although the HPC is a different domain, the I/O limitations faced by data-intensive research applications apply to a scalable and fault-tolerant service as well. There is one major difference between the two, that is the difference in quality as HPC servers are built to last; the machines used in the groupware scalability research are commodity machines, facing frequent failures.

Dorier et al. [25] dived into the problem where concurrent-processing bursts of I/O requests in HPC clusters causes a bottleneck. Dorier et al. developed Damaris, an approach that uses a dedicated I/O core on each Symmetric Multi-Processor (SMP) node. Using the shared memory of its host, the system is able to efficiently process asynchronous I/O tasks. This completely hides the jitter and related I/O costs, increases sustained write performance 15 times, and allows a near perfect scalability. This, however, requires a shared inter-node memory system. Cloud scalability uses a shared-nothing to become more failure resistant, in other words, cloud instances share no memory or physical hardware to speed up the computation, and each instance is isolated from the others.

Tlili et al. [61] developed a peer-to-peer reconciliation infrastructure using an Operational

Transformation (OT) approach. Edits are replicated to other peers optimistically, where other OT solutions fail to work on peer-to-peer infrastructure, the proposed solution of Tlili et al. assures liveness and eventual consistency even though the network is dynamic and faces a lot of sudden changes. The peer-to-peer logging and timestamping infrastructure is called P2P-LTR, utilizing a distributed hash table to track the changes. The P2P-LTR solution excels when more than 15 users are editing the same document. A peer-to-peer infrastructure is more dynamic, faces more failures, and has only limited resources to keep its consistency. Therefore, the eventual consistency strategies are interesting as scaling servers up and down changes the cluster behaviour more like that of a peer-to-peer network, albeit one with far more resources in terms of bandwidth and performance.

3 Service requirements

This Section describes the service requirements that an email service should meet in order to be successful. More specifically, this Section answers the first sub question on "What is required by the customers of groupware solutions in the area of data redundancy?". To start, the performance requirements are analysed. Followed by the availability requirements, and lastly the managerial requirements to keep the service running. Section 3.3 discusses the issues related to managing the setup, initiating the discussion to answer "How could a failure resilient single-point of entry be defined to ease set-up for end-users of the cluster?", which is the third sub question of this research.

3.1 Performance requirements

Email can be classified as non real-time and asymmetric. It uses a best effort protocol without Quality of Service (QoS) reliability to deliver emails. This best effort protocol processes emails based on first in, first out [19]. However, two mails sent by the same user to the same destination will not necessarily be delivered in order, as the paths of the mail messages might vary while in transit.

Server to server delivery of emails tolerate hours of delay, as email is a store and forward type of service. In other words, when an email is delivered externally, the email might be queued until the server is able to process it. However, the communication between the end-user and the server should meet the expectation of the user that the changes are applied within certain limits.

Where the server has hours to complete an external delivery, the user should get feedback that the email is in transit in a time frame of two to five seconds [19]. Actually, the later case is different, as the user expects the server to accept the mail message, ensuring it will not be lost, but whether the email is sent instantaneously or within an hour does not really matter [19]. As long as the user can continue to use the groupware service in the meanwhile. In terms of reading mails, a user experiences opening mail as instantaneously when the request would be processed within 100 ms, according to the usability guidelines of Nielson [50].

The CIO of SBI Life, Mr J.B. Bhaskar, published the requirements they had to meet to move their email solution to the cloud in 2012 [13]. At SBI Life, the email service had major performance issues during the morning and evening hours, as their servers were not able to scale to deal with the load increase. Furthermore, at the end of the month, and even worse at the end of the year, the service was unable to process the incoming mails. On a daily basis, they delivered 120 thousand messages inside their domain, and 60 thousand leaving their domain. This results in a data transfer of roughly 20 GB per day, email only. Their peak demand is about 25 thousand emails per hour. In their calculations, they only looked

at valid emails, leaving out the 315 spam messages each day that should be filtered for each user, as was described in Section 2.1.

3.2 Availability requirements

Forrester Consulting conducted a survey research on the criteria companies use in their purchase decision for email platforms [30]. Forrester Consulting was commissioned by Microsoft to conduct this research in December 2010. Forrester concluded that there is a gap between the IT vision and business vision when it is related to email. Although email seems like just another tool to communicate, it is used thoroughly between businesses to reach goals and become profitable. This makes email a business-critical resource for companies today [30]. Businesswise, not being able to reach the email service, is a scenario most companies want to avoid. Businesses would also want to use the latest email and collaboration software to communicate efficiently without downtime to upgrade to newer versions. Seen from the IT side, these two are both hard to realize. Each upgrade to the next platform involves some downtime, however, being late with applying updates is a risk in terms of security as well.

From a business aspect, availability is very important. However, zero downtime solutions are too expensive to realize. According to Forrester Consulting, an email service should guarantee an uptime of 99.9 per cent to be placed among the best email solutions available. Furthermore, if the system goes down, businesses require the data to be replicated continuously, with a recovery time objective (RTO) of one hour or less [30]. Since email can be really valuable to businesses, it would be better to deliver mail late than losing it. Among other requirements, businesses find it important that their email solution supports mobile devices, and is capable of storing large mailboxes [30, 15]. In the second quarter of 2012, the Radicati Group reported that about 34 per cent of the email users worldwide access their mail through a mobile device [34].

In order not to annoy the end-users, separate devices should get updated using at least some form of eventual consistency that takes at most a couple of minutes to complete. However, when previous email data is consistent, it should remain consistent from that moment on, with the exception of new or edited emails.

Whether a company is open to cloud email solutions, depends on their business nature and legal requirements. Compared with on-site email solutions, cloud-based email is a lot cheaper to realize, as the costs for redundancy are limited and included in the nature of the email service provided. Comparing medium sized companies with enterprises, the medium sized companies are more likely to opt for an email solution as a service, as they require the same RTO and redundancy as enterprises, but are too small to manage it themselves. Although enterprises are capable to setup redundant email solutions internally, migrating these to the latest version takes too long according to the responsible decision-makers [30].

Moving email to the cloud allows businesses to focus on what they do best. Leaving upgrades of the platform to the IT specialists of the email service provider. In case anything goes wrong, the responsibility can be offloaded to a domain specialist, this decreases risks and costs of the email solution.

In terms of the user expectancies, any recoverable failure of the service should not annoy the user. The user might notice a hick-up, but should not interact manually in order to recover the connection. Furthermore, among the devices that a user would use to browse email, the latency between an update and it being visible on other devices is acceptable up to a couple of minutes. In terms of performance, reading is more important to finish quickly than writing, as the user performs more read than write requests.

3.3 Management requirements

From a system administrator point of view, the system should be fully autonomous in terms of determining on which server the users should be placed. The less management this system requires, the cheaper it would be to keep it running.

The system should be self-healing, as soon as a node fails it should automatically ensure that the data that was provided by that node is replicated. This replication process should finish fast enough such that the probability of failure on the other replica nodes is highly unlikely.

Furthermore, in order to scale with the increasing demand of storage, adding a server to the cluster should be as easy as booting it up. Even better yet, if the boot process can be automated by an automated cluster manager, it will be able to scale without any administration tasks.

In terms of performance, the cluster should be able to add new nodes and remove idles nodes automatically. The process should drop nodes to cut costs if their service is not necessary, while adding new ones if user demand requires so.

For the end-users of the service, the configuration to connect to the service needs to be simple. A single point-of-entrance should allow all users to connect, without requiring the users to remember the cluster they are placed on. Additionally, in case the connection with the service got interrupted, for example if a node in the cluster failed, a simple reconnect should suffice to get back to what they were working on.

3.4 Conclusion

In terms of data redundancy and availability, to setup an email service among the top line we require:

- it to deal with 105 new emails per account per day, the delivery should be guaranteed to 99.9 per cent;
- the incoming mail layer to be able to filter 315 spam emails per account per day;
- the service to accept messages sent in at most five seconds, while it has hours to deliver the email to the other party, as long as it does not disrupt the process;
- an open request to be processed within 100 ms in order to experience it as instantaneously;
- mail should be delivered late, rather than on time if that risks losing it;
- read operations to be prioritized compared to write operations in terms of performance;
- the data to be replicated continuously;
- should be built to have the data accessible 99.9 per cent of the time;
- to have a fallback mechanism in place that takes at most one hour to take over (RTO);
- should support large data sets between one and five GB per user;
- should support concurrent connections to the same data, with a consistent view within a matter of minutes;
- server management and user placement should be fully autonomous;
- the service to be self healing, with automated node recovery;
- adding more servers to the cluster should be as easy as booting it;
- cluster management should be able to automatically add and remove node as load requires;
- and lastly, the service should have a single endpoint to connect to, however, this cannot become the single point of failure.

In other words, the user should be able to access the service anytime, anywhere, and the user should not be bothered by heavy usage of other users. Therefore, the service should keep data redundant, should write fast, but read even faster, and should be able to deal with concurrent sessions accessing the shared data.

Sections 3.1, 3.2, and 3.3 provide a detailed guideline on what is required to become a scalable, fault-tolerant, and fully automated groupware service. With these requirements, the next section will describe what problems are faced to realize these.

4 Problem analysis

In this section, the internal data structure of Zarafa will be explained. The design of the database has been changed several times along the development of new major versions of the product. Apart of features such as utf-8 encoding support, the improvements focussed on getting the best performance on a single server level. This section will analyse the internal data structure, while discussing why these design choices were made, and what the benefits and disadvantages of such a design are. Followed upon these sections, is the analysis of the scalability of the platform. This starts with showing how a scalable set-up would operate, as well as showing the limitations of this architecture.

4.1 Performance problem

The current architecture of Zarafa server relies on the MySQL relational database, with InnoDB as its storage engine. MySQL in combination with InnoDB is fully ACID compliant, enabling transactional support, with crash recovery. The table is locked at row-level for write operations, while supporting consistent reads to optimize its performance in concurrent multi-user sessions. MySQL is able to scale up to hundreds of gigabytes of data. According to Oracle, some of their clients use it with 5 billion rows and a couple of hundred thousand tables [52]. However, since MySQL is ACID compliant, it faces the limitations in scalability by guaranteeing consistent writes.

Furthermore, the Zarafa product requires each server process to connect to its own database. As long as that process is the only one writing to the database, it can ensure that the data in the database matches its internal caches without being forced to check the database for recent updates regularly. This is a typical example of a scalability trade-off, focussing on single-server performance instead of multi-server environments. In order to guarantee uptime of the data, Zarafa has developed a special replication agent that will synchronize small multi-master clusters with each other. The data is replicated by cloning the data into a new MAPI object on the replica server. The data is cloned at fixed intervals, leaving the newest version on the server in case a conflict arises.

However, a more preferred and supported solution would be to set-up a failover slave replica using standard MySQL redundancy techniques. These MySQL set-ups offer more control, and have been implemented across many industries to achieve redundancy. MySQL limits replication to a couple of servers, although technically you could set this up using multiple separate database clusters that are randomly assigned each others backup slaves, this is very labour-intensive. If a server fails inside such a cluster, the slave server will need to deal with the immediate load increase. Even though one could set-up multiple Zarafa servers on a single machine, this still requires all the slaves to deal with an enormous amount of extra

load. For example, if each machine would have four Zarafa servers running, and host four slave copies of other servers, if it would fail the load on the slave machines increases by 25 per cent.

A very important requirement that limits the throughput of groupware data is the requirement that all data needs to be stored persistently on a storage device. Since groupware is very data intensive, and part of the data is accessed heavily, the performance characteristics of the underlying storage device are very important. At this moment there are two disk types that have completely different storage implementations, these are mechanical disks generally referred to as hard drive disks (HDD) and solid-state disks (SSD). Both of these disks have their own advantages, mechanical disks perform best at sequential read operations, whereas solid-state disks perform best while processing random read operations. In the design of the Zarafa server, the database architecture is optimized to utilize the performance advantage for mechanical disks. In terms of performance per buck, the solid-state disks offer the best solution. Performance is an issue for fresh data, as these are accessed most. As shown in Figure 2 shows, after a month the performance of the data is less of an issue, for data that is older it would benefit larger storage disks, this is where mechanical disks would meet the demand best.

4.2 Scalability problem

Changing the database layer to a distributed database would be hard to accomplish, due to the optimized use of MySQL database features, as discussed in the previous section. Especially when multiple servers need to connect with the same distributed database.

Strictly speaking, this design limits the available options in terms of database scalability to vertical scalability. With vertical scalability, the database server scales up by changing the server hardware to high-performance hardware. Another way to scale the database is to add more machines that together are responsible for hosting the database, this type of scalability is called horizontal scalability [17].

To share the load across multiple Zarafa servers, in which each connects to its own database, Zarafa implemented an agent that copies the latest changed content from one server to the other. Technically, this synchronization agent is able to developed to synchronize two servers, but with some modifications three or more servers would also be an option. However, the synchronization process delays the delivery of messages, and while copying the messages conflicts might arise. The implemented conflict resolution is to take the last version and present that to the user. If multiple servers are synchronized with this agent, the probability of conflicts increases. Therefore, improving the conflict resolution algorithm of this agent would become a necessity. However, with this synchronization method the overhead of copying messages and conflict resolution technique will become the first bottleneck in terms of scalability.

By design, each Zarafa server connects to one database, which could be configured to be replicated to another server. The servers that contain a replica of another node together form a cluster. In the field of email scalability this is called a database availability group (DAG). This structure, however, forces the administrator to assign users to a specific server. Microsoft Exchange uses a similar structure, in their architecture the maximum number of servers in a single DAG is limited to sixteen [45].

Another limitation to the DAG architecture is how new users should be dealt with. If one cluster reaches its limit, each new user needs to be moved to another cluster. Unfortunately the users that are placed in the new cluster would not be able to interact with the data on the other cluster directly. For a single enterprise this would force the company to segment the users into departments, to hide this limitation as much as possible.

Although their recent changes have strengthened the scalability of Zarafa, it is limited to set-ups of tens of thousands users and labour intensive to setup. The scalability limitations require a redesign of the storage layer to make it scale horizontally even further.

4.3 Availability problem

The Zarafa server preferably connects to one database that runs on the same machine to take away the network overhead. To guarantee the availability of the database, a replica of the database is hosted on another server. In the current design, each user is assigned to a specific server, with one or more replica servers for availability. This, however, introduces a very important limitation in terms of availability, if a server fails, the entire load is moved to its replica. The load cannot be distributed across the other servers, as the data is not available to those servers. Thereby, the replica will have to deal with the extra load. Either the servers have extra capacity, or all the users that are assigned to the failed node or its replica will notice performance degradation.

Since the servers replicate each other, the set-up forms a small cluster. If servers are added to the cluster, the nodes that will replicate the data of the new node need to be restarted; therefore the accounts on that cluster will not be available for a brief period of time.

In terms of user experience, if the server fails, the users would notice a hick-up, as their client needs to reconnect, followed by the failover performance degradation. Another issue with this set-up is the risk that a single entity could crash a server, in case the users are grouped together, the whole group will experience the crash on all slave servers as well. While if each user would be randomly assigned to a server, the master and slave servers of a single user might fail, but for all other users this would result in a slight hick-up, but the system keeps running.

4.4 Management problem

In managerial tasks, the overhead required to add new users and keeping such a cluster operational at large scale become another bottleneck to deal with.

Theoretically, you could configure a complete cluster to replicate as one big chain. Configuring the servers to replicate the data in such a structure and assigning users to specific servers becomes harder with each server that is added to the cluster. Each server needs to be configured to connect to its cluster, this requires careful partitioning of data and users. Each user would be assigned to a specific master node that is replicated by one or more replicas. However, to which server the user is connected to when logging in is very important, as network latency would become a major bottleneck for the performance. Therefore, upon login, the user should be connected to the node that is closest to the data of that specific user. This configuration would have to be published across the cluster, such that the connection of the user can be reconfigured to connect to the responsible node at the start. The administrators of the cluster would have to assign the users to their cluster using a database for user authentication, for example LDAP.

More recently, Zarafa introduced some features that should ease management of cluster set-ups. Through reverse-proxy support, the end-users would have a single point of entry to connect to, which will redirect the users to their designated server. In the same version, Zarafa also introduced support for multiple LDAP servers to make the authentication process redundant, providing higher availability.

When a cluster is increased in size, the other nodes need to be configured to use this new node as a replica and vice-versa. This requires quite a lot of tasks to set-up correctly. At some point the overhead of adding another node to the cluster is too high, such that a new cluster needs to be created. To be effective in terms of scalability, each of these processes need to be automated.

4.5 Load Simulator

In order to analyse the performance of Zarafa compared to other products, Zarafa utilized the Microsoft Load Simulator 2003. Newer versions of this product have been released, but unfortunately Microsoft switched from the MAPI interface, that is supported by Zarafa, to an internal Remote Procedure Call interface to execute the tasks on its Exchange servers. Therefore, the newer versions of the load simulator only operate on Microsoft Exchange servers.

Until recently this was not a real problem, as the previous version still operated correctly. However, when the scalability tests ran in a virtualized project, a bug of the load simulator

made it crash completely. What caused this behaviour is unknown up until now, but the analysis of the crashes suspect that the load simulator needs to have guaranteed CPU capacity that cannot be guaranteed on virtualized machines.

This made testing the set-up impossible, to solve this problem we decided to develop a load-simulator that would be able to connect to any MAPI compliant cluster. For the development of the load-simulator, the part that I worked on was the test suite of the Zarafa WebApp client, later extended to interface to the prototype type to validate this thesis as well.

The load simulator is based on the MAPI Messaging Benchmark 3 (MMB3) defined by Microsoft [43]. The MMB3 specification defines the exact workload that users would otherwise generate on a typical workday. This includes login in and off, retrieving contacts from the address book, composing messages, browsing through the inbox, replying to emails, flagging messages, copying, moving, deleting, etc. All of these tasks use a set of email templates that are used to simulate real-life workload. It connects to the server through MAPI or WebApp and simulates the workload of thousands of users. It allows configuring the amount of time it should simulate, the number of users, and how exactly the users should login. Each of the tasks are profiled and at the end of the test a performance score can be determined based on the average latency measured. As the load simulation is sending emails to users, the recipients of the emails sent will react on these events and simulate real on going mail conversations.

4.6 Conclusion

With the developed load simulator, tests could be performed to determine the recovery process of the current Zarafa server, as well as performance measurements when a large group of users uses the mail cluster.

In terms of its single server per database design as discussed earlier, the architecture limits the scalability as it will become more complex with every server that is added. Especially since recovering or scaling up require the administrators to intervene. As the users are statically assigned to a certain server, the load cannot be balanced automatically when multiple heavy users have been assigned to the same server. Furthermore, when a failure occurs, the replica node will have to deal with the additional load. Leaving the cluster unbalanced until administrators distribute the load. To conclude, the scalability is limited and labour intensive to achieve with its current architecture.

The next section will analyse data replication strategies to enable automated scalability and load balancing.

5 Data replication analysis

In this section, the data replication techniques that could be used in order to achieve $n + 1$ scalability will be discussed, answering the second sub question (Q2). To determine which replication strategies could be applied, a list of requirements is presented first. Followed by listing of possible database solutions, whether they match, and what their strengths and weaknesses are. Once the list of database solutions is filtered, the remaining candidates are tested with a benchmark test to profile their performance under groupware usage load patterns. Based on the benchmark results, the best performing database will be selected as the database user for the scalable groupware service prototype.

As described in Section 2.2, all emails are stored inside the MySQL database using three important tables, in the current storage architecture of Zarafa 7.1, these are the:

- Hierarchy table; holds all the relations between entities.
- Properties tables; holds all the properties related to the entities.
- Transposed properties table; holds a transposed truncated view of the entities, sorted on the folder they are stored in and the property type.

A detailed look at the email clients, and the structure of email messages in MAPI, show that email messages can be split up into three parts. The first being the envelope of the message, these include the headers, in MAPI terms these are called the properties. The second is the body of the message, this is another type that behaves quite different compared to the properties of messages. And last, but certainly not least, are the attachments of the message. These hold the most data in general. These three data structures are covered in this order in detail in the following three sections. Followed upon that, the topic of indexing these messages is covered, as the user would like to sort the messages in a certain manner.

