DEGREE PROJECT IN COMMUNICATION SYSTEMS, FIRST LEVEL
STOCKHOLM, SWEDEN 2015

Automatic Log Analysis System Integration

Message Bus Integration in a Machine Learning Environment

CARL SVENSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

Automatic Log Analysis System Integration

Message Bus Integration in a Machine Learning Environment

Carl Svensson

2015-06-08

Bachelor's thesis

Industrial advisor:
Weixi Li

Examiner and academic advisor:
Gerald Q. Maguire Jr.

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

Abstract

Ericsson is one of the world's largest providers of communications technology and services. Reliable networks are important to deliver services that live up to customers' expectations. Tests are frequently run on Ericsson's systems in order to identify stability problems in their networks. These tests are not always completely reliable. The logs produced by these tests are gathered and analyzed to identify abnormal system behavior, especially abnormal behavior that the tests might not have caught. To automate this analysis process, a machine learning system, called the Awesome Automatic Log Analysis Application (AALAA), is used at Ericsson's Continuous Integration Infrastructure (CII)-department to identify problems within the large logs produced by automated Radio Base Station test loops and processes. AALAA is currently operable in two versions using different distributed cluster computing platforms: Apache Spark and Apache Hadoop. However, it needs improvements in its machine-to-machine communication to make this process more convenient to use. In this thesis, message communication has successfully been implemented in the AALAA system. The result is a message bus deployed in RabbitMQ that is able to successfully initiate model training and abnormal log identification through requests, and to handle a continuous flow of result updates from AALAA.

Keywords

Big Data, Machine learning, Message passing, Machine-to-machine communication

Sammanfattning

Ericsson är en av världens största leverantörer av kommunikationsteknologi och tjänster. Tillförlitliga nätverk är viktigt att tillhandahålla för att kunna leverera tjänster som lever upp till kundernas förväntningar. Tester körs därför ofta i Ericssons system med syfte att identifiera stabilitetsproblem som kan uppstå i nätverken. Dessa tester är inte alltid helt tillförlitliga, producerade testloggar samlas därför in och analyseras för att kunna identifiera onormalt beteende som testerna inte lyckats hitta. För att automatisera denna analysprocess har ett maskininlärningssystem utvecklats, Awesome Automatic Log Analysis Application (AALAA). Detta system används i Ericssons Continuous Integration Infrastructure (CII)-avdelning för att identifiera problem i stora loggar som producerats av automatiserade Radio Base Station tester. AALAA är för närvarande funktionellt i två olika versioner av distribuerad klusterberäkning, Apache Spark och Apache Hadoop, men behöver förbättringar i sin maskin-till-maskin-kommunikation för att göra dem enklare och effektivare att använda. I denna avhandling har meddelandekommunikation implementerats som kan kommunicera med flera olika moduler i AALAA. Resultatet är en meddelandebuss implementerad i RabbitMQ som kan initiera träning av modeller och identifiering av onormala loggar på begäran, samt hantera ett kontinuerligt flöde av resultatuppdateringar från pågående beräkningar.

Nyckelord

Big Data, Maskininlärning, Meddelandesändning, Maskin-till-maskin kommunikation

Acknowledgments

I would like to show my greatest appreciation to my supervisor Weixi Li, manager Jens Wictorinus, and other colleagues at Ericsson for giving me this opportunity and helping me out during this thesis. I would also like to thank Professor Gerald Q. Maguire Jr. for giving great advice on the report writing and supporting me during the thesis process.

Stockholm, June 2015
Carl Svensson

Table of contents

Abstract i
Keywords i
Sammanfattning iii
Nyckelord iii
Acknowledgments v
Table of contents vii
List of Figures ix
List of Tables xi
List of acronyms and abbreviations xiii
1 Introduction 1
1.1 Background 1
1.2 Problem definition 1
1.3 Purpose 2
1.4 Goals 2
1.5 Choice of methodology 2
1.6 Delimitations 3
1.7 Structure of the thesis 3
2 Background 5
2.1 Awesome Automatic Log Analysis Application (AALAA) 5
2.1.1 AALAA Core 5
2.1.2 AALAA Client 6
2.2 RabbitMQ 6
2.2.1 Advanced Message Queuing Protocol (AMQP) 7
2.2.2 Open Telecom Platform (OTP) 7
2.3 Related work 8
2.3.1 Performance evaluation of RESTful web services and the AMQP protocol 8
2.3.2 Building a Scalable Event Processing System with Messaging and Policies 8
3 Methodology 9
3.1 Research Process 9
3.2 Research Paradigm 10
3.3 Data Collection 10
3.4 Experimental design 10
3.4.1 Test environment 10
3.4.2 Hardware/Software to be used 11
3.5 Assessing reliability and validity of the data collected 11
3.5.1 Reliability 11
3.5.2 Validity 11
3.6 Planned Data Analysis 11

3.7 Evaluation framework 12
4 Implementation 13
4.1 Software design 13
4.1.1 Design of the AALAA system 13
4.1.2 Design of the Message Bus 16
4.2 Software implementation 19
4.2.1 AALAA Core implementation 19
4.2.2 AALAA Client implementation 20
5 Analysis 21
5.1 Major results 21
5.2 Reliability Analysis 23
5.3 Validity Analysis 23
5.4 Discussion 24
6 Conclusions and Future work 25
6.1 Conclusions 25
6.2 Limitations 26
6.3 Future work 26
6.4 Reflections 26
References 27

List of Figures

Figure 2-1: Overview of the AALAA Core [3] 6
Figure 2-2: Simple example of how messages can be routed through RabbitMQ [16] 7
Figure 4-1: Planned architechture of the AALAA system 15
Figure 4-2: Example of a request message sent to the AALAA Core 18
Figure 4-3: Example of a progression update message sent from the AALAA Core 18
Figure 4-4: More detailed overview of the implemented message bus including flow of events 18
Figure 5-1: Time to initiate request in AALAA core based on message size 22

List of Tables

Table 3-1: Hardware specifications of the used test bed 11
Table 5-1: Time to initiate request in AALAA core based on message size. 95% confidence level applied. 21
Table 5-2: Difference checks for a predict action between the old and new Spark cores 22
Table 5-3: Difference checks for a predict action between the old and new Hadoop core 23

List of acronyms and abbreviations

AALAA	Awesome Automatic Log Analysis Application
AMQP	Advanced Message Queuing Protocol
CI	Continuous Integration
CII	Continuous Integration Infrastructure
JSON	JavaScript Object Notation
MOM	Message Oriented Middleware
OTP	Open Telecom Platform
RBS	Radio Base Station
REST	Representational State Transfer
RRS	Result Reporting System
UUID	Universally Unique Identifier

1 Introduction

Big data is a term associated with large and complex data sets that are inconvenient to process with traditional methods. Such data sets have recently become an interesting area for analysis. Today many companies are using analytic methods on big data to identify possible improvements for various kinds of appliances [1]. Ericsson, as a world leading provider of communications technology and services, every day faces challenges in delivering high-performing & cost-efficient networks and infrastructure [2]. Automated test loops and processes are regularly run within their networks to identify errors. However, it has been shown that it is very hard to write sufficient test cases to identify all abnormal behavior in these test runs. Because of this, large amounts of valuable log data from performed tests are used by Ericsson's CII-department to improve the whole testing process by using this data as input to an automatic machine learning system. This systems is called the Awesome Automatic Log Analysis Application (AALAA) [3]. By using a self-learning system to process large quantities of logs, abnormal behavior in logs that are similar to *known* faults can be identified [3].

This thesis project was carried out at Ericsson. It aims at implementing a message bus that will be capable of communicating with two different distributed clustered computing implementations of the AALAA system: Apache Hadoop and Apache Spark. This communication will be improved by implementing message communication, rather than the earlier polling approach when initiating requests and retrieving progression updates from ongoing calculations.

1.1 Background

The AALAA-system is divided into two main parts: *AALAA Client* and *AALAA Core*. The client is responsible for initiating requests in the core. The core applies machine learning techniques on provided logs. Two different versions of the core have been implemented. The first version is based on Apache Hadoop and the second one is based on Apache Spark. Apache Hadoop is an open source framework that allows distributed processing of big data across clusters of computers. It uses two main components in its implementation: the Hadoop Distributed File System (HDFS) for storage and the programming model MapReduce for processing [4]. Apache Spark is an open source framework similar to Hadoop, however rather than using a disk-based MapReduce model, it allows data to be directly loaded into memory for repeated queries [5]. This provides quicker processing of data that is regularly accessed in the clusters, with a speedup of up to 100x in memory performance and 10x in disk performance [6]. The initial version of the core was implemented in Hadoop [3], but because the performance was insufficient (mainly due to sequential executions and several passes of input data stored on disk) a Spark version was developed [7].

