CHALMERS

Migrating and testing distributed cloud based web applications

Master of Science Thesis in Networks and Distributed Systems (MPNET)

Daniel Arenhage
Fabian Lyrfors

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2012

DANIEL R. ARENHAGE, FABIAN M. LYRFORS

Examiner: Elad Michael Schiller

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2012

Acknowledgement

We would like to thank our supervisors Elad Michael Schiller at Chalmers University of Technology along with Fredrik Hilmersson and the Baseline team at Enfo Zystems.

Abstract

With the rapid development of today, developers are getting more dependent on the ability of easy and fast deployment into production. A cloud introduces the possibility of running ones system in a scalable and adaptive environment providing easy access with extensive maintenance possibilities. Extending with the possibility of having highly scalable resources available at any given time, for a reasonable cost, is a strong argument for considering cloud solution for anyone working with new development. However, the act of migrating a locally developed system, or adapting a system for cloud migration might introduce unexpected complications not initially considered. Migration is a term describing the process of adopting an existing application to be run in an unfamiliar environment. It is an important process for ensuring successful deployment in a well-defined manner, and may involve complicated steps adopting the system to the new environment. A cloud often qualifies as such an environment where applications are hosted in a virtualized off-site system.

The contribution of our work is two-fold; studying the effects of migrating and implementing tests aimed for the cloud, and comparing cloud based services to their local counterparts. This has been conducted by developing and migrating an application named *Track and Trace* to a cloud. The results of this report provide valuable considerations as well as practical examples of solutions on problems encountered during development and migration. Migrating an application to a cloud is complicated by the fact that a cloud environment in many aspects has different properties than a local environment. It complicates data consistency, software modularity and testing due to the distributed nature of a cloud. The main difficulty in migrating an application to a cloud is to be aware of its peculiarities and to develop the application avoiding these problems.

Keywords: Cloud, IaaS, Amazon EC2, distributed system, migration, MongoDB, database, message queue, testing, Selenium, Enfo Zystems

Preface

This is a 30p Master Thesis report for the Department of Computer Science and Engineering at Chalmers University of Technology. The proposed work was initiated by the company Enfo Zystems with intentions of developing and testing a new module in an upcoming system. Fredrik Hilmersson is the assigned supervisor at Enfo Zystems and the examiner at the Department of Computer Science and Engineering is Elad Michael Schiller.

Both Daniel Arenhage and Fabian Lyrfors have contributed equally in completing the assigned task by Enfo Zystems as well as gathering information for the written report. Together they have written the report based on experiences gathered while working with developing the module as well as from material from research papers and technical literature.

Software utilized in this project

Amazon Linux AMI 64-bit	http://aws.amazon.com/amazon-linux-ami/
Apache ActiveMQ 5.5.1	http://activemq.apache.org/
Apache Camel 2.9.1	http://camel.apache.org/
Grails 1.3.7	http://grails.org/
IBM WebSphere MQ 7.0.1	http://ibm.com/software/integration/wmq/
Java 1.6.0 JDK	http://www.oracle.com/technetwork/java/
Maven 3.0.3	http://maven.apache.org/
Microsoft Office 2010	http://office.microsoft.com/
Microsoft Windows 7 Enterprise x64	http://www.microsoft.com/
MongoDB 2.0.4	http://www.mongodb.org/
Selenium WebDriver 2.0	http://seleniumhq.org/
Spring Security 3.1.0	http://static.springsource.org/spring-security/site/
SpringSource Tool Suite 2.8.1	http://www.springsource.com/developer/sts/
VMware Player 3.1.4	http://www.vmware.com/products/player/

Table of Contents

1. Introduction

This report describes the observations during development and migration of a system named *Track and Trace*. It is a completely new developed system based on specifications from a prior similar system no longer fulfilling the requirements at Enfo Zystems. The intentions with developing a completely new system were to make it loosely coupled and scalable, developing it with no considerations to the old client/server model.

The technical industry of today is bound by cost driven development and requires time efficient deployment. A lot of the systems today are implemented using a common client/server model which on many levels are hard to decouple and re-implement in a more well-structured and service oriented manner. Since the technical industry is continuously growing, it is faced with the problem of having consistency and compatibility issues when dealing with integration of different computer systems. Companies today does not always share a common architecture for handling data and are therefore dependent on running applications in specific environments or platforms. The act of migrating these systems, making it more service oriented and less dependent on point-to-point integrations is the process of decoupling systems and provide a platform-independent environment ensuring consistency between systems.

Cloud computing is the concept of moving applications and services from a local environment to run in an abstract offsite environment. The cloud environment often provides scalable and virtualized hardware resources in order to remove some of the additional dependencies of having a locally running environment. This act of moving an application will from here on be referenced as migrating a system.

Services and applications in general have traditionally been built with a simple client/server approach. Often the environment on which both the server and client are running in is known, and can be reasoned about, which lets the developer tweak the application and the environment to match. As a result of being built on the traditional client/server model they are often designed as hard-coupled systems. In terms of testing, the traditional way is a straightforward and standardized approach of building unit testing, integration testing and system testing for ensuring the soundness of the developed system. This way of testing has been applicable for the typical client/server module for a long time due to the distinct separation between the two entities. Today we are still dependent on these tests as they still play a key role in determining systems soundness as a whole.

The problem with this traditional way of implementation is that it is not suitable when one wants a highly scalable and loosely coupled solution. A hard-coupled system requires costly hardware upgrades in order to increase its performance to cope with increased requirements, something that is not only complex but also time inefficient. The idea of providing easily adapted and highly scalable applications is something no longer applicable when developing in the commonly known client/server model.

Clouds solves many of the traditional problems by providing an elastic, service oriented and efficient infrastructure supporting scaling. Many of the issues with having a system run as a common client/server comes from the difficulties in adaptability. A cloud introduces the possibility of dealing with ones applications in a service oriented manner which allows for easy reusability, replacement and addition of new functionality. A cloud offers the possibility of quick and effortless management of computing instances which gives the possibility to automatically adapting the system's performance to the current workload. This requires an application running in an elastic cloud to be designed with modularity and distributed computing in mind, something that does not apply for the

client/server model. A more commonly known term for this is horizontal scaling and simply increases performance by addition of a cloud instance.

1.1 Our contributions

Together with the knowledge regarding problems with the commonly known client/server model and system requirements for the Track and Trace system, considerations and adaptations has been made in order to provide a loosely coupled and scalable application suitable for the cloud. The Track and Trace system, which is more elaborate explained in *Case study system* (chapter 4), is built with a cloud deployment in mind and uses the results from this report to make design decisions.

A large portion of the result focuses on a message queue service provided by Amazon, called Simple Queue Service (SQS), and provides suggestions on how it can be used as a replacement for a traditional message queue. It has certain characteristics such as multiple deliveries and out-of-order deliveries which is uncommon in a local message queue and may be unacceptable for an application. Our report contributes to this area by providing simple, yet practical, suggestions for solving these problems. It discusses for example how unique identification numbers can be generated effectively in a distributed setting and how this fact can be used to solve the mentioned problems without using performance expensive distributed data sharing.

The topic on scalability is also covered in later sections in the result chapter. To successfully harness the elastic properties of a cloud an application must be developed as modules. These should preferably not share any data but this is often not possible, which requires the use of distributed or network connected data structures. *Performance and scalability* (section 5.1.3) gives an overview of how an application can be modularized, provides performance measurements using Memcached as a distributed cache, and finally an explanation on how the Track and Trace application was modularized. The application is made into two major modules, the front- and back end, which is only sharing data via a database, and the development uses the performance measurements done in order to justify not to further modularize the back end.

It is shown *Storage and data resilience* (section 5.1.2.3) how the performance in a cloud can differ from a local environment. The system is virtualized in order to allow multiple users to share the same hardware, which has a negative performance impact. Tests are done running the MongoDB database both in a local network and in the Amazon EC2 cloud. The results clearly showed that there are performance differences and also that the performance can vary over time, depending on the current system's load. The virtualization also has the effect of hiding important details which can be important regarding the system's performance. Not knowing for example the storage type (hard drive or solid state drive) makes it hard to reason about the cloud's performance.

The other large aspect of this report is the ability or possibility of providing a sufficient test coverage for the complete case study system. Since the traditional way of testing still is a key part of any development in order to assure system soundness this is not something that is seen as a traditional problem. However, it presents new and interesting problems when traditional test implementations needs to be migrated to the cloud. This is something we experienced during our development and migration of the Track and Trace system. As described in section 5.4.2 (Extending cloud testing), with following subsections, we observed and describe that there are concrete differences in how one should approach testing for a cloud compared to the local environment. Isolating the problem and assumptions regarding performance are no longer reliable or predictable problems which leads to reconsiderations in implementation.

Our approach to provide a sufficient test coverage was to provide a level of automated end-to-end testing. The idea of providing end-to-end testing is the ability of stating that a system is working as intended going from low level testing to high level testing. The way this was achieved during the development of the system described in *Case study system* (chapter 4) , was to separate the traditional low level testing from the suite of testing the user-level. For our case study system we have implemented four levels of test suites namely user test, integration test, system test and user test. The implementations are described in section 5.3 with the intention of providing a high level view of providing end-to-end testing for the case study system.

1.2 Purpose

The purpose of this report is to study the effects and potential problems of migrating a distributed web-application to a cloud. A migration is the act of preparing and moving an application to a new environment, which in this case is a cloud. Our intention is to come to some conclusion in what potential affects that might follow with migration and what considerations that that should be taken into account. We also aim to conclude some of the differences in terms of testing, comparing the local environment to a cloud.

1.3 Problem description

The act of migrating a distributed web application to a cloud introduces new obstacles in how one initially approaches implementation and planning for a new distributed setting.

Considerations in terms of scalability, reliability and availability needs to be addressed in a more generic and well defined manner in order to have a complete and successful migration. Differences in services residing in a cloud compared to local services may also need to be addressed. Another important aspect is testability. How does one actually test an application living inside the cloud, is it possible to migrate current internal test-builds and how do we test the complete cloud application from outside the cloud?

1.4 Related work

In [1], the authors describe their experiences in migrating an open source software to a cloud environment. They very briefly describe decisions and observations providing the reader with not more than an slight idea of how their system have been modified in order to properly migrate to a cloud. With our report we try to extend the initial idea of presenting observations with providing concrete example and potential solutions to observations encountered. This provides the reader with a field of application and an easier way of producing future work. Finally, observing one large aspect separating our work from the work presented in [1]. The authors of [1] have not observed much literature (stated in paragraph 1, section 5) for developing and providing a system aimed for the cloud. They have therefore no solid foundation strengthening their claims regarding pre- and development work for developing an application that is migration friendly.

The author of the paper [2] extensively covers the fundamentals of cloud computing as well as describes some of the key subjects that comes with studying cloud computing. In addition the author has performed some practical testing and measurements supporting some of his claims regarding system performance and capacity. However, the sections covering the aspects of moving systems to the cloud follows the same pattern of mostly describing techniques and literature relevant to the topic, and not so much how this actually can be performed. This is something that we have addressed by stating concrete results for a specific case study. In the future work the author presents

what have not been covered in the report. We took the liberty of trying to answer some the author's proposed future work.

1.5 Scope

This report will focus on techniques and test methods relevant to the needs of Enfo Zystems. Our intentions are to provide a high level view of the migration process for the system described in *Case study system* (chapter 4). The report highlights services relevant to the system existent in the chosen cloud provider, namely Amazon EC2.

1.6 Report outline

The beginning of the report covers the technical background. Here we describe what different techniques and software that have played the major part during development and testing throughout the project. We also describe some of the concepts in defining a cloud based environment.

The chapter following the technical background gives a general description of how we approached the project in terms of method. How we familiarized with the different environments and technologies and how we began our project work.

The third part of this report gives a fairly brief description of our case study. The case study gives an overall description of the system re-implemented, in order to give an understanding in what kind of system we are trying to migrate.

Following the case study we present our results in how different problems were, and potentially could be solved. In the results our main focus is to present a variety of considerations that should be taken into account when developing or choosing one approach over another.

In the discussion part of the report we highlight some different aspects that we found interesting or that turned out to bring some potential problems in future development. We also give a general description of our "9-step-database-migration-plan" based on our own migration procedure.

The last part of our report is our conclusion. Here we briefly state our conclusions based on our experiences throughout the project, and gives proposals for potential future work.

2. Technical background

The technical background describes some of the techniques and software used during development and migration. This chapter aims at providing the reader with a basic technical knowledge of the different aspects that is relevant for better understanding the report.

It explains the concept of integration where systems are decoupled to form a general view of the system. Next on is the core concept of this report, cloud computing, explaining the three cloud models broadly and narrows in on Amazon's EC2 solution. Databases are an important part of the case study and both traditional relational databases and a new term NoSQL databases are explained along with important concepts. The last major part is explaining testing in general, with focus on techniques used to perform user tests in the case study.

2.1 Integration

One of the big challenges today with our continuously growing demands on scalability, availability and accessibility, is integration of new systems. Not only does integration today play a huge role in linking systems though different services, it is also plays a key role in being able to deliver a consistent and reliable environment with high availability for all parties, see [3].

The concept of using integration for increased availability is becoming more and more commonly used within large organizations. Larger organizations no longer need to rely on their own hard-coupled integrations between different services and applications; instead they can now rely on a third party organization that makes use of well-structured architectural standards such as a *Service-oriented architecture* (SOA). This efficiently loosens the hard couplings and re-implements their previous connections as loose couplings through integration. For a more details description see [3].

2.1.1 Integration architecture

The two most commonly used approaches for integrating applications today are point-to-point integrations or making use of a service bus.

When using Point-to-point integration one usually refers to using a hard-coupled integration. One application is connected to some other applications which are dependent on the same protocols and formats in order to successfully communicate, explained in figure 1.

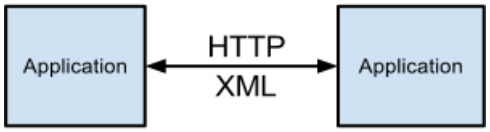

Figure 1. Point-to-point integration

This basically deviates from the idea of having a platform-independent environment and is often what integration companies today are trying to dissolve and implement in a loosely coupled architecture instead. As described in [4], a more sustainable approach is making use of a service bus for integration.