5.1 Property storage

The properties table holds all properties of entities such as mails, calendar items, recipients, and attachments. As all data is indexed using the transposed properties view, the properties table is merely used to store all properties related to a single entity. In the current architecture of Zarafa 7.1 the properties table allows each property to be accessed independently. With the design of the InnoDB storage engine in mind, these properties use a primary-key structure that will place the properties near to each other on the file system. This allows InnoDB to read all the properties related to an entity using a minimal number of IO operations. In other words, it is designed to read and write the data per entity. An alternative way to store the data would be to write it as a single binary object. Both these storage strategies will be discussed in the following two sub sections.

5.1.1 Single property read/write structure, record per property

The record per property structure, allows the server to read and write single properties of entities directly. As discussed in Section 2.2.2, the design of this structure focuses on minimal data transfer to and from the disk for single properties. By structuring the primary key to store related data near to each other, as depicted in Figure 9, the disk will be able to write the data as a stream, instead of chunks. This design focuses on the performance advantages of mechanical hard drive disks (HDD) to benefit from their sequential read and write speed, as discussed in Section 4.1.

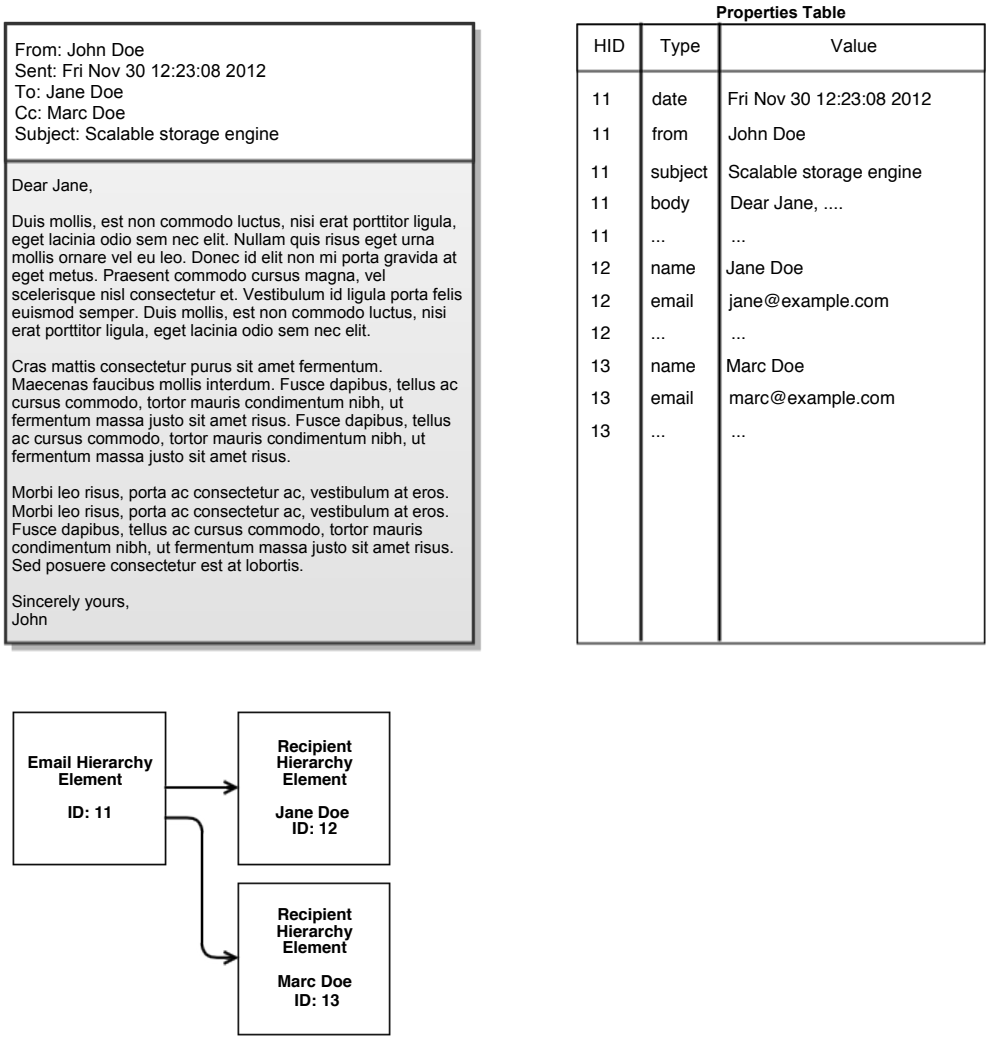

Figure 9: Email entity storage, storing each property (Type) of an object (HID) separately.

The most common operations that make use of updating a single property are:

- Setting a flag on an email;
- Marking it as read;

- Linking a reply to the original message.

These operations would execute with a minimal amount of data to be touched in the process. However, when multiple properties of a message need to be updated, this structure will require multiple rows in the properties table to be updated. The database will have to lock them one by one in order to guarantee the write operation.

There are a few operations that benefit from an architecture that tracks single properties. However, the most frequent operations are reading and writing the entire entity. As discussed in detail in Section 2.2.2, reading a message requires all properties of the message to be read. In order for this operation to perform well, the properties need to be retrieved with a minimum of disk and network operations. In other words, the data needs to be stored near to each other. Enforcing this requires a level of flexibility of a distributed database that could limit our options heavily.

5.1.2 Entity stored as a blob

An alternative would be to store the complete entity as a document or binary large object (blob), with all properties stored inside, as depicted in Figure 10. This would reduce the overhead of storing separate rows per property, and makes sure that the data is always written together. With each update or extension of the entity, the storage engine will rewrite the complete object. In most storage engines, this would force the engine to place the object as one data block, forcing it to write it sequentially. With both SSDs and HDDs this means that the data is written and read very quickly. Another advantage of this data structure, is the fact that the MAPI message object can be serialized or deserialized very quickly. Since the headers of a message are stored in one blob, the replication process is also easier. The client application would not have to query multiple servers, nor does the storage layer have to be told which properties are related to replicate them to similar servers.

In the document / blob storage, it could occur that the object grew to a size that does not fit to the same location as where it was previously stored. The storage engine will therefore find a place where it can store the entity entirely; this might be out-of-order with the hierarchy id order. With HDDs this would matter, as these disks would benefit of having all related data stored sequentially, SSDs do not have this issue as they have a very good random read/write performance.

Compared to the single row per property architecture where indexing the data is fast using the transposed properties table, a blob or document structured database would have to index the data differently to allow fast sorting of messages in folders. Blob structured databases are not able to analyse the internal structure of messages. Alternatively, the messages could also be stored as documents inside a document database. Some of the document databases

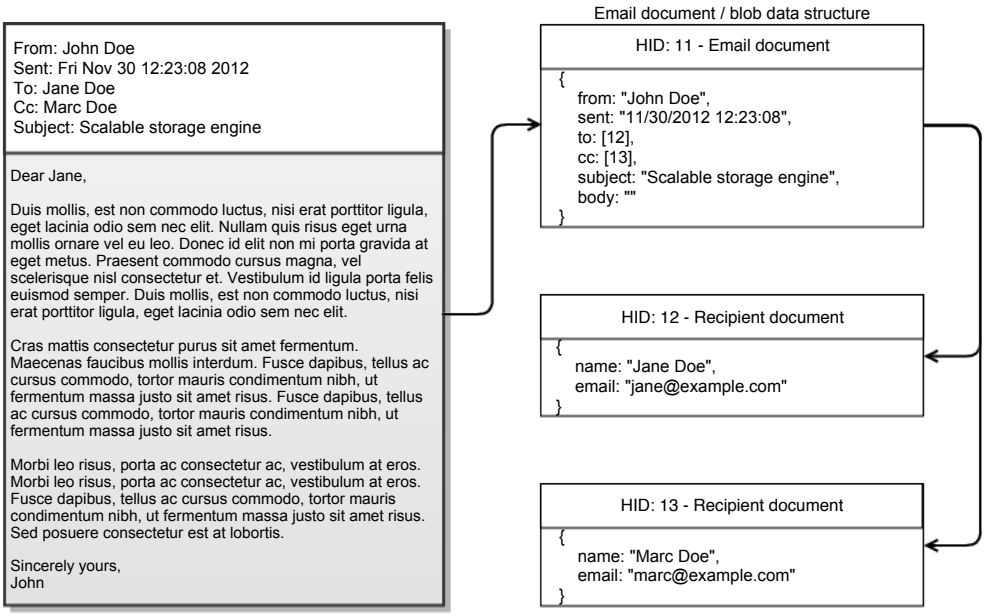

Figure 10: Email entity storage, storing each entity (HID) as a blob.

are very efficient in indexing the data. Unfortunately, the MAPI specification, as discussed in Section 2.2, states that the folders should be sortable on a wide group of properties. Keeping an index for all is too expensive, more details on this limitation and possible solutions are further discussed in Section 5.4.2 and 5.4.3.

5.1.3 Conclusion

The performance problem, discussed in Section 4.1, is caused by the fact that a lot of data segments are present in a groupware service, getting the best performance requires a lot of IOPS. Although new storage devices offer tremendously higher IO performance than older disks, the database structure should perform well with the IO constraint. Therefore, the level at which the single row per property structure is built up is far too detailed and IO consuming. Especially since most of the operations involve reading or writing the complete message anyway. Writing documents or blobs at once puts the data on the same location on disk, this protects the storage layer from writing a single document in fragments.

Even mechanical disks would perform better using this structure, as these disks perform optimally when data is written sequentially, and by design of blob storage databases the data is written this way. When new headers are added or some are removed, the internal structure changes. However, in case of blob storage the database does not need to organize the data, allowing it to write it sequentially. Therefore, there is less overhead when storing, reading, and writing the properties. The replication process would gain from this structure as well, as all related header data of a message can be replicated by replicating the blob. Reading all

the headers of a single message would be available with a single read request.

Reading and writing using this data structure requires less IOPS for normal message handling. The built-in ability to list messages ordered by one of the properties does not outweigh the constraints it introduces. Opting for blob storage requires designing another method to construct an index of the messages, this is explained in Section 5.4.3. However, when two concurrent write operations change some properties of the message, only one of them would survive in most blob storage layers. If a Multi Version Concurrency Control (MVCC) type of storage layer is chosen, the user or an automatic resolver could fix these conflicts, as discussed in the literature survey [37].

5.2 Body storage

In this section, the characteristics of the body data are analysed per storage architecture to compare them and decide which storage class would be able to serve the body data best.

When a message is delivered to the server, the body of the message gets written once. Whereas, when a user is typing a new message, the body of the message gets updated every couple of minutes, as the message is saved as a draft. In both cases, the likelihood that a message gets updated once it is sent or received is very small. Only other data types that are also stored in the database next to messages, such as calendar events, could see some updates over time. But even in this case, the updates are most likely taking place in the headers of the object, for example to move an appointment.

In other words, leaving aside possible files that are attached to a message, the body of a message involves a lot of data that is rarely updated. Thus, the characteristics of the body of a message:

- Some updates when it is still relatively new;
- Likelihood that the body gets updated decreases rapidly over time;
- And the body of the message contains a lot of data compared to the headers.

Naturally, all the characteristics of a body point to blob storage. Since Section 5.1 concluded that the headers should be stored as a blob to perform optimally, the body could be included in the same blob as the headers, or stored in a different blob as the characteristics are different. The advantages and disadvantages of these are further discussed in that order.

5.2.1 Headers and body as one blob

Section 2.2 showed that the headers of a MAPI message contain a truncated version of the body. This truncated body is stored for client applications that show a short summary

of a message directly from inside the list view. Instead of keeping this truncated version, the complete body of the message could be included in the blob that contains the headers. Putting the body and headers in the same blob allows the server to read and write all data related to one message in one operation. When the data is replicated it is ensured that the complete message is available at the responsible servers.

5.2.2 Body as a separate blob

The access and update characteristics of the headers and body differ tremendously. The headers of a message are read at least as often as its body. Additionally, the headers are read to list and sort the messages, whereas the client only retrieves the full body of a message when the message is opened. The headers of a message face an average of 1.7 write operations over time, as discussed in Section 2.1, while the body does not. When both are stored in the same blob, the body data would have to be written every time the headers get updated as well. Additionally this involves more data in the replication process, as the update involves the complete blob. Therefore, to utilize the relaxed requirements of the body, the body should be placed in a separate blob. Having both stored in separate blobs allows the body to be stored in a long term blob storage that faces a lot of reads and inserts, with some rare updates. Furthermore, since headers face more updates, keeping them small makes it cheaper to keep track of older versions, allowing the user to solve write conflicts.

To relax the system even further, the body can easily be cached, as it gets updated rarely and updates are most likely performed by the user itself. Compared to reading headers, the read operation of bodies may take a bit longer as well, but should still be in acceptable ranges in terms of performance.

5.2.3 Conclusion

When comparing the read and write operations of headers and bodies, the differences in requirements are very clear. As bodies face a lot less read and write operations, as discussed in Section 2.1 and5.2.2, the best way to take advantage of these relaxed requirements is to store the bodies into a separate blob.

5.3 Attachment storage analysis

In the current Zarafa architecture, the attachments are stored inside the database or on the local file system of the server. As only one server is able to access the database at any moment in time, and the disk is only locally accessible, this design has to be revised.

Attachments take about 81.5 per cent of the total storage, as was shown in Section 2.1. Moving this data to an environment where multiple servers can access the attachments would allow the database to be a lot smaller, and therefore easier to move around.

In this section several methods are discussed in detail. In the first section it discusses how network shares could be used, followed by an analysis how a distributed file system could serve as well. Last, but not least, the use of a globally accessible storage service as the target location is discussed.

5.3.1 Using network shares

Putting each of the servers in a virtual private network, where each of the servers can access the attachments stored on another server was one of the options considered. This would require all servers to store the attachments on disk, setting up a shared network service and each of the servers should be able to determine where exactly the attachments related to an entity are stored.

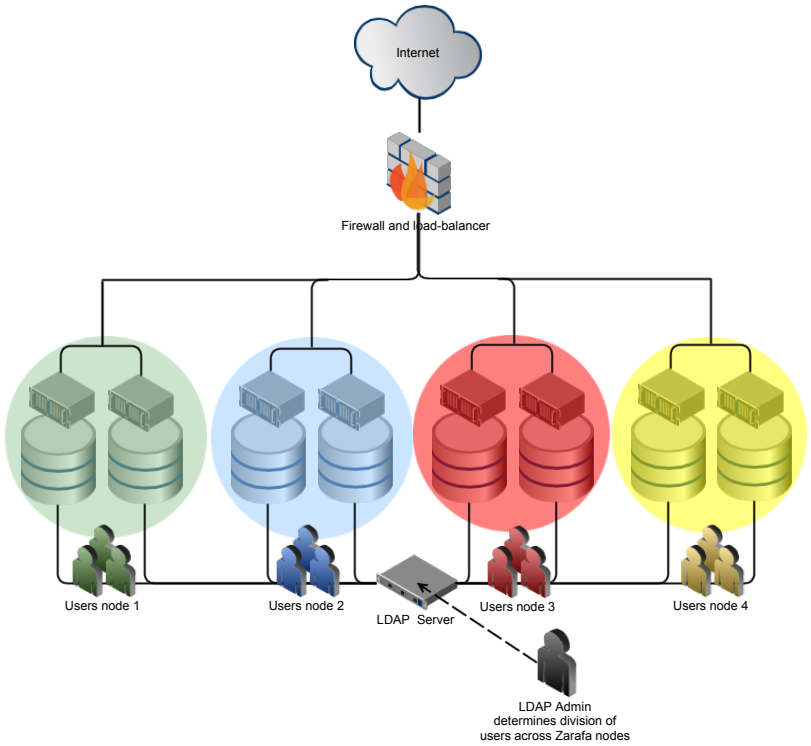

Figure 11: Using network shares to share the attachment data between the servers.

Advantages

- Transfer speed and latency: The advantage of this set-up is that each server in the

multi server set-up is able to connect to the other server very quickly, transferring a file from one node to the other is very fast due to the local area network connection.

- Manageable: All data related to a single user is stored on a specific server. In case a server starts to malfunction, only part of the cluster might go down. By setting-up a primary/secondary replication strategy, faults can be limited to a single problem domain.
- Development: No programming is required to implement this, accessing files on disk is already available. The replication between two servers, however, would still need to be implemented.

Disadvantages

- Syncing data: In case each user group is stored using a redundant server set-up, all data need to be kept in sync between the servers inside that group. As soon as the primary goes down, all data that have not been synced will become unavailable.
- No load balancing: Since the attachment data is stored on the same servers as the database for that user group, it experiences the same loads. In other words, a group where some users search a lot, might slow down the usage of attachments as well. In order to load balance the users across other nodes, more data needs to be copied to the other servers, all attachments and data in the database of the users that are migrated to another server.
- Disk IO: Accessing attachments requires additional disk IOPS, therefore slowing down the performance of the database server in case they both operate on the same disk. However, adding dedicated disks with fast IO for database storage could simply solve this.

5.3.2 Using a distributed file system

Another option is to set-up a distributed file system that is responsible for hosting all the attachment data to the groupware server cluster. This would require a cluster of file servers that together form the distributed file system. A distributed file system that has a single access point and manages redundancy internally is preferred. The distributed file system could be hosted on the same machines as the groupware servers, however, placing the file servers on dedicated machines enables them to use more memory for caching and decreases the amount of data that needs to be replicated in case of a failure.

Advantages

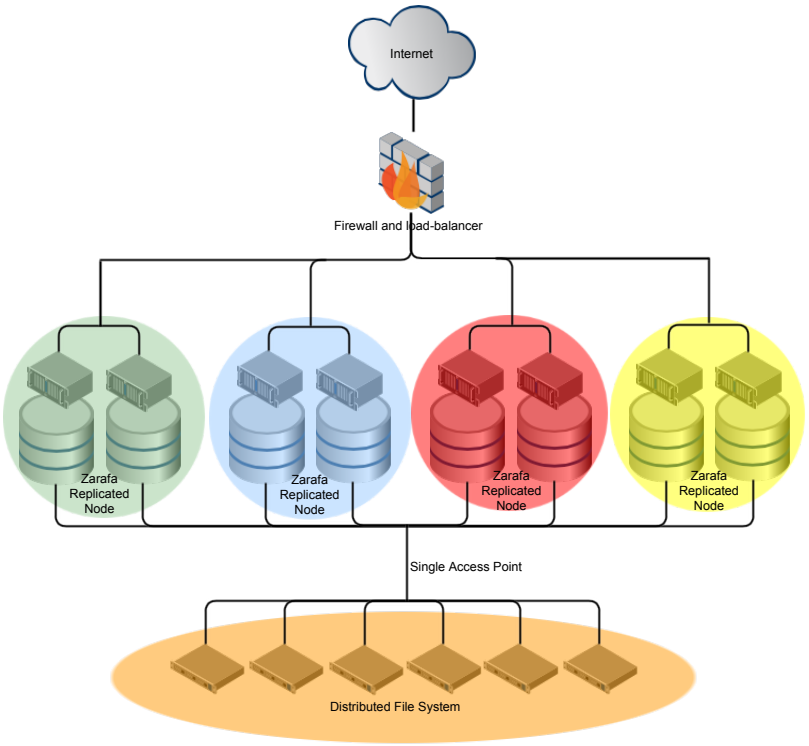

Figure 12: Using a distributed file system to share the attachment data between the servers.

- Automated replication: The DFS is responsible for replicating all the data across its nodes. In case a DFS node fails, all the data on that node need to be replicated to the other nodes to bring the cluster back in a stable and fully replicated state. On most DFS implementations this requires at least three replicas. How the replication is implemented depends on the type of DFS, the best replication strategy would have the redundant copies spread across all other nodes, not on a single other node. This allows the cluster to replicate the data very quickly by involving all the nodes in the replication process. In a large cluster, such a quick replication burst could replicate all data of a single node in about three minutes, copying terabytes of data in parallel.

- Single access point: To access an attachment, the server does not need to know where the data is stored. A single access point does not necessarily mean that it would have a single point of failure. Each server could, for example, connect to a random node in the cluster until it fails, that node will redirect the requests if the data is hosted on another node.