1.2 Problem definition

At present, both Hadoop and Spark versions of the AALAA Core are implemented. A REST-service is currently used to communicate between the AALAA Client to remotely initiate requests in the Hadoop core, while the Spark core currently is standalone – hence it requires manual initiation. The results computed by the Hadoop version are stored in a database deployed using MongoDB, while the results of the Spark version are stored in a local output file. The problem addressed in this thesis project is to integrate the client with both the Hadoop and Spark versions of the core using a RabbitMQ message bus. This will enable request initiations using input data provided by the client, such as logs to be processed, along with the training or prediction action that will be used to process this data. The message bus should also be used to update results continuously to the initiating client

– i.e., the client should receive outputs as the computation progresses. The final results should be stored by the client into a MongoDB database.

1.3 Purpose

The aim of this thesis project is to investigate how a machine learning system for big data handling can be expanded & improved, as well as to reflect on the chosen implementation and how the expansion will affect the usage of the system. The purpose of the project is to improve AALAA by implementing a better communication method in order to enable the users of the system to choose which version of the AALAA Core to use when they send a request.

1.4 Goals

The main goal is to enable the user to send requests destined for either or both versions of the AALAA Core. These requests will be handled instantly if resources are available. Collecting results and visualizing the data is an important part of this goal. This goal will be achieved by adapting both the initiating client and the current versions of the cores to be compatible with a message bus.

The following sub-goals were defined for the project:

1. Understand the current implementation of the Hadoop and Spark cores,
2. Integrate both versions of the core with a message bus,
3. Listen to new test data by integrating the client with a Result Reporting System (RRS),
4. Modify the client to be able to initiate requests and receive results,
5. Enable results updating in MongoDB through the client,
6. Combine all of the above parts and deploy the system,
7. Evaluate the resulting system.

1.5 Choice of methodology

The selection of paradigm for designing, implementing, and evaluating the solution developed in this thesis fell on a Design Science oriented approach. Design Science is a methodology within information systems that focuses on how to solve important problems in an unique way or how to improve existent solutions in a more efficient way [8]. This proved to be a relevant methodology since the main goal was to improve the AALAA system in terms of design and implementation in order to achieve better functionality and better performance. This approach made use of both qualitative and quantitative methods for defining the objective of the solution and to evaluate the proposed solution through functional correctness and performance tests. The questions to be addressed by the implemented artifact are: (1) Does the proposed solution solve the defined problem? and (2) Is the solution sufficient in terms of performance?

An agile programming methodology was used while developing all of the software relevant to the proposed solution. The agile programming methodology is commonly used within software development when adaptive planning, continuous improvement, and flexible response to changes are emphasized [9]. Important advantages associated with this programming method is that a minimal amount of administrative overhead is incurred while planning and executing the work, and throughout the entire process receiving constant feedback from the product's owners regarding the implemented parts. Agile is also convenient to use because of its ability to be tailored to meet the different needs of a project. For example, since this project was carried out by a single developer,

only the relevant and applicable parts of the agile method were utilized. The core agile features used while developing the software consisted of a backlog with tasks to implement, a schedule with weekly goals, and an iterative approach with small steps at a time.

A possible methodology for software development that was not chosen for this project is the waterfall model. This method is based on a sequential approach in contrast to agile's iterative approach, which makes it less flexible and less open to changes during the development process. The name waterfall comes from the following five steps that are executed in sequential order during the development process: requirements, design, implementation, verification, and maintenance [10]. The waterfall methodology was not considered appropriate for this thesis project because of its large overhead, complex process, and inability to respond to rapid changes.

1.6 Delimitations

This thesis will focus on improving communication between the AALAA Client and the AALAA Core, and to improve the functionality of the client. The overall logic and structure of the core will **not** be changed, and **no** improvements will be made to the performance related to the training or prediction algorithms (these are described in Section 2.1).

1.7 Structure of the thesis

Chapter 2 describes the background areas that might be relevant in order to understand this thesis. **Chapter 3** describes the methodology used while implementing and testing the artifact. **Chapter 4** presents the design decisions and implementation steps taken while developing the artifact. **Chapter 5** analyzes the work that has been done by presenting collected data and evaluating the artifact. **Chapter 6** presents conclusions drawn from this project and also presents work that has been left for the future.

2 Background

This chapter will give background information of areas that might be relevant in order to understand this thesis. In Section 2.1, a basic overview of AALAA will be given to present some of the characteristics, architecture, and functionality of the current system. Section 2.2 explains the message broker software by going into those details of RabbitMQ that were relevant for the machine-to-machine communication in the implemented artifact. Section 2.3 presents some related work on message bus implementations in larger systems.

2.1 Awesome Automatic Log Analysis Application (AALAA)

AALAA was developed at Ericsson as an attempt to solve the time consuming task of manually identifying abnormal logs collected from Radio Base Station (RBS) test runs. As described in Section 1.1, these tests are regularly run within Ericsson's continuous integrated (CI) systems in order to identify failed runs. However, these tests do not always find all of the faults.

AALAA consists of a client and a core, where the core is responsible for machine learning and the client is responsible of sending out requests to the core and receiving updates regarding the core's progress. The core was developed in two different versions: Hadoop and Spark. The core trains models by processing large batches of logs where the test outcomes are **known**, and tries to identify abnormal logs from other logs based on these models. The AALAA Core is described in Section 2.1.1 and the previous architecture of the AALAA Client is described in Section 2.1.2.

2.1.1 AALAA Core

Each AALAA Core version has two main modes available for execution: training and prediction. The training phase takes a batch of log files produced by test runs where the outcome of the tests are **known** and the status of each log is provided as *passed* or *failed*. From these logs, the algorithm builds a model by executing three steps:

1. **Feature extraction** – From each log file, information that serves as a base for the training is extracted. This data consists for example of text character occurrences, timestamp statistics, and word counts. Most of the potential improvements to the machine learning algorithm should be found here, enabling faulty logs to be more easily be identified by extracting the correct features.
2. **Feature transformation** – In this step, data normalization and dimensionality reduction are performed on the extracted features. Normalization is done in order to provide consistent statistical properties throughout the whole data set (as otherwise differences in values might be too great). Reduction is used to remove irrelevant values from the features that have been extracted, as otherwise the number of extracted attributes could be quite large.
3. **Learning a model** – The feature data is used by a variant of Self-Organizing Feature Map (SOFM) [11] to output a model. This model is used later together with the prediction algorithm.

The prediction phase uses a set of logs to predict which the abnormal logs are in that set. This prediction is compared for all logs with the best model created during a training phase, where the choice of the model is based on a scoring system called *F-score* that uses k-fold cross validation [12]. All logs in a set that have been classified as abnormal by the prediction algorithm are produced as output. Figure 2-1 (Weixi Li 2013 [3]) shows the steps taken while performing a training or prediction action.

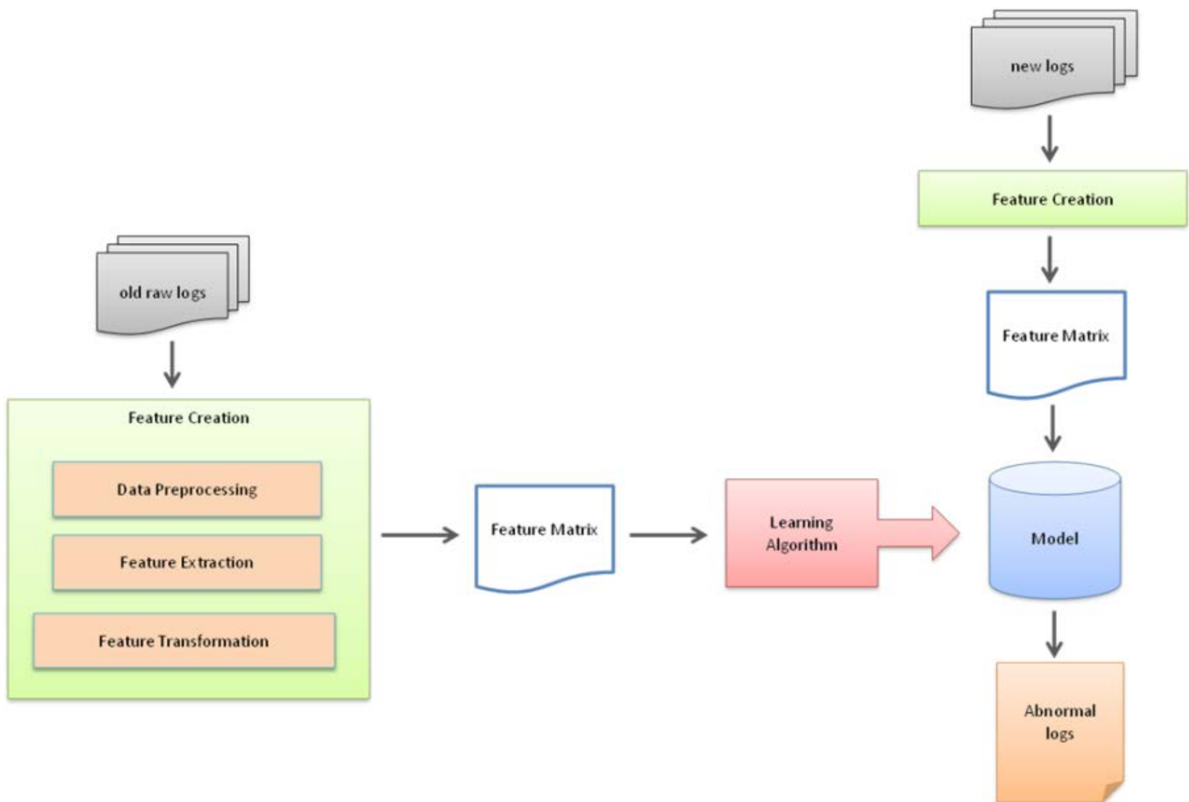

Figure 2-1: Overview of the AALAA Core [3]