The authors at [5] describes an *Enterprise Service Bus*, or ESB, as an architectural model for implementing a loose coupling for applications, services and clients in a platform, protocol and format independent environment. The ESB acts as a middleware messaging service and takes care of everything from routing to transformation of data. Services and applications connected to the ESB puts their messages on the bus which guarantees that the messages arrive correctly to the intended destination with the correct formatting for the recipient. A small illustration can be seen in figure 2.

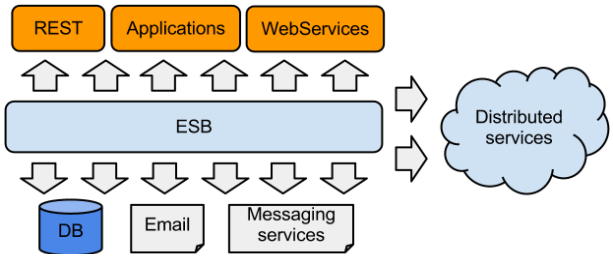

Figure 2. Enterprise Service Bus connected to various applications and services

2.2 Cloud computing

Cloud computing is described in [6] as a general and rather diffuse term to describe services hosted in a remote location, such as an intranet or the Internet. This deviates from the traditional way of handling applications which is installed locally on each client, and services running on the local network.

The term can be divided into three models; *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS) and *Software-as-a-Service* (SaaS), see [6]. These categories can be seen as a layered model where IaaS are the lowest level, then PaaS and on the highest level SaaS. The layers represent the services provided and the level of control a user has over the cloud.

2.2.1 Infrastructure-as-a-Service

In [7] the Infrastructure-as-a-Service (IaaS) model is described as the layer closest to the hardware. The user is responsible for managing the operating system and available resources, but the hardware and infrastructure is handled by the cloud provider. The real underlying structure of the cloud may be hidden as a way to form a generalized view of the system. This provides a well-defined environment which can be fully or partially independent of the underlying hardware and software. It can hide some technical details of the distributed nature of a cloud and give the user a view of a homogeneous system.

IaaS gives the user the largest responsibility but also allows for running applications and services of one's choice.

2.2.2 Platform-as-a-Service

In this model the user is provided a complete platform on which applications and services can be built. The authors of [7] describes that custom applications is restricted by the environment created by the host provider, regarding for example operating system, software libraries, programming language support and support services such as databases and web servers.

PaaS can provide developers and administrators tools to administer and test applications running in the cloud. Administration can involve basic tasks such as starting and stopping existing applications, resource-, user- and permission management, and presentation of statistics and logs. In [6] the authors describe more advanced features that can include support for development, debugging, testing and deployment of new applications. Additional supported services may be offered, such as specialized databases, queue managers and file storage. *Results* (chapter 5) expands on this area.

Compared to IaaS this model is less flexible but relieves the user of the maintenance of the operating system and included services.

2.2.3 Software-as-a-Service

The Software-as-a-Service (SaaS) model is defined in [7] as the topmost layer in the cloud stack. It provides customizable applications to be used without any extra development from the user's side. Such applications are often used to move from traditional local applications to use a similar application in the cloud. It does not require any installation or local storage for the end user as the application often runs in a web browser.

SaaS is described in [8] as the most restricted of the three models, but relieves the user of almost all the maintenance work. The user can seldom install custom applications and are bound to the applications provided by the cloud host.

2.2.4 Amazon EC2

Amazon EC2 is a web-service aimed to provide a highly scalable and configurable cloud computing environment. It provides a highly configurable platform in the IaaS layer which gives the developer the flexibility of choosing his/her own operating system, cloud services and configurations. Initially launching ones EC2 instance, one is able to choose an instance type suitable for whatever requirements the system to be deployed requires. An Amazon EC2 instance comes with a variety of instance compositions such as high-memory or high-CPU in order to provide whatever necessities that might be required. In conjunction with Amazon EC2 one is able to utilize all other functionality supported by EC2 provided by Amazon such as *Elastic Block Storage* (EBS) for sustainable data storage, *Elastic IP Addresses* for providing static IP's, *Virtual Private Cloud* (VPC) for enhanced security and so on. [9] contains the full documentation on Amazon EC2.

2.3 Database

Next we describe conventional relational- and NoSQL databases, and how they differ. The MongoDB NoSQL database which is used in the project is introduced and functions such as sharding and replication is explained.

2.3.1 ACID properties

The ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*) concept stipulates four properties that databases shall conform to in order for a transaction to be processed reliably. Transactions are defined as an isolated read or write to the database. The following paragraphs describe the ACID concept with reference to [10].

The *Atomic* property states that any modification to the database is successfully committed or not committed at all. This follows an "all or nothing" rule and guarantees that no partial modifications can occur which leaves the database in an inconsistent state.

The second property *Consistency*, states that any transaction must leave the database in a consistent state. Each database defines more rigorously how a state is defined. An example is validation of written data so that it conforms to the defined rules.

Isolation states that transactions are isolated from each other and that one transaction should not interfere with another transaction. No transaction that affects the same data can be run concurrently since the outcome would be unpredictable. This can be solved by implementing a lock on the data if a transaction is modifying it.

The last property *Durability* refers to the durability of data after a transaction. It states that a committed transaction shall be stored permanently in the database and not be lost due to application crashes or other unexpected shutdowns.

2.3.2 Relational database

Relational databases are currently the most common database type. As described in [11], it stores data in well-defined tables and are built on the relational model theory. A table's structure is predefined to contain certain data and constraints on what type of data, and which values, can be stored. Tables can also form relationships to indicate that data in one table references data in another table. Relationships are used to normalize tables, meaning that redundant data are removed from one table, and it instead makes use of a relation to the other table. Ideally there is no redundant data which makes updating the relationship trivial as the data is only stored once.

Relational databases use the method of *joining* tables to return data from a table which references to another table. This is used to return one single view of a table when in reality it is divided into several tables.

The common way of communicating with relational databases is by the language *Structured Query Language* (SQL). It is often used in interaction with the database and supports operations such as insert, update, delete and querying of data.

Relational databases often adhere to the concept of ACID as described earlier, in section 2.3.1 (*ACID properties*).

For a more detailed description on relational databases and its concepts see [11].

2.3.3 NoSQL database

The authors of [11] describe NoSQL databases as an umbrella term to describe databases which does not use SQL as query language. Many NoSQL databases also depart from the relational model theory used in relational databases, and may not support all of the ACID properties. A wide variety of NoSQL databases exist, ranging from simple key-value storage to complex databases with dedicated query languages and distributed storage.

2.3.3.1 MongoDB

In [12] MongoDB is defined as a NoSQL database which in some aspects is similar to common relational databases. It supports complex queries using its own query language similar to JSON (introduced in section 2.5) and can use indexes to improve query performance.

The author of [12] highlights that differences between MongoDB and relational databases are in many ways more profound than the similarities. Tables, which in MongoDB terms are called collections, does not have a fixed schema defining the contained data. Instead it stores each document, which is similar to a row in a relational database, in the BSON data format (introduced in section 2.5) to allow documents to contain arbitrary data structures and data types.

There is no support for joining tables, which is in stark contrast to relational databases. The reasons are that the operation of joining tables can be costly for the database if the tables are big and that joins are hard to manage if the collection is distributed over several databases. The data is instead embedded directly into a document, much like in a relational database which has not been normalized. This information and more details on MongoDB schema design are found at [13].

In [14] it is stated that MongoDB is specifically built to scale well and make it easy to add additional storage and computational power. This is accomplished by the concept of *sharding* where data is automatically spread over several databases. To add more storage space an additional database is connected to the shard group and existing and new data is automatically evenly spread to all databases. Sharding also can accelerate certain queries where each individual database can process the query in parallel, see [15].

In favor for speed MongoDB is not fully ACID compliant. In [16] the author explains that MongoDB supports simple atomic operations but lacks support for true transactional queries. Another way to increase speed is to use memory-mapped files to store and access both the database and the related journal file. This gives the end user less opportunity to tweak the amount of RAM memory available to MongoDB as this is handled by the operating system. The journal file is an optional but recommended feature to increase both the speed and durability. Entries written to the database is periodically written to the journal file instead of the main database file. Data is simply appended to the journal file which is fast, and then later written to the real database file.

See *Appendix A* and *Appendix B* for a deeper explanation of MongoDB.

2.4 Testing

This section describes some of the most commonly utilized testing approaches. The intention is to provide some basic knowledge in the different testing methods related to the developed system described in *Case study system* (chapter 4).

2.4.1 Unit testing

Unit testing is the foundation on which larger system depends on for validating that a sub-part of the system is behaving as expected. The author of [17] describes that the basic idea is to separate and isolate a part of the system as much as possible in order to perform an expectancy test. The test itself should be considered as if it was run inside a black box separated from all other logic in order to determine if it actually performs as intended. In [17] a common method of how to do unit testing is to add assertion statements to the tested code. These statements compare the current program state to an expected, valid state. If the results differ the execution can be aborted and an error message logged.

On larger system one can usually compose a series of unit tests, test each part by itself and makes sure that it is working correctly. When each part seems to work correctly, one can start integrating parts and build additional expectancy tests in order to validate that the components works as expected together. In [17] this is described as integration testing.

2.4.2 Integration testing

In [17] and [18] integration testing is described as the concept of weaving together one or more already existing and validated unit tests and testing how the separate components work depending on each other. It is usually conducted in the fashion that each sub-tested part is added iteratively, forming a group of components. These components are then tested as a group and will in the end make up the whole system. When the grouped test has been verified to work, one can draw the conclusion that all other sub-parts also are working as expected. This implies that the system now is ready for system testing.

2.4.3 System testing

Conducting system testing for a service or application is described in [17] as a crucial part in determining that a product lives up to the specified requirements. A common way of doing this is basically to compose a series of pin point test on the product testing e.g. performance, stress, reliability or security. According to [19] one is often very keen in reaching some conclusion to e.g. what environment the system has the highest performance, or what formatting that should be used considered to the systems load.

At the point of system testing one usually have a “finished product” that is placed in its intended environment. System testing is not so much changing the coding or application structure, but to test the system as a whole.

2.4.4 End-to-end testing

In [20] End-to-end testing is described as a set or series of tests that aims to assure functionally to some focused part of the system. It is a tool used when performing revision work on intermediate code that may affect multiple parts of the system. Having a set of tests reconstructing the same procedure can indicate that the system as a whole might experience failure since the new revision.

In order to further describe end-to-end testing one could imagine a simple web-interface connected to a back-end running a database. A simple case would be, a user clicks the request button and a request is sent to the server. The server looks at the request and issues a query to the database for the requested data. The database in its turn collects the requested data and returns with a response to the server. The server looks at the data and applies correct formatting and returns to the client application. The requested data reaches the client application and is displayed to the user. Following this sequence of operations, one tests each sub-part along the way making sure that everything is working correctly.

2.4.5 Selenium WebDriver

Selenium is described in [21] as an open source tool suite for providing automated tests of graphical web-applications. It has high flexibility in providing support for many different languages as well as multiple browsers. The WebDriver interacts with a real web browser in order to test the graphical interface. Code is written to interact with the web browser’s DOM to simulate user input. Tests can then be written to compare the graphical data available in the web browser to the expected values.

Selenium WebDriver is a well-supported API with aim to provide an interface for developing high-level graphical testing. It supports a variety of programming languages, each with their own benefits and libraries that might come in handy when developing graphical testing, see [21].

2.5 JSON and BSON

JavaScript Object Notation (JSON) is defined in [22] as a text based, data-interchange format aimed to be easily readable. Even though its close relationship to JavaScript it is language independent given that there are support for almost every programming language and is easy to read/generate for both humans and computers. It is derived as a subset of JavaScript and makes use of being built on simple, well defined data structures. Its close relation to JavaScript makes it ideal to use in AJAX communication since JavaScript natively supports JSON format.

BSON, or *Binary JSON,* is a binary format closely related to JSON, used mainly by MongoDB for storing documents, see [23]. It differs from JSON by discarding human readability and size in favor for fast computer parsing. BSON is supported in quite a range of language including object oriented-, scripting- and functional languages according to [24].

3. Method

This section describes the project workflow and how information to this thesis has been gathered. It covers the initial work of defining the project specification, development of the Track and Trace application and finally migration it to a cloud. Figure 3 describes our project workflow, having each iterative process provide information the final report.

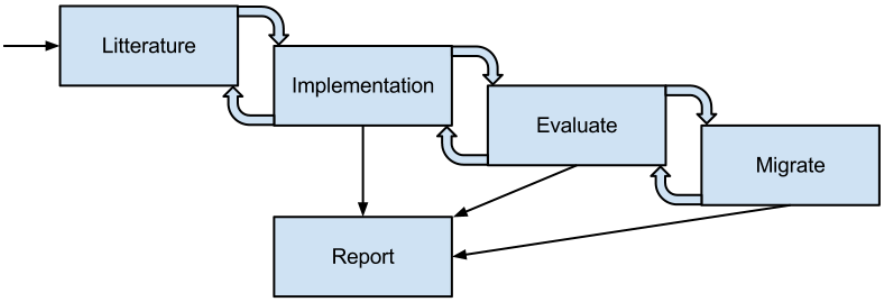

Figure 3. Project workflow

For the project reviewed in this report we started by familiarizing us with the existing systems, development environment and related technologies at Enfo Zystems. As it was an existing development already, several frameworks and software were already chosen which required initial literature studies. The list of software included:

- Oracle Java 6
- SpringSource Groovy & Grails
- SpringSource Tool Suite
- Apache Camel
- Apache ActiveMQ
- IBM WebSphere MQ

Software motivation

Enfo Zystems provides the service of high quality integrations utilizing well-established enterprise software from companies such as Apache and IBM. The backbone of Zystems is built upon some of these software's, requiring the new system to support these techniques. The choice of implementing the system using the framework Grails is grounded in Grails having a variety of plugin support for both Apache Camel and ActiveMQ. It is also a suitable framework based on the system requirements, and the compatibility having its roots in Java. Since Java plays a key role in developing integration solutions at Enfo Zystems, one key prerequisite for choosing new software is the support for Java.