- Quick user migration: Since the attachment data is stored in a DFS, moving around users does not involve copying the attachments from one server to the other. Allowing the user data to be moved around quicker.

- Low latency: Although not as fast as true local storage, storing the files in the same

data centre allows a low latency and high throughput access to the attachment files.

- Costs: By choosing your infrastructure wisely, the DFS could be cheaper than a globally accessible storage service.

Disadvantages

- Extra cluster of servers: For performance, a dedicated cluster that hosts the DFS would be preferred. This requires more dedicated hardware to host the DFS.
- Management: The servers will need an administrator to manage it, since you add more machines, the probability that one fails increases.
- Scalability: The administrator will need to add more nodes and disks in order for the cluster to scale.

5.3.3 Globally accessible storage

Related to the idea of a DFS, the servers could also connect to a scalable storage platform that offers cloud storage on the Internet. Using a scalable storage platform allows all the servers to access the same data. Since the data is hosted elsewhere, the replication and administration of the platform do not need to be managed. This eases the implementation and allows the administrators to focus on the scalability of the groupware servers, leaving the scalability of the attachment storage to a third party.

Advantages

- Fully replicated: The replication process is managed by the third party that offers the storage service.
- Load balancing: Depending on the storage service, the data is automatically load balanced across the nodes at the third party, this results in a higher performance for files that are accessed more frequently.
- Scalability: This solution automatically scales with the growth of the cluster.
- Pay for usage: Storage is only paid for as soon as it is used. Using the storage layer from the start involves no start-up costs, therefore, it is much cheaper to use in the beginning phase.

Disadvantages

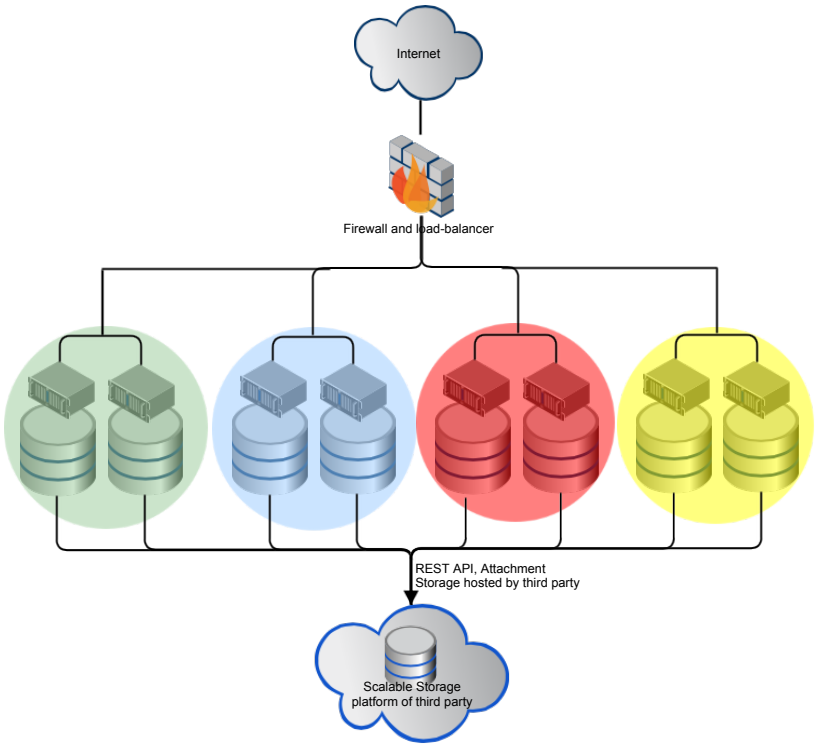

Figure 13: Using a globally accessible storage service to store all attachments on.

- Costs: As soon as you store a lot of data, the costs of managing it in-house will be lower than the extra price for the service the third party offers.
- Latency: To retrieve an attachment, the server needs to open a connection to the third party storage provider, since the data might not be stored in the same data centre as the groupware servers, this involves additional latency per attachment.
- Lock-in: As the amount of data stored inside the storage service increases, it will become difficult to transfer all the data to another storage provider.
- Security: In case the security of the storage provider is breached, all the attachment data could become accessible.

5.3.4 Conclusion

Considering the limitations and advantages of the shared network storage, distributed file system, and globally accessible storage, the last option as a service allows immediate performance at a fraction of the costs of hosting it on dedicated hardware. A globally accessible storage as a service enables unlimited storage of attachments, with a single point of access that scales, and balances the load internally. Moving the attachment data to this storage service solves the scalability and load-balancing issues for 81.5 per cent of the data, in terms

of total size. However, which service provider is chosen is a very important decision that is not easy to change later on. Which service provider could offer such a service is analysed in Section 6.3.

5.4 Listing messages

The ability to create a sorted list of the messages in a folder is dependent on the choice of the properties data structure of the messages.

With the single row per property the transposed properties table, as described in Section 2.2.2, the database could sort all messages in a folder quickly. For example, listing the messages in the inbox on their date is a simple query, requiring only a few IOPS to store the rows that contain the date property of the messages in the inbox. With blob storage, the database is not aware of the internal layout, and can therefore not query them efficiently. With some document databases, it is able to perform queries on the internal layout. The database must keep an index to do this efficiently, this will be discussed further below.

5.4.1 Single row per property

Section 2.2.2 explains the data structure of the transposed properties table and how it allows the server to list all messages in a certain order. The first query uses the transposed properties table to retrieve the message order in that folder. Afterwards, another query will retrieve all other headers of those messages, such that the client application is able to show the other header fields in the index as well.

An advantage of this structure is its ability to sort new messages immediately when they have been written to the database. A major issue here is the level on which data is written to the database, as this structure requires the properties to be written individually. This involves a lot of overhead in storage, and requires a transaction to perform the write operation. As discussed in Section 5.2 of the literature survey [37], these types of transactions are not scalable as they focus on consistency and partitioning.

5.4.2 Document structure

With a document structure that allows efficient querying of internal message properties, the database engine would be able to sort the messages in the right order. However, keeping an index for all properties of the messages requires a lot of storage and processing time at write operations, as the database will need to determine where the message is placed on the list of each of its indexed properties. Once the database has figured out where to put it, it needs to list the message reference and property value in the index. As described in Section 2.2, the

data structure of MAPI allows 256 types of properties in its sort query, if one would keep an index for each property, the database would have to perform $256 + 1$ write operations. One for the message itself, and one for each indexed property.

Considering the fact that the database would have to make sure its write operation does not get lost, each write operation would have to lock the index it is working on until it is finished. If the database would not take the write operation of the index seriously, the message would be stored in the database, but no reference allows the client applications to find it. In other words, a single write operation would require locking 256 indexes, one at a time. The probability that multiple processes are using these as well increases rapidly. This has serious consequences for the write performance and overall efficiency of the database storage layer.

The problem with document structures that use the index functionality of its database, is that the indexes need to be determined up front. While the properties that get sorted on differs per user. When a document database is chosen, the most popular selection of indexes need to be determined, all users that sort their folder using a different property require a lot more resources in CPU and memory terms. As these need to be sorted on the first query, keeping the index in memory as long as the user is connected, and updating it when new messages arrive.

5.4.3 Blob structure

With a blob architecture, the database layer is not aware of the internal structure of the messages. Therefore, the messages cannot be sorted by the database itself and have to be tracked otherwise.

There are several possible solutions to deal with this limitation. The first would be to keep track of the messages sorted on the sort property of the client, for as long as the client is connected. Another option would be to keep track of the sorted messages in a separate blob that is updated in an ACID manner. These two strategies are further compared below.

In-memory index The advantage of an in-memory index is that it is very quick to update. The server could construct the index on the first query of that folder by retrieving all messages and ordering them as requested. As all the data required when building an index is available, this can easily be kept in memory. However, this requires some time to construct, especially if there are a lot of messages in a folder. As this happens when the user opens the folder for the first time, and gets slower over time, this will definitively be noticed by the end-users. Besides, keeping the indexes while the users are connected will require memory that can be utilized more efficiently in other use cases. Although updates can be parsed in-memory, it

still requires other processes to be locked out of the index while it is writing. Besides, when a user requests another sort order, the server should either keep track of multiple in memory, or sort the messages as requested.

Blob index Alternatively, the server could also store the sorted list of messages in the database in a separate blob. Storing the index would make it available directly when the user connects and opens a folder. This structure operates the same way as a document database index, however, one major difference is that with a self-constructed index the property that is used to sort on can vary per folder. Instead of keeping track of all 256 properties as indexes, this structure would allow us to keep the most common sort views as an index. This structure would require more storage and involve some latency when the index is requested, however, the latency of a read request would by far outweigh the processing time required to sort when in-memory indexing is used, and storing data on disks is a lot cheaper than keeping it in memory.

By keeping track of a deferred update list, as described in Section 2.2.2, new messages can be kept in a separate list to be processed later. Better yet, since the index of messages is constructed outside of the database layer, the first time the user gets back to the folder, or when a server has time to process them, the messages can be inserted in the sorted list. This would allow messages to be delivered by writing it as a blob in the database, and adding it to the deferred updates journal, requiring no locking on the indexes unless the user is online. As discussed in Section 3.1 on the performance requirements, the message has to be delivered in several hours, this sort and write operation can be delayed until the server has time to catch up.

5.4.4 Conclusion

Keeping each property separately or using a document based index mechanism introduce high costs to enable the flexibility of MAPI when sorting folders. Considering the requirements, the best solution would be to use the blob index as its easier and cheaper. To decrease its latency, the index could be cached in-memory on the server as well. This allows the server to utilize the best of both worlds; fast retrieval of the index and fast updates. This architecture lets the server decide which properties need to be indexed on a folder level, allowing the server to efficiently balance the costs of storing an index versus generating it. Furthermore, using the deferred update structure allows the server to prioritize index updates, postponing deferred updates for users that do not seem to use the folder anyway.

5.5 Conclusion

The different characteristics of headers, bodies, and attachments play a major role to replicate the groupware data to achieve a $n + 1$ scalability.

The headers and body of a message are best stored as separate blobs in the database, or even different databases as they have different requirements in terms of consistency and data size. The blob structure decreases the level of detail, simplifying the replication process and reducing operational costs.

However, since the headers are placed in a blob, an index needs to be created to enable fast access to the list of messages in a folder. With a custom indexing written in blob form, the flexibility requirement of MAPI to sort on any property is met at acceptable costs. Indexes need to be created on the fly, and should be written to permanent storage in case it is accessed frequently.

Attachments are best stored inside a globally accessible storage service. This eases the design in terms of scalability and load balancing, as the service provider is able to deal with these issues at a far bigger scale.

As this Section showed, the data structure determines which techniques are available to replicate all user data, thereby partly answering the second question (Q2); the next section continues to answer this question by covering specific replication implementations through different database providers.

6 Storage comparison

The literature survey on online scalable and fail-safe data technologies [37] formed the fundament for the research on databases that match the requirements listed in Section 3.2. The scope of the research are databases that were presented in the literature survey, as well other databases that were not covered due to their similarities, but are used frequently in production environments. Based on the data analysis presented in the previous section, the candidates that met most of the important requirements are: Cassandra, MongoDB, Riak, Voldemort, and HBase, these are listed in no particular order.

6.1 Property storage solutions

In order to make a good selection of candidate data technologies, they need to meet the requirements listed in Table 5. The list of requirements is based on the requirements stated by customers, end-users, and system administrators of groupware solutions, as discussed in Section 2.1. Based on the design as presented in Section 3, the list of requirements has been extended with the technological requirements to realize this design.

Cassandra is an Apache licensed open-source key-row database supported by Datastax [6]. Various companies used it in production, including Facebook with more than 600 nodes and 120 TB of storage. Twitter, Cisco, and Reddit are some of the many companies that still use Cassandra in production. The database allows unlimited number of columns per row, each with a variable length. All data is distributed using a consistent hashing algorithm [38], without the need of a central node to coordinate the cluster. Cassandra is Rack-aware, with multi data-center support to guarantee uptime [23].

MongoDB is an AGPL open-source document-oriented database [1] developed and supported by 10gen. The data is serialized using a JSON data model, allowing a rich query language with range requests, and indexing support. MongoDB supports master-slave replication for high availability, with support for location awareness data storage. Sharding between its servers is automatic, as long as the servers have enough resources to add more data to their shards. However, adding new shard nodes to the cluster for write performance, or adding more replica servers for increase read performance, requires a system administrator to intervene [20]. Important note on MongoDB is the 16 MB limitation of the document size [64].

Riak is an Apache licensed key-value store that is based on the paper of Amazon's Dynamo database [10, 24]. Basho Technologies is the main developer, offering support for its Enter-

	Requirement	Type	MoSCoW
1	The database must have proven history of success.	Support	Must
2	The database must have an active community or development team.	Support	Must
3	The database must be able to store blobs of 20k, such that it can store complete entries at once.	Storage	Must
4	It must have a replication mechanism built-in.	Replication	Must
5	It must guarantee at least three replicas for each entry.	Replication	Must
6	The database must randomly distribute the slaves for each entry written, such that all the data on one server is replicated on a random set of other servers, not just a fixed few.	Replication	Must
7	When a new server is added, it must bring it up to speed without decreasing the replication factor.	Management, Replication	Must
8	The database must be aware of the load on its nodes, such that the average load can be determined.	Management	Must
9	It must detect failure of one of the servers instantly.	Management, Replication	Must
10	The database should have an option to get professional support.	Support	Should
11	These blobs should be efficient with append requests, in other words, the database should be able to write 10 percent more data to the item without requiring more processing time than a normal in-place update request.	Storage, Performance	Should
12	Write operations that were queued to a failing server, should fail too, unless the minimum requirement of two replicas is met.	Management, Replication	Should
13	The database should have incremental scalability properties, adding a new node should be as easy as dropping one	Management, Performance	Should
14	It should be able to detect hot zones, moving the data from those servers to other locations automatically.	Magagement	Should
15	If the database requires master nodes to organize everything, the master should have its own redundant replica as well.	Replication	Should
16	The master server should be able to recover its state when all masters fail.	Replication	Should
17	When new data is accepted, it should guarantee that it is written to at least two nodes.	Replication	Should
18	When a new server is added, the network should have balanced the data across the nodes within 24 hours.	Replication	Should
19	New servers should be able to help with the load increase, within half an hour.	Performance, Replication	Should
20	Write requests should take at most 500 ms.	Performance	Should
21	Read requests should take at most 200 ms.	Performance	Should
22	The write requests should be non-blocking, such that multiple processes can write to the same data entry if necessary.	Performance	Should
23	It should be able to operate on commodity hardware, in other words, it should be able to work while facing continuous failures across the cluster.	Replication, Storage	Should
24	Data should be consistent in at most five minutes.	Replication	Should
25	It could replicate the data of a complete server in 15 minutes, moving it to the existing servers in the cluster, such that data loss due to failure of multiple servers is less likely.	Replication, Performance	Could
26	It could add new nodes as load increases, to deal with load spikes fully autonomously.	Management	Could
27	It could create more replicas for hot entries, such that the load is spread across multiple servers automatically.	Management, Performance	Could
28	In the replication process, if the database could recognize the location of the server, i.e. which rack it is in, and is able to make two servers in the same rack less likely to become each other's replication server for data served is a pre.	Replication, Management	Could
29	It would be efficient, if the database supports range queries based on the primary key of the entries.	Storage	Could
30	If it supports range queries, it should place the entries near to each other in terms of primary key distribution, such that the query only needs the least amount of IOPS to execute it.	Storage, Performance	Could

Table 5: The list of requirements for property storage

prise edition. Keys are distributed across a 160-bit ring, following a SHA1 based consistent hashing [10, 11]. All Riak nodes are equal, no master node is required. Riak is eventual consistent, with built-in methods to detect and resolve collisions automatically. Read consistency can be enforced per request, setting a minimum of nodes that should agree upon the value.

Voldemort is inspired by Amazon's Dynamo database [55], just like Riak. Voldemort has been developed by LinkedIn to solve high-scalability storage problems, however, neither LinkedIn, nor any other company is backing up Voldemort by offering support contracts. Each server holds one or more partitions of the ring, these partitions are equally divided without requiring a master node. It is able to balance the load by moving partitions if these become hot zones, allowing both write and read operations to be scaled linearly.

HBase is based on the proprietary column-oriented BigTable project of Google, made available under the Apache license [7]. Just like BigTable, HBase runs on top of a distributed file system, in this case the HDFS file system to distribute the data. It provides linear and modular scalability, with many contributing companies that offer support, like Cloudera [21]. All data is automatically sharded with redundancy built-in to provide failover. HBase supports atomic increments [57], with multi versioning as the underlying concurrent write resolver, updates become new versions of the same object, only the last object is returned, which is resolved at read time [58, 57]. HBase runs on top of HDFS, through this requirement, HBase fully adopts all automated failure recovery, and location awareness features as built in HDFS [58, 7]. This also includes automatic balancing of data between nodes to equalize the load, as well as moving data out of hot-zones [63].

Impossible alternatives Databases such as Megastore and Google F1 have been analysed as well, unfortunately these do not have an open-source counterpart available yet, using the proprietary software is no option unfortunately as their developers are not offering it. Megastore has been used as the database layer to store email data at huge scale. The article of Google F1 looks very promising; unfortunately the system is not available nor any open-source projects that are based on this engine. BigTable has an open-source counter-part named HBase, although BigTable itself is not available.

The Dynamo database is Amazon proprietary, based on the pricing model of Dynamo it would be unsuitable for use cases that require a lot or IOPS with 20KB blob values. Riak is an open source alternative, based on the Amazon Dynamo paper [11].

Another database that has been looked into is CouchDB, but it does not meet the requirements, as it is designed to replicate single node data to multiple nodes [40]. However, the

size purpose of this research is to find a data store that scales far beyond anything that a single node could manage.

6.2 Body storage solutions

The header data is split from the body data to optimize the write and read queries when the header is updated or read in a batch. The header is read and written to far more often than the body of the message, as was discussed in Section 5.1. However, for both blob objects a key-value database would suffice, as long as it would know how to deal with long message bodies. Therefore, the discussion for the header property storage solutions applies to the body storage solutions as well.

6.3 Attachment storage solutions

In Section 5.3, the conclusion was drawn that the attachments were best stored on a cloud storage provider. However, moving data to a cloud storage provider means that the provider should meet up in terms of uptime, performance, availability, and scalability. Therefore, in order to determine which storage provider to opt for, an analysis is presented including the following five candidates, being: Amazon, Google, HP, Microsoft, and Rackspace. The analysis compares the scalability of the providers, the throughput speed, the latency of the provider, and how failure tolerant the cluster has proven to be. Based on these criteria, the best candidate for storing the attachment data is selected.

6.3.1 Amazon S3

Amazon's S3 platform is the biggest and most mature storage provider. As of June 2012, Amazon S3 stores more than 1 trillion objects in their cluster [4]. Nasuni published an annual report about the state of the different cloud providers that exist [49]. According to Nasuni, Amazon is the second fastest storage provider in terms of writing files, with an uptime of 100 per cent during the test period of 31 days [49]. While S3 showed no errors while writing data, the read process faced about 0.0018 per cent of read failures. Nasuni concluded that the scalability within a container performs well, showing no degradations in performance if a container was filled with millions of objects.

6.3.2 Microsoft Azure

Microsoft Azure performed best in terms of reading, writing and deleting files during the Nasuni benchmarks [49]. With a 99.996 per cent up time during the benchmark tests, no

read or write errors, an average response time of about 500 ms, and just slight variations in terms of processing time, the Azure storage service has proven to be mature [49].

However, some of their latest performance improvements are only available to new customers, according to Microsoft's own publication on this matter [46]. According to Nasuni, the cluster scaled well, showing no significant performance degradation if the number of objects inside a container increased.

6.3.3 Google

Nasuni's availability tests, showed that the Google storage platform was the slowest during the response time benchmark, although their uptime was 100 per cent. Throughout the tests it showed no write errors, 0.0030 per cent read errors, and with increased container sizes it scaled well.