2.1.2 AALAA Client

The previous AALAA Client consisted of three different modules: a *Log Collector*, a *Train Client,* and a *Prediction Client.* The Log collector is responsible for copying the raw logs that will be used by AALAA for its calculations. The Train client is used to initiate training requests to the core, while the prediction client is used to initiate prediction requests to the core. Information about batches of logs is polled every 10 minutes from a system called RRS. RRS uploads log data produced by a test case when the test case has completed. When the train or prediction client discovers a new test case, then the information needed for initiating a train or predict request to the core is stored through a REST-service into a database from which the core will subsequently poll for this data. Once certain milestones for calculations in the core have been reached, the same database will be updated with progress results directly from the core. Currently, only the Hadoop Core is integrated with the client. Further information on this will be presented in Section 4.1 starting on page 13.

2.2 RabbitMQ

RabbitMQ is a message broker written in Erlang for the purpose of translating messages from a sending protocol to a receiving protocol [13]. Message brokers are a core part of the Message Oriented Middleware (MOM) infrastructure and acts as a pattern for validating, transforming, and routing data [14]. RabbitMQ uses message passing, which in our appliance means abstracting the AALAA Client from the AALAA Core by implementing a bus with listeners that continuously listen to a flow of messages between different parts of the system. In contrast with polling that relies on waiting for another device to change state before continuing, message passing allows the producer

to send a message and then continue on as normal. It relies on the receiver of the message to take the necessary actions. RabbitMQ uses the standard protocol called "Advanced Message Queuing Protocol" (AMQP) for message passing and the Open Telecom Platform (OTP) for its server implementation. These two components are further described in Sections 2.2.1 and 2.2.2.

2.2.1 Advanced Message Queuing Protocol (AMQP)

AMQP is a standard used to enable message brokering applications to send messages between a wide variety of systems with different designs [15]. This protocol consists mainly of rules defined in a theoretical manner that applications should follow. The goal for AMQP is to enable clients and messaging middleware to communicate without needing to know anything about each other's internal design. This protocol standardizes message passing on an enterprise scale and creates a mutual ground for information exchange. AMQP is typically preferred when high quality, reliable, and safe message delivery is required. By ensuring rapid delivery with message acknowledgements, AMQP has proved to be an ideal standard for asynchronous messaging with multiple senders and recipients [15].

RabbitMQ makes use of AMQP by having senders publish their messages to an *exchange* (which can be seen as a mailbox). The exchange sends copies of these message to one or more predefined *queues* by using an internal set of rules. Any producer that is connected to a queue can at any time pull a message that has been delivered to that queue. Multiple consumers can subscribe to the same queue, enabling work to be distributed in parallel over a set of workers [16]. A simple example of message routing can be seen in Figure 2-2 [16].

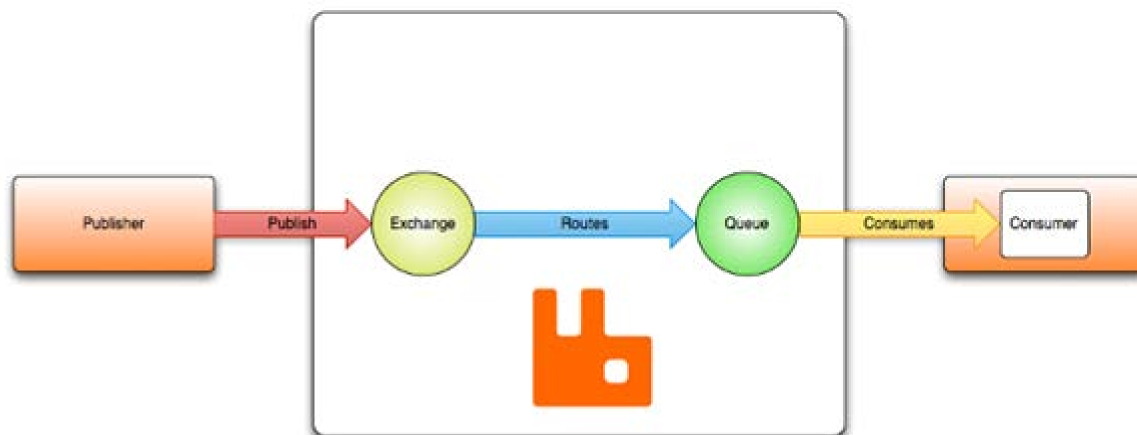

Figure 2-2: Simple example of how messages can be routed through RabbitMQ [16]

2.2.2 Open Telecom Platform (OTP)

OTP is an open source release of Erlang released by Ericsson [17]. It consists of three different components: the Erlang language itself, libraries with interfaces & tools, and a package of design principles [18]. The main goal of OTP is to provide a platform for developing uniform applications and systems with a common structure that enables easy scaling of systems in a fault tolerant way. OTP is known for its concurrent method of handling tasks, while at the same time providing high reliability. RabbitMQ makes use of OTP in the implementation of its server, thus it can take advantage of these characteristics.

This project utilizes RabbitMQ to implement message communication between the AALAA Client and Core. Message bus controllers are implemented in both the client and the core to enable

request initiations and to receive results. A further description of this architecture and implementation can be found in Chapter 4.

2.3 Related work

This section will present related work connected to messaging in larger systems, particularly those using the AMQP protocol. Many examples can be found in the research literature. In this section two relevant appliances are presented.

2.3.1 Performance evaluation of RESTful web services and the AMQP protocol

Joel L. Fernandes et al. presented a conference paper in which performance evaluations of RESTful web services were compared to the AMQP protocol [19]. Their intention was to solve the problem of simultaneously sending massive quantities of data to databases via REST services – as this can have a large negative impact on a system's general performance [19]. Their goal was to address this problem by proposing a new solution that sent messages over AMQP. Tests showed that the best architecture for exchanging large amounts of data between these two services was to setup a RabbitMQ server, use a Back-End service to receive messages from RabbitMQ queues and to store these messages in a database. This proposed architecture is similar to the planned architecture for a tracker module in the AALAA Client, where progression updates will be tracked and routed into databases, rather than having the cores directly communicating with these databases.

This work relates to both REST and AMQP. Both are parts of the task that is to be researched in this thesis project. The previous way of communication in AALAA was through a REST service storing results into a database when the client wanted to initiate training or prediction requests. The proposed new method of communication is to integrate RabbitMQ and to makes use of the AMQP protocol, as was described in Section 2.2.

2.3.2 Building a Scalable Event Processing System with Messaging and Policies

Sumit Dawar et al. presents a policy based event system which they designed and implemented as a scalable solution for handling a large number of events in a telecommunication environment [20]. Their solution implements message passing using RabbitMQ. In this distributed system, several components were connected to a message bus which makes it relevant to the appliance in this thesis as they share some modular characteristics. The purpose of the implementation in their research was to listen to events. This will also be the case in out proposed solution for AALAA as we expect the client to listen for test case completion events from the RRS system. The result of their research was a system that could handle more than 1 million events per second. Their work provides additional proof that RabbitMQ can realize high performing middleware for message routing, although AALAA currently does not require this high a throughput*.

* Tests are currently run hourly, daily, and weekly. This means approximately hundred possible events each week.

3 Methodology

The purpose of this chapter is to provide an overview of the research method used in this thesis. Section 3.1 describes the research process. Section 3.2 details the research paradigm. Section 3.3 focuses on the data collection techniques used for this research. Section 3.4 describes the experimental design. Section 3.5 explains the techniques used to evaluate the reliability and validity of the data collected. Section 3.6 describes the method used for the data analysis. Finally, Section 3.7 describes the framework selected to evaluate the implemented solution.