Both WebSphere MQ and ActiveMQ are solution software for implementing queue based functionality for the integrations. Enfo Zystems provides its service utilizing either of the two hence requiring the new system to support both technologies. Both technologies serve the same purpose with the difference of WebSphere MQ being close-source in comparison to ActiveMQ being open-source with slight variance in support.

Workflow

The existing systems served as a specification for our development of the new system, and provided insight to how the already existing implementation was connected to the back-end technologies.

In order to get an understanding of the shortcomings and problems with the old system developed by Enfo Zystems, we spoke to both users and customers about how the system could be

improved. With this information we were able to determine functional- and non-function requirements as well as an overview of system requirements.

Theoretical studies have been carried out in order to strengthen development choices. A large part of the theoretical studies involved studying databases recommended by Enfo Zystems, especially reading best-practice recommendations, and how they behave in a cloud environment. The other large part of the studies was to learn about software testing in general, and cloud testing in particular. From this the choice of database could be done and some of the important differences in cloud testing was identified.

During the planning and implementation phase of the system, considerations and best practices found during the *theoretical studies* laid as a foundation for developing a migration friendly system. The implementation phase consisted of building the system in an iterative manner, continuously implementing new ideas, adding new functionality and conforming to new requirements. Developing the new system involved dealing with message queues, distributed data and testing. Iteratively adding and removing functionality for these technologies and in parallel adapting the system to fit in a cloud environment, contributed with experiences and problems in developing the system which provided considerations and solutions for this report.

The final part of this project consisted of migrating the system to a cloud. After finishing our development, we now had a system ready for deployment. This process involved configuring a cloud host, migrating the system and finally test that the requirements were met. During migration we encountered new problems not initially considered, which led to reimplementation and reevaluation of the current system. This presented new considerations and thoughts regarding migration and also practical examples presented in this report. In order to verify that the requirements for the system were met, we performed expectancy tests letting the system operate with intended relevant data. These results were then presented to Enfo Zystem in order to verify that the system operated at desired level.

4. Case study system

This chapter covers the case study for the developed system on which the report is built. The intention with this chapter is to briefly describe the application developed at Enfo Zystems, in order to provide a basic understanding of the system that the report is referring to.

The purpose of the system, named *Track and Trace* (TNT), is to provide an endpoint for saving and displaying logs. The system developed at Enfo Zystems is divided into two major parts, the back end which is responsible for saving new logs, and the front end which provides a graphical user interface.

The old Track and Trace is an application developed by Zystems Baseline in order to provide easy access to log events and information regarding different transactions to their customers. The TNT applications gathers log events from a logging queue located in the ESB and is presented in a graphical interface to the client. The event is grouped as transactions, depending on some identification, and gives the client the possibility to follow their transactions as a series of events through the system.

4.1 Old Track and Trace

The old track and trace is around six years old and does no longer provide sufficient performance and usability to the growing clientele. It has severe problems with handling large volumes of data and therefore its design needs to be reconsidered.

The major flaw in the old TNT was the way it queried its database. As the amount of stored data grew searches became increasingly slow as the database did not make proper use of indexes. One specific customer of the TNT generates around one million log events per day which was enough to make searches unusably slow. Neither the TNT application nor its database was built with scalability and deployment into a cloud in mind. As the system does not scale horizontally any performance upgrade requires the investment in more powerful hardware which is cost ineffective.

4.2 New Track and Trace

The new and improved Track and Trace aims to provide high performance, usability and scalability to all clients. It should be backwards compatible, supporting XML and JSON formatted data, in order to support clients developed for the old TNT. During our work we have developed a completely new track and trace system based in requirements received from Enfo Zystems.

In order to provide higher performance a new database has been considered along with a new JSON data format. The user interface has been reworked with a new layout and better usability, providing extra functionality for efficient usage.

The main method to allow for fast searches in log events amounting hundreds of millions of entries was to change the database. The database achieves this speedup in two ways; it adds indexes to important data fields and changes the database schema fundamentally. Customers of the TNT can choose which data fields are particular important to their needs and can add indexes to these to speedup searches on these.

Both the system and the database have been built in order to scale better and be easier to migrate to a cloud than the old system. This has been made by dividing the systems into several modules in order to prepare it for a migration to a cloud. Each customer of the application can choose whether to install it locally or use a dedicated cloud instance. This allows customers handling sensitive data, or data regulated by laws, to be stored and handled locally.

4.2.1 Back end

The primary purpose of the back end is to provide a common interface for saving logs to a database. It provides multiple endpoints used by systems wishing to save logs. Several endpoints exist due to their different nature and the need of the system sending the log.

Message queues are a middleware between the logging system and the back end, and can provide guarantees on data consistency and order of delivery. This endpoint is used by systems which requires a high degree of data consistency and may afford the extra complexity of message queue infrastructure. They also support mechanisms to place messages on secondary queues, called dead letter queues, in case an error occurs in the main queue. This ensures that no logs are lost in case of for example a database error, back end application error or the main queue being full.

The second endpoint uses simple HTTP requests to connect to the back end. This method is simpler than the message queue alternative and requires virtually no additional infrastructure. It provides no additional data integrity and requires the back end to be available at the time of delivery. This method of delivery is suitable to use where the delivery of a log is not crucial and the additional features and complexity of messages queues are not needed.

In the process of saving logs, the back end must convert the received logs into a common format, suitable for storage in the database. The database used is MongoDB which natively supports JSON objects. The back end therefore converts all incoming logs into JSON format which can then be sent to the database. To improve performance and decrease latency of saving logs the back end does a minimal amount of error checking and other data processing.

The second task of the back end is to temporarily save some data from each log in memory to periodically build statistics from the saved logs. This process is delayed to after saving a log, in order to reduce latency of each save. The nature of the statistics gathered makes it beneficial to save data in memory for a certain time before calculating the statistics and saving it to the database.

Figure 4 describes the two tasks of inserting data and calculating statistics. Each of these tasks run in separate threads where the insertion of logs is triggered by incoming logs and the statistics calculation is executed periodically.

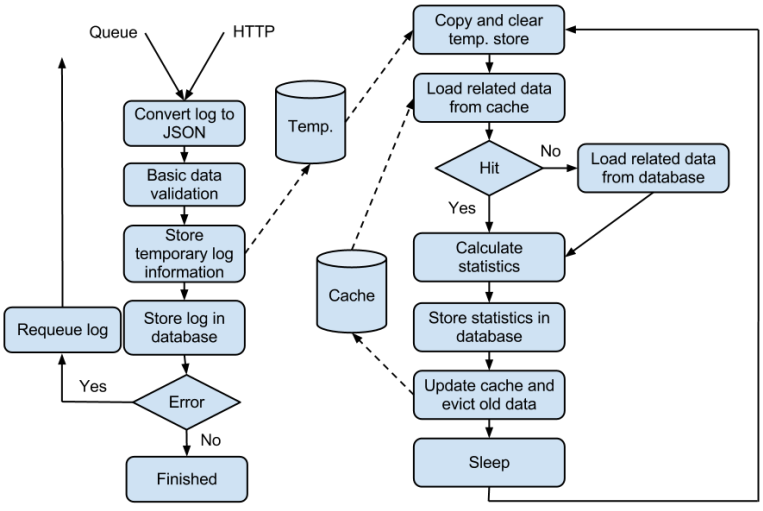

Figure 4. The two major tasks of the back end; saving of logs and statistics generation

4.2.2 Front end

The very core of the front end is a HTTP interface where queries for data can be made. It accepts JSON formatted requests containing the data fields used in the database query and other database options. The queried data is then returned to the client formatted as JSON. This interface provides an

abstraction level towards the database and gives clients a simple way of querying data without caring about the underlying database. It also gives the front end the ability to limit access or filter certain data fields for different users.

The data interface is separated from any graphical interface to give the flexibility of implementing different graphical interfaces using the data provided by the database. The front end also contains a graphical interface to provide a graphical representation of the saved logs. The interface gives the ability to search the logs, sorting, and to view statistics. The graphical interface provided to the browser only contains the bare minimum of HTML data and does not contain any data from the saved logs. The interface instead relies heavily on JavaScript both to query data using AJAX and to build the graphical HTML page from the data retrieved.

5. Results

This chapter describes the results of studying the effects of migrating a distributed web-application to the cloud. The results cover two main areas; the effects of migrating and implementing tests aimed for the cloud, and implications of utilizing cloud based services.

In *Cloud computing considerations* (section 5.1) we start by explaining what areas to take into consideration before migrating an application to a cloud. It describes requirements that should exist before a migration, how cloud services can differ to their local counterparts, and how these differences can be solved.

How to implement testing for a cloud based system is still vague and unstandardized which leaves a lot of open options on how to properly implement and deploy a test environment. The following sections, *End-to-end testing* (section 5.3) and *Cloud testing* (section 5.4), describes a provision of sufficient tests for a cloud application, along with considerations and a basic practical solution for testing a cloud application externally.

The results are based on theoretical studies as well as practical experiences in developing and migrating the system described in Case study system (chapter 4).

5.1 Cloud computing considerations

Due to differences between a cloud - and a local environment one need to make sure the application to be migrated is compatible with the new environment, and that the cloud host fulfills the needed requirements. The following sections explain requirements that should be evaluated and also how and why cloud services differ compared to local services. It focuses on services provided in the Amazon EC2 cloud relevant to the developed system such as databases, message queues and file storage.

The following section, *Requirements* (section 5.1.1), describes the need of describing requirements for the cloud. Its performance will due to being virtualized and shared between multiple users differ from a local environment. This makes it important to compile requirements and validate these in real cloud before commencing the migration process. The subsequent sections dig deeper into the technical details of how a cloud migration can affect ones application. A recurring topic is the effect of distributed services used in a cloud which gives rise to new problems not normally seen in local services. Suggested solutions are given for a client to overcome these problems by mimicking the behavior of local services.

5.1.1 Requirement considerations

As detailed in *Cloud computing models* (section 2.2), cloud services can be divided into three service models. Each of these models provides different levels of control over the system and as a result a higher level of administrative responsibility. Cloud hosting with the least amount of configuration and administration may only allow certain software or services to be used. The opposite is when a user is allowed to install and configure its own operating system and all related software.

Before choosing cloud hosting provider one need to have carefully prepared requirements in regards to for example pricing, performance, supported software, included services and service level agreement. The exact requirements are dependent on the type of application that will be run. Several factors affect the technical requirements:

- application characteristics such as memory, CPU, I/O, storage and network usage
- amount of data sent and received from outside the cloud
- services included

- supported operating systems and software
- approximated user load
- level of data security

One should perform validation of the requirements before commencing a full migration. Especially performance measurements are important to conduct as the performance of a virtualized cloud may differ to the performance of dedicated hardware and may fluctuate as other users share the same hardware.

The system described in *Case study system* (chapter 4) have several requirements; lots of RAM-memory, fast disk access, resilient data storage, pricing and support for custom applications and operating system. Most of these requirements have been compiled by studying the behavior of the application in a local environment when running a normal work load. The MongoDB database used is characterized by high memory usage to improve its performance, as explained in *MongoDB* (section 2.3.3.1) and *Appendix A*. To further improve the database performance and to provide data durability, fast and resilient disks are desirable.

Only IaaS cloud's supports running a custom operating system and custom applications, which limited suitable cloud hosts. Amazon EC2 was chosen as a candidate early on due to its good reputation and available services. It provides several different cloud instances aimed at different workloads, such as high memory instances which fulfilled our first requirement. The disk throughput and latency is dependent on the chosen instance type and storage type, but proved to be sufficient on a medium sized instance, which is further explained in *Storage and data resilience* (section 5.1.2.3). Testing over a period of time showed that disk latencies of smaller instance types varied greatly and could interfere with our application. The data resilience was deemed to be sufficient as several disk devices can be configured to form a RAID array and the data is also automatically replicated by Amazon.

5.1.2 Services

This section describes services often found in IaaS and PaaS clouds, with focus on services provided in the Amazon EC2 cloud, and how they differ from similar services running in a local environment. NoSQL database, message queue and storage services are services relevant to the system described in *Case study system* (chapter 4). This section contains both solutions to problems experienced during migration of the system, and an elaboration about potential problem areas identified.

5.1.2.1 Database

A database service provided by a cloud host may have certain properties that one should be aware of before migrating an application to the cloud. One common characteristic is the property *eventual read consistent* when reading from a database. What this means is that a read may not yet reflect the latest write, but that the database guarantees that it will eventually reach a consistent state. Eventual consistency reads are used instead of strong consistency reads, where all databases agree on a state, to avoid an expensive synchronization mechanism to keep multiple databases synchronized. It would require distributed databases, such as replicas and shards, to synchronize the data at each write, which would negatively affect the performance and limit the scalability.

Amazon provides two non-relational databases running in their cloud, SimpleDB, see [25] and DynamoDB, see [26], which supports both eventual read consistency and strong read consistency. MongoDB can also be configured to support these two consistency models. If reads from a replica set is allowed MongoDB uses the eventual read consistency, and otherwise strong read consistency.

Both Amazon's services and MongoDB uses distributed databases to support data replication and sharding, and therefore have the eventual consistency property.

When evaluating the cloud databases one must know if eventual read consistency is acceptable and compatible with the application to be migrated. It may be impossible to migrate an application to a different database if the application assumes consistent reads from the database. One should also investigate if the cloud database can be used in a local environment while developing the application. This can ease the development process and lower the cost of the cloud.

In the system described in *Case study system* (chapter 4) the MongoDB database was chosen due to its features matching the requirements, its good reputation and unlike the databases from Amazon it has left the beta stage. The major database requirements regarding a cloud migration was:

- reliable storage, both long and short term
- scalable, both regarding storage space and performance
- dynamic data structure, with index support

The requirement above that influenced the choice of database the most is support for dynamic data structures. As explained in the technical background, *Relational databases* (section 2.3.2), relational databases often require the table to be fixed both in structure and data types, and known at creation time. This requirement by itself made NoSQL databases the most obvious choice due to their lack of a fixed data structure. MongoDB's support for horizontal scaling by using shards, as further explained in *Appendix B*, made it a good match to use in a cloud environment. It integrates well with the idea of an elastic cloud which adapts seamlessly to the user's need.