6.3.4 Rackspace

Rackspace is one of the founding parties of OpenStack. The Rackspace storage service showed performance spikes of 26.1 per cent, with 0.000001 per cent write failures, and 0.0012 per cent read errors. Performance varied enormously during the availability tests, spiking from about 900 ms to 1.9 seconds. The platform was online 99.962 per cent of the time through the availability benchmarks. The service performed worse while more objects were added to a single container, showing that the service is not linearly scalable in terms of objects inside a single container [49].

6.3.5 HP

The cloud storage service provided by Hewlett Packard is based on OpenStack. During the benchmark tests of Nasuni, the service replied 99.977 per cent of the time, with performance variations up to 23.5 per cent [49]. Throughout the read and write tests the service returned some errors, 0.00017 per cent in writing, and 0.0099 per cent while reading data. Like Rackspace, the service did not scale very well, showing significantly slower performance when millions of objects were stored.

6.3.6 Conclusion

Microsoft Azure has shown to be the best performing storage provider overall, especially the stable performance that is delivered is outstanding compared to the other service providers.

Therefore, the proposed storage provider to work with is Microsoft Azure, as it scales well, performs best of its class, and showed the least amount of performance variations.

6.4 Conclusion

In terms of the properties and bodies of email, several key-value databases have been matched with a list of requirements to form a candidate list. In this analysis, databases like Cassandra, Riak, Voldemort, and HBase have been discussed as the candidates to run tests on. The benchmark tests for these databases are presented in the next section.

As presented in Section 5, the attachments of emails have different characteristics and could be stored inside an external storage provider to move 81.5 per cent of the data outside the cluster. Providers that were compared include Amazon S3, Microsoft Azure, Google, Rackspace, and HP file storage. Of this subset, Microsoft Azure showed best performance in tests performed by Nasuni [49].

7 Key-value benchmarks

In order to determine the database architecture to use, the databases discussed in Section 6.1 have been tested on their performance in reading, updating, and inserting data, as well as their ability to recover from network failures and full node failures. Section 2.1 showed the statistics on mail usage online, on average the body and headers of emails comprise around 26 KB of data. Since the headers are stored separately from the bodies, the databases are tested with 10 KB of data per entity to simulate the storage of these data types.

The scenarios and the logic behind these, are both covered in the first section. Followed upon this, is the section that explains the experimental setup, showing all the ins and outs of the benchmark cluster, and why this has been set-up this way. Hereafter, the performance results of each of the databases is showed one-by-one, covering MySQL Cluster, Cassandra, Riak, Voldemort, and HBase, in that order.

7.1 Benchmark scenarios

The performance tests are executed on a three-node cluster and a six-node cluster, to determine whether the database meets near-linear scalability. At the start of the performance tests, the databases are filled with about twenty times more data than the nodes have as memory. This forces the database to use the IO storage layer to store and retrieve the data, allowing the test to simulate the massive amount of data accessed, where most of the data queries are in segments that are not available in the cache of the database.

100% inserts The tests started with the insertion tests, immediately generating data for the following test stages. In this test the performance degradation of the inserts were measured.

50% reads - 50% updates In this stage, the database is benchmarked with a 50 per cent read, 50 per cent update test, this test stresses the database in terms of updates and reads, one way or the other this will show where the logic of resolving conflicts resides, in the read operation, write, or whether it is offloaded in a background process.

80% reads - 50% updates The test that followed executed 80 per cent read, 20 per cent update queries to simulate the massive amount of reads and writes that would be required for the headers of emails.

95% reads - 5% inserts To simulate reading bodies and receiving new emails, the database is projected a workload to perform 95 per cent read queries, and 5 per cent inserts.

100% reads The last workload tests how the database will perform if there would only be read operations fired at it.

Temporary network failure Once these tests have been performed on both clusters, the six-node cluster is used to perform the failure tests. The failure tests start to test how the database responds if a node suddenly becomes unavailable, but picks up a couple of minutes later, simulating a network disconnect of one of the nodes. Facing an 80 per cent read, 20 per cent update workload to test whether the read, update, or both operations would show increased latencies, during such a failure or recovery process.

Full node failure and recovery Followed by another test, in which a full node fails, and a new node is inserted in the cluster to take over the load a few minutes after. This test used the same workload profile as the temporary disconnect test.

7.2 Experimental setup

For this set of experiments, nine Amazon EC2 m1.xlarge instances were used. These machines have 15 GiB of memory and provide high I/O performance. Using high-end machines for the set-up makes it is less likely that there are more machines running on the same server that hosts the virtual machine. The experiments used a dataset of about ten times the size of the available memory, such that the performance of the IO layer would be visible. This simulates the use of groupware data, in which the data does not fit in memory and is stored on disk as well. The set-up cluster is further discussed in detail in Appendix A.

7.3 MySQL Cluster

Hypothesis: In theory, the MySQL Cluster allows scalable and fault-tolerant storage of data. Separate SQL nodes are able to connect to the data nodes in the cluster to execute SQL queries on this data. Due to this extra layer, the complexity of transactions increases, and since the MySQL cluster is designed to be fully compatible with the normal MySQL database, it is expected that the functionality comes at the cost of performance and availability, when using the disk storage of this cluster.

Results: The latency results are presented in Figure 14, it shows that the MySQL Cluster performs less optimal when nodes are added. However, the latency of both read and write requests is quite low, the benchmark showed that even in a six-node cluster the read and write requests are able to finish in about 20 ms. However, in terms of failure tolerance, the database showed a lot of side effects when the cluster would try to recover from a node failure,

as is seen in Figure 15 and 16. With spikes up to 100 ms during recovery, and complete failure of the cluster when the old node was replaced by a new one. Due to time constraints and limited documentation on the recoverability, the cause of this crash has not been analysed in depth.

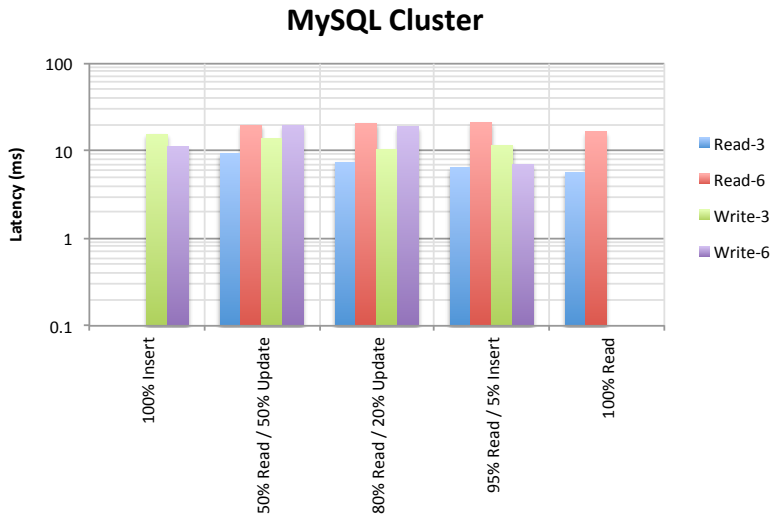

Figure 14: MySQL Cluster benchmark results: While inserting new data, the six-node cluster performed better than its three-node cluster counterpart. Whereas the three-node cluster had more throughput while performing update and read operations. The graph shows that the difference is significant, especially while performing read only operations. This shows some limitations in terms of the scalability of the database.

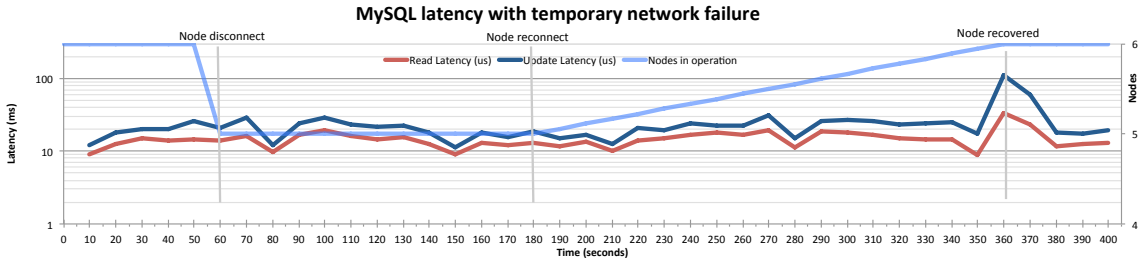

Figure 15: Six node MySQL Cluster: Temporary node disconnect. The graph shows that the performance is quite consistent while it endured a temporary disconnect of one of the nodes. The latency increased from 20 ms to 35 ms during the failure. When the node finished its recovery process, the responsibility hand-off caused a short latency spike up to 156 ms. The light blue line shows that a node failed after one minute, the recovery process started two minutes after. Full recovery of the node is depicted by the blue line being drawn at the sixth node line after three minutes of recovering.

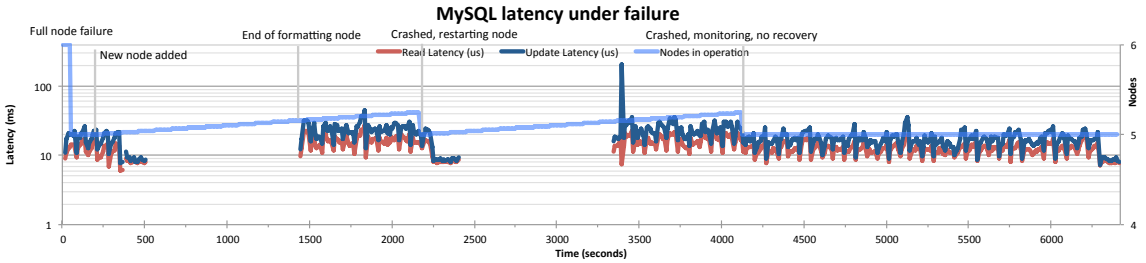

Figure 16: Six node MySQL Cluster: Full node failure. According to the documentation of MySQL Cluster, the database should be able to recover from a full node failure. Unfortunately, after more than two weeks of figuring out why it fails, I decided to leave the test as having failed and continue my efforts on the other databases. The graph depicts one of the many attempts to restart the cluster after a node failure, as the lines show, it failed to recover at each point where the line disappeared. The light blue line shows the recovery process and where it restarted, as this line shows the recovery.

Strengths and limitations: A major strength of the MySQL Cluster is its ability to setup a single database cluster that is able to perform complex SQL queries, as well as execute key-value queries directly on the subsystem used by the SQL node. The scalability of the two layers, being the data nodes and query nodes, are defined separately, allowing the cluster to add additional query nodes if the a temporary spike in queries occurs. While running the benchmark, the cluster crashed many times with a very generic error message, "Table is Full". According to the documentation [51], the error could be caused by several error scenarios. After researching all the available information on this topic, the cause of the error was the use of a blob field in the database to store data. Apparently, the cluster saves the first 256 bytes of each blob or text field in memory, while storing the rest on disk [53]. Translated to the email domain, storing 256 bytes per record in memory is a lot, per one thousand users this would require about 9 GB of additional memory per year, leaving spam messages out of scope. Alternatively, the blobs could be segmented into multiple varchar columns, however, the limit for each row is set to 14 KB of data [51]. As the average body size is determined to be 10 KB, discussed in Section 5.2, some emails might be too big to store in a single record. During the configuration process, two size limits had to be predefined. First, it required to add log files and data files manually to the cluster, these are required to store the records in. And second, the maximum amount or rows should be defined. These both show that the cluster is not built to be scalable in terms of the amount of data.

Conclusion: The hypothesis did not hold, the MySQL database became slower as more nodes were added due to the complexity, however, the cluster was unable to recover after a full node failure occurred.

7.4 Cassandra

Hypothesis: Cassandra is designed to perform best on write operations with extreme throughput characteristics. It is completely decentralized, with advanced redundancy and failover support. Hundreds of nodes and hundreds of terabytes of data are no problem for Cassandra in terms of scalability. Since Cassandra is designed to determine the correct value on read, the read operations will most likely be a lot slower than write operations.

Results: Figure 17 shows that the database performs very well write operations, with a latency around 1 to 2 ms. Reading data, however, requires about 65 to 80 ms per request. Although this is still acceptable in terms of the performance, when a node failure occurs, as is depicted in Figure 18, the latency can spike up to 180 ms. In terms of full node recovery, Cassandra is the winning party, it was the only database that recovered a full node in less than 1200 seconds during the tests, shown in Figure 19, whereas other database could easily take about 9000 seconds or more. Surprisingly, the write performance faced the most performance hits while a new node was in the recovery process, with some requests taking 180 ms, whereas reading did not take longer than 100 ms in this process.

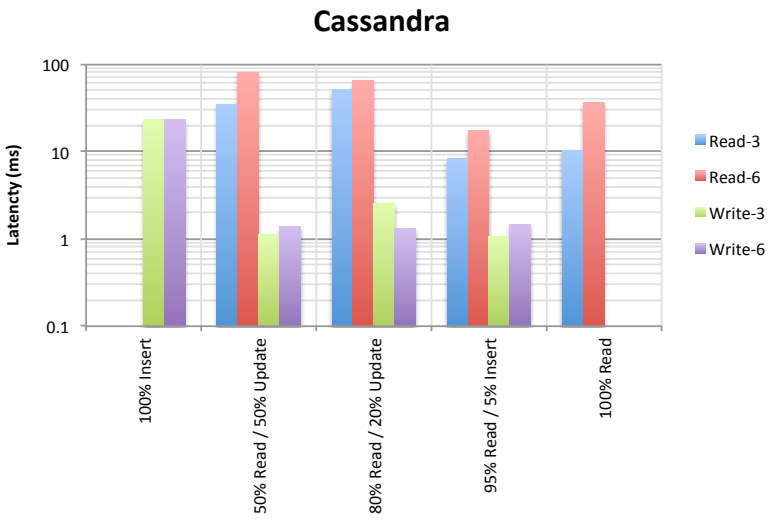

Figure 17: Cassandra Benchmark Results: The fact that Cassandra is built to write a lot of data is clearly shown in this graph. Read operations determine the latest version, and therefore require more time to execute. With writing data the difference between the three and six-node cluster is not significant. With reading, the communication overhead to resolve which version is the latest limits the throughput for the six-node cluster a bit.

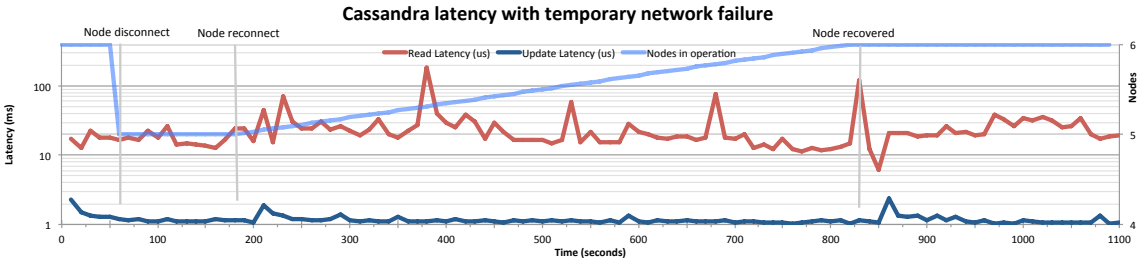

Figure 18: Six node Cassandra Cluster: Temporary node disconnect: Throughout the recovery process of Cassandra, the write process was able to perform at a consistent rate. The read, however, showed a couple of peaks that took about ten times longer than it normally would.

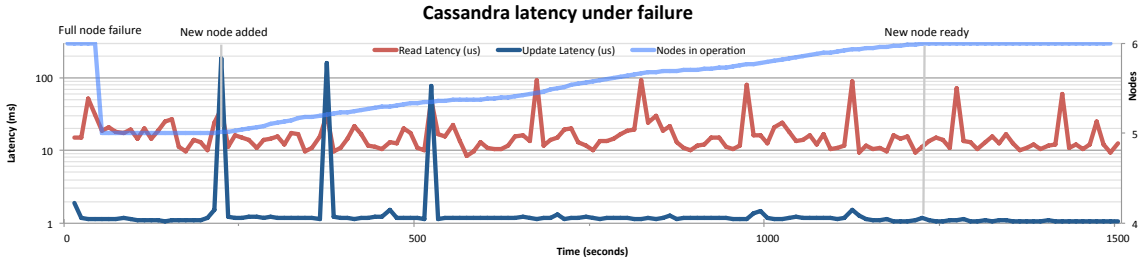

Figure 19: Six node Cassandra Cluster: Full node failure: Noteworthy, are the spikes in update latency at the beginning, these map directly on top of the read latency spikes that occurred rhythmic throughout the tests. Outstanding, was the performance of Cassandra to recover a node that was lost, as soon as a new node was added, the replicas copied the data concurrently to the new node, taking around 17 minutes to complete a full copy of about 20 GB.

Strengths and limitations: Datastax, the group behind Cassandra, provided very clear performance instructions, with a special segment on tuning the performance on virtual instances as well. The benchmark clearly showed that the conflict resolving logic is placed in the read operation, making it fairly unstable in terms of the required latency. The write operation, however, was very consistent throughout the benchmarks. The three-node cluster was able to perform read operations more quickly than the six-node cluster, as it needs to retrieve the latest updates on all the nodes, but as the cluster is bigger it experiences more concurrent operations, increasing the overhead on the read operations.

Conclusion: The hypothesis did hold, the Cassandra cluster clearly showed that the database performs best with write operations, even when failures occur.

7.5 Riak

Hypothesis: Riak supports guaranteed writing [10], ensuring that each write operation will be performed. Riak uses vector clocks to achieve this, allowing a resolver algorithm to merge the write operations if multiple concurrent writes occurred. In terms of availability, no management nodes are required, as all nodes are equal in Riak. This improves the reliability of the cluster, as there is no single point of failure. Riak is easy to scale up, and reaches an almost linear performance increase when new nodes are added [10].

Results: Riak performed very well throughout the tests. Read operations were almost twice as fast as write operations, where write operations took around 20 ms to complete, as is depicted in Figure 20. Even with 80 per cent reads, 20 per cent updates, the database was able to execute read requests in 7 ms on average. In case of a disconnected node, the read operations were executed with a stable average latency of 8 ms. The write performance did notice the disconnect though, its latency spiked up to 350 ms, with an average of 75 ms. With a full node failure, the maximum read latency measured was 30 ms, with an average of 7 ms, as is shown in Figure 22. The recovery process degraded the performance to write data, latency degraded up to 375 ms, with an average of 43 ms.

Strengths and limitations: A major strength of Riak was its documentation, it covered all aspects from failure of nodes, expanding the cluster, and even documentation how to perform benchmark tests. Furthermore, the developers behind the database wrote detailed performance instructions. However, the topic of replacing a failed node with a new node was not completely clear, through some extra research on this matter and trial and error the test that includes node failure succeeded. The YCSB tool crashed while performing node failure tests, main reason for this was the inability of the connector to recover from executing requests

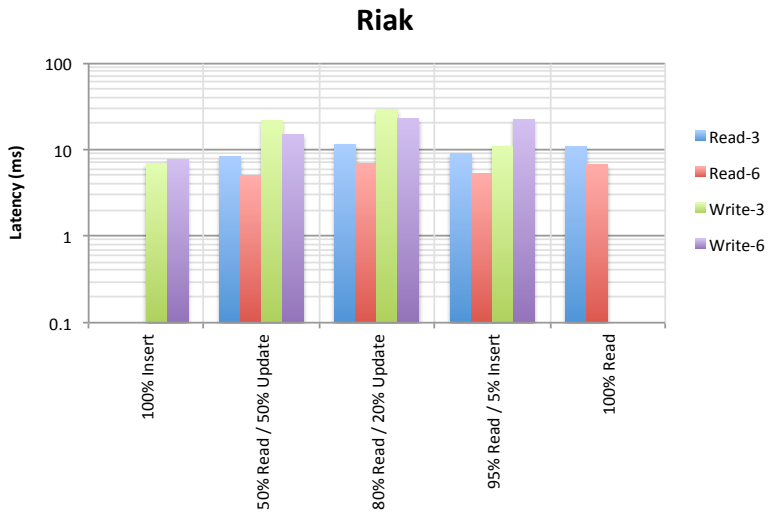

Figure 20: Riak Benchmark Results: Except for inserting new data, Riak performed better with each of the operations in the six-node cluster, compared to the three-node cluster. Riaks design to focus on well performing read operations is clearly visible in the graphs. The write performance of this database is well below 30 ms for all of the operations.