3.1 Research Process

The Design Science methodology used during the research process was based on a six step model presented in *"The Design Science Research Process: A Model for Producing And Presenting Information Systems Research"* by lead authors Ken Peffers and Tuure Tuunanen[21]. The steps of this model were carried out sequentially as is natural for this methodology. Below, a description of each of these steps is given, while also presenting the work process that was carried out and applied to each step.

1. **Problem identification and motivation** – The defined problem was identified and researched in order to obtain a better understanding of its scope and complexity. Once a sufficient understanding of the problem had been achieved, its importance motivated further work.
2. **Objectives of the solution** – Objectives of a potential solution to the problem were addressed to classify the problem either as qualitative or quantitative problem. A combination of these two paradigms was chosen since the solution needed both types of analysis to improve the current implementation of AALAA and to integrate new modules and functionality not previously present.
3. **Design and development –** The artifact was designed and implemented according to agile methods by using an iterative process throughout the development. The problem was divided into smaller tasks, where each task was separately implemented and evaluated. Once all of these tasks had been solved, they were put together to provide a complete solution for the defined problem.
4. **Demonstration –** The proposed solution was demonstrated through tests of functional correctness and performance by comparing the new AALAA with the old AALAA. New functionality that was previously not present was demonstrated through experiments.
5. **Evaluation –** The artifact was investigated in terms of functionality by evaluating the results of the tests performed in a demonstration. These results were compared with the previous version of AALAA and conclusions were drawn as to whether the new functionality and performance was sufficient to offer as a proposed solution or not.
6. **Communication –** The work carried out during the five previous steps is presented in this these to communicate the problem's importance and by presenting an artifact that suffices as a proposed solution.

3.2 Research Paradigm

The research paradigm associated with this project is of both a qualitative and quantitative type. This problem is fairly unique since the implementations needs to be tailored to suit the requirements of the currently implemented AALAA system. This includes changing existing modules, implementing new modules, and improving the method of communication; while still maintain the underlying functionality. Little research has been found where a problem of the same characteristics has been solved. All of the relevant information needed for building a solution was gathered from a wide arrange of sources. An inductive approach will be applied while researching background areas, a deductive approach will then be used to evaluate this research and to construct a solution that will be tested in terms of functional correctness and performance (specifically throughput).

3.3 Data Collection

The data needed to prove that the solution is feasible will be collected from tests of functional correctness and performance of the implemented artifact. Gathering proof of functional correctness is the most important part in order to ensure that the core functionality of the machine learning algorithms remains the same, even after the message bus has been implemented. It is also important to prove that the results received by the AALAA Client are correct and identical to the direct output of the AALAA Core. Data to prove functional correctness will be collected by running difference checks (comparisons) between the output data from the previous standalone versions of the program and the new version of the system. Comparisons will also be made of the direct output from the core and the messages received by the client. Performance data will be collected by measuring the time it takes to initiate a request to the core via the message bus and to receive a result. A sufficient number of tests will be made to compute average values, with a desired confidence interval of 95%. Since the standard deviation between observed test runs were moderately large (ca ± 25%), a larger amount of test runs (100) was chosen and applied for each test case. These results will be presented in Section 5.1.

3.4 Experimental design

This section describes the test environment in which all of the tests were performed. Specifications are given for all hardware, software, and data used for these tests.

3.4.1 Test environment

The tests were carried out in an environment present at Ericsson that consisted of a cluster with three different machines. These three machines were divided into one master node and two slave nodes. Both the AALAA Client and the AALAA Core were deployed in this cluster. The client operated in the master node and the core was operable in all three nodes. The requirement of having slave nodes for this implementation is due to the fact that the core uses the Apache Spark and Apache Hadoop cluster frameworks to speed up big data computations. Both the RabbitMQ message bus and the MongoDB databases used to store results were deployed in the master mode. The logs required for testing consisted of a rather small test set (~232MB in size). This was considered sufficient to test the message bus implementation. Larger quantities of logs have previously been used for testing the machine learning algorithms in the core, therefore using large logs was not considered a requirement for the testing in this thesis. This was advantageous due to long processing times when dealing with large logs [3,7].

3.4.2 Hardware/Software to be used

The hardware specifications for all three nodes can be found in Table 3-1.

Table 3-1: Hardware specifications of the used test bed

Machine	CPU	Memory	OS
Master node	Intel Xeon X5660 @ 2.80GHz, 6 cores	32 GB	SUSE Linux Enterprise Server 11, 64 bit
Slave node 1	Intel Xeon X5660 @ 2.80GHz, 6 cores	16 GB	SUSE Linux Enterprise Server 11, 64 bit
Slave node 2	2x Intel Xeon X5660 @ 2.80GHz, 6 cores	16 GB	SUSE Linux Enterprise Server 11, 64 bit

The software used while testing included the Spark and Hadoop frameworks installed on all three nodes for performing clustered machine learning in the AALAA Core. A RabbitMQ server was installed in the master node - where the message bus was also deployed. Further information about the structure of this message bus can be found in Section 4.1.2. Two MongoDB databases were installed in the master node for storing result updates coming from the communication with the AALAA Client. The three modules used for testing the implementation: Spark core, Hadoop core, and AALAA-Tracker (part of the client) - were all compiled with Maven [22] and run as .jar (Java Archive) files.

3.5 Assessing reliability and validity of the data collected

This section describes how the reliability and validity of the collected data can be assessed.

3.5.1 Reliability

Reliability of the data will be achieved by performing multiple test runs using the same parameters for each test case. Any differences in results will be evaluated and the final results that are collected will be presented as average values computed over the combined data. Parameters that might affect reliability (such as time differences when sending messages of varying sizes) will also be evaluated.

3.5.2 Validity

Validity will be assessed to ensure that the tests actually targets those areas that we intended to measure. This will be done by evaluating whether the test cases are sufficient to prove that the system is reliable and whether these test cases establish a suitable framework for measuring the system’s intended functionality and performance.

3.6 Planned Data Analysis

Data analysis of the collected data regarding its performance will be done by presenting the average times from multiple runs of each test case. These results will be compared to the old polling method of processing requests. Conclusions will hopefully be reached as to whether the newly implemented artifact improves the system’s performance and functionality or not.

While evaluating the functional correctness of the implementation, any differences in output between the old and the new system will be investigated to find and fix the root cause. Discoveries of such differences will be presented in the analysis chapter together with information about what the problem was and what done to solve it.

3.7 Evaluation framework

The previous AALAA-system was only partly integrated with the Hadoop core using remote request initiated by the client. This earlier implementation used polling to initiate requests which proved to be inconvenient due to the long times before a request could be processed by the core. The proposed solution enable full integration with the client for both the Spark and Hadoop versions of the core. A message bus is used to initiate fast requests in both versions of the core and also to send fast progression updates back to the client. The convenience of using the new system will be evaluated by comparing it with the performance and functionality of the old system. Parameters will be accounted for, such as the improved functionality due to the additional integrated modules, as well as the advantages of using a message bus rather than polling.

4 Implementation

This section will give an overview of the software that was implemented during this thesis project. It describes the main design decisions that were taken concerning the overall system architecture, as well details about more specific program functionality and how both expected and unexpected problems were solved. Section 4.1 describes the design decisions taken for the system as a whole and for the message bus that was built. Section 4.2 describes in greater detail how the software was implemented, while giving details of the modules connected to the message bus and their functionality.

4.1 Software design

This section describes the architecture of the proposed software solution. Section 4.1.1 describes the design that was planned for the AALAA-System. Section 4.1.2 gives more details of the design of the message bus, in particular describing the architecture of the message exchange that was built and the format of the messages routed through it.

The design of the software was made with regard to the initial requirements that were predefined for this project. The requirements that the design was based on, and possibly limited by, were (1) that a message bus should be implemented and (2) that multiple modules needed to be integrated with this message bus in order to maintain full functionality of the AALAA system. There was a requirement to maintain the current system's functionality, but also the aim was to provide extended functionality by integrating modules that previously were not integrated. Since all of the modules needed a message broker compatible with all parts, RabbitMQ was initially proposed as the middleware for the message bus. Further research on both the system and the general characteristics of RabbitMQ showed that RabbitMQ would provide a suitable solution, mainly due to its flexibility, publish-subscribe like structure, and near guarantee of fail-proof delivery [13]. Its appliance in the developed artifact was motivated by conducting research to find out if it would suffice to solve the targeted problem or not. These motivations were done by looking at related work that had been done with AMQP and RabbitMQ before (Section 2.3).

For the overall architecture, a proposal was laid out by Ericsson (most of which can be seen in Figure 4-1). This proposal was broadly followed, design decisions that was taken concerned the specific parts of this architecture. Decisions had to be made for example on module architecture, message bus structure and message formats. Most of the main design decisions were made in agreement with my supervisor and others at Ericsson with insight into the problem.