While Amazon does not natively support MongoDB it is possible to run it as a custom application using Amazon EC2. MongoDB does not need any special configuration to run in a cloud environment but there are some things to take into consideration:

- If the inserted data is unevenly distributed over multiple shards, a load balancer process is run periodically to even out the data. This process moves data over the network between MongoDB instances. One need to make sure this increase in network traffic and disk I/O does not negatively affect the overall performance. This is especially important if the low level storage and network architecture is not known. If the amount of data needed to be moved is expected to be high an eventual network traffic cost must be calculated for. This problem can be alleviated by choosing a good shard key, as explained in the official MongoDB documentation, see [27].
- Expanding the database storage space and performance is done by adding new shards. Similar to the bullet above, this makes use of the load-balancer and will temporarily degrade performance. Expanding capacity is most intuitively done when the maximum capacity is reached but in the MongoDB case this will put further strain on the system. It is therefore advisable to add a new shard well before maximum capacity is reached.
- Both the database and journal file is written to disk at regular intervals. Care should be taken when choosing storage medium and file system for the cloud instance, to provide high random read and write performance and high IOPS. In the case of an Amazon cloud the EBS storage is suitable due to its high performance. More on this is explained in *Storage and data resilience* (section 5.1.2.3).

The eventual read consistency of MongoDB is not a major problem for the application since the database is seldom read. A write to the database will in most cases have time to reach a consistent

state as reads happens infrequently. Objects saved in the database will only be updated for a short period of time after they are first created which gives distributed databases time to stabilize. Other data in the database, such as statistics, is generated periodically and therefore not consistent at every moment. The user is therefore made aware of that eventual consistencies can occur and that a short wait often clears the problem.

In the case of MongoDB the eventual read consistency property is only applicable if the replication feature is enabled and a client is explicitly allowed to read from a potential inconsistent slave database. If the reader allows reads from slaves a warning message can be shown to the user to warn for the possibility of inconsistent data. This reads are not enabled in the application and the database is therefore strong read consistent.

5.1.2.2 Message queue

The application described in *Case study system* (chapter 4) relies heavily on message queues to receive its input data. This section studies the process of adopting an application to use the *Simple Queue Service* (SQS) queue solution provided by Amazon.

The major obstacle in using SQS, or similar services, in our application, is to adopt existing systems to this queue platform. There already exist standard solutions for message queues which are well-supported, provides a multitude of additional features and are more enterprise oriented compared to SQS. These does not integrate in the cloud as seamlessly as native cloud services but can if needed be run as custom applications in the cloud. Adopting our application to use SQS would require invasive changes to multiple parts of the system. Both the back end consuming messages and the producers of messages would need to be changed in order to support this new queue service. This approach is infeasible since the message queues are a business critical part of the system, on which every other part relies, and therefore require intensively tested services and enterprise level support. Because of this the developed application have no support for SQS, nor have it been seriously considered, so this section explores SQS more theoretically.

The SQS queue manager, more in detail described at [28], is hosted in the Amazon EC2 cloud and provides functionality normal to queue services but has several uncommon characteristics:

- Inserted messages are guaranteed to be delivered at least once. This means that a client, when doing multiple reads from a queue manager, can receive the same message multiple times. This property is due to how the queue is distributed in order to improve performance and reliability. To guarantee that a message is to be delivered once, and only once, would require the queue manager to synchronize distributed queues. This distributed synchronization would decrease performance and increase complexity and is therefore often omitted.
- Queues are not synchronized which can lead to a client receiving different answer from queue managers.
- Messages are not guaranteed to be delivered in *First-In-First-Out* (FIFO) order. This property is similar to the one above and is due to the queues not being synchronized.

Redelivering messages

An application using SQS must be designed with the above bullets taken in consideration. If processing the same message more than once is an error the application should have some mechanism to detect a redelivered message. This mechanism is dependent on the nature of the messages read from the queue. An easy approach is to temporarily save data in a local cache to

uniquely identify a message. If the message has not been seen it is processed and added to the cache, otherwise it is a duplicate and therefore discarded. This method requires that a message can be uniquely identified, and may require the addition of a unique data field to each message. An easy approach is to use an incrementing counter which is inserted into each message and serves as a unique identifier.

This method of keeping temporary data has four easily identifiable major drawbacks:

- It requires that some data must be saved until one can be sure that no more duplicates of a message can be received. If messages are read from the queue at high speed the amount of saved data can pose a problem.
- Multiple consumers may need to share a common data structure used to identify the messages received.
- There must be a way to determine that no more duplicates can exist.
- It makes distributed producers (writers) of messages more complex as the created identification number must be unique.

Shared cache

The first and second bullet is dependent on if distributed consumers (readers) are supported and the amount of data expected to be saved. One drawback using a distributed data structure is additional communication overhead as will be discussed in more detail in *Performance and scalability* (section 5.1.3). In either case of a local or distributed storage, a fast data structure suitable to hold the required amount of data and with good read and write complexity shall be used.

If the data is calculated to fit in memory, a key-value store such as a hash map, skip list or tree can be used. Distributed alternatives are databases optimized for key-value store, such as MemcacheDB, or pure caches such as Memcached.

No more duplicates

How the third bullet is solved is dependent on both the queue manager and how the application saves messages received from the queue. If a message is read and the queue manager indicates that no duplicates exist, the message can safely be removed from the temporary storage as it will never be seen again. If the queue manager does not support such functionality another simple approach is to save a temporary message for a limited amount of time and then remove it. This method might not be feasible as it does not in every case guarantee that a message is not delivered more than once. If a late message duplicate is received after the initial message have been removed from the temporary storage an erroneous delivery is made. If a message contains a timestamp this can be compared to the receiver's clock and when the time delta is large enough the message is deleted. The characteristics of the queue and the time delta used defines the probability of a message is incorrectly labeled as a duplicate. This method must therefore only be used if it is acceptable to lose old messages. The last method requires that the receiver can validate if the message is a duplicate by contacting an external source. If the temporary store does not contain the received message, an external store, for example a database is queried. This method is depends on that the receiver has access to such a database and is willing to accept the additional overhead when receiving new, never seen messages, by contacting an external source.

Unique identification generation

The last bullet is only relevant if the messages received do not contain any uniquely identifiable data. The creation of a unique identifier depends on if there can exist multiple distributed producers of

messages. If the producer(s) only exist locally a simple incrementing counter can be used, as mentioned earlier. This approach is still possible when dealing with distributed producers but might be unsuitable due to additional overhead of synchronizing the counter. Another approach is that each producer creates an identification number which has a high probability of being unique. This might be solved in a similar manner to how MongoDB generates its identification numbers (see *Appendix A*). The identification number is created in a way that does not require multiple producers to be aware of each other.

FIFO ordering

As SQS does not guarantee FIFO delivery an application requiring this property must implement it. This can be done by keeping track of every received message and if necessary sort the messages in the correct order. This requires that each message contains a field suitable for sorting, for example an incrementing counter. It can be implemented in a similar way to how a unique identification can be generated, as described earlier. One major problem with reading messages from the queue and not directly processing them is the possibility of data loss in case of an application crash or similar event. Queues, and similarly databases, often allow transactional operations which guarantee that no data is lost in case of an unexpected event. The problem with reading messages from the queue and storing them locally is that the messages are removed from the queue after they have been saved locally. In case of an application crash these messages are lost unless the client implements a persistent storage.

5.1.2.3 Storage and data resilience

The characteristics of storage in a cloud are both dependent on the actual hardware used and how the storage space is virtualized. A storage device may be shared between multiple users and should therefore not be able to be accessed as a raw device by the users. Instead it is virtualized to hide the low-level details, which have the effect that information that can be useful to the users is hidden. Information such as device types (hard drive or solid state drive), interface (SATA, SAN or SAS), block size, buffer size, IOPS and latency gives the user possibility to optimize the application and services used.

Amazon provides three storage services with different properties. *Instance storage* are attached to most EC2 instances and is recommended to use for temporary or less important data, as it bound to the lifetime of an EC2 instance and is less rigorously replicated. *Elastic Block Storage* (EBS) is a block level storage suitable to store frequently changing data, such as databases. This data is replicated automatically and it is possible to use multiple EBS volumes to use replication, such as software RAID. This storage is network connected and as such limited by the internal network speed, but is optimized for low latency. Both the Instance storage and EBS employs the strong read consistency. The last storage service, *Simple Storage Service* (S3) differs from the rest by employing eventual read consistency, and a different pricing model. It is network connected but the speed is more limited than EBS, and may have higher latency.

Durability and MongoDB

The system in *Case study system* (chapter 4) is dependent on the underlying storage since it affects the MongoDB database. Data is written to a journal file, a database file and a log file which may pose a problem if the underlying storage uses hard drives, due to their high seek latency. Therefore one should use different hard drives, or at least different platters, to store the different files in order to

improve performance. Multiple EBS volumes are used in a software RAID 1+0 array to alleviate this problem by striping data to multiple hard drives.

In order to reduce the impact of high disk latencies MongoDB does batch commits of data. Data written to the database are not immediately written to disk but instead kept in memory and periodically written. The direct result of this is that the *durability* property of ACID is violated as data may be lost in the short time period before being persisted to disk. The journal file can be set to synchronize to disk at intervals between 2-300 milliseconds, where one can choose if to prioritize write speed or data durability. To find a suitable write interval one must perform benchmarks in the cloud by applying load similar to the real application. MongoDB gives the connected client possibility to choose the level of data resilience at each write. This is called *write concern* and indicates how concerned the client is about the data being written successfully. One can influence both how data is flushed to disk and how many replicas must receive the data before returning.

During testing of our application we have found that the level of write concern greatly affects the write performance in terms of writes per second. The default behavior (denoted NORMAL) is to not report errors in the database which can lead to data loss without the client ever noticing. Several different levels of write concern can be used to alleviate this by waiting for the database to acknowledge the write. The simplest and fastest form of safe write concern is that the database acknowledges the received data directly to the client without writing it (denoted SAFE). Another possible write concern is to let the client wait for the database to flush its buffers and write the data to disk (denoted FSYNC_SAFE). This guarantees that no data is lost but slows down write performance severely as data is only persisted periodically. The last studied write concern is only applicable if replicas are used (denoted REPLICAS_SAFE). It gives the possibility to wait for any number, or the majority, of the replica databases to acknowledge the write. The data are not flushed to disk but one instead relies on that at least one replica will successfully persist the data to disk.

MongoDB performance

The following performance data is measured on an Amazon EC2 instance (*m1.medium*) with a dedicated RAID 1 array using two EBS volumes for each of the data and journal files. It is running the complete back end, an Apache ActiveMQ queue and the MongoDB database on the same server. MongoDB uses the default journal flush interval of 100 milliseconds, has sharding disabled and replication enabled. The replication database consists of one additional instance also configured as described above. The last test (REPLICAS_SAFE) using replication uses an additional EC2 instance (*t1.micro*) to host the second replica set, configured as described earlier. Both these instances are described more in detail in *Appendix D*.

Table 1 shows the performance measurements saving messages to the database in Amazon EC2:

Write concern	Messages per second
NORMAL	2800
SAFE	2080
FSYNC_SAFE	25
REPLICAS_SAFE	295

Table 1. Insertion performance of MongoDB in Amazon EC2

Table 1 shows the performance of MongoDB database situated on Amazon EC2. Worth noting is that the NORMAL write concern most likely is skewed due to being CPU bound. On this test the CPU utilization was constantly 100%.

The next test is done in a local environment in order to compare the difference in performance in the specific cloud. These tests are performed on ordinary customer grade computers, but using the same configuration as in the Amazon EC2 test. Table 2 shows the performance in a local environment:

Write concern	Messages per second
NORMAL	8550
SAFE	4920
FSYNC_SAFE	24
REPLICAS_SAFE	395

Table 2. Insertion performance of MongoDB locally

The both tests from Table 1 and Table 2 clearly show how MongoDB's performance suffers from safe writes. The common denominator of the write concerns except NORMAL is the additional delay they introduce to the writing client; in all cases the client must wait for the database to acknowledge the write before resuming operation. We have identified the disk flush delay and network latencies as the two contributing factors for this performance impact. Both of these factors can be hard to reason about in a cloud environment since the underlying infrastructure is often hidden to the user. During the development and testing of our application we have found on several occasions that the default (NORMAL) write concern may lose data. It is therefore a must to use a write concern of at least SAFE.

To make our application less sensitive to high latencies it uses several concurrent threads to write data to the database. As the limiting factor is the high latency and not the database throughput in itself this method allows for less time wasted on waiting for a response. Testing with three concurrent threads showed a close to linear increase in performance.

5.1.3 Performance and scalability

In order to better utilize the capacity of a distributed cloud the application must be designed to scale horizontally, which means that the application can increase its performance by adding additional cloud instances. This can be done by designing the application as modules where each module runs on its own cloud instance. The process of modularization for a cloud application is not the same as a local application. In a local environment fast in-memory data structures can be used, but this is not possible in a cloud since the application is distributed over multiple instances. To highlight the differences this section contains a comparison between different cache solutions.

The module concept can be divided into (at least) two classes; either a module processes a sub-part of an algorithm, or there are multiple modules of the same type computing the same algorithm. These two algorithms are described further below.

Divide and conquer

The first class depends on the inner workings of the algorithm and requires it to be sufficiently parallelizable. The maximum speedup factor is limited by Amdahl's law [29] which states that the speedup is limited by how much of the original serial algorithm can be parallelized. This means that there will be a point where it is not possible to parallelize the algorithm any further and that the system therefore does not scale beyond this point. Another issue is the additional overhead created by communication between modules. As each module only computes a fraction of the algorithm the

amount of communication between modules will be high. In a cloud environment the exact location of a module might not be known and suboptimal communication methods such as TCP or HTTP requests might be used instead of for example *Inter-process communication* (IPC) or shared memory.

Parallel execution

The second module class looks at the application at a higher level and creates modules that each executes the same algorithm but on different input data. This method is not dependent on how parallelizable the algorithm is but requires that the input data can be processed in parallel. This means that the input data must be possible to divide among several modules and that there are no internal dependencies between the input data. Given that there are no data dependencies this algorithm may scale towards infinity, by adding more modules.