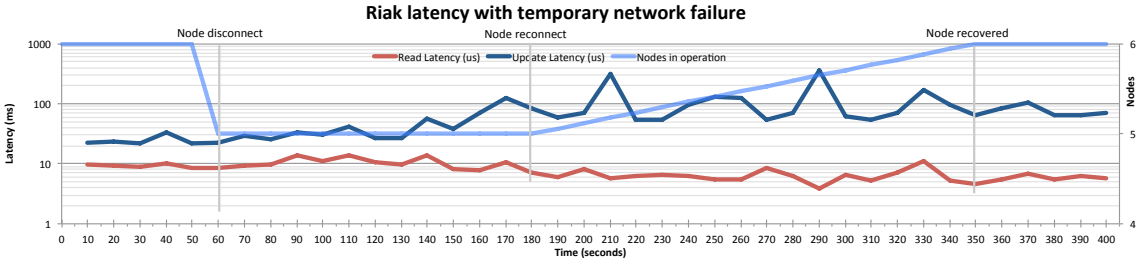

Figure 21: Six node Riak Cluster: Temporary node disconnect: Throughout the temporary disconnect tests, the read latency was around 10 ms. With the write operation, the failed connection to one of the nodes caused some of the update requests to last about 400 ms, on average the write latency was around 100 ms.

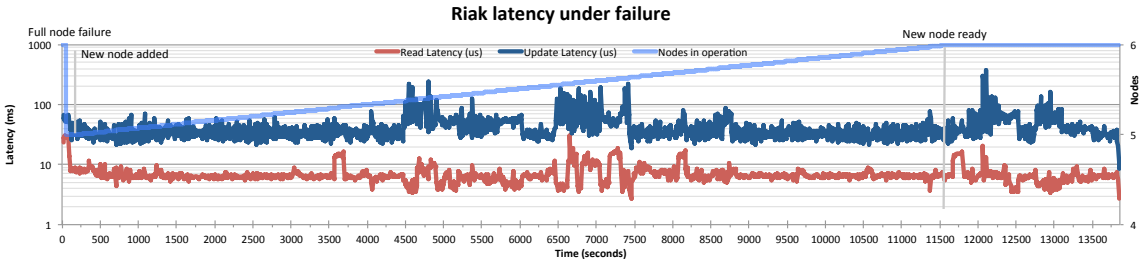

Figure 22: Six node Riak Cluster: Full node failure: A full node failure resulted in the same behaviour as a temporary node disconnect. The write performance takes most of the hit, with a few spikes that remain below 300 ms in terms of latency. There was some turbulence in the read performance around 6500 seconds to 7500, after this recovery period the latency was very consistent at around 10 ms.

on a failed node. Through some code changes in the connector used in YCSB, the benchmark tool was able to operate while enduring failures. Although the performance degraded quite a lot while trying to recover the failed or disconnected nodes, the performance only caused write operations to slow down, whereas read requests executed stable like no failure had occurred.

In Riak, it is possible to specify the replication factor, which by default is set at 3. However, according to the documentation of Riak, setting a replication factor does not mean that the data is replicated to specifically that number of different nodes [12]. Especially, if a small cluster is used, the changes are high that some of the data might be replicated less than that. Statistically speaking, this would be no issue in a large cluster, but it is something to be aware of.

Conclusion: The hypothesis did hold, the Riak nodes performed better when more nodes were used. Scaling up was as easy as booting new Riak nodes.

7.6 Voldemort

Hypothesis: The way Voldemort uses versioning to resolve concurrent write requests, the database is most likely performing best on write and update operations, forcing read requests to deal with the resolving operations, also known as read-repair [55]. Since there is no master required, and it supports automatic replication and balancing of data across the cluster, this key-value database would be suitable for high availability workloads, specifically those who require non-blocking write operations.

Results: With the exception of 50 per cent read and 50 per cent write operations, the cluster performed extremely well. Read and write operations both finished in about 10 ms on average, however, when the cluster got bigger the performance did degrade noticeably, as can be determined in Figure 23. In contradiction to the hypothesis, the read performance of the cluster was a bit faster than the write performance. Both by far meet the performance requirements as stated in Section 3.1. In terms of scalability, the cluster scales nearly linearly when more nodes are added, this is a big advantage showing that the overhead of adding more nodes is almost neglectable.

When the cluster experienced failures, the performance degraded a bit, with spikes to about 200 ms on a full node failure, shown in Figure 25. Recovering the node took a lot of time compared to the other clusters, however, with a few outliers, the performance of the cluster stayed well below the 50 ms line. With a temporary node disconnect, shown in Figure 24 the cluster recovered steadily as well.

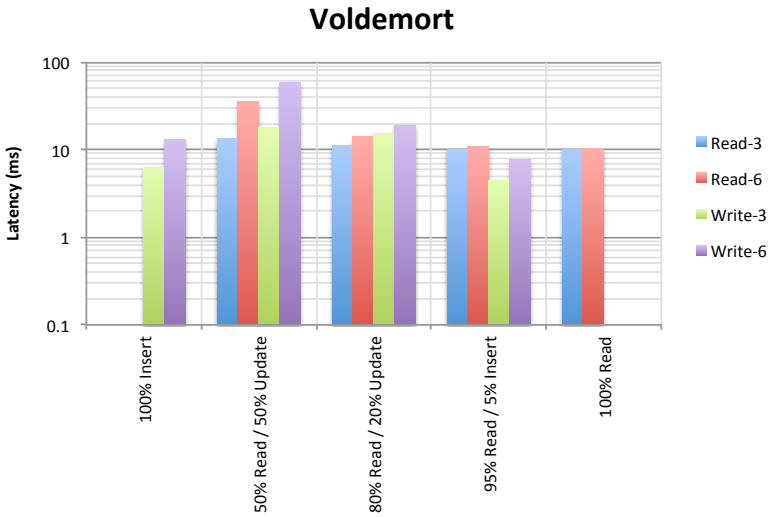

Figure 23: Voldemort Benchmark Results: Comparing the three and size node clusters, the operations appear to perform a bit worse at scale. On average, Voldemort is very well performing in terms of read and write operations. A test at larger scale should be executed to determine the true scalability limit of this database, due to resource constraints these tests have been limited to at most six nodes.

Strengths and limitations: In order to make Voldemort perform at its best, the cluster was set-up using some performance instructions for running Voldermort on EC2 instances as written down by True [62] in 2010. The results showed that the performance degradation, while it experiences failures, is neglectable. Although the cluster scaled in a linear fashion, the configuration by far does not. In order to add nodes to the cluster, the configuration files of all the cluster nodes had to be updated to discover the new node. This requires all nodes

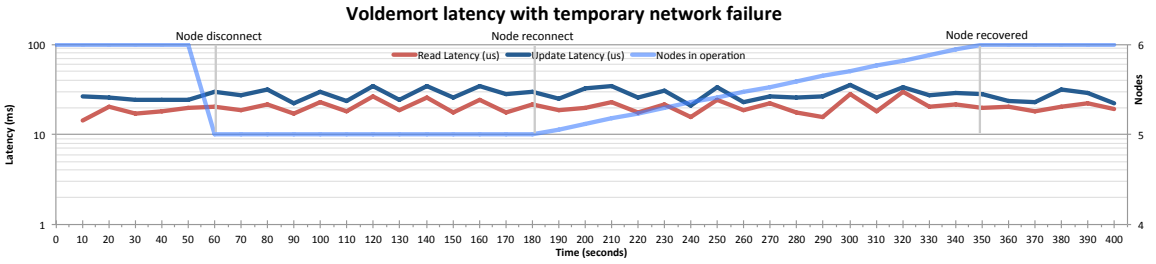

Figure 24: Six node Voldemort Cluster: Temporary node disconnect: The performance of the cluster was very consistent, even though one of the nodes temporary lost his connection with the rest of the cluster.

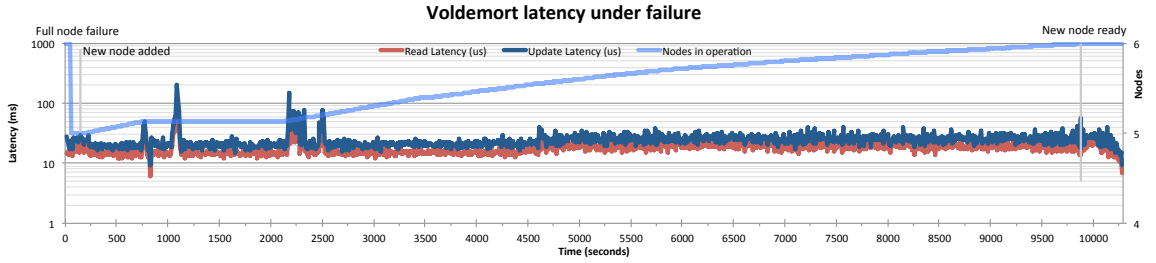

Figure 25: Six node Voldemort Cluster: Full node failure: The read and write throughput of Voldemort are outstanding, except at the start of the node recovery process, the read performance averaged around 20 ms, whereas the write operations faced a latency of about 28 ms. Setting up a cluster and the manual tasks involved while performing recovery tasks do not favour the database.

to be instructed to find the new node, instead of an automatic discovery of nodes, making auto provisioning of resources a difficult task. The later option has been put on the agenda of the Voldemort development team for years now [55], why this has not made a release yet is unclear.

The available documentation of Voldemort is very brief and unstructured, while community support and examples are lacking as well. This is a very important aspect, as these are key to a good integration. With limited resources on documentation, the platform is not a good candidate to integrate, especially since there is no company backing the product either. The connector integrated into the YCSB did not support multi-threaded access to the cluster, causing concurrent write operations to fail when they update the same record. In order to perform the benchmarks, the connector has been improved. Rewriting part of the connector showed that there is no clear documentation how to solve conflicts, forcing an in-depth analysis of the source code to determine the real cause of this problem.

When a node becomes completely unavailable, the default implementation of the connector is to wait and retry, blocking the client thread several minutes before giving up [27]. This can be resolved, but requires some additional development work before it would be suitable for production usage. Another issue with Voldemort is its inability to rebalance data automatically, possible solutions have been researched by Gao et al. [31], however, since its publication none have been implemented yet.

Conclusion: The hypothesis did not hold, throughout the tests its write operations were slower than read operations most of the time. While failures occurred, the read operations performed more consistently than write operations as well.

7.7 HBase

Hypothesis: HBase has proven track record in large-scale data storage on top of a Hadoop cluster. As concluded in Section 6.1, the database is built to utilize the sequential write performance of HDFS, therefore this database will perform very well during write operations, at the cost of slower reads where data needs to be analysed to determine the latest version. The availability and failure tolerance of this cluster are very good, as it is built on top of Hadoop. Considering that it is designed to perform well with writing data, it would be a good candidate for writing data logs to.

Results: The results showed that the HBase is write focused, in fact, writing was more than 10 times faster than reading in most scenarios, as is shown in Figure 26. The differences between the two were even further apart when the cluster endured a failure. Reading data

is just within the boundaries, but especially when the cluster experiences failures, the read performance degrades with latency peaks up to 580 ms. The latency measurements when failures occurred are depicted in Figure 27 and 28.

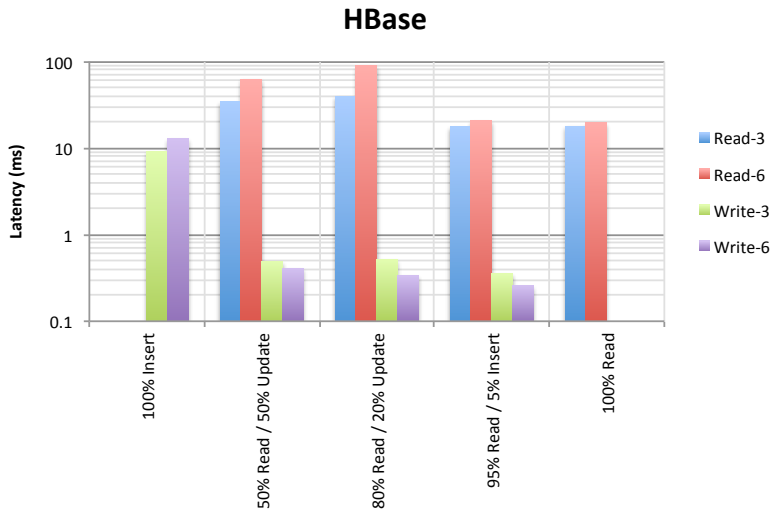

Figure 26: HBase Benchmark Results: Like Cassandra, HBase is a database that favours write operations in terms of performance, the graph depicts this clearly. In a larger cluster, the read operations take more time to decide on the latest value.

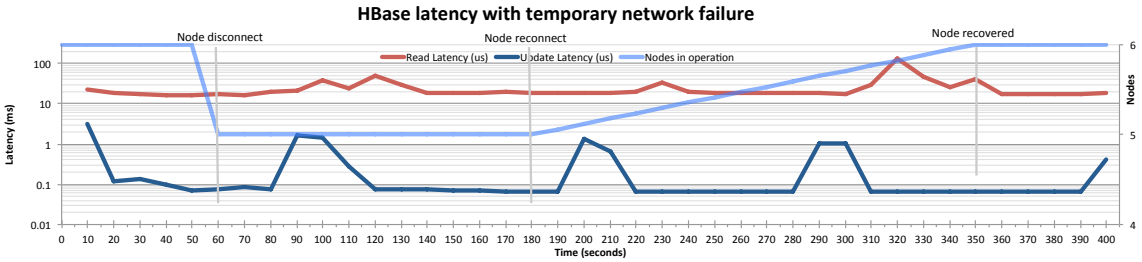

Figure 27: Six node HBase Cluster: Temporary node disconnect: The write throughput was extremely high with HBase, even in the event that a node disconnected, the highest latency measured for writing was still lower than 2 ms. The latency while reading data is a bit higher, this was measured at around 30 ms on average.

Strengths and limitations: If writing data would be the most important performing operation, with a stable average write latency far below 1 ms HBase would be the clear winner. However, as read performance is most important in the case of groupware data, with continuous failures faced by large clusters this database is not a good candidate. A major strength of HBase is its abstraction, since it runs on top of Hadoop, it is able to build on top of the failure tolerance logic of Hadoop. By design, another cluster runs ZooKeeper to monitor the usage of the Hadoop cluster resources. This allows nodes to connect to the HBase cluster

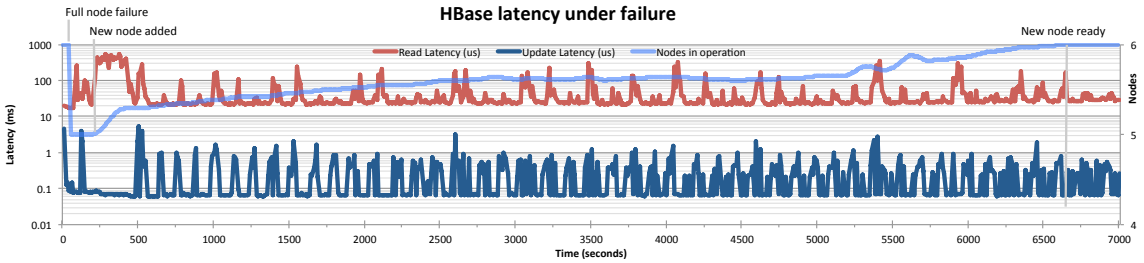

Figure 28: Six node HBase Cluster: Full node failure: Due to the logarithmic scale, the write performance looks very instable. With a few exceptions, the HBase database performed all of the write operations within 5 ms. Reading data did not perform that well, when the node started its recovery process, the read operations took around 400 ms to complete, whereas on average the read operations took about 57 ms to complete.

through a simple single point of entry without creating a single point of failure, as ZooKeeper itself is highly available as well. HBase allows new nodes to deal with the load without a single restart, just letting the node join the cluster is enough to get it configured and running. However, there exists a lot of documentation on the Internet that is out dated and not fully compatible with the latest version of the HBase cluster. If one would use a popular Linux distribution such as Red-Hat Enterprise Linux, the installation process would be fairly easy, as an automatic installer for clusters exists. However, with the benchmarks, the Amazon Linux AMI was used as the Linux distribution, as this distribution is optimized to perform on top of provisioned EBS devices used during the test.

Conclusion: The hypothesis did hold, the write performance was outstanding, especially in comparison when compared to the read latency. These characteristics would fit perfectly to write data logs to.

7.8 Conclusion

The benchmark results showed clear differences in the structure and performance of the databases. The experience with setting up the databases and performing the different benchmark scenarios gave a lot of insights in the flexibility of the database, how well it was documented, and whether the database supports failover natively or a connector should implement this logic.

In short, MySQL Cluster would be a good candidate if the cluster were relatively small, allowing complex SQL queries to be executed on the data. Cassandra and HBase showed outstanding write performance, even in the event of failure. Whereas, Riak and Voldemort covered the other segment, where read operations are most important to perform well.

8 Proposed solution

With the insights acquired in Sections 5, 6, and 7, a proposed solution to store groupware data, across multiple servers that together behave like one robust service provider, allowing scalability in the order of $n + 1$ is given. In other words, this section uses the answers to the sub questions to answer the main research question of this thesis. The fifth sub question (Q5), that questions how a cluster could operate fully autonomously, achieving scalability, failover, and load balancing has not been answered yet; this topic will be discussed throughout this section.

8.1 Architectural design

Following the requirements discussed in Section 3, the proposed architectural design entails multiple clusters of servers to meet these requirements. These clusters are depicted in Figure 29, the clusters are coded with letters to ease referencing them.

Clients are able to use the Web Client or their own client software to access their email. The web servers host the Web Client in cluster A. All web-based sessions will access the email service like a MAPI client would. In case the user uses their own MAPI compatible client, such as a desktop application or a mobile device with an email client installed, the client will connect directly to cluster B through the MAPI protocol.

Since MAPI requires the server to keep track of the client state, each user is assigned to a server that will keep track of this state as long as the client is connected. Cluster B is able to track which server is responsible for the user that connected. The servers in cluster C holds this session data.

The Zarafa load balancer cluster, depicted in cluster B, is responsible to track the state and redirect clients to the right servers. The nodes in this cluster keep a cache of the state of the nodes in cluster C, on failure this is easily recovered by querying the nodes of that cluster. The Zarafa mail servers, depicted in cluster C, keep the state of the clients that connected. This state includes an index for the folder that the user is viewing at this moment, as well as caching of mails and folder attributes. All of this data is recoverable by retrieving the data from the database layer, or by forcing the clients to reconnect; the reconnect will instruct the client to inform the server about the state it expects.

All the Zarafa mail servers are connected to the database load balancer, this load balancer allows ease of configuration on the Zarafa mail servers. As Riak is accessible through a REST interface, a simple HTTP load balancer is sufficient. The load balancer should be able to detect the state of the nodes, by polling one of the servers in the database cluster it can determine whether they are all up and running. The load balancer redirects the requests to

one of the available database servers, depicted in cluster F, which will process or redirect the query internally. The Riak database nodes hold all email data except attachments, the attachments are written to an external storage provider. The mail servers in cluster C and the write agents in cluster E access the attachments.