4.1.1 Design of the AALAA system

As can be seen in Figure 4-1 (on page 15), the proposed solution is based on a message bus implemented in RabbitMQ. This message bus connects an upper part, the AALAA Client, and a bottom part, the AALAA Core. The client is responsible for sending requests to be handled by the core and to receive continuous progress updates and results from the core as the calculations progress. The AALAA Client also handles all communication with the database for storage of progress updates and results. The core implements the machine learning techniques and does so by receiving either a training or a prediction request together with parameters (such as a batch of logs and a model type). Based on this input the core will then process the logs while continuously providing progress updates and results back to the client.

4.1.1.1 AALAA Core

AALAA consists of two different versions of the core, one implemented in Apache Spark and one implemented in Apache Hadoop. The Hadoop version was previously integrated with a REST-service for manual polling of requests from a database at certain time intervals, while the Spark version was standalone and could only be executed manually. A requirement of the new architecture was to integrate both modules with a message bus so that the monitoring client could initiate requests to either or both modules and fetch calculation updates and results for storage into a database. The Spark and Hadoop modules were implemented by different developers at different times. It was discovered that both of these core modules would require specific modifications in order to support the new message bus architecture.

4.1.1.2 AALAA Client

The approach for designing the client was different from the approach for designing the core, since the client's structure needed to be changed by not only changing the existent modules, but also by implementing new modules. The old client retrieved log paths for a completed test case, collected these logs, and decided whether to train or predict based on whether a model for that test case was already stored. The requested information would be stored into a database that the Hadoop version of the core would poll information from. The new client implements message communication to initiate executions in the core by providing log paths and a train or predict parameter in a message routed through the message bus. Logs still need to be collected by a separate module (*Log Collector)*. However, the module tracking completed test cases in the RRS system for core-initiations (*RRS MB Controller)* requires changes in order to support message communication.

Finally, a tracker responsible for listening for progress updates from the AALAA Core needed to be developed (this tracker is called *AALAA-Tracker*). The tracker's sole responsibility is to route progress updates to the correct database (Spark-DB or Hadoop-DB). These update results from the calculations are done to satisfy a request from the requesting client. Thus it later provides information to the RRS MB Controller to decide whether to send training or prediction requests. Due to the limited duration of this thesis project, only the AALAA-Tracker was implemented in the client. When testing the system, request messages to initiate core executions were manually provided directly into the message bus web interface.

4.1.1.3 Message Bus Controllers

AALAA requires reliable communication between different modules (as described in Section 4.1 and 4.2). This communication had to be handled by controllers that could route messages between these modules via the message bus. These controllers needed the functionality of opening connections with the message bus and to provide functions for modules to send and receive messages. The main principles that shaped the architecture of the controllers were to: (1) Maintain a uniform structure and (2) Make the implementation simple. Therefore, it was convenient to make the design for the controllers' uniform by implementing a single version of a controller and deploying it as an instance in each module. Only small adjustments were made to each controller to satisfy specific module requirements. By doing so, similar behavior on each side of the message bus could be guaranteed, while at the same time simplifying the logic by abstracting the controllers from the modules. Since most of the modules that existed were implemented in Java, a decision was made to implement all of the controllers in Java as well. The only exception was the Spark AALAA Core module that was written in Scala [23]. However, since Scala is based on Java and provides Java support, a Java version of the controller could easily be integrated and used with the Scala code. An overview of the message bus integration can be seen in Figure 4-1.

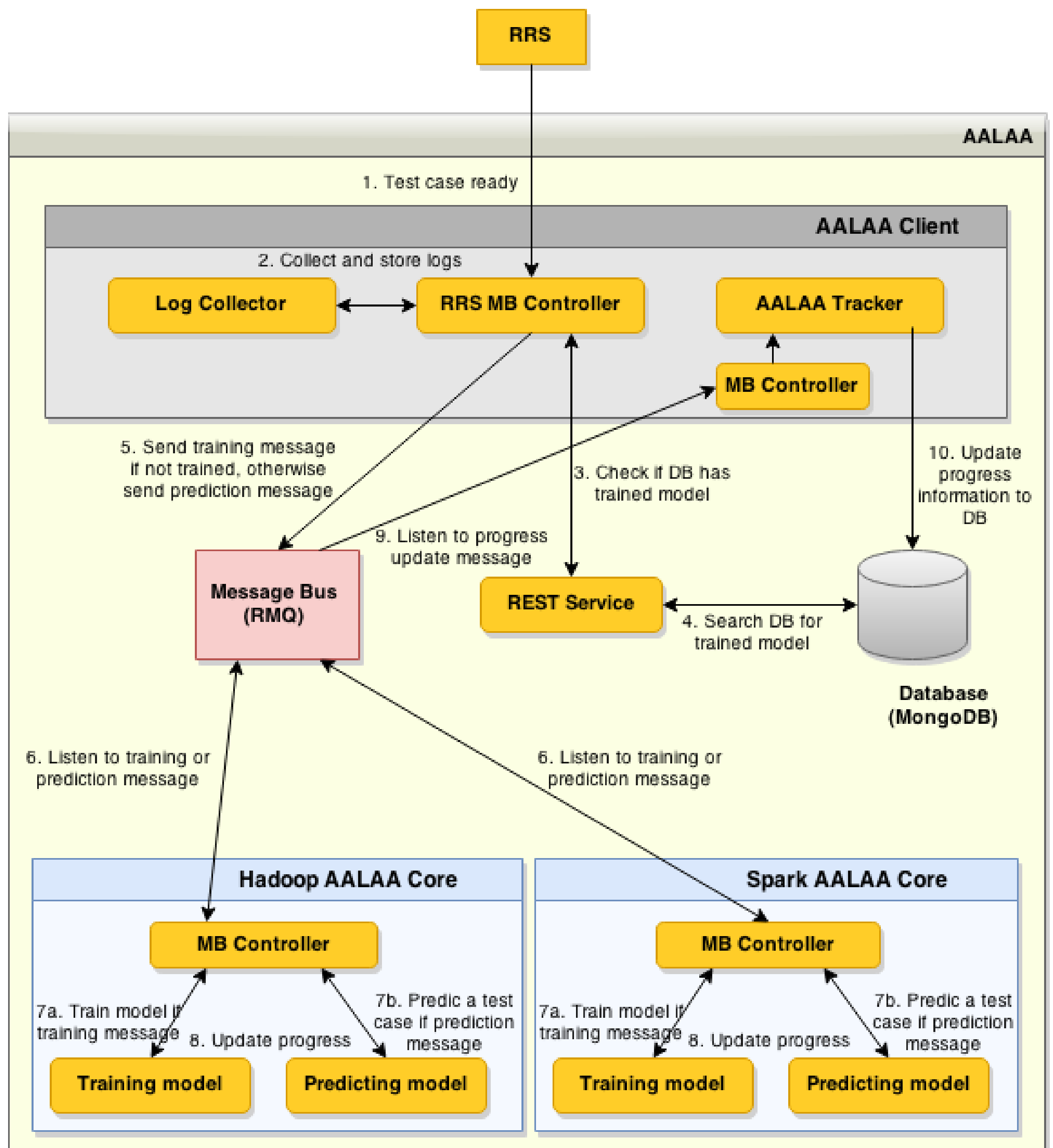

Figure 4-1: Planned architechture of the AALAA system

4.1.2 Design of the Message Bus

Designing the message bus was an important part of the process of constructing the system's architecture. A good design would facilitate communication between all of the modules, therefore a lot of thought was put into this design. Figure 4-4 (on page 18) illustrates a more detailed version of the AALAA system that actually was implemented, including a more detailed overview of how the message bus was constructed. The main decisions that had to be done concerning the message bus were: (1) How to set up the message queues and (2) How to design the message formats. As stated previously, the message bus was implemented in RabbitMQ and the selected message format was JavaScript Object Notation (JSON). This message format was selected because it provided an easy and uniform way to interpreting messages in all modules. JSON is also used with MongoDB, so by using this format, messages could be dumped into the databases without having to make any re-formatting.

A first stage of designing message busses in RabbitMQ is to set up an *exchange*. As described in Section 2.2, an exchange is where all messages that wish to be routed are sent to. The exchange decides where to route the incoming messages based on an attached routing key that specifies which *queue* the message are to be sent to. The implemented message bus consists of one exchange that is bound to four different message queues. Figure 4-4 explains how the message flow works in the implemented artifact. A decision was made to separate the Spark and Hadoop versions of the core when designing the queues. This was done because the different core versions are separate programs and thus it was natural to utilize separate Message Bus Controllers in their implementations. It also seemed to be reasonable to separate them as a client can route a request to either or both of these core versions.