Two classes of modules are visualized in figure 5.

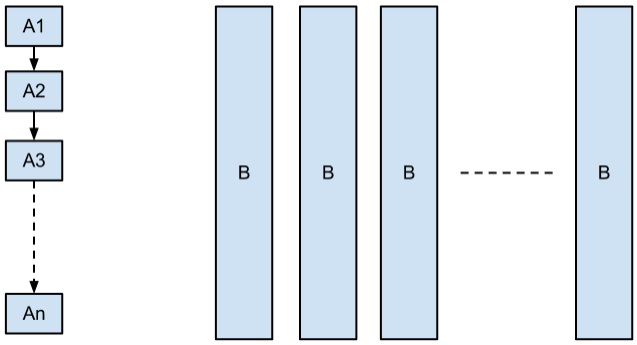

Figure 5. Two classes of modularity

One major obstacle when creating modules are the need to share data in a distributed environment. Modules running locally on a single computer can use optimized data structures and access methods. Multiple local modules can easily share a single thread-safe in-memory data structure with low communication overhead. A distributed system on the other hand may not have an obvious location where data can be shared. The idea of centralizing data goes against the concept of modularizing the system as it adds a potential bottleneck and may inadvertently serialize the algorithm. It may limit the performance growth potential unless the data storage in itself supports some form of sharding. It also adds a single point of failure to the system when multiple modules are dependent on a centralized node. Modules should ideally be designed as a shared-nothing architecture, where each module is independent of the others, to minimize the need of performance expensive communication.

Distributed cache performance

During the implementation phase of our application described in *Case study system* (chapter 4) the need for sharing data between modules emerged. This chapter explores the performance impact of using a distributed cache instead of simple in-memory data structures.

The tests are performed on Amazon EC2 instances (described in *Appendix D*), using Memcached [30] as a distributed cache, and a thread-safe Java hash table implementation [31] as a local data structure. The cloud instances are running the Amazon Linux AMI (detailed in *Appendix D*). Both the operating system and the Memcached server use the default configuration, except that Memcached is given 1024 MB of object storage.

Table 3 shows the Memcached server running on the same server as the client.

Type	# Threads	# Messages/Thread	# Total messages	Time [s]	Messages / s
Insert	1	500 000	500 000	75	6 667
Read	1	500 000	500 000	74	6 757
Insert	10	50 000	500 000	78	6 410
Read	10	50 000	500 000	79	6 329
Insert	100	5 000	500 000	85	5 882
Read	100	5 000	500 000	82	6 098

Table 3. Local Memcached server running in Amazon EC2

Table 4 shows the Memcached server residing on one of the Amazon EC2 instances, and the client on the other instance. This clearly shows the large performance impact a distributed cache has, where the single threaded insert is almost eight times slower than the local Memcached server. These findings are similar in those of *Storage and data resilience* (section 5.1.2.3) where MongoDB displayed severe slowdown when used in a distributed configuration.

The degraded performance is due to the increased latency introduced in the network communication. As both inserts and reads from the cache is done in serial the latency effects the dead time. This is somewhat alleviated by running several threads in parallel inserting or reading the cache, which makes the aggregated dead time less noticeable.

Type	# Threads	# Messages/Thread	# Total messages	Time [s]	Messages / s
Insert	1	500 000	500 000	574	871
Read	1	500 000	500 000	595	840
Insert	10	50 000	500 000	209	2 392
Read	10	50 000	500 000	291	1 718
Insert	100	5 000	500 000	85	5 882
Read	100	5 000	500 000	81	6 173

Table 4. Network Memcached server running in Amazon EC2

Table 5 shows the use of a local in-memory hash table. It circumvents the complexity and overhead of using a dedicated cache server but is limited to use in a single cloud instance. Unlike usage of Memcached this solution does not use TCP/IP connections but rather accesses the RAM memory directly. As RAM memory has higher throughput and latencies of many orders of magnitude lower than network connected solutions the findings in table 5 is expected.

Type	# Threads	# Messages/Thread	# Total messages	Time [s]	Messages / s
Insert	1	500 000	500 000	0.75	666 667
Read	1	500 000	500 000	0.28	1 785 714
Insert	10	50 000	500 000	0.69	724 637
Read	10	50 000	500 000	0.28	1 785 714
Insert	100	5 000	500 000	0.65	769 230
Read	100	5 000	500 000	0.25	2 000 000

Table 5. Local in-memory Java hash table

Performance and scalability for the case study system

Our application is divided into several modules, of which the two biggest are the front- and back end. The need for sharing data is solved by using a database as persistent storage and simple HTTP requests between the modules for rare, less important data. The front end is not further modularized, but as it does not hold any important state or data there can exist several instances of a front end. This adds the possibility to utilize multiple front ends as load-balancers if necessary.

The back end on the other hand is divided into several modules:

- read messages from a queue and saving them to a database
- periodic statistics generation
- periodic database cleanup rules update
- periodic database cleanup

The main work for the back end is to read messages from a queue and save them to a database. This in itself would be a perfect candidate for a module to concurrently run in several cloud instances. It is problematic due to how each saved message generates metadata, which is grouped into larger blocks and are dependent on earlier metadata. This metadata is made available to the statistics generation module(s), and would require the need for a distributed data structure if several modules should work in parallel. From the earlier findings the performance was sufficient but the added complexity of additional software was discouraging. Instead a simple thread-safe in-memory shared data structure was chosen which gives high performance but limits the modules to be running in a single instance.

The statistics generation module uses the metadata generated by the module described earlier. Further modularization of this module is complicated by how the metadata is grouped together, which requires that chunks of the metadata are processed by a single module. Creating independent modules requires that the metadata resides in a distributed data structure where the data is already divided into the correct chunks, or that the data processed does not generate any module transcending metadata. This will lead to a more complex handling of metadata, both at insertion and retrieval of data. The added complexity of such a system was deemed to be infeasible, and this module does therefore not support distributed processing.

The module which updates the database cleanup rules is responsible for periodically read a set of rules present in the database. It acts as a controller and handles the logic of scheduling tasks. This module does not share any data structures with the previous mentioned modules and are therefore a suitable candidate for modularization. Due to the nature of this module, care must be taken for using proper synchronization if several modules are concurrently running, in order to prevent a task from being scheduled multiple times. Although this module is possible to modularize it is deemed very unlikely to ever cause any performance problem due to its infrequent and lightweight workload.

Each periodic database cleanup task is completely independent and does not keep any state at all. It is depending on the previous module to be scheduled but requires no intervention after this point. In addition it might run for long periods of time and be both computation and I/O intensive. They are intended to be run very infrequently and are not sensitive to high startup latency. Each task is run sequentially and is not possible to modularize further. These facts make a task suitable to be run at its own cloud instance. It the cloud host provides a truly elastic cloud a new cloud instance can be created and destroyed for each task invocation. This can lead to economical saving where only cloud instances which are used exist.

5.1.4 Distributed systems and connectivity

Systems externally connected to the cloud can and should be considered as a distributed system. In a distributed system where an application needs to gather data from different locations there are a few scenarios that might occur that one should keep in mind in consideration to architecture and connectivity:

- Front- and back end might not be located at the same location or server.

- Firewalls and other gateway issues might arise when a non-local application is trying to access database or server applications from outside the local environment.
- The local- and cloud environment might not have the same protocol support.

Communication between systems is something that needs to be addressed early in order to avoid difficulties later in development. A general and good approach is making the application or system generic in sense of communication and data formatting.

Having an application run in the cloud, there is interesting scenario in the way one deals with fetching information from other complementing distributed systems. The coming scenarios describes two different problems and possible solutions with consideration to system requirements of the application described in *Case study system* (chapter 4).

Internal system scenario

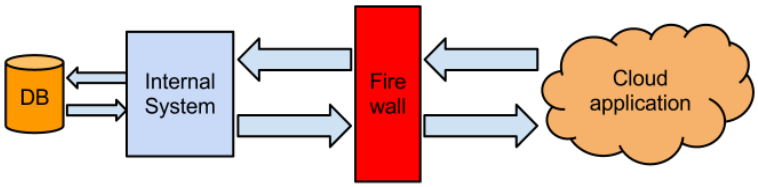

Figure 6. Communication with internal system

In the above scenario (figure 6) an internal system is connected to a local database and a cloud instance needs to request information from the internal system's database. The internal system logs information and stores it in the database which is then is meant to be requested by the cloud application. Here the cloud application is dependent on a system outside the cloud which adds additional complexity in terms of connectivity and administration. The internal system might be located inside a company's network which requires firewalls to be configured to allow the cloud application access. It also makes the cloud application dependent on the outside system in terms of availability.

This was implemented as figure 6 shows by having the cloud application perform requests to the internal system directly. The internal system fetches the information and returns it to the cloud application. However, this is not an optimal solution when the system requires low latency towards the database. This type of implementation is an option when the number of requests from the cloud application to the external database is low and the additional overhead is acceptable, or that the data must not be located inside a cloud either due to legal or business reasons.

This scenario was implemented to deal with infrequently used, non-critical local services running inside a company's internal network. The solution does not provide sufficient performance when dealing with a system performing frequent queries to the database and that requires low latency communication. This problem is solved in the following scenario.

Push information scenario

This scenario describes a part of the system that requires high performance and low latency querying the database. Having the information stored on a local database in the cloud is not an issue due to legal or business reasons and therefore one can utilize the fact that the cloud database are more easily accessible by the cloud application.

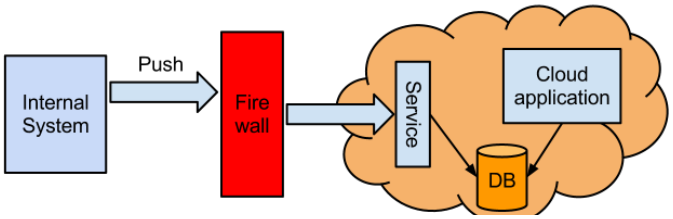

Figure 7. Push information

In the above figure 7, the system consists of the application, a communication service and a database. Instead of having the cloud application request information from the internal system, the internal system pushes all relevant information to the communication service in the cloud. The communication service will then store the information in the local cloud database which the cloud application is able to read.

With this implementation one avoids all the complications of having to access secure locations inside company environments and instead one can define which providers that should be able to push information to the cloud application. A benefit of this approach is that the application is relieved of the responsibility of retrieving data from the internal system, and instead reads its local database which gives lower latency. As the database is local to the cloud application it can provide a higher level of consistency and availability compared to scenario 1.

5.2 Security

When evaluating the viability of a cloud based solution one need to take security into consideration. Comparing security in a cloud compared to a local network, one can not necessarily make the same assumptions. As both hardware, such as network infrastructure and storage, and software such as databases, may be shared between multiple users, steps need to be taken in order to prevent any third party from reading sensitive data.

Internal network communication between cloud instances, which is normally done in a trusted local network, will in a cloud involve communication over untrusted networks. To secure this communication the use of encryption can be used. If the applications used does not natively support any form of secure communication the use of tunneling can be used. For example, two cloud instances connect using *Secure Shell* (SSH), and the applications are configured to communicate over the secure tunnel. This approach have the advantage of being transparent to the application, but adds increased CPU usage and a more complex network infrastructure.

How data residing in shared cloud services is secured, such as in message queues or databases, is largely dependent on the service's encryption support. The payload of a message placed in a message queue may be encrypted before it is placed on the queue, and later decrypted by the client reading the message. This approach may not work in a database if the content of the database should be queryable, unless special encryption support is natively available. If the data to be stored is sensitive the alternative to a database service is to administer a custom database which supports the needed level of security. To secure stored data, which may include databases, message queues and normal files, one can use a file system and operating system supporting full disk encryption.

The system in *Case study system* (chapter 4) is moderately secured. The front end supports HTTPS (HTTP Secure, using SSL (*Secure Socket Layer*)) to provide authentication and encryption. A similar approach is used for the back end which uses SSL to authenticate both the client and back end. This requires both parties to share certificates which greatly improves security but adds additional administration work at first setup. This must be done as the back end does not validate the

received data and malicious input would be disastrous. The Amazon EC2 cloud provides an ingress filtering firewall to further secure the systems by limiting the access to the services. Securing the network communication between cloud instances or file storage has been deemed to not be necessary.

5.3 End-to-end testing

Providing automated end-to-end testing for the system described in *Case study system* (chapter 4) is possible considering a combined test environment combining traditional test methods such as unit tests and testing tools external to the cloud. This section describes the process of combining these difference approaches in order to provide a case sufficient level of end-to-end testing.

In order to provide end-to-end testing for the web-application a lot of different aspects needs to be considered. The main idea is to iteratively build tests on top of each other until finally one is left with a system that has been tested from the bottom and up starting with unit testing and ending with user testing. Another possible requirement is that it should be able to run through these test automatically before every deployment which introduces some difficulties when one has to combine unit testing with user functionality testing.

Figure 8 shows the whole suggested test process described in the following sections.

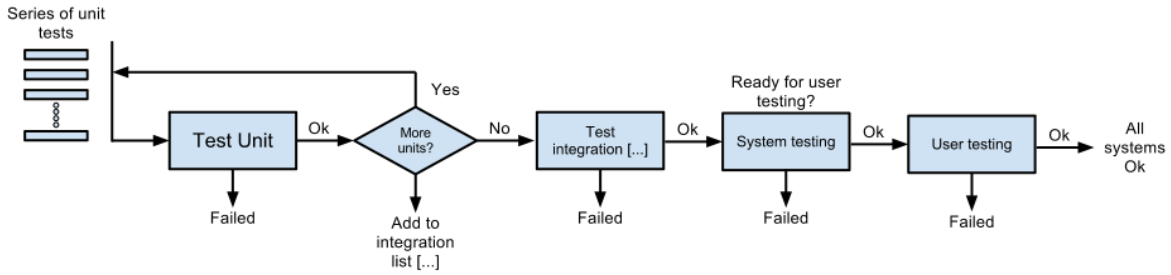

Figure 8. End-to-end testing Track and Trace

5.3.1 Unit test

The unit tests has been implemented and designed in order to assure that specific components performs as expected in the sub-parts of the system. Components are extensively tested in sense of expectancy and utilize assertions in order to detect anomalies. If enough unit tests have been included, one can be certain that an error does not lie within the specific units but in the next level of integration. Unit testing is usually rather straightforward but there are some approaches where one could extend unit testing to a more advanced setting.