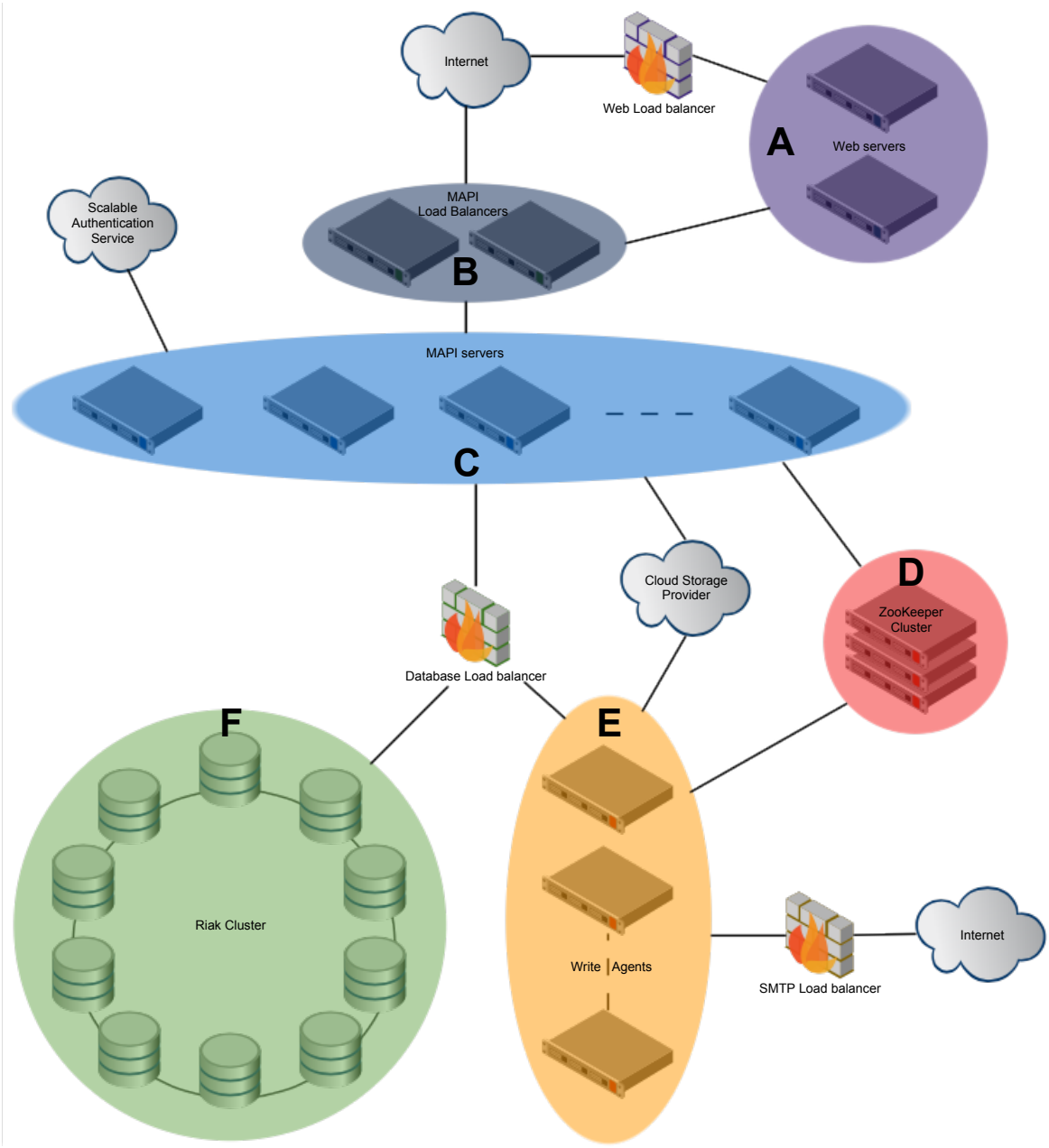

Figure 29: Architecture design of intermediate components

Mail is delivered to the cluster through the SMTP protocol, passing the SMTP load balancer as the first node. This load balancer will forward the request to one of the available Write Agents, depicted in cluster E. These agents are each responsible for all write operations to

folders, these include delivery of mails, sorting the inbox, moving emails, deleting emails, etc. Internally, all these tasks are assigned to an agent through the ZooKeeper cluster, depicted in cluster D. The ZooKeeper cluster is able to track state of connected nodes very accurately. It offers exclusive file access and locking mechanisms based on the paper of Burrows [16] on the commercial implementation of the Chubby locking service.

Authentication is required at the level of the mail servers, however, since this is outside the scope of this research, it is assumed that a scalable and fault tolerant authentication service is available. The single point of entry of the web servers and Zarafa load balancers, the mail interface, and database cluster layers are discussed individually in the following sections in that order.

8.2 Automated user segmentation

Each email is referenced by a unique key, this key references to the body and header of the email. Each email needs to be placed in a folder, in order for the user to access it.

The scenario might occur where two processes write to the same folder concurrently, in this case one of the emails might be lost in the process. This would happen when they both read the folder index at the same point in time, place the received email inside. When these processes would have written the new index, the version that finished last will be available. The other message would be lost in the process. Since iterating over all records in a key-value database is like searching a needle in a haystack, the process of storing an email in a folder should guarantee that the email is referenced correctly. Therefore, at any moment in time there should be at most one process granted write access to a specific folder. In the proposed design, the write agents are responsible for updating the folder indices.

To guarantee that only one process is writing to a specific folder, a process that would like to write to the folder needs to have acquired the lock on that folder. The locking mechanism is provided by a ZooKeeper cluster, as further discussed in Section 8.4. Other processes will need to wait until the lock is released, before they can acquire the lock and start writing to the folder.

It is important that a node failure will not lead to starvation of other processes, the lock should be released in that case. Additionally, if multiple nodes interact with the same user, deadlocks could occur. In order to solve this problem, a segmentation algorithm is proposed to assign each user to a specific server. Making that server the responsible node to perform all write operations on the folders of that user. The algorithm should keep track of the users that have been assigned to a certain server, as these users need to be assigned to another server if it failed. Keeping a long list of all the users and their assignment is very inefficient, as the hand-off process would require locking each user individually. Therefore the proposed

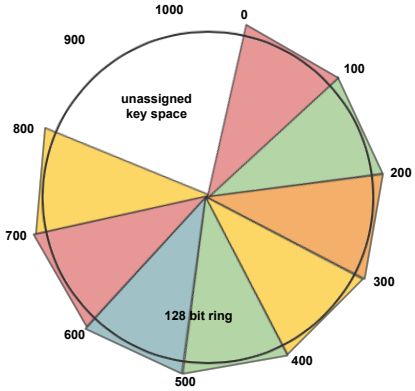

Figure 30: Division of the complete 128-bit key ring into virtual nodes, all virtual nodes hold an equal chunk size in terms of key range. Where the users are assigned on the ring is randomly determined by their UUID. The unassigned key space depicts the range of virtual nodes that are not assigned to a node and should be recovered. The colours of each of these virtual nodes depict to which server these are assigned.

algorithm uses an approach similar to Riaks implementation to divide data across the nodes in the cluster with so-called virtual nodes, depicted in Figure 30.

By splitting the user list into a fixed number of virtual nodes, the hand-off process requires a single lock to be acquired per virtual node. For example, if we would have four servers, each having four virtual nodes, each server would have to check whether it can access that virtual node. Each virtual node might represent a couple of thousand users. Choosing a fixed number of virtual nodes allows us to hash the user key and determine in which virtual node it is located. However, fixing the number of virtual nodes completely, would limit the maximum number of servers to that amount of virtual nodes. Therefore, the algorithm should be able to scale the number of virtual nodes both up and down.

The leader executes Algorithm 1 to balance the virtual nodes over the available nodes. The algorithm takes the previous active configuration and balances that with a minimal amount of move operations. In other words, if a node is added to the cluster, the virtual nodes that are released at each of the nodes will be the bare minimum to equalize the load.

In case a single node only operates with one virtual node, the leader will decide to split the virtual nodes such that further growth is possible quickly. The leader will execute Algorithm 2 to instruct each of the nodes to release the virtual nodes it holds and take part in the new configuration like any normal configuration update is processed. Each of the nodes will process these configuration changes by executing Algorithm 3. If the number of virtual nodes has changed, then it should release all the virtual nodes it has and continue when it is ready. The next steps will check whether the node is part of the cluster, if it is, the node will release all virtual nodes that are acquired and are no longer part of its configuration. Followed by acquiring all the virtual nodes that were recently assigned to the node.

Algorithm 1 Balance algorithm to distribute the virtual nodes across the nodes evenly

```
1: procedure BALANCE(virtualNodes, nodes)
2:     minPerServer = floor(virtualNodes.length / nodes.length)
3:     leftOver = virtualNodes.length % nodes.length
4:     remainingVnodes = []
5:     for node in nodes do
6:         shouldHave = minPerServer + (node.number < leftOver ? 1 : 0)
7:         if node.vnodes.length > shouldHave then
8:             remainingVnodes.push(node.vnodes.popLast(node.vnodes.length - shouldHave))
9:         end if
10:    end for
11:    for node in nodes do
12:        shouldHave = minPerServer + (node.number < leftOver ? 1 : 0)
13:        if node.vnodes.length < shouldHave then
14:            node.vnodes.push(remainingVnodes.pop(shouldHave - node.vnodes.length)
15:        end if
16:    end for
17: end procedure
```

Algorithm 2 Segment virtual nodes, scaling up or down with the number of virtual nodes.

```
procedure SEGMENTVNODES(virtualNodes, nodes)
    newNrOfVnodes = 4 * nodes.length
    remainingVnodes = [0 . . . newNrOfVnodes]
    for node in nodes do
        node.vnodes = remainingVnodes.pop 4
    end for
end procedure
```

Algorithm 3 Process the staged configuration on all nodes

```
procedure processStagedConfig(stagedConfig)
    if currentConfig.nrOfVnodes != stagedConfig.nrOfVnodes then
        releaseAllVnodes()
        once finished: processStagedConfig(stagedConfig)
    end if
    nodeConfig = stagedConfig.findNodeConfig(this.uniqueId)
    if nodeConfig != null then
        releaseTheseVnodes = this.myVnodes - nodeConfig.vnodes
        for vnode in releaseTheseVnodes do
            vnode.release()                          ▷ Release the vnode immediately
        end for
        for vnode in this.myVnodes do
            vnode.acquire()          ▷ This will wait async until it acquired the vnode
        end for
    else
        releaseAllVnodes()                    ▷ Release all the vnodes immediately
    end if
end procedure
```

8.3 Single point of entry

For the users, the setup process to connect with the cluster should be as easy as possible. The fact that there is a cluster behind the service should not be visible to the users. In other words, the complete service should behave as one big, extremely reliable, server that the user connects to. Just one domain to connect to, no further cluster knowledge should be required, as discussed in Section 3.3. By discussing a proposed solution for the single point of entry, this section will answer the third sub question (Q3), how to set-up a failure resilient single-point of entry to ease its usage for end users.

How the user connects to the service depends on their client, with his or her own MAPI-compatible client on their mobile, tablet, or pc the user would connect to the mail load balancers, depicted as cluster B in Figure 29. If they use the web client to connect, the web mail interface behaves like another client seen from the cluster itself, as the web servers connect to the mail service like any other client would. Internally, each user gets assigned to a certain MAPI server that will keep track of their connection details. Allowing the server to cache the user data and tracking where the user is browsing currently, this is a requirement to enable the stateful MAPI protocol.

To a user, the service would have a single point of entry if the domain name to connect with would be equal for all users. Strictly speaking, if the user does not need to know about the exact cluster assignment, the service would look like a single server to the user, even though the users are placed on different nodes in the cluster.

Which methods would enable a single point of entry for an email service is discussed first. However, the two portals of the email service, being the web client and the MAPI mail service, require different approaches. How these services are accessible as a scalable service with a single point of entry, without having a single point-of-failure, is discussed separately in the concluding section.

8.3.1 Possible solutions

With each solution, it is important that the user is not required to specify direct routes to servers and that the solution does not become the bottleneck of the service. The software should determine to which server the user gets connected, and should immediately make the server unavailable if it fails. This allows the cluster to scale and reliably replace nodes if they appear to have failed in the meanwhile.

This section discusses the following solutions, DNS, reverse proxies, and load balancers.

DNS is a very important aspect in the design of a single point of entry, whichever additional solution is chosen, the DNS layer will always be put in operation to ease the configuration. As new users will not be able to connect to the cluster if the DNS service failed, this is a very important service to setup to be fault tolerant.

Besides being the telephone book for the Internet, DNS can be set-up to deal with some of the single point of entry difficulties as well. DNS can connect users to the cluster by distributing the queries over a set of servers. There are many ways to distribute the load across the servers in this manner, the most common and easiest method to use with DNS, is to use a round-robin approach. A round-robin approach walks through the set of servers and connects the incoming user connection to the server, based on a fixed order. However, the cluster will need to update the DNS record in case a server fails or a new server is added to the cluster. The way DNS works makes it possible that connecting clients will fail for up to 60 seconds even after a node has been removed from the set.

Reverse proxy servers provide access to other resources by redirecting all the requests through their own interfaces. In other words, all the requests will first go through the reverse proxy, which will query the servers internally and return the results. Reverse proxies are easy to set-up, can be used as the gatekeepers of the service, and would allow rescuing of queries that failed somewhere inside the cluster without the user noticing. However, as the reverse proxy itself can also fail, the user could notice a failure anyway, the problem just moved to the proxy layer. One major advantage of a reverse proxy technique is its ability

to provide SSL termination services, moving the encryption load on the mail servers to the reverse proxy.

Load balancer servers, operate like a DNS service, but provide a faster and more intelligent solution to the problem. They process the establishment of a new connection, by connecting the user to a specific server in the cluster behind it. Which server it picks depends on the mechanism that is chosen, options include round robin, intelligent selection of a node that faces the least load, or other factors that it can monitor. Compared to DNS, this allows a smoother operation of continuously changing clusters, as intermediate DNS servers might cache the results, possibly routing new clients to failed servers.

8.3.2 Conclusion

The cluster will face failures continuously. If a server fails, the users that were connected to that server could notice a hick-up, but should be able to recover quickly. However, new users should only be connected to servers that operate correctly. To guarantee this last requirement, the service should be able to react instantly to failures, such that any interrupted server would be removed from the redirection list of servers. Along with the requirement to connect all devices of a single user to the same server, the best solution would be to use the load balancer implementation.

Since the web client connects to the MAPI service like any other client would, the problem of providing the web client as a single point of entry is similar to other web service solutions. The proposed solution utilizes a simple HTTP load balancer to recover from server failures immediately. The HTTP load balancer, however, should be smart enough to assign users to the under-utilized servers. In case partitioning occurs, the load balancer should recognize that a specific server is not able to communicate with the MAPI service and consider that web server as failed.

8.3.3 Implementation of the load balancer

At the MAPI server level, the load balancer servers are actual Zarafa mail servers, as these are able to look-up where users are stored internally to cleverly deal with this logic. The assignment of the users to specific servers is stored inside the ZooKeeper cluster, depicted as cluster D in Figure 29. The ZooKeeper cluster is designed to enable highly reliable distributed coordination, Section 8.4 discusses the internals of ZooKeeper. If a MAPI server is not able to communicate with ZooKeeper, Riak, or the storage provider, the server should be flagged as failing. Failing servers will immediately be removed from the MAPI server pool. Additionally,

any server that faces heavy load is removed from the pool for new connections, until it has recovered to a normal utilization level.

8.4 Cluster coordination

To keep track of the state of the cluster, a locking service should be used. All nodes should have access to the state information, therefore it should be important that the service guarantees exclusive write access and has locking capabilities. In 2006, Burrows wrote a paper on the Chubby lock service [16]. Chubby is used by Google Inc. to appoint the master server in their Google File System [18] in a cluster of thousands of nodes. In 2010, Hunt et al. [35] presented a paper on ZooKeeper, offering coordination services that are wait-free, scalable, and reliable. ZooKeeper is based on the internals of the proprietary Chubby lock service, as presented by Burrows. ZooKeeper meets all of the requirements necessary to provide a reliable distributed coordination to elect a leader node and coordinate segmentation of data across the nodes in the cluster.

As described in Section 8.2, only one node should be able to write to a folder at the same time. In the proposed architecture, the nodes that are responsible for writing are referred to as Write Agents. A leader node among the Write Agents executes the most important part of the user segmentation algorithm, how these nodes operate is further described in Section **??**.

The ZooKeeper nodes orchestrate the Write Agents. The ZooKeeper nodes are responsible for electing a leader among the Write Agents, track tasks that should be executed by the Write Agents. Furthermore, it holds the active and staged cluster configuration settings as set-up by the leader of the Write Agents. When a new configuration is made active, the ZooKeeper cluster will ensure that only one node is able to access a certain folder at a time. Furthermore, since ZooKeeper enables ephemeral locking, each node in the cluster locks their own ephemeral file that others use to determine which nodes are alive. In case a node loses its connection with the ZooKeeper cluster, the lock is released immediately, notifying all other nodes about the failure. This structure is used to track the active MAPI servers as well.

Any task that requires exclusive write access to the folder index, is first written to the batch task directory of that user on ZooKeeper. Through ZooKeeper and the segmentation algorithm discussed in Section 8.2, only one Write Agent can monitor the batch task directory of a user at a time. This Write Agent will process the task and finalize it by deleting the task from the batch directory. Tasks that require this locking service include sending, moving, copying, deleting, and delivering emails, as well as sorting folders.

The ZooKeeper cluster has important tasks to fulfil, therefore it should be fully fault tolerant. Since all the tasks require at most 100 bytes of data to be stored inside ZooKeeper, the data is replicated very quickly to the replica nodes in the ZooKeeper cluster. Additionally,

ZooKeeper guarantees that the data is replicated before confirming the request; this ensures that locks would survive failures inside the ZooKeeper cluster as well.

8.5 Autonomous scalability

Section 8.2 presented the logic behind automatically assigning users to servers and the recovery of this data. In order to ease the scalability of the platform, it is important that an automated process is able to measure the current load on the cluster to scale up or down in terms of servers.

By design, each of the clusters in the proposed solution are able to recover their data through other nodes in the service cluster. By booting a new server, it will automatically join the cluster through ZooKeeper, allowing the leader node to add the new server to the cluster. This ensuring easy scalability, adding more nodes is as simple as booting clones of original machines such that they join the cluster automatically.

Puppet and Chef are two autonomous scalability agents that are able to measure the load on the different servers. If load increases, agents like these will allow a clone to be booted up when the load increases on the service without further administrator interventions required.

8.6 MAPI servers

The MAPI servers are responsible for all client connections; therefore these servers are responsible for a consistent view of a mailbox even though the same user uses multiple clients. This Section will cover how such a consistent view on the mailbox is achieved, thereby it answers the sixth sub question (Q6) how data consistency could be realized if data is redundantly stored and users access the stores using different clients.

The groupware server layer is depicted as cluster C in Figure 29. This layer is responsible for access management, making sure only authorized users can access the emails, as well as keeping track of the state of client devices throughout their session. These servers process composed emails, provide full access to editing emails, and access to move, copy, delete, and sort operations on the emails inside folders.

The state that these servers have of a session is completely recoverable. In case a server fails, the clients will be forced to reconnect to another server through the single point of entry layer. Once the client connected to another server, it will inform this server about the state that the device has, such that this server will be able to continue where the connection was lost. By querying the Riak database, all email and folder data can be restored.

The MAPI servers are able, through ZooKeeper, to track where the session of a user resides in the cluster, allowing them to make quick decisions where data should be redirected. This

segmentation of users across the MAPI servers is determined through the ZooKeeper cluster. As soon as a server becomes responsible for a user, it will lock the user file in the ZooKeeper cluster. ZooKeeper guarantees that only one server is able to lock a user at a time. If the same user opens another session, the file will be locked, allowing the single point of entry layer to redirect the connection to that specific server immediately. The lock is ephemeral, as soon as the responsible node faces timeouts, the ZooKeeper cluster will release the lock and allow other servers to become responsible for that user.

All MAPI servers have full write access to the email bodies, headers, and attachments. However, sorting a folder, or moving, copying, or deleting an email is processed asynchronously on one of the Write Agents. This later tasks are asynchronous, as the write agents are responsible for managing the folders. The write agents hold the latest copy of the folder and process all requests in batches to optimize the write process, this hides the processing time required from the user. The request is added to the queue of that user through the ZooKeeper cluster, on which the responsible server will be notified to process the request. If the request is fulfilled, the Write Agent will delete the request data on ZooKeeper, which informs a MAPI server immediately if they monitor this task. Since all tasks are written to the ZooKeeper cluster, failure of a Write Agent or MAPI server will not prevent the task from being executed, as long as ZooKeeper confirmed it is written of course.

When a user sends a message, the headers and body of the email are written to the Riak database. A batch task is written to the ZooKeeper directory of that user. When the write agent found a timeslot to process the send request, it will send the message using the SMTP protocol and upon success move the message to the sent messages folder. The MAPI server will be able to process this request very quickly, leaving the time consuming task of delivering the message to the recipients through SMTP to the write agents. The email is safe as soon as the data is written to Riak and a task is created on ZooKeeper, a node failure could only delay the delivery of the message.