4.1.2.1 Message Bus structure

Each message bus controller in the AALAA Core is attached to two different message queues, one for incoming messages and one for outgoing messages. As the bus controllers are listening to separate incoming message queues, the message sender must provide the correct routing key when sending a request to the exchange. This structure greatly simplifies the implementation of the core modules, as all incoming messages from a queue can be consumed without needing to check if the content of this message is designated for this module or not. The decision to use separate outgoing message queues to be consumed by the AALAA-Tracker was not as easy to make since both queues would make use of the same consumer. Before making this decisions several things had to be taken into consideration, mainly looking at how the AALAA-Tracker would operate and how it would process the messages it was to consume.

It was decided that the AALAA-Tracker was to communicate with two different databases, one storing updates from the Hadoop version (Hadoop-DB) and one storing updates from the Spark version (Spark-DB). By listening to two separate queues that were each configured to dump messages into the database designated for that queue, two instances of the tracker could be spawned as two different threads and each thread could update its designated database concurrently. This was considered to be a good programming model as it offered good error-tracing abilities, while enabling potential future expansions by adding additional queues and spawning additional threads.

4.1.2.2 Message format

While designing the format of the messages to be routed through the message bus, both decisions about the format and message contents had to be made. JSON was selected as the format mainly due to its lightweight format and easy readability. This decision was considered especially appropriate since only a relatively small amount of information needed to be transferred within the messages. The Message Bus Controllers were implemented in Java and Java provides good libraries

to process JSON-formatted data. While designing the contents of the messages, it was important to understand what information was required to initiate executions in the core and what information was to be sent back as results. When requesting a train or predict action in the core, only one message is sent with the information necessary to initiate the execution. Progress updates were to be sent back to the client during execution with information about the request's current state. These messages were designed to be sent continuously from the core to the tracker since the requested calculations can sometimes take a long time. On completion of an execution task in the core, a final message is sent along with the results produced by the requested action.

A basic example of the message format for incoming requests to the core is presented in Figure 4-2. Figure 4-3 shows an example of what a progress update message could look like.

AALAA-Core initiation:

```
{
 "uuid":"33121f30-fc01-11e4-b939-0800200c9a66",
 "requestType":"train",
 "modelID":"testmodel",
 "trainLogInfo":[
          {
      "verdict":"PASS",                    //Only included in case of train
      "path":"/path/to/trainlog1.txt"
      },
          {
      "verdict":"FAIL",                    //Only included in case of train
      "path":"/path/to/trainlog2.txt"
      }

 ]
}
```

uuid	Unique identifier for mapping results with corresponding entry in database
requestType	Type of request to carry out, can contain “train” or “predict”
modelID	Name of the model to train or to predict on in case a model is already trained
trainLogInfo	JSONArray listing all logs to perform training on. The “verdict” field is the outcome of each test and can contain “PASS” or “FAIL”. The “path” field should contain the full path to each log.

AALAA-Core progress update:


```
{
 "uuid":"33121f30-fc01-11e4-b939-0800200c9a66",
 "requestType":"train",
 "progressionStatus":"Saving model",
 "saveModelAt":"2015-05-08 10:35:35.725"
}
```


uuid	Unique identifier for mapping results with corresponding entry in database
requestType	Type of calculation performed, can contain "train" or "predict"
progressionStatus	What the current state in the calculations is
saveModelAt	Timestamp of the information associated with progressionStatus, this field can contain various types of information

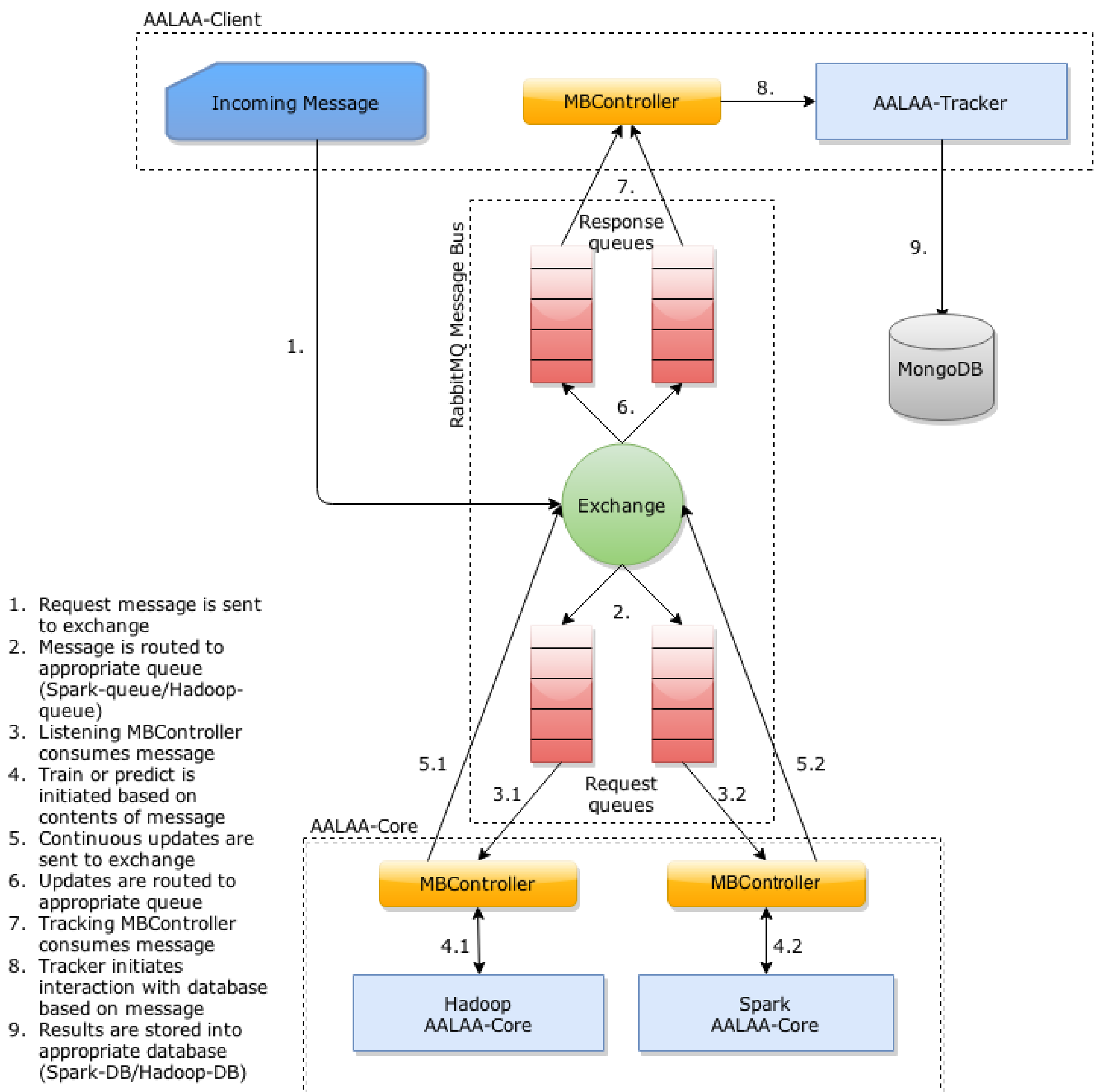

Figure 4-4: More detailed overview of the implemented message bus including flow of events

4.2 Software implementation

This section describes the actual implementation of the software based on the architecture defined in Section 4.1. Section 4.2.1 describes the main decisions made while modifying the AALAA Core and its associated message bus controllers. Section 4.2.2 goes into detail about the work that was done when changing the AALAA Client to support the new architecture. This section also describes what parts of the client were implemented and what parts were left for future work.

4.2.1 AALAA Core implementation

The implementation of the AALAA Core mainly consisted of modifying the two existing machine learning modules (i.e., the already implemented versions of the core for Spark and Hadoop). The work started by (1) Identifying how the new message bus controllers could be integrated with the existing code and (2) Locating what data that needed to be extracted from the calculations in order to send back relevant progress update messages. The message bus controllers for both core versions were implemented as its own Java class. The main function of this class is to communicate with their individually assigned message bus queues for immediate message retrieval and to send back progress updates to the requesting client via a specified exchange. The configuration for host, port, message queues, and exchange of messages via the message bus is provided in a configuration file for both versions. The modules read this information and pass it to their controllers in order to open the required connections. A helper class was implemented for each version of the core to interpret and translate incoming and outgoing JSON-formatted messages.

The trickiest part of the implementation was to connect the modules to their associated message bus controllers in a way that would not require major modifications to the modules themselves. The implementation philosophy was to change the existent modules as little as possible in order to avoid any interference with their current functionality. The modules were designed to start by listening to their assigned message bus queues. Once a message is received by a controller, the module makes use of the helper class to extract the necessary information from the incoming message. For example, this information will indicate whether this is a training or prediction message –hence initiating the requested processing and indicating what logs would be used for these calculations.