- Test sequences that periodically execute application specific blocks or methods in order to verify that they still are behaving as expected.
- Having precomputed test data or a preconfigured database for testing expectancy.
- Pseudo-random stress functions during low latency periods in order to verify stability.

As far as how extensive unit testing can become for any application, it is usually limited by size and complexity.

5.3.2 Integration test

Before integration testing starts one first have to make sure that all unit tests has passed as Ok. When all unit tests are finished, integration testing is started by combining different units which underlying functionality are intended to work together. These tests can be seen as a list of units and

are expectancy tested as a group. The following example is a function with intention of entering a log event into the database.

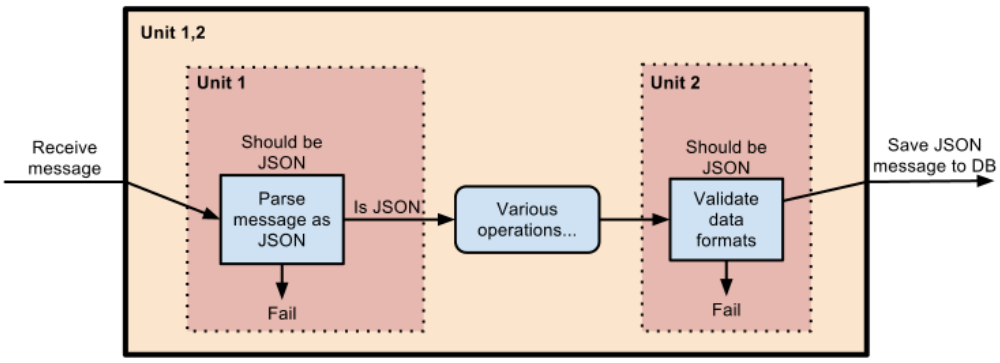

Figure 9. Integration test example

In figure 9 there are two separate unit tests, *Unit 1* and *Unit 2*. These units are individually tested and needs to check Ok in order for the integration test *Unit 1,2* to check *Ok*. A passed integration test does therefore not only guarantee the success of its sub-parts, the unit tests, but also that the module as a whole works as expected.

In the example above the system validates that the message received at *Unit 1* is in a valid JSON format, and therefore can be sent to *Unit 2*. *Unit 2* in its turn validates message content. Running integration test *Unit 1,2* will validate that the received message has been saved in the database as a valid JSON message.

Iteratively one computes a series of these integration tests making up the whole system. By wrapping a complete test around all these integration tests one now have a well-defined structure in how to test the system as a whole. With this, one now have a complete series of tests making up the whole system which can be used in order to perform relevant system testing.

5.3.3 System test

System testing is the broad term of testing that a system conforms to the specified requirements. In this context a system refers to a system that has passed integration testing. This phase of testing can contain for example performance, load/stress, compatibility and scalability testing.

Performing system testing provides information about the overall soundness of the system. The system described in *Case study system* (chapter 4) receives log events on a queue which is read and later inserted into the database, one wants as high throughput as possible going from queue to insertion in database. This is a crucial part since the clients utilizing the system are dependent on the information that is added to the queue to be available from the application with low latency. Figure 10 describes how a message is fetched from the message queue to the back end service which performs a series of operations before saving the information to the database. The back end service examines every log event in order to distinguish anomalies or errors before passing the information to the database.

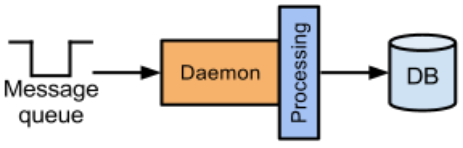

Figure 10. From message queue to database

Data retrievement from the message queue to the back end service could be a potential choke point since it is the back end that is responsible to deliver the data to the database. This because, not before the back end has delivered the received message to the database can it fetch another message for processing. It is therefore periodically tested to detect if there are large variances in gathering data from the queue, as well as the time for a log event to be inserted in the database. In the scenario of a very high load there could be a delay in delivery to the database and with that a latency for the client to get up-to-date data from the application.

Although both the back end and the database have very high performance the application is still dependent on the interaction between the two modules to ensure high performance. It is therefore important to test the complete system in order to obtain a realistic measurement of performance.

5.3.4 User test

When all prior tests are Ok, one moves on to the last step of end-to-end testing; to test the graphical interface of the application. To test the graphical user interface Selenium 2.0 WebDriver is used. The graphical interface testing is built in Java, using the Selenium driver, which makes it easy to integrate other techniques that might be useful for testing the application. The graphical user testing suite is separated from the rest of the test build and is run externally from the cloud. This makes it possible to provide a different aspect of testing, providing the possibility of validating and testing the system separated from cloud environment. By this methodology one is able to give a more concrete picture of how the complete exposed system is performing focusing on performance, stability and usability.

The structure of writing browser tests is very straightforward. By using Selenium WebDriver one can manipulate the web browser to simulate user input by accessing the DOM object. The following is a basic procedure of how a graphical interface is tested:

1. navigate to the desired page
2. find the desired HTML element by for example, class name, tag name or id
3. perform operations on that element or locate the next related element
4. validate response

When the above steps are performed from top to bottom without exceptions, the graphical interface are most likely working as intended. Since the graphical interface are dependent on the underlying functionality working properly, one can be sure that since the graphical test did not generate any errors, the underlying functionally also worked without error. If an error on for example a page redirect occurred due to some error in underlying functionally, Selenium will return an exception and an appropriate message are given to the console. By well-structured error messages one can get a good overview of how the system performed during the testing phase and maybe get a hint of what might have gone wrong during the run.

When performing a query from the graphical interface, the database is expected to deliver correct data back to display for the client. By adding some functionality of making it possible to connect to the database within the Selenium test program, one can make sure that the information sent, and the information received corresponds to the appropriate entry in the database.

Example test run

In the beginning of the test program a connection to the database is made from within the test program. A query is passed to the database for a random log event which is returned and formatted appropriately in order to work within the Selenium test case. With the information received from the

database the program queries the same log through the graphical interface. The information returned to the graphical interface from the query is compared to the information earlier acquired from the database. If this information checks out as equal the program is working correctly and the Selenium program can continue with testing other functionality.

5.4 Cloud testing

There are distinct differences in testing comparing the local environment to the cloud. Testing a cloud is complicated by, and important due to, the distinct difference in the implementation environment. Security-, performance- and problem isolation testing are hard to implement as these areas may be out of control for the user. In the following section we will highlight observations made during the development and migration of the *Case study system* (chapter 4) in regard to implementing testing for a cloud environment.

5.4.1 General

Having an application run in a cloud environment in contrast to running it in the local has some differences in terms of testing. Testing a cloud application does therefore not end with testing only the actual application but also the underlying cloud infrastructure and the services it depends on.

Testing, such as unit testing and similar, gives an indication that the application is behaving as intended before actual deployment in relation to the services to be used. Once the application resides in the cloud, a whole new set of aspects are presented in terms of performance, usability and usage intentions. A real issue in the cloud is testing the capability and reliability from within the cloud between different services from the cloud provider. Once the application resides within the cloud, one is bound by performance and reliability that comes with the cloud solution. Not only will the application probably be running in parallel with other applications on the same server, it will probably also be bound by the same hardware which might cause problems or concerns in form of security and reliability.

5.4.2 Considerations for cloud testing

Moving the application to the cloud will in some cases present some differences in how to implement a certain test or test case depending on the cloud environment. Some levels of testing are more migration friendly given that they are less dependent on the current environment. Unit tests are generally relatively easy to migrate since it usually only depends on very specific or small factors in order to compute, whereas integration and system testing are more sensitive to the execution environment.

Migrating the system described in *Case study system* (chapter 4) presented some interesting considerations to keep in mind when initially developing ones test suite, alternatively at the actual migration.

Load, stress and performance testing will no longer consider only the application itself. Since the application now is running within a cloud, additional aspects such as external communication need to be considered when determining systems soundness as a whole. Therefore it is interesting to produce some automated test cases, stress and load testing both the application itself from within the cloud comparing testing the complete cloud application from outside the cloud in order to determine possible bottlenecks more specifically. A more elaborate example of this is given in *Case scenario* (section 5.4.3).

A test case closely related to performance is to test an application's ability to scale performance wise. This is a valid test if the application is built to support scaling in a cloud. This test

can be run by generating high load on the application and measure the system's response. These measurements can then be used to track performance changes related to system changes and also to compare different cloud hosts.

If the process of migrating the application results in new or modified support services, a compatibility test can help to verify the soundness of the system. This test can be seen as an extension of normal integration testing but focuses more in the integration of external systems, such as different cloud based services. Even though a well-defined API is used to communicate with the services provided by the cloud host, unexpected changes to external system can break functionality of existing applications. This is especially true to systems which is out of control to the user, and only provided as "ready to use" services. The exact structure of a compatibility test is dependent on the system, but this test can generally contain logic that can be run periodically or if the cloud provider announces changes to its services. The test can verify that the API used is still valid by comparing results received from the service with an expected valid result.

5.4.2.1 Security

Security is always a factor, whether it is in the cloud or in a local server setup. However, dealing with security and access from within the cloud, there are less concrete factors stating that the user trying to access the service or application is a valid user, except for example a provided user name and password. In relation to the local environment running behind the security of firewalls and gateways of a company, where users of the local network is assumed to be trusted, the cloud does not have the same extensive protection for accessibility. In a more local setting one can take parameters into account such as IP subnets, domain, role and other parameters more concretely stating who you are and what rights you have to access the system.

Testing that the cloud system provides sufficient security is a rather crucial part to monitor in order to be able to state that the application and the services behind it are secure.

5.4.2.2 Performance

Performance is a very wide concept applicable to almost all parts of a system. Still there are some distinct differences that cannot be found in a local setting. Performance testing can be performed as usual within the application, testing e.g. throughput for specific parts of the system.

A new aspect that comes with a cloud application testing is the act of testing the actual cloud. All communication to and from the cloud is bound by the performance, accessibility and stability of the cloud provider, something that one as a mere customer of the cloud service has no control over. Since ones applications and services are virtualized and the real hardware might be distributed, this implies that you also are sharing physical hardware, memory and storage with other applications, not necessarily your own. Performing e.g. stress tests and other performance determining test methodologies against the cloud is therefore quite dependent on its overall state. This could possibly generate some diffuse results depending on the load during that time. This implies that measurements must be viewed over a larger time span in order to eliminate possible variances.

5.4.2.3 Problem isolation

An application running within the cloud is bound by whatever performance and availability that is provided by the cloud provider. An application depending on various services relies on the stability and accessibility of that specific service at any given time during run-time.

Given the scenario that an application is utilizing a series of services which during a longer period of time has been working correctly. At some point the system crashes and through the error

reports one can determine that the problem has occurred in relation with one of these services. In a local scenario the problem can be easily isolated to that specific service or the sub-part of the system that utilizes that service. Isolating the same problem in a cloud environment is not quite as easy. Within the cloud, the service in question is not necessarily localized even within your own virtualized environment. Moreover the service could be utilized by a number of other applications in parallel, distributed and replicated from some centralized unit. This makes it extremely hard to isolate the actual problem and even harder to produce tests that could determine if it is the actual service, one's own application or some other application that is the actual cause to the problem.

5.4.3 Test scenario for case study system

Given the scenario and application described in *Case study system* (chapter 4), testing of the deployed cloud application is done both from within the cloud in terms of application testing, as well as testing the exposed application services from outside the cloud using Selenium WebDriver.

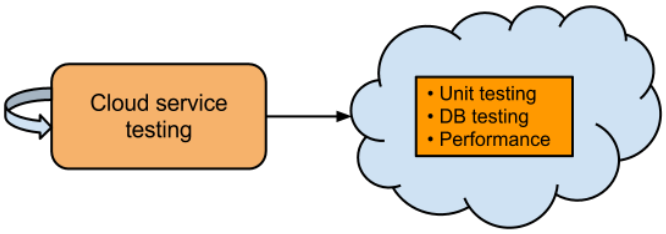

Figure 11. External and internal cloud testing

Figure 11 describes the testing done with the system described in *Case study system* (chapter 4). The application itself contains all necessary tests briefly described in *End-to-end testing* (section 5.3) in order to show that the application is running as intended with expected results. The application is run with a debug build containing tests in application level and database. From outside the cloud another test program is run periodically from a server which runs the test script described in *User test* (section 5.3.4). The test program running the graphical testing script is also combined with data input testing as well as stress testing, testing the system during low load periods with intention of validating system performance.

By this methodology one covers a few, but not all of the different aspects that are testing a cloud application. One aspect not considered in this model is cross service and cross cloud application testing, internally and externally, which is difficult to implement as a mere customer. In order to perform these kinds of test it is necessary for the cloud provider to have some form of support.

5.4.4 Data input testing

Interesting methods of executing system testing is the idea of data input testing, or fuzz testing. The idea is to test the parts of the system that is exposed to external systems which may not be trusted. The tests are done by generating random or semi-random data as input to the system and monitor the system's response. Since most parts of the system are dependent on some form of input, this method of testing is applicable in most levels of system testing.

Fuzz testing is favorable used in conjunction with other tests such as unit testing to provide more substantial test coverage. Our application used two forms of fuzz testing, both by generating semi-random to the queue and by the Selenium WebDriver to extend the testing of its graphical interface. The first test created syntactically correct data which in different ways differed from normal data and saved it to the input queue. It was used to successfully detect errors in the back end

which received and parsed the data. It highlighted the dangers of assuming a certain data format from data received from an external source. Performing fuzz testing using Selenium tested the front end. For example the logic parsing search queries was tested with unusual data which exposed several anomalies.

6. Discussion

This chapter contains a discussion around the research method used and the results of this report. It will consolidate our results to discuss the findings, how they can be applied in other projects and to justify their validity.

It also contains a step-by-step guide used in our project to migrate a MongoDB database from a local environment to a cloud. This guide is created from migration experiences and contains tips and guidelines for how to successfully migrate MongoDB while using features such as sharding and replication.