In case the user uploads an attachment to send in an email, it is first stored in Riak before it is written to an external storage service. In return, the user will get a unique attachment id that is required to refer to the attachment and link it to the email that the user is composing. This allows the user to upload attachments while writing, and ensures that the data is save as long as the client did not lose the id on the client side. Since the MAPI server responsible for storing the attachment temporarily, failure of the server node will not result in lost attachment data. The write agent will retrieve the attachment from the Riak database when it is delivering the email to the recipients.

8.7 Write agents

The write agents are responsible for processing incoming SMTP emails and all tasks that require exclusive write access to a folder index. These servers hold no data, only a cache of the folder index and email headers in that folder to speed up sorting and other tasks. All of this data is reconstructed quickly by querying the Riak database and the ZooKeeper cluster.

The users are segmented in virtual nodes, of which each node holds a few. The virtual nodes are divided across the nodes through the virtual node segmentation algorithm, as discussed in Section 8.2. This algorithm is executed on the leader of the write agent cluster. The leader is elected at the start of the cluster, the first node that started becomes the leader. ZooKeeper guarantees that only one leader is elected. All the other nodes that join the cluster will monitor their preceding node, such that a chain of servers monitors each other. If the leader fails, the second node will become the new leader to take over its tasks instantly. A failure of any non-leader write node will be discovered by the leader, in case a node fails, the leader will start the virtual node segmentation algorithm to divide the users across the current set of nodes. The same algorithm is executed in case a new node is added to the cluster.

When the cluster is booting up, the leader should parse and clean up all the previous configurations. As soon as it is finished processing these, the leader will notify the other nodes that are ready by writing a leader ready file to ZooKeeper. The other nodes will react to this event by joining the cluster immediately.

When a new message is delivered through SMTP, the write agent will split the message into separate header, body and attachment objects. The header and body are stored in the Riak cluster, whereas the attachments are written to an external storage provider. The header will hold a reference to the attachments, and both the header and the body of the email will be stored in their designated buckets in Riak using the same key. This key is written as a batch task to the ZooKeeper directory of that user. The write agent node that is responsible for that user, this might be the same node, will react on the new batch task by processing it. This division ensures that only one node is able to write to the folders of a user. Since this design operates fully asynchronously, threads are not put in a locked state waiting for another write process to finish. This design allows the write agent to sort the incoming requests based on the folders they share. If multiple new emails should be delivered to the same folder, this is a lot quicker to do in one batch, than one by one. Only once the email is written to the Riak cluster, the Write Agent will confirm the delivery of the message. This ensures that, in case the Write Agent or a temporary Riak failure occurred, the email will not be lost in the delivery process.

Sending a message starts just like receiving one, except that the message is written to the outbox of the user instead of their inbox. Sending a message uses the SMTP protocol for

external recipients. For all internal recipients, the write agent will clone the message and process it like it has just been received through the SMTP server. In case a failure occurs and the write agent was not able to accept the message, the outbox still holds a copy, the process starts over until another write agent is found to process the mail delivery. If a failure occurs while processing an external SMTP email delivery, emails that were delivered to that server were either not confirmed to have been received successfully, or are safely stored in Riak with a reference in ZooKeeper.

8.8 Blob storage

The blob storage layer is responsible for keeping track of all the headers and bodies that are stored by the groupware service. This entails all the emails, calendar items, folders, address book contacts, and other data that is linked to a user store inside the groupware platform. It should ensure this data is available even though nodes fail, and it should offer a high throughput in terms of reading and writing data.

Based on the results of Section 7, the Riak database has been selected as the best candidate for storage of large amounts of groupware data. Proposed is to store all user data including email, and folder indices with a replication factor of three.

Access to the storage layer is provided through a load balancer. The load balancer eases the configuration of the servers to connect to a random storage server. All the servers connect to the Riak cluster through the load balancer, this eases the set-up of the other servers, as the load balancer will automatically get updated if storage servers fail or are added in the process.

8.9 Attachment storage

As discussed in Section 5.3, attachment data behaves very different than the headers and bodies of emails. The attachments are rarely updated, and if they are downloaded it happens less frequently than reading the message it self.

As the attachment data entails the largest amount of data, to be precise about 81.5 per cent, this data is best moved to a storage service provider. Especially since the attachment data has typical storage access characteristics, storage providers are able to serve this type of data at lower costs due to the scale at which they store data. Section 6.3, showed that Microsoft Azure is the best performing cloud storage provider at this moment.

8.10 Conclusion

Each of the clusters depicted in Figure 29 is designed to scale up or down autonomously. Using ZooKeeper, load measurements, and the proposed user segmentation algorithm, the cluster will be able to recover from failing nodes automatically. The blob storage is automatically balanced and able to deal with faults automatically, with Riak as its back-end.

All of this is designed to be hosted without increasing the complexity for the end users. Even though a dynamic cluster of servers is hosting the groupware service, the ZooKeeper state data together with the MAPI load balancers are able to provide access through a single point of entry. The next section will validate the proposed design using a prototype that gets benchmarked using the load simulation as described in Section 4.5.

9 Validation

In order to validate the proposed solution presented in Section 8, a prototype is developed in Ruby. Figure 31 shows the implementation of the prototype in terms of clusters and their connectivity.

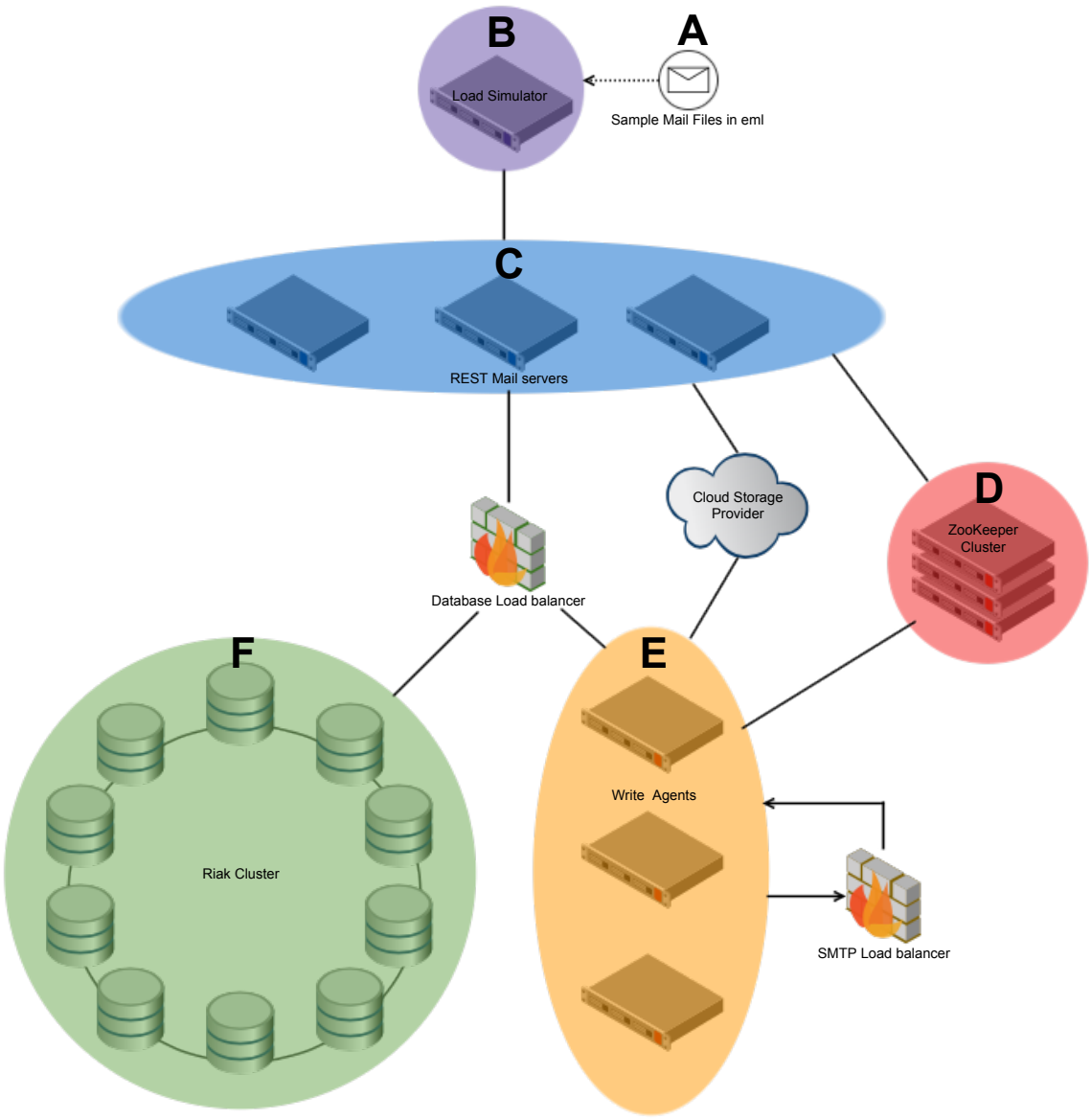

Figure 31: Validation prototype developed to validate the failure resilience and scalability while performing load simulations.

The load simulator is developed as part of this thesis to simulate load on both the previous

platform and the scalable prototype, this simulator is depicted as cluster B. The load simulator uses the MMB3 standard to perform a load simulation on the service. As discussed in Section 4.5, tasks that are simulated include session management, composing emails, replying to emails, reading messages, sorting mails, deleting mails, moving, etc. The load simulator has been configured to measure the latency of each operation. Furthermore, to validate that no emails would be lost in the process, the messages received in the inbox were validated with the list of messages that were supposed to have arrived. Additionally, the client would throw an exception in case it noticed a disruption of the service, in order to check whether the service is available to the end users throughout the tests.

The Mail servers, depicted as cluster C, are developed in Ruby on Rails. These servers perform all the tasks that the MAPI client would do in the load simulation. The mail servers are connected to both ZooKeeper and the Riak database. Through ZooKeeper it registers its availability in the cluster. The Mail servers use the Riak database to retrieve all the user data including email bodies, headers, folder indices, and the user profile itself. All of this data is recoverable; the mail server holds no unrecoverable state information. In order to validate the single point of entry of the service, all of the operations are executed round robin on one of the available mail servers. More specifically, it does not matter to which of the mail servers the user got connected to, as the mail service manages the segmentation internally.

The delivery agents are developed in plain Ruby, as these perform the background tasks in the cluster. The delivery agents are fully SMTP compliant servers, in order to test the scalability of this part in the architecture, all of the emails that would otherwise be delivered internally, are now delivered over SMTP to a random other delivery agent. To determine which delivery agent to connect to, it queries the ZooKeeper cluster on the available SMTP servers at that moment. Aside of this modification, the delivery agent is implemented with all features and algorithms as discussed in Section 8.7.

9.1 Test cluster

The test cluster comprised of 13 64-bit Amazon EC2 instances, of which six m1.xlarge servers were used, three for the Riak nodes, shown as cluster F, and three for the ZooKeeper cluster, shown as cluster D. These nodes have 15 GiB of memory, four virtual cores of two compute units each. The three Riak nodes used the same disk set-up as during the key-value benchmarks in Section 7. Since slow I/O in the ZooKeeper cluster would limit the total throughput of the cluster, these instances were configured with a four disk RAID-0 file system. Failure of one of the disks would make the node fail, therefore a redundant set-up in this cluster is even more important.

The mail servers and the load simulator, depicted as clusters C and B respectively, were running on four c1.xlarge instances, these have 7 GiB of memory, eight virtual cores, each

having 2.5 compute units. The delivery agents were using the m3.2xlarge instances, each instance has 30 GiB of memory, and 8 virtual cores with each 3.25 compute units each.

9.2 Workload measurements

The cluster was initialized with a thousand users, each having 100 messages in their inbox. Ten messages that each has very different characteristics were randomly used to fill the inbox. Each message got assigned a random sender that exists in the cluster, as well as a random set of other recipients in this cluster. While performing the workload, the load simulator uses the same logic to compose new messages. It retrieves messages from the inbox, replying to some of them. Thereby it influences the inbox of these other users as well, such that a real mail conversation is simulated.

The load simulator follows the MMB3 standard [43], where the workload comprises that of normal users on a typical eight-hour day at the office. The simulated workload performed these tasks in 15 minutes, thereby increasing the load on the cluster by a factor of 32. Therefore, the simulated workload actually equals that of 32 thousand users for 15 minutes. This has been decided to take away most of the initialization time required to bootstrap the load simulation.

The tests have been performed four times, the average of these results is presented in Figure 32. The top graph shows the average utilization per cluster group, whereas the bottom graph shows the latency of the services provided. The delivery agent utilization graph shows a small increase in the workload when a node failed. The Riak servers show an increase of 20 going up to 40 per cent when another node failed. With the web servers and ZooKeeper the load did not increase significantly. Important to notice is the relation with the latency shown in the lower graph. The latency experience by the user is depicted as cluster C. While failures occur, this latency did not increase. The latency in mail delivery, however, did increase significantly. This latency is hidden from the user, as this is executed in the background by one of the delivery agents.

9.3 Limitations

The prototype focused on the aspects that need to be validated. The user authentication is not implemented in the prototype as this is outside the scope of this research. The web client and related web load balancer are not implemented as these operate like any other client would, this service scales independently.

In order to simulate the clients, a RESTful server has been implemented as the MAPI mail server. The MAPI clients, however, expect the server to operate stateful, whereas RESTful

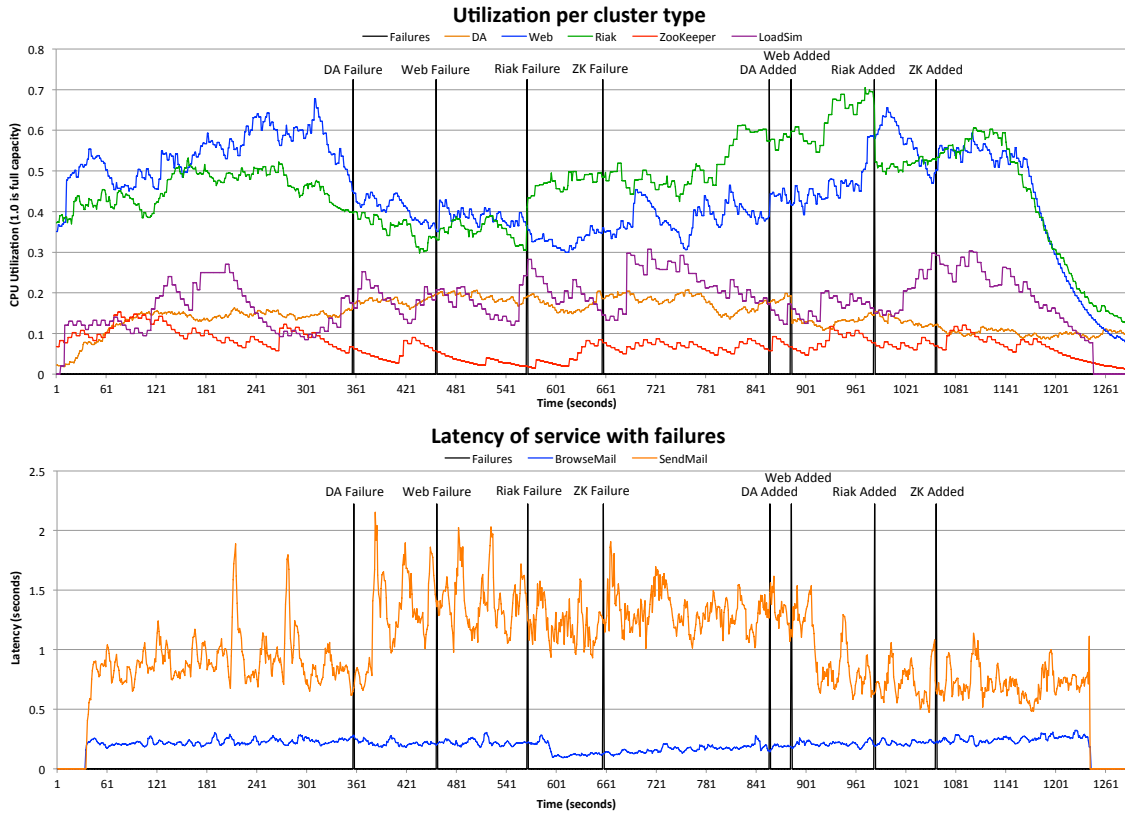

Figure 32: Validation measurements of the utilization per cluster group (top) and average latency of tasks performed (bottom).

servers do not hold state by design. Implementing a fully MAPI compatible server has been considered, but after four weeks of developing a scalable prototype in the Zarafa server codebase, I made the decision to move to another programming language that I'm more familiar with to validate the architecture. In terms of scalability, allowing MAPI connections does not form a bottleneck, as the segmentation of users across the mail servers could be implemented as it has been at the delivery agents. Strictly speaking, this aspect of the architecture is validated by the prototype. The statefulness of the server and the recovery process of this state is covered by the delivery agents as well, as these require an index to be rebuilt when another recovers the state where another delivery agent failed.

Furthermore, due to resource limitations, the scalability of the platform has been tested up to a thousand users, simulating a workload of 32 thousand users. To perform these tests, a total of 12 high-end servers have been used, all orchestrated by a single load simulation server with enough resources to simulate this load. Tests have been performed to show that it scales up to 12 servers; nevertheless, in order to validate whether it would operate with millions of users, a larger test cluster would be required.

9.4 Conclusion

The prototype showed that the clusters that were proposed in Section 8.1, are able to scale and recover from failures without noticeable disruptions to its end users. The prototype showed that it was capable of dealing with the simulated workload of 32 thousand users.

The load simulation showed a clear separation of performance, where the user would not notice any hiccups if one of the back-end servers fails. The results, depicted in Figure 32, clearly showed this effect.

When the cluster continued in a recovery state, the delivery of emails took a bit longer than they usually would. However, even though several nodes had failed in the cluster, no message was lost in the process. When failures occurred, the cluster took more time to deliver emails, this time is within the requirements as defined in Section 3.

When the mail server failed to which the user was directly connected, the connection was reset, forcing the user to reconnect. After reconnecting the user got assigned to another server, processing all requests like nothing happened. This has been validated by throwing exceptions in the load simulator if the expected outcome of operations was not matched, throughout the tests no exceptions were thrown.

10 Conclusions and future work

To conclude, the main findings that cover the main research question of this thesis are presented in the first upcoming section. Followed by areas of future work that have not been covered in this research, but might be interesting to look at.

10.1 Conclusions

A self-managing, scalable, elastic, and fault tolerant architecture is important to provide a cost effective groupware service. The characteristics of groupware data require a different architectural approach, in order to be scalable without requiring the involvement of administrators.

On the basis of the MAPI protocol, this thesis presented an architecture to serve groupware data on a cluster that meets these requirements. The growth in scalability is important as this allows a large group of users to utilize the same cluster, allowing their different usage patterns to even out the load on the cluster as a whole. This allows the operational costs to go down, as its resources are utilized more efficiently.

A prototype of the architecture has been developed, on which experiments were performed. The results of these experiments showed that the architecture scaled well, and continued to be fully operational while failures at different levels occurred. The architecture operates through a single point of entry, through which users will be able to access their groupware data without having to worry about the whereabouts of their data. This is important as servers will malfunction or become unreachable, these disruptions should, however, not disturb the service provided.

10.2 Discussion

The aim of this thesis has been to research and prototype a scalable and fault-tolerant solution in the groupware domain that provides its services as a cluster that behaves like one single robust service provider. The characteristics of groupware data, especially in combination with the stateful MAPI protocol, form a strict requirement list that conventional scalability and fault-tolerant architectures do not meet.

The service requirements were determined for the ideal groupware service provider. These requirements include the ability to access the service from anywhere, at anytime, where the user has a consistent view on the data even though multiple clients are used. Furthermore, the back-end should write fast, and read even faster. It should load balance automatically, with new servers nodes being added autonomously. The management of the cluster should be

as easy as adding more hardware, where the cluster will manage itself in the event of failures or increased load.