Once a train or predict action was started in the core, certain milestones throughout the calculations would be tracked in order to send progress updates through the controller to the specified exchange when a milestone was reached. A routing key was attached to each update message so that the exchange in the message bus would be able to route it to its corresponding response queue (i.e., the queue listened to by the AALAA-Tracker). Progress updates were used rather than sending back one large message at the completion of the request because of calculations on large batches of logs that might take hours to complete. By continuously sending updates back to the client and storing the results into a database, the current progress of a calculation using a certain model could be checked at any time by looking in the database.

Implementing the above steps was different for the Spark and the Hadoop versions of the core. The old version of the Hadoop module was already polling for initiation requests from a database and stored progress updates directly into this database during its calculations. Changing this module to support message communication consisted mostly of listening to a message queue rather than polling from a database and redirecting progress updates to a message bus exchange rather than storing them directly in the database. The Spark version was a standalone program using manual requests and did not communicate with any database to store its results. Therefore, this module had to be changed to take its parameters from a message while identifying and extracting the data required for progress updates.

4.2.2 AALAA Client implementation

Some parts of the AALAA-Client were not implemented due to the limited duration of this thesis project. Figure 4-1 describes what was planned and Figure 4-4 gives a good overview of what the final version of AALAA actually looked like. For the client, the *AALAA-Tracker* was implemented while the *RRS MB Controller* and the *Log Collector* were left for future work.

The AALAA-Tracker was implemented in Java. Its purpose was to listen to progress update messages sent by the core during calculations and to store this information into a database. This module did not exist, hence it had to be implemented from scratch. The message bus controller associated with the tracker only needed to be able to listen for messages. For this reason no outgoing message functionality was provided. As described in Section 4.1.2.1, one thread was assigned to each of the two queues receiving messages from the AALAA Core. This simplified the implementation by having the cores read the necessary configuration for these queues and databases from a configuration file. This information is passed as parameters when spawning the threads. Once spawned, the threads listen to their assigned queues and on the arrival of a message the thread executes the following actions (in order): (1) read an incoming message, (2) extract the attached UUID, and (3) route the progress update to the associated entry, i.e. the entry with the same UUID in the database.

As the RRS MB Controller was not implemented, normal requests cannot be sent to initiate requests (as they would be in real deployment). For this reason messages were manually constructed and injected directly into the exchange through a web interface in order to initiate executions in the core. The RRS MB Controller would listen to messages broadcasted by the RRS, a system used for reporting test results and sending information about each finished test case. Once a finished test case message is received, the logs would be collected through the Log Collector. The RRS MB Controller would then extract these log paths and automatically initiate a training or prediction request in one or more of the AALAA cores and it would provide the appropriate action and log paths in the request message. This part of the implementation had the lowest priority to complete because the development team already had good knowledge of this part of the system in contrast with the other parts. For this reason these two components were left to be completed later.

5 Analysis

This section presents the major results collected from the experiments that were carried out on the implemented artifact (see Section 5.1). Two major areas was examined during the evaluation: (1) Does the solution solve the defined problem and (2) Is the solution sufficient in terms of performance. To evaluate (1), a qualitative analysis was performed by reasoning about whether the implemented artifact's functionality would suffice as a valid solution or not. This will be discussed in Section 5.4. For (2), a quantitative analysis was done by collecting data from several test cases covering performance and functional correctness aspects of the implementation.

5.1 Major results

In this section, any data collected connected to performance and correctness tests will be presented. Table 5-1 shows the average time consumption for sending and receiving messages through the implemented RabbitMQ bus depending on the size of the sent message. This provides as a sufficient test for measuring how long it would take to initiate a request from the AALAA Client to the AALAA Core. As can be seen in Figure 4-2, the message size when sending out a request depends on three static fields and one non-static field (the number of log paths).

Table 5-1: Time to initiate request in AALAA core based on message size. 95% confidence level applied.

Number of log paths in message (N)	Size of log paths (85 bytes per path)	Number of tests performed	Average time spent (ms)
1	85 B	100	0.8097 ± 0.04 ms
10	850 B	100	0.9133 ± 0.04 ms
100	8500 B	100	1.2402 ± 0.06 ms
1000	85000 B	100	2.2354 ± 0.1 ms
10000	850000 B	100	12.058 ± 0.59 ms

Since the only parameter that varies is the number of log paths in a request, different values were used when generating test messages. This value was increased by 10x to cover as wide a range of number of log paths as possible (at least for the range that might be used in a real deployment). Messages for each test case were sent 100 times as decided by the standard deviation and the desired confidence level of 95%. The average values from these observations with its associated confidence intervals are presented in Table 5-1. Time measurement was started before a message was sent to the bus, and stopped immediately after the same message was consumed in the core. Since the requesting RRS MB Controller of the client had not yet been implemented, a simple test program was written to send artificial messages. Figure 5-1 shows a plot of the time per bytes cost results presented in Table 5-1.

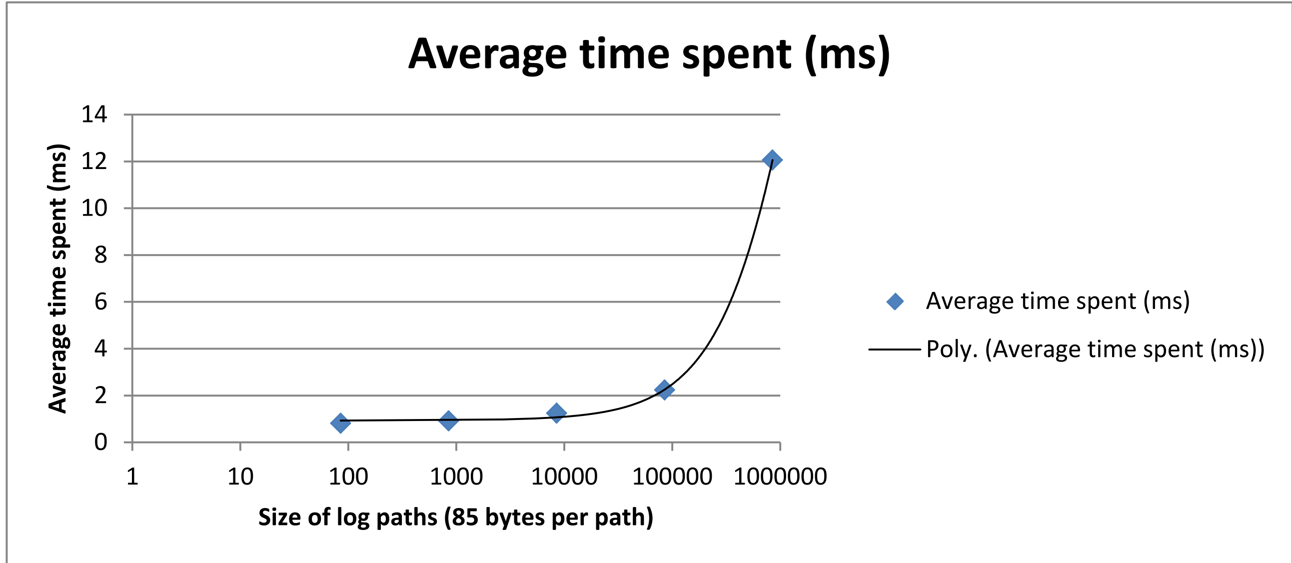

Figure 5-1: Time to initiate request in AALAA core based on message size

The time to send out progression updates was not measured. These updates are generally static in length and only consists of a few fields, as shown in Figure 4-3. The exception is the final result of a predict action where all abnormal logs found are listed in the message, however this message will always be smaller than the initiating request message. As will be discussed in Section 5.4 on page 24, the time it takes to send a message through the bus is extremely small in comparison to how long the complete calculations will take, therefore a more extensive analysis of this area was considered unnecessary.

Table 5-2 and Table 5-3 shows the data collected from tests of the functional correctness of the implementation. They show that the functionality of the modified modules in the core remained intact. Comparisons are presented between the outcomes of 3 different test cases for each version of the core, where abnormal logs were predicted using already trained models in the core. The outcomes shows no difference in output data between the old and the new implementation, therefore the functionality of a predict action in the Spark and Hadoop core are considered correct (or at least no less correct than they were with the previous implementations). As one might notice, the number of logs used in the tests between Table 5-2 and Table 5-3 differs. This is due to much faster processing times in the Spark core than the Hadoop core. Tests on the Hadoop core had to be scaled down to process fewer logs in order to complete tests within reasonable time.