The results can be naturally divided into two parts which will be discussed separately; the first part concerns services and how they are affected by a migration, and the second part discusses considerations for cloud testing.

Worth noting is that these results are limited to the application developed in the case study. They do not cover any general case in migrating an application to a cloud, but instead focuses on the specific needs of the mentioned application. The characteristics and high level view of cloud services and testing described, is valid independent of the application. However, the solutions to the problems may only be applicable to our case. The possible solutions to the problems explained in *Services* (section 5.1.2) should not be seen as definite solutions to the problems. They are simply solutions given in order to raise awareness of how cloud services may differ from local services. Additional work needs to be done both to give solutions to general cloud computing problems and to examine individual cloud host's services.

6.1 Cloud services

The major findings in the results concerning cloud services relates to the profound differences between local services compared to cloud services. In preparation for the development and migration process of our application it became clear that we needed to carefully consider which services was usable.

MongoDB with its functionality suitable for an elastic cloud, such as sharding and replication, showed that one need to take both the application to be migrated and the cloud into consideration. The application may have to deal with the eventual read consistency property if using replication, which is an area where we have not presented any real solution but only how we avoided this problem. This problem was not very deeply studied as it did not affect our project, and it is possible to opt out of this behavior by disallowing reads from slave databases. One needs to be wary of this problem and thoroughly investigate if ones application can tolerate this property.

MongoDB performance

MongoDB was initially chosen mainly due to its good write performance and ability to scale horizontally. It was known from the beginning that MongoDB does not adhere to ACID in order to increase its performance. The results from the performance measurements done in *Storage and data resilience* (section 5.1.2.3) clearly shows how MongoDB's performance declines when one requires safe writes. One often stressed benefit of MongoDB is its fast writes. This has been showed to be true only when using the default, unsafe configuration, which in our case provided unacceptable data security.

The measurements presented are only valid for our application in the specific environments used. These results may therefore not apply strictly to an EC2 instance or clouds in general, but should be seen as a general hint. The techniques tested will due to their nature without doubt limit

the performance, regardless if tested in a cloud or locally. The actual performance numbers will differ in different environments but their ratio will most likely stay fixed. The application relies on a chain of services in addition to the MongoDB database, such as message queues and other applications. The chain is famously never stronger than its weakest link, and therefore our measurements could have been influenced by limited performance of external services. Even if the measurements could benefit from a larger sample it is our belief that these finding point in the correct general direction.

The finding corresponds with our expected results. The introduction of latency when writing data to a database will evidently reduce a client's performance. To reduce this performance impact we used several concurrent threads to write data which showed near linear performance increase. This is also to be expected since the database is more efficiently used. These finding are similar to other performance measurements in similar context. In [32] the author describes a simple test case inserting data using different write concerns. The environment in which this test was run in is unknown but the test clearly shows a similar pattern to our findings.

Message queues and scalability

The section on cloud based message queues highlights the differences between the traditional queue managers we used compared to the cloud based Amazon SQS. The solutions presented are mostly theoretical as none of them was implemented for the purpose of solving the complexities added by the cloud based queues. These solutions can still be considered to be valid as they are based on experiences developing other parts of the system as well as the internal working of the MongoDB database. The process of modularizing our application gave ideas on how solve the issues of cloud queues using concepts of distributed data structures and generation of identification numbers in a distributed system.

An often advertised property of a cloud is its elastic or scalable structure which requires the application to be designed with these properties in mind. To enable an application to scale in a cloud environment it must be divided into modules which each are run on separate cloud instances. The process of parallelizing an application is already hard in the traditional local scenario, and is as shown in the results to be even more complicated in a distributed setting. It is no longer possible to use simple and fast local data structures since the application is distributed over multiple instances. As is shown in the result this has substantial performance implications, especially when used in a single threaded application.

A proposal for future work would therefore be to research and generalize the process of modularizing cloud based applications. This way one could automatically analyze different modularization method's potential.

6.2 Testing

Looking at the results presented in *Results* (chapter 5) one can draw the conclusion that migrating or implementing a test suite for the cloud is possible. The results presents a concrete case of implementing a running test build on a cloud environment as well as the ability of testing the cloud application as a whole externally from some other test program. Experiences during development of the system described in *Case study system* (chapter 4), presented some new and valuable considerations presented in the results. These considerations do not necessarily reflect a general case and could possibly be quite restricted to this specific system. However, some of the aspects are quite general which presents the possibility of adapting at least thoughts and considerations that could be valuable to keep in mind for any new, to be developed, distributed system.

Given the fact that the system have been successfully migrated and tested in a cloud strengthens the claims presented in the results, and also presents a concrete case overview of implementation which can be adaptable for other systems then the one presented in this report. The system has been practically tested with live data from within a production environment at Enfo Zystems, providing a lot of valuable both new considerations and re-considerations to earlier implementations. The build deployed were successfully tested with a test program implemented as described in *Case scenario* (section 5.4.3), running an implementation of a Selenium test program in parallel with data input testing verifying the solution. The research on fuzz testing (data input testing) can be expanded in order to better cover GUI testing or the case of a cloud solution. It can e.g. look at more specialized input tests aimed at common cloud based services or cloud hosts.

Having practically tested the system showed that these implementations are easily adaptable to be tested automatically, especially user testing in parallel with fuzz testing. Given the nature of these tests, the input data can either be totally random or generated to resemble actual correct data. The response can then be monitored in order to detect errors such as null pointer references, memory leaks, format conversion errors and buffer overflows. However, there are uncertainties with fuzz testing in if it actually covers a large enough input space to be relied upon. If the input space is very large it is infeasible to test all combinations of data and it is therefore better to use semi-random generation of input data to limit the test space. Unless every possible combination of inputs is executed the result of this test only serves as an estimate of the system's resilience. Another disadvantage is that the tests do not necessarily present the shortest chain of events to reach an invalid state. It can therefore be hard to debug the problem if it depends on several data inputs.

Some of the considerations presented in this report have not been strengthened by any practical solutions. *Extending cloud testing* (section 5.4.2) elaborates on the potential differences between a local and cloud environment without providing a potential solution or implementation example. However, these considerations are the result of observations made during migration and development, and are only intended to provide considerations for anyone planning on moving ones system to the cloud.

Selenium WebDriver

Selenium WebDriver brings some really interesting and valuable testing features to the web-development sector with a well-defined API and easy usability. Still there are some convenience issues that should be addressed in sense of handling synchronized method calls. There is a workaround for the more advanced setting that allows you to make what is similar to a synchronized call. WebDriver is natively asynchronous and does not take latency issues into account. This will complicate things when writing operations that are dependent on prior information that might not have been returned due to latency. There is a workaround to this problem by wrapping the intended element fetch in an explicit wait block. This wait block in its turn is a function which periodically checks if the element has been returned, and not until the element has returned will the function return as result to the original fetch. This brings a lot of overhead to your written code and should be implemented in a more convenient way.

Another inconvenience is that the DOM object is reset each time a web page is refreshed or redirected. This means that data must be manually retrieved from the DOM object before it is reset, which adds additional repetitive code. References to HTML elements must be re-queried after a page is reloaded.

Even though WebDriver has some unfinished quirks it is still a good testing methodology for web-applications. Given that the web-application also resides in the cloud, one can still run the test case on the cloud-server which can give you indications of other errors in the environment. Given the scenario that something goes wrong while testing the application from outside the cloud; an administrator could run the test program from within the cloud checking if the error is generated by the actual application. If no error is found from the internal test, one can start wondering if the problem might be located elsewhere in communication between the cloud and workstation.

6.2 MongoDB migration plan

The method of migrating a database from a local environment to a cloud host is largely dependent on the database used. This section discusses the steps of migrating a local MongoDB database to a cloud. As discussed earlier MongoDB provides functionality such as sharding which suits excellent in a cloud environment.

This migration plan has been compiled from several successful deployments of MongoDB databases onto Amazon EC2 instances.

1. Evaluate and define metrics related to the cloud instance such as performance, storage, durability and pricing. Determine if database features such as replication shall be used.
2. Choose if the database application should be provided as a service by the cloud host or administered by you. Evaluate if any special software version or configuration is needed, and confirm they are compatible with the cloud chosen. Verify the cloud provides the needed level of reliability and uptime.
3. Prepare the database infrastructure. Create documentation of the cloud instances related to the database. Create configuration scripts and start the needed amount of cloud instances.
4. Configure the cloud system and software. Assign static host names to all instances participating in a shard or replica set.
5. Set up a test database in the chosen cloud and assure it meets the requirements from step 1. Use test cases that as close as possible simulates real system load, for most reliable measurements. Go back to step 2 if the performance is not satisfactory.
6. Evaluate live data and choose a good shard key based on the access pattern and data distribution of existing data. The key cannot be easily changed and it is therefore vital this choice is well thought-out.
7. Create a metric used to determine if a new shard needs to be added. This can involve database performance or storage space. As adding a new shard temporarily puts additional strain on the database a new shard must be added well in time before the database reaches its hard limit.
8. Enable database sharding from the start, even if it is not immediately used. This allows expansion of both database performance and storage space while increasing the database infrastructure.
9. If using replication the applications using MongoDB must be made aware of this. As the replicas are eventually consistent with the master database their content may differ. An application querying MongoDB must explicitly allow reads from replica databases to indicate it allows eventual consistency reads.

7. Conclusion

The purpose of this report is to evaluate and highlight considerations for migrating a distributed web-application to the cloud. Our develpoment of the system named Track and Trace has laid as a foundation to results and observations presented in this report. It addresses differences between services provided in a cloud in relation to their local counterparts, and how the implementation of a test suite might need to be reconsidered when migrated to the cloud. This report provides the reader with knowledge about some of the aspects of cloud migration, focusing considerations in terms of scalability and testability.

Migrating an application with the intentions of utilizing the elastic properties of a cloud, considerations in terms of application's scalability need to be taken. The design of the application must be reviewed to determine its modular- and parallelizability potential. This is important in order to assure that the design actually will scale as expected in a cloud environment. In order to assure scalability, care must be taken in order not to limit the application in a distributed environment. Modules should therefore minimize the amount of shared data and dependencies as communication in a distributed system can be costly in terms of performance. As the performance tests have shown in Performance and scalability (section 5.1.3) the usage of network connected data structures can severely limit the performance. We therefore conclude that the primary target for modularization should be modules which shares no, or seldomly, data with other modules, or can make use of multiple concurrent connections to a network storage in order lessen the impact the network has on the performance. The Track and Trace application developed, on which this report is built upon, was made into two separate modules. When reviewing performance measurements of network data structures it became clear that this was not a viable solution.

Services within a cloud and services running in a local environment will not necessarily behave in the same fashion. The ability for a service to scale performance-wise is a necessity in order to assure that it will be able to adopt the elastic properties of a cloud. The cloud provider might therefore need to change the service in order to support scaling, which gives a different behavior compared to a local service. The database MongoDB, described in Database (section 5.1.2.1) will if used with the replication feature have the eventual read consistency property which is uncommon in local services. One need to investigate if this is an acceptable property prior to commencing a migration. The Amazon SQS queue manager, described in Message queue (section 5.1.2.2) does not guarantee that distributed queues are synchronized and therefore have some unusual properties. A message can be delivered more than once and out-of-order, which requires the client to take measurements to correct this behavior. The can add additional data to each message or store temporary messages locally, in order to identify it uniquely or to allow it to be sorted in the correct order. This report provides simple, yet practical, suggestions for solving these problems. It discusses for example how unique identification numbers can be generated effectively in a distributed setting and how this fact can be used to solve the mentioned problems without using performance expensive distributed data sharing.

Having developed and migrated a complete system with the intention of adapting it to a cloud environment we conclude that in order to assure sufficient test coverage in the cloud, one does not necessarily need to reconsider existing test cases but rather expand them in order to cover newly introduced aspects in the new cloud environment. While unit tests often does not require any major change, integration and system testing usually need reevaluation or expansion in order to cover the new aspects. This should be done as performance, security and services utilized in a cloud environment might differ greatly compared to a local environment.

Providing a level of end-to-end testing for the complete system is possible with some adaptation for the case study system. The main problem solved was the ability of combining low level testing such as unit testing with the high level testing that is user testing, and perform it in a automated fashion. This was solved by separating the user testing from the rest of the test suits and construct a externally running test program triggering internal test cases externally from the cloud. As this worked for our system we claim that it is also possible with adaptation for any system sharing the same type of loosely coupled architecture.

Having specified the actual use and intention of the application in a detail, with a good idea of requirements for the finally deployed cloud application one will have an easier transition in migrating an application from the local setting to the cloud. As always, specification and research are key elements for any development to be successful which are hard during the planning phase. However, having shared some of these experiences in this report, we hope that some considerations can be evaluated even at an early stage of a new project.

7.1 Future work

The migration process has given us an insight to the potential problems migrating an application to a cloud. One must take care to understand how an application can be affected by utilizing cloud based services and how these new complications can be solved. Cloud services, compared to local services, display different behavior due to their distributed and elastic properties. To validate that the migrated application still works as expected one should make use of integration- and system testing.

Although the results in this report reflect a single case of migration, many of the considerations and solutions presented can be used in other contexts. If an application is to be migrated to a cloud, whether it uses cloud services or not, the finding are general enough to be used as guidelines for future work.

7.2 Extensions

The general approach of this report highlights a few, but far from all approaches in dealing with migration to a cloud. Interesting future work within this field would be to have a more specific approach, concentrating on one field of application. One interesting field would be to investigate effectiveness in scaling for the different database and storage solutions supported by the cloud provider. Another proposal for future work would be to research and generalize the process of modularizing cloud based applications. This way one could automatically analyze different modularization method’s potential.

Bibliography

[1] M. A. Babar and M. A. Chauhan, "A Tale of Migration to Cloud Computing for Sharing Experiences and Observations", *SECLOUD '11, pp. 50-56,* 2011.

[2] C. Mikalsen, "Moving into the Cloud", Department of Informatics, University of Oslo, Oslo, Master thesis, 2009.

[6] N. Antonopoulos and L. Gillam, "Cloud Computing - Principles, Systems and Applications", 1 ed., London: Springer-Verlag, 2012.