In the current design of the Zarafa server, a lot of effort was put into optimizing single server performance. Through the set-up of distributed access groups, a cluster could theoretically scale up to 18 servers, this does require a lot of overhead to administer and segment the users across the servers in the cluster. To show the limitation in scalability, a load simulator has been developed. The load simulations confirmed the communication overhead in case multiple servers are made responsible for a group of users.

Analysis of the data in the current design showed that the level of detail at which properties are stored form a major bottleneck in terms of scalability. By simplifying the data structure into three data blocks, being the body, header, and attachments of the messages, the problem of writing and locking dozens of fields is simplified to a minimum of two write operations per e-mail. It was shown that the header is updated more frequently than the body, by keeping them apart, each update operation of the header limits the size of data that needs to be written to a minimum.

Based on the literature survey that has been conducted, several key-value databases were compared with a set of requirements. The comparison left MySQL Cluster, Cassandra, Riak, Voldemort, and HBase as the candidates for the storage back-end of the service architecture. Related benchmarks on these key-value databases were out-dated, used a small dataset, or used a small row size to test with. Therefore, several rounds of benchmarks with the YCSB benchmark tool have been performed to determine the throughput and failure-tolerance of the key-value databases. In the benchmark tests, the databases were tested with intensive workloads using a dataset that has 20 times more data than the servers have as memory, to test the I/O subsystem of the database. The two databases that met the requirements best were Voldemort and Riak. However, Riak has been selected as the key-value database back-end, as the documentation on Voldemort is very limited and no professional support options are available. The benchmark results form a valuable resource for other research projects that require insights in the I/O subsystem of the different key-value stores, or research projects that use larger values just like the groupware domain requires to store email and other user data.

The proposed service solution for Zarafa separated the servers that users interact with, from the back-end that is responsible for delivering emails, updating folders, etc. Each of the responsible server groups are designed to resolve issues with failing nodes immediately, such that a failure might result in a short reconnect, but no noticeable hick-ups or service downtime is preventing the user from accessing their account data. The architecture automatically segments the users across the available servers, where failures automatically move these users to their new assigned servers. The architecture is designed to operate as one single robust

service provider in the eyes of its users. The MAPI requirements are met by assigning users to specific MAPI servers that are able to replicate the data from the storage cluster if necessary. In the back-end, all mail is delivered through a separate cluster of nodes called the Delivery Agents. These nodes process all incoming SMTP traffic, by ensuring that mails are successfully written to the storage back-end on three nodes or more, the agent guarantees that emails are delivered and failure of any delivery agent could not result in the loss of email messages. The research showed how Riak and storage service providers could be used to keep all data safe and accessible at all times. Adding nodes to the cluster is as simple as cloning one of the nodes, the process of scaling up or down with an automated provisioning tool is really straightforward.

The prototype validated the design of the proposed solution for Zarafa's use case, where email is processed by a scalable and fault-tolerant cluster of servers that together behave like one single robust service provider. With this validation, the main research question of this thesis has been answered and validated. The validation was executed using a 32 thousand user load simulation on a cluster of 12 servers. These tests showed a clear separation of performance, where failing back-end nodes were not noticeable in the user interaction with the cluster.

10.3 Future work

The prototype has been validated using a 12-node cluster, in order to show more insights on the scalability characteristics of the cluster this should be tested using a cluster double that size. These tests were not possible due to resource constraints. The limitation of this resource constraint might implicate that the platform appears scalable, but really might be a few steps away from the scalability bottleneck.

The same resource constraint limited the key-value database tests to a maximum cluster size of six nodes and one load simulator. Since the database has been tested with a three-node and six-node cluster, the scalability of the database is measured at the start of its scalability capabilities. To test the true scalability performance of the databases, benchmark results on a cluster that is several magnitudes larger would allow us to determine the real scalability of the platform.

An interesting aspect that has been excluded from the scope of this research is the ability to search through the user data. Search operations might require a different storage strategy for the bodies of the emails. Furthermore, this research focussed on the full scope of email data, being fresh and older data, the topic of archiving old email data was briefly touched upon, but could allow further advancements in terms of performance and scalability.

The security issues related to storing mail data in one big cluster is left open for future work, email data could be a valuable resource to some advertising parties, but should be kept

private at all times.

The financial achievability of the service discussed in this thesis is not covered by this research. Future work might look into the different levels of outsourcing to set-up a cluster as the one presented.

With the key-value benchmarks, the failure tolerance tests of the MySQL Cluster have been aborted after two weeks of attempts, Oracle guarantees 99.999 per cent availability of the cluster, however, due to time constraints and limited documentation on this subject, the cause of the crash during the availability test has not been analysed in depth, as discussed in Section 7.3.

How user session data should be tracked in such a scalable service has been considered as available up until now, future research might look into the scalability of the user session data to keep all clusters in sync on the available session data.

Another interesting topic to research is whether automatic resolvers could be implemented to allow multiple nodes to write to a single key. For example, writing multiple emails concurrently to the same folder index, if the default conflict resolver of Riak is used these actions will result in the loss of header references in the folder. However, if the database would be able to determine the patch set of the change that is applied by each of the operations, the two might be merged and applied as one.

11 References

[1] 10gen. Mongodb overview. *Product support*, December 2012. URL: http://www.10gen.com/products/mongodb.

[2] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, 2009. doi:10.1109/CLUSTR.2009.5289188.

[3] Amazon. Ebs devices. *Amazon Web Services*, November 2012. URL: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html.

[4] Amazon. The first trillion objects. *Amazon S3*, June 2012. URL: http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillion-objects.html.

[5] Amazon. Increasing ebs performance. *Amazon Web Services*, November 2012. URL: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSPerformance.html.

[6] Apache Software Foundation. Cassandra. *Apache Projects*, January 2013. URL: http://projects.apache.org/projects/cassandra.html.

[7] Apache Software Foundation. Hbase website. *Apache Projects*, January 2013. URL: http://hbase.apache.org.

[8] Jason Baker, Chris Bond, James C. Corbett, Jj Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. *5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, pages 223–234, January 2011.

[9] O. Bälter. Strategies for organizing email messages. *Proc. HCI*, pages 21–38, 1997.

[10] Basho Technologies, Inc. Concepts. *Riak documentation*, December 2012. URL: http://docs.basho.com/riak/latest/references/appendices/concepts/.

[11] Basho Technologies, Inc. Riak vs dynamo. *Riak documentation*, December 2012. URL: http://docs.basho.com/riak/latest/references/dynamo/.

[12] Basho Technologies, Inc. N replication factor. *Riak documentation*, January 2013. URL: http://docs.basho.com/riak/latest/references/appendices/concepts/Replication/#So-what-does-N-3-really-mean-.

[13] J.B. Bhaskar. Email and collaboration asa service on public cloud - an experience. *SBI Life*, September 2012. URL: http://www.slideshare.net/connect2mithi/sbi-life-email-and-collaboration-as-a-service-on-public-cloud-an-experience.

[14] Gordon Blair, Fabio Kon, Walfredo Cirne, Dejan Milojicic, Raghu Ramakrishnan, Dan Reed, and Dilma Silva. Perspectives on cloud computing: interviews with five leading scientists from the cloud community. *Journal of Internet Services and Applications*, 2:3–9, 2011. 10.1007/s13174-011-0023-1. URL: http://dx.doi.org/10.1007/s13174-011-0023-1.

[15] Jason Bloomberg. Email as a service not as easy as it sounds. *CIO - Applications*, October 2012. URL: http://www.cio.com/article/717884/Email_As_a_Service_Not_As_Easy_As_It_Sounds.

[16] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1298455.1298487.

[17] Juan Cáceres, LuisM. Vaquero, Luis Rodero-Merino, Álvaro Polo, and JuanJ. Hierro. Service scalability over the cloud. In Borko Furht and Armando Escalante, editors, *Handbook of Cloud Computing*, pages 357–377. Springer US, 2010. URL: http://dx.doi.org/10.1007/978-1-4419-6524-0_15, doi:10.1007/978-1-4419-6524-0_15.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. doi:10.1145/1365815.1365816.

[19] Yan Chen, Toni Farley, and Nong Ye. Qos requirements of network applications on the internet. *Information, Knowledge, Systems Management*, 4(1):55–76, 01 2004. URL: http://iospress.metapress.com/content/YUBMB4NV3YU3U6UX.

[20] K. Chodorow. *Scaling MongoDB*. O'Reilly Media, 2011.

[21] Cloudera. Hadoop support, including hbase, hdfs, etc. *Cloudera, Inc.*, February 2013. URL: http://www.cloudera.com/content/cloudera/en/products/cloudera-support.html.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. doi:10.1145/1807128.1807152.

[23] Datastax. Deploying cassandra across multiple data centers. *Cassandra Developer Center*, March 2011. URL: http://www.datastax.com/dev/blog/deploying-cassandra-across-multiple-data-centers.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007. URL: http://doi.acm.org/10.1145/1323293.1294281, doi:10.1145/1323293.1294281.

[25] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf. Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter. Rapport de recherche RR-7706, INRIA, April 2012. URL: http://hal.inria.fr/inria-00614597.

[26] RobertoR. Expósito, GuillermoL. Taboada, Sabela Ramos, Jorge González-Domínguez, Juan Touriño, and Ramón Doallo. Analysis of i/o performance on an amazon ec2 cluster compute and high i/o platform. *Journal of Grid Computing*, pages 1–19, 2013. URL: http://dx.doi.org/10.1007/s10723-013-9250-y, doi:10.1007/s10723-013-9250-y.

[27] Alex Feinberg. Voldemort. *Voldemort GitHub Wiki*, July 2011. URL: https://github.com/voldemort/voldemort/wiki/Client-side-failure-detector-implementations.

[28] Danyel Fisher, A. J. Brush, Eric Gleave, and Marc A. Smith. Revisiting whittaker & sidner's "email overload" ten years later. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 309–312, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1180875.1180922, doi:10.1145/1180875.1180922.

[29] Forrester, Inc. Tier your workforce to save money with cloud-based corporate email. *Forrester Information & Knowledge Management Professionals*, August 2009. URL: http://download.microsoft.com/download/5/5/C/55C69DCB-6D2B-4433-9D95-D6Fb5BD9FE86/Forrester%20Tier%20Your%20Workforce%20to%20Save%20Money%20with%20Cloud-Based%20Corporate%20Email.pdf.

[30] Forrester, Inc. How to choose the right email solution for your business. *White Papers*, May 2011. URL: http://g.microsoftonline.com/0BXPS00EN/1128.

[31] Lei Gao. Voldemort rebalancing. *Voldemort GitHub Wiki*, September 2012. URL: https://github.com/voldemort/voldemort/wiki/Voldemort-Rebalancing.

[32] Steven D Gribble, Matt Welsh, Rob von Behren, Eric A Brewer, David Culler, N Borisov, S Czerwinski, R Gummadi, J Hill, A Joseph, R.H Katz, Z.M Mao, S Ross, and B Zhao. The ninja architecture for robust internet-scale systems and services. *Computer Networks*, 35(4):473 – 497, 2001. <ce:title>Pervasive Computing</ce:title>. URL: http://www.sciencedirect.com/science/article/pii/S1389128600001791, doi:10.1016/S1389-1286(00)00179-1.

[33] Rolf Harms and Michael Yamartino. The economics of the cloud. *Microsoft Corporation*, page 22, November 2010.

[34] Quoc Hoang. Email statistics - 2012-2016. *Radicati Group*, April 2012.

[35] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1855840.1855851`.

[36] Thomas Karagiannis and Milan Vojnovic. Email information flow in large-scale enterprises. *Microsoft Technical Report*, pages 1–15, May 2008. URL: `ftp://ftp.research.microsoft.com/pub/TR/TR-2008-76.pdf`.

[37] Simon Bernardus Kok. Online scalable and fail-safe data-technologies. *Literature Study*, December 2012.

[38] Avinash Lakshman. Cassandra - a structured storage system on a p2p network. *Facebook Developers*, August 2008. URL: `http://www.facebook.com/note.php?note_id=24413138919`.

[39] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/o performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 40:1–40:12, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1654059.1654100`, `doi:10.1145/1654059.1654100`.

[40] Joe Lennon. Exploring couchdb. *IBM developerWorks*, March 2009. URL: `http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html`.

[41] J. Logan and P. Dickens. Interval based i/o: A new approach to providing high performance parallel i/o. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on*, pages 289–297, 2011. `doi:10.1109/ICPPW.2011.45`.

[42] Min Luo and Haruo Yokota. Comparing hadoop and fat-btree based access method for small file i/o applications. In Lei Chen, Changjie Tang, Jun Yang, and Yunjun Gao, editors, *Web-Age Information Management*, volume 6184 of *Lecture Notes in Computer Science*, pages 182–193. Springer Berlin Heidelberg, 2010. URL: `http://dx.doi.org/10.1007/978-3-642-14246-8_20`, `doi:10.1007/978-3-642-14246-8_20`.

[43] Microsoft, Inc. Mapi messaging benchmark 3. *Microsoft Exchange Server*, December 2007. URL: `http://technet.microsoft.com/en-us/library/cc164328(v=EXCHG.65).aspx` [cited September 17th, 2012].

[44] Microsoft, Inc. Microsoft security intelligence report. *Microsoft Corporation*, June 2012. URL: http://www.microsoft.com/security/sir/.

[45] Microsoft, Inc. Database availabilty group. *Exchange 2013 Help*, January 2013. URL: http://technet.microsoft.com/en-us/library/dd979799%28v=exchg.150%29.aspx.

[46] Microsoft, Inc. Windows azure's flat network storage and 2012 scalability targets. *MSDN Blogs: Windows Azure*, 2012, November. URL: http://blogs.msdn.com/b/windowsazure/archive/2012/11/02/windows-azure-s-flat-network-storage-and-2012-scalability-targets.aspx.

[47] Robert L. Mitchell. Corporate e-mail in the cloud: Google vs. microsoft. *Computer World*, April 2010. URL: https://www.computerworld.com/s/article/9176036.

[48] Fred Moore. Information lifecycle management. *Horison Information Strategies*, October 2003.

[49] Nasuni. The state of cloud storage - a benchmark comparison of performance, availability and scalability. *Industry Report*, pages 1–17, February 2013.

[50] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann Pub., 1994.

[51] Oracle. Error documentation. *MySQL Cluster Documentation*, 2013. URL: http://dev.mysql.com/doc/refman/5.5/en//table-size-limit.html.

[52] Oracle. Mysql 5.5 reference manual. *MySQL Documentation*, 2013. URL: http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html.

[53] Oracle. Mysql cluster disk database. *MySQL Cluster Documentation*, 2013. URL: http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-disk-data-storage-requirements.html.

[54] Sundar Pichai. Chrome & apps. *Google I/O: Your web, everywhere*, June 2012. URL: http://googleblog.blogspot.nl/2012/06/chrome-apps-google-io-your-web.html [cited February 22nd, 2013].

[55] Project Voldemort. Voldemort design. *Project website*, November 2012. URL: http://www.project-voldemort.com/voldemort/.

[56] Radicati Group. Email statistics - 2009-2013. *White Papers*, May 2009.

[57] Ryan Rawson. Hbase. *StumbleUpon NoSQL Meetup*, June 2009. URL: http://www.docstoc.com/docs/9912857/HBase-nosql-presentation.

[58] Ryan Rawson and Jonathan Gray. Hbase. *Hadoop World NYC*, september 2009. URL: http://www.docstoc.com/docs/12426213/HBase-at-Hadoop-World-NYC.

[59] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. *SIGOPS Oper. Syst. Rev.*, 33(5):1–15, December 1999. URL: http://doi.acm.org/10.1145/319344.319152, doi:10.1145/319344.319152.

[60] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability, and performance in porcupine: a highly scalable, cluster-based mail service. *ACM Trans. Comput. Syst.*, 18(3):298–, August 2000. URL: http://doi.acm.org/10.1145/354871.354875, doi:10.1145/354871.354875.

[61] M. Tlili, R. Akbarinia, E. Pacitti, and P. Valduriez. Scalable p2p reconciliation infrastructure for collaborative text editing. In *Advances in Databases Knowledge and Data Applications (DBKDA), 2010 Second International Conference on*, pages 155–164, 2010. doi:10.1109/DBKDA.2010.21.

[62] Kirk True. Ec2 testing infrastructure. *Voldemort GitHub Wiki*, August 2010. URL: https://github.com/voldemort/voldemort/wiki/EC2-Testing-Infrastructure.

[63] Jason Venner. *Pro Hadoop: Build scalable distributed applications in the cloud.* Apress, 2009.

[64] Miles Ward. Mongodb on aws. *Amazon Web Services*, January 2012. URL: http://info.10gen.com/rs/10gen/images/AWS_NoSQL_MongoDB.pdf.

[65] Stege Whittaker and Candace Sidner. Email overload: Exploring personal information management of email. *Proc. CHI*, pages 276–283, 1996.

Appendices

A Key-value experimental setup

For the key-value benchmark experiments, nine Amazon EC2 m1.xlarge instances were used. To continue on the setup description introduced in Section 7.2, each machine has 15 GiB of memory, 8 64-bit EC2 Compute units, of which there are 4 virtual cores, each having 2 compute units. The machines used a provisioned EBS drive as their target storage, using the EBS-Optimized interface of 1000 Mbps. These machines have high I/O performance, making them a suitable candidate for running database services in the cloud [3].

Although these machines are first generation EC2 instances, the m1.xlarge EC2 instance supports EBS-optimized storage, ensuring consistency and provisioned performance. The second generation EC2 instances are not capable of using EBS-optimized storage, but provide additional CPU and memory performance. Since database services are more I/O intensive, than CPU, the EBS-optimized storage outweighs the additional CPU and memory performance. Besides, as this research is initiated to determine a good candidate for email storage, we require the database layer to deal with a lot of data, a lot more data than could possibly fit in memory at a reasonable price level. The perfect database would be able to read quickly from the disk, and flush new updates to disk, without risking consistency or decreasing performance when it flushes.

The chosen operating system used on the storage instances is the Amazon Linux AMI 2012.09, as this is optimized for EC2 instances and fully supports EBS-optimized storage. All of the instances were located in the same availability zone, in this case EU-west-1c was chosen as the availability zone. The same availability zone was selected as the geographical scalability of the groupware service is outside the scope of this research, as discussed in Section 1.3.

Each of the machines uses a small EBS volume of 8 GB hard drive to boot up. The attached EBS-optimized device is able to store 60 GB of data, with a 500 provisioned IOPS. Amazon guarantees that at least 99.9 per cent of the time, the I/O throughput is 10 per cent of the provisioned IOPS or more. According to the Amazon Web Services documentation [5], the drives have a 5 to 50 per cent reduction in IO performance when data is read or written to for the first time. To make sure that this does not influence our tests, the disks have been initialized by reading all data once, using:

dd if=/dev/md0 of=/dev/null

In the tests focus was put on the performance of the IO layer of each of the databases. To simulate a lot of data, the test data should be several times bigger than the available memory, such that the key-value store will actually have to read a lot from disk and flush changes quickly. However, generating 60 GB of data per server, in a cluster of six servers, would take a lot of time just to generate all of that. Especially since the data has to be generated for each database, the decision was made to limit the available memory on the storage machines

such that the server would need to access its disks more often. The available memory was set to 2 GB, of which each of the databases was allowed to take 1 GB for internal processing, leaving 1 GB for the kernel to keep track of the file pages and caches these.

This forces the key-value stores to flush their changes to disk quickly, and limits the use of caching as this would not be available with the retrieval of old mails either.

The benchmark setup used two clusters of servers, the first cluster had three storage servers, while the second cluster used six storage servers. Each of these clusters had their own benchmark server to generate the load. The instance type for these benchmark servers was the c1.medium type, these have more CPU, faster memory, and a very fast network connection. Throughout the benchmarks, the resources of the benchmark servers were monitored to make sure that this would never become the bottleneck for the test itself. With each database, the benchmarks were started at the same time on both clusters, such that general disruptions and load on the network would interfere both tests equally. To equalize this even further, the test size of the larger cluster was also doubled compared to the first cluster, such that the duration in an optimal linearly scalable database would take the same amount of time.