Table 5-2: Difference checks for a predict action between the old and new Spark cores

Test case	Total logs processed	Abnormal logs old AALAA	Abnormal logs new AALAA	Log name comparison
1-Spark	1236	176	176	176/176 MATCH
2- Spark	1497	49	49	49/49 MATCH
3- Spark	1403	31	31	31/31 MATCH

Table 5-3: Difference checks for a predict action between the old and new Hadoop core

Test case	Total logs processed	Abnormal logs old AALAA	Abnormal logs new AALAA	Log name comparison
1-Hadoop	19	7	7	7/7 MATCH
2-Hadoop	29	12	12	12/12 MATCH
3-Hadoop	9	5	5	5/5 MATCH

While testing the preservation of the training functionality in the core, only unreliable data could be collected. Since the machine learning algorithms connected to a training action produce slightly different outcomes every run, even though the same input logs are provided, no direct outcome comparisons could be performed. Validating the functionality of a train action consisted of looking whether the outcomes from the same input data were within a reasonable range relative to each other, and by analyzing the code to ensure that the modifications that were made did not interfere with core functionality. Multiple predict actions was also tested on the same trained models to ensure that outcomes would not differ. The results showed no difference between the performed predict actions.

5.2 Reliability Analysis

The experiments performed for testing the artifact's performance are considered reliable. While testing time consumption of the message bus, see Table 5-1, 100 tests were performed for each test case to ensure reliable average values. Some differences were observed in time per bytes costs as the size of the messages grew. This is due to the overhead present when routing a message, independent of its size. This resulted in overhead having a larger impact on the smaller message. Based on the results presented in Table 5-1, the overhead observed when sending a message was approximately 0.8 ms.

While testing functional correctness, multiple tests were performed to confirm that no changes had been made to the machine learning functionality provided by the AALAA Core. When making these changes, great care was taken to only modify parts of the code asserting correct input and output data, therefore the functional correctness tests was merely a confirmation that no such changes had taken place. Complete reliability can however not be confirmed for functional correctness of a training action in the core because of the varying outcomes that this process naturally produces. The assurance that no changes were made to this functionality due to the newly added code therefore requires us to trust in the care taken when making the modifications to this code.

5.3 Validity Analysis

The results for measuring the implemented message bus are considered valid based on the metrics proving that messages were correctly delivered from start to end while functionality of the AALAA Core remained intact. Validation of the planned solution in its entirety could however not be achieved since not all parts of the AALAA client were implemented and complete tests could as a result not be done in real deployment.

5.4 Discussion

Looking at the results presented in Table 5-1, a pattern can be seen for decreased time per bytes cost as the message size grows. This is due to the relatively large overhead present while routing messages of small sizes. Performance measurements can be found at RabbitMQs official blog where similar results are noted but for larger parameters [24]. The results presented in Table 5-1 are similar to the results presented in [24], where overhead is larger for small messages, but becomes a small factor for larger messages. An important note to the observed message processing speed is that its impact on the complete execution time of a train or predict action is negligible. The execution time of calculations in the AALAA Core varies depending on the number of logs it processes. However, this time usually ranges between a few minutes and several hours. Messages sent that add a few extra milliseconds (or even seconds) will therefore not substantially affect the total execution time.

The polling for requests previously implemented would in the AALAA Core poll a database for test cases and on discovery of a new test case initiate calculations. The implemented message bus arguably improves the architecture, scalability, and performance of AALAA based on the presented results. This conclusion is based on the decreased times between test case discovery and the increased functionality achieved by the additional integrated modules.

Looking at correctness, results have shown no evidence of any functionality in the core being affected by the new implementation. Only input data and output data transferred between modules through the message bus has been treated, great care has been taken into assuring that this data is routed from start to end without changing information or getting lost on the way. A complete analysis of the new artifacts correctness and performance can only be done once the Log Collector and RRS MB Controller left for future work in Section 6.3 has been implemented.

The design decisions presented in Section 4.1 have been shown to be correct. The artifact both solves the given problem and does so in a more efficient way than before. When constructing a system of this scale a lot of things can be done differently, especially since the message bus can be designed in different ways. However, since the requirements of the implemented artifact were quite clear and thorough, the design decisions became fairly straightforward. The conclusion is that the implemented artifact currently partly solves the given problem and will suffice as a solution once the (near term) future work has been completed.

6 Conclusions and Future work

This chapter presents conclusions drawn from this thesis project. Section 6.1 gives some general conclusions regarding the achieved goals, insights gained, and things that could have been done differently. Section 6.2 presents some of the limitations of the efforts performed and limitations of the presented results. Section 6.3 describes things that were left for future work and suggests some work that could be done to following up this thesis project. Finally, Section 6.4 gives some general reflections regarding different aspects relevant to the performed work.

6.1 Conclusions

The main goal of enabling the user to send out requests to the AALAA Core was not entirely met since not all of the work was managed to be completed in time. This meant that request messages destined for the core had to be manually constructed and sent directly into the message bus exchange through its web interface. The future work related to this is presented in Section 6.3. Looking at the sub goals that were defined, we can conclude can that most of the functionality was implemented. Both versions of the core were integrated with a message bus, the client was modified to be able to receive progress updates from this bus, and the client successfully managed to store these updates into MongoDB. The different parts of the solution were connected and deployed, so that despite the need to manually send a request message, the rest of the planned flow of events worked as intended.

The goal of implementing a solution that integrated previously unintegrated parts, namely the Spark version of the core and the AALAA-Tracker in the client, was also achieved. Based on the results, some conclusions can be drawn – specifically that the functionality of both core versions remained intact since comparisons between the old solution and the modified solution showed no difference in the prediction outputs when run with the same input data. Request initiations and progress updates were also evidently improved in both performance and architecture as decreased initiation times and added functionality was achieved.

Several insights were gained from completing this project. Since AALAA is a rather large and complex system consisting of several modules, a lot of work was required just to understanding its functionality, before I could begin to consider how to modify it. Planning the work and building an architecture were by far the most important parts to focus on, since having a good thought process was essential while modifying and implementing each module - so that it would be compatible with the rest of the system. A relevant and uniform message structure was also important to construct in order to make message translation and interpretation easy on both ends of the bus.

If I were to do the same work again, a greater focus would have been put on researching AALAA more extensively in order to fully understand all of the requirements *before* doing any actual work. Some minor issues arose after the first weeks when requirements of the planned architecture changed slightly – causing some unnecessary work on researching parts that were in the end not required.

6.2 Limitations

The most obvious limitation of the work performed is that the complete solution was not finished, as the RRS MB Controller that is supposed to send messages to initiate requests in the core was not implemented. The planned RRS MB Controller was supposed to implement automatic checking for whether a trained model for the incoming test case was already stored in the database and based on this make a decision to send out a train or predict message. Since this was not done, it meant that messages had to be manually injected directly into the message bus. This made testing of the system in a real time environment impossible. As a result, tests had to be done by manually constructing messages containing information about the log paths and whether a training or prediction request was to be performed.

6.3 Future work

The next stage in the implementation phase is to build the RRS MB Controller of the AALAA Client. This controller should listen to the test case messages sent out by the RRS system. It should then decide whether to use the Spark or Hadoop version of the AALAA core for machine learning processing, and then open that version of MongoDB through REST to check whether a model for that test case is trained or not. If a model is not trained, then the log paths and test verdicts for each log should be extracted and stored with a generated UUID into that database. A message should then be sent to the RabbitMQ exchange in with a routing key designating the Spark version or Hadoop version. The message that is sent should contain a “train” parameter, model name, a generated UUID, and the log paths with its paired verdicts. If a model is stored in the database, then only the log paths need to be extracted and saved into the MongoDB. A similar message should then be sent to the exchange containing a “predict” message instead of “train”, the model name, a generated UUID, and log paths without test verdicts. The Log Collector also needs to be updated in order to handle requests from the RRS MB Controller so that logs can be copied once a new test case has been discovered.

6.4 Reflections

Positive economic gains from of the project can hopefully be achieved as the convenience of using AALAA has increased by enabling an automatic and fast initiation of computations once a test case has become available in the RRS. For this effect to take place, all of the future work (in Section 6.3) needs to be implemented to support an actual deployment. The Spark version of the core has also been integrated with the client in addition to the Hadoop version that earlier was only partially integrated. Spark has in several cases been shown to outperformance the Hadoop version in terms of processing speed [7], therefore this addition might be of increased value to Ericsson.

Ethical aspects applicable to projects involving machine learning consist of having too much automation at the expense of employees running out of work and therefore being considered redundant. This is generally a good thing for companies - since this would result in reduced expenses while maintaining revenues. However, it is still an ethical aspect to consider. AALAA is an experimental feature at Ericsson's CII-department and performs a relatively specific task, therefore its scope is currently not considered to cause lack of work for the current employees working in this department.

References

[1] O. Kwon, N. Lee, and B. Shin, “Data quality management, data usage experience and acquisition intention of big data analytics,” *Int. J. Inf. Manag.*, vol. 34, no. 3, pp. 387–394, Jun. 2014.

[2] “This is Ericsson - this-is-ericsson.pdf.” [Online]