[7] P. Mell and T. Grance, "The NIST Definition of Cloud Computing", SP800-145, National Institute of Standards and Technology, Gaithersburg, 2011.

[8] D. Agrawal, K. S. Candan and W. Li, "New Frontiers in Information and Software as Services - Service and Application Design", 1 ed., Berlin: Springer-Verlag, 2011.

[10] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM Computing Surveys,* vol. 15, no. 4, pp. 287-317, 1983.

[11] H. Garcia-Molina, J. D. Ullman and J. Widow, "Database Systems: The Complete Book", 2 ed., Upper Saddle River, N.J, USA: Pearson Prentice Hall, 2009.

[12] K. Banker, "MongoDB in Action", 1 ed., New York: Manning Publications Co, 2012.

[17] G. J. Myers, "The Art of Software Testing", 2 ed., Hoboken, N.J, USA: John Wiley & Sons, 2004.

[29] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", AFIPS spring joint computer conference, Atlantic City, 1967.

[37] C. Ibsen and J. Anstey, "Camel in Action", 1 ed., Stamford: Manning Publications Co, 2011.

Online references

[3] Microsoft, " SOA in the Real World", [Online]. Available: http://msdn.microsoft.com/en-us/library/bb833022.aspx. [Accessed 18 04 2012].

[4] Mule Soft, "Eliminating Point-To-Point Integration Pain with Mule ESB", 2012. [Online]. Available: http://www.mulesoft.org/eliminate-point-to-point-integration-mule-esb. [Accessed 18 05 2012].

[5] Mule Soft, "What is Mule ESB", 2012. [Online]. Available: http://www.mulesoft.org/what-mule-esb. [Accessed 18 05 2012].

[9] Amazon Web Services LLC, "Amazon Elastic Compute Cloud (Amazon EC2)", 2012. [Online]. Available: http://aws.amazon.com/ec2/. [Accessed 14 05 2012].

[13] R. Murphy and M. Gill, "MongoDB - Schema Design", 2011. [Online]. Available: http://www.mongodb.org/display/DOCS/Schema+Design. [Accessed 10 02 2012].

[14] D. Merriman, "MongoDB - Philosophy", 2011. [Online]. Available: http://www.mongodb.org/display/DOCS/Philosophy. [Accessed 10 02 2012].

[15] T. Hannan, "MongoDB - Introduction", 2011. [Online]. Available: http://www.mongodb.org/display/DOCS/Introduction. [Accessed 10 02 2012].

[16] K. Banker, "MongoDB Fundamentals", 2012. [Online]. Available: http://docs.mongodb.org/manual/faq/fundamentals/. [Accessed 02 05 2012].

[18] Microsoft, "MSDN Integration testing", 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa292128%28v=vs.71%29.aspx. [Accessed 13 02 2012].

[19] Microsoft Corporation, "Testing Methodologies", 2005. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff649520.aspx. [Accessed 02 02 2012].

[20] Y. Sikiri, "msdn.microsoft.com", 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc194885.aspx. [Accessed 02 02 2012].

[21] Selenium Project, "Selenium 2.0 WebDriver", 2012. [Online]. Available: http://seleniumhq.org/docs/03_webdriver.html. [Accessed 14 03 2012].

[22] D. Crockford, "RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON)", 2006. [Online]. Available: http://tools.ietf.org/html/rfc4627. [Accessed 16 04 2012].

[23] G. Magnusson Jr and B. Hackett, "MongoDB - BSON", 2011. [Online]. Available: http://www.mongodb.org/display/DOCS/BSON. [Accessed 03 02 2012].

[24] BSON, "BSON - Binary JSON", 2011. [Online]. Available: http://bsonspec.org/. [Accessed 03 02 2012].

[25] Amazon Web Services LLC, "Amazon SimpleDB", 2012. [Online]. Available: http://aws.amazon.com/simpledb/. [Accessed 11 05 2012].

[26] Amazon Web Services LLC, "Amazon DynamoDB," 2012. [Online]. Available: http://aws.amazon.com/dynamodb/. [Accessed 11 05 2012].

[27] E. Horowitz and K. Chodorow, "MongoDB - Choosing a shard key", 2012. [Online]. Available: http://www.mongodb.org/display/DOCS/Choosing+a+Shard+Key. [Accessed 03 05 2012].

[28] Amazon Web Services LLC, "Amazon Simple Queue Service", 2011. [Online]. Available: http://aws.amazon.com/sqs/. [Accessed 14 03 2012].

[30] Memcached, "memcached - a distributed memory object caching system,", 2009. [Online]. Available: http://memcached.org/. [Accessed 25 05 2012].

[31] Oracle Corporation, "Collections (Java Platform SE 6)", 2011. [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html. [Accessed 25 05 2012].

[32] S. Gulati, "How MongoDB Different Write Concern Values Affect Performance On A Single Node?", 2011. [Online]. Available: http://whyjava.wordpress.com/2011/12/08/how-mongodb-different-write-concern-values-affect-performance-on-a-single-node/. [Accessed 18 05 2012].

[33] R. Murphy and D. Merriman, "MongoDB - Object Ids", 2011. [Online]. Available: http://www.mongodb.org/display/DOCS/Object+IDs. [Accessed 10 04 2012].

[34] S. Kleinman and D. Merriman, "MongoDB - Sharding Introduction", 2012. [Online]. Available: http://www.mongodb.org/display/DOCS/Sharding+Introduction. [Accessed 15 04 2012].

[35] SpringSource, "Grails - The search is over," 2011. [Online]. Available: http://grails.org. [Accessed 10 02 2012].

[36] S. Leer, "Getting Groovy in a SOA", Hogeschool van Arnhem en Nijmegen, Arnhem, Nijmegen, 2009 [Online]. Available: http://www.gayadesign.com/scripts/uploads/researchpaper.pdf. [Accessed: 02 03 2012]

[38] Apache, "Apache Camel", 2011. [Online]. Available: http://camel.apache.org/. [Accessed 10 02 2012].

[39] Amazon Web Services LLC, "Amazon Linux AMI", 2012. [Online]. Available: http://aws.amazon.com/amazon-linux-ami/. [Accessed 25 05 2012].

[40] Amazon Web Services LLC, "Amazon EC2 Instance Types", 2012. [Online]. Available: http://aws.amazon.com/ec2/instance-types/. [Accessed 24 05 2012].

Appendix

A. MongoDB ObjectID

ObjectID is the default data type used as an ID by MongoDB and is used to identify an object. It does not have any property that guarantees uniqueness of documents but instead gives a high probability of uniqueness. Each ObjectID is most often created by the application, or rather the driver, inserting new documents into a database, but the database will create such an ID if it is missing.

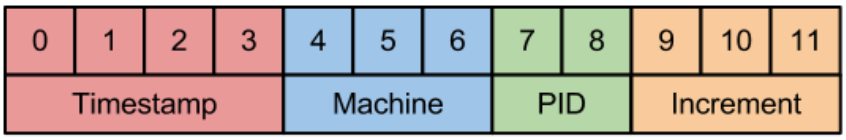

Figure 12. ObjectID structure showing the four different fields. Based on **[33]**

As seen in figure 12 an ObjectID consists of four parts, totaling 12 bytes.

- **Timestamp**. UNIX timestamp representing the number of seconds since January 1, 1970 (UTC).
- **Machine**. The first three bytes of the MD5 hash of the machine host name, MAC/IP address or virtual machine ID.
- **PID**. Process or thread identifier of the process generating the ObjectID.
- **Increment**. Incrementing or random number.

As none of the fields by itself is unique, MongoDB relies on the fact that the possibility of a collision is very small. Both the machine ID and PID will most likely differ between multiple clients accessing the database.

Both the timestamp and increment field must be in big-endian to ease the process of inserting new documents. Due to how the indexes are created using a B+ tree it is more efficient to insert values with increasing order. It requires less computations and less amount of the index to be available in RAM memory. Another result of storing the timestamp in the most-significant-bytes is the possibility to order documents by insertion time without using a special timestamp field.

B. MongoDB architecture

The architecture of a MongoDB database is dependent on the features used. The simplest configuration is a single database but addition of sharding and/or replication adds additional complexity.

MongoDB consists of several processes, see [34].

- **Daemon:** The core MongoDB process, running the *mongod* executable. Handles database management, replication and in simple configurations connection to clients.
- **Sharding controller:** Only used in sharded configurations. Running the *mongos* executable. Front end for clients to connect to instead of the *mongod* processes. Handles routing and coordination of the sharded *mongod* processes in order to give clients the view of a single system.
- **Config server:** Only used in sharded configurations. Running a special configuration of the *mongod* process. Holds metadata related to the shaded data. Is connected to the *mongos* processes in order for them to route reads and writes to the correct *mongod* process. If used,

there must be either one or three config servers to support the distributed locking mechanism used.

- **Arbiter:** Optionally used together with replication. Running a special configuration of the *mongod* process. Used when the number of replication nodes are even and an additional node needs to be added in order to break the symmetry and allow voting. Voting happens when the master replication node is found to be unreachable and a new one has to be elected.

In the simplest case only a daemon process is used. It handles all database management and does not provide sharding nor replication. Clients connect directly to this process.

Adding sharding complicates the architecture somewhat. There is no change from the client's perspective as the details are hidden by the sharding controller. One or more sharding controller can coexist and are each connected to the daemon processes and config servers. The sharding controller uses the metadata from the config servers to route client requests to the correct daemon processes. The shard controller updates each config server if the database needs to be balanced by moving excess data between two daemon processes. This is done using a distributed locking mechanism in order to guarantee consistency.

Replication is the only feature that requires the client to know about the underlying architecture. Normally a client only knows about one daemon or shard controller to connect to, but to provide redundancy one or more *replica sets* must be given to the client if using replication. How the client connects to the replication sets are driver dependent but the most common method is to connect to them in the order they are defined. If a connection cannot be established to the first replica set the driver moves to the next.

A figure representing an architectural overview of a MongoDB system using both sharding and replication can be seen in figure 13. The three upper boxes represent shards, which is replicated to the boxes below. Notice the absence of an arbiter since the number of replica nodes is odd.

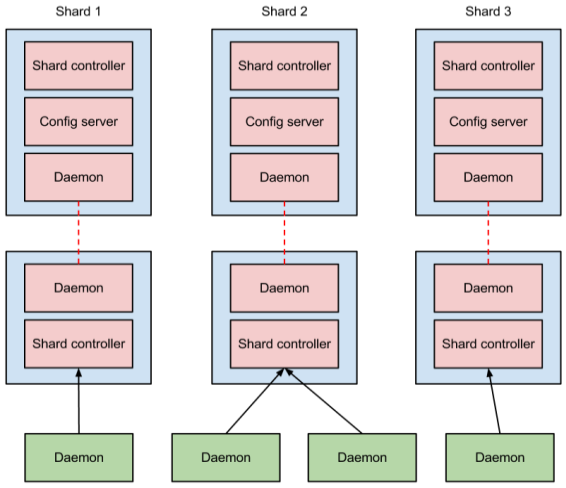

Figure 13. MongoDB architecture using sharding and replication **[31]**

C. Framework

This is an extension to the technical background describing the framework used when developing the application. This section does not have any concrete connection to the results of the report; it is strictly included in interest of sharing additional information in what techniques that have been used when developing the application.

Grails framework

The Grails framework is defined by [35] as a stable and powerful framework for building dynamic and consistent Java applications. It is built on the Groovy programming language with all its benefits and also makes use of being built on top of Spring which introduces a lot of possibilities of integrating other new components (topology presented in figure 14). Grails works out of the box with a standard development environment supporting Java applications with possibilities to integrate all libraries and plugins one might need in one's environment. Grails has a lot of powerful features built in such as easy use of scaffolding, CRUD and database connectivity for a wide range of components as well as a ready to use web server for easy commence.

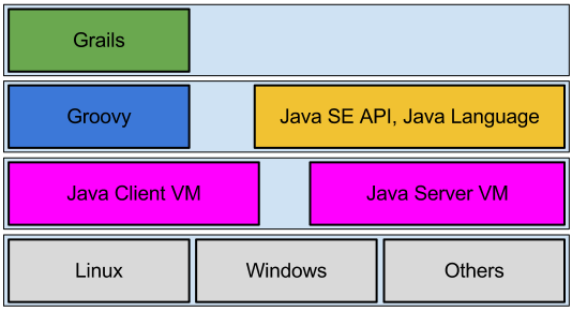

Figure 14. Grails is built on top of Groovy **[36]**

Groovy

Grails is built on the dynamic language Groovy which is an object-oriented language built for the *Java Virtual Machine* (JVM), see [35]. Groovy provides an easy to use language with its core in Java, with additional features from languages such as Ruby and Python. Groovy deviates from Java in being very compact in its syntax and supporting some interesting features such as closures and dynamic typing. Since Groovy has its core in Java it can therefore use all features existent in Java including all Java libraries for high usability. With this it is also interpreted as Java byte code when compiled and can therefore be used in any Java environment.

Apache Camel

Camel is an integration framework providing a powerful route-builder with aim to glue communicating end-points together. With its own integration language camel provides powerful features for defining complex routes and integrating these routes into one's web-applications. A key principle of camel is that it does not assume anything in terms of what format or volume of data that is to be sent within the route. It brings a high level of abstraction which gives the system the possibility of using the same API regardless the form of communication or type of data. This information and a more detailed documentation are found in [37].

Camel makes use of so called “Bean Binding”, see [38], in order to bind its route to a designated end-point for processing. Since the data is passed in whatever format it was initially sent, the data can just be forwarded to the designated end-point and from there be processed in whatever fashion suitable. Camel is well compatible with Spring, see [38], and is therefore very suitable for integration with Grails framework which also provides support for Camel with corresponding plugins.

D. Amazon EC2 instances

This section lists the technical specification of the Amazon EC2 instances used in this report. The operating system used is Amazon Linux AMI (64-bit) [39]. The following list is based on Amazon's official documentation found at [40].

Medium instance (m1.medium)

- 3.75 GB RAM
- 2 EC2 Compute Units
- 410 GB instance storage
- 64-bit platform
- Moderate I/O performance

Micro instance (t1.micro)

- 613 MB RAM
- Up to 2 EC2 Compute Units
- 64-bit platform
- Low I/O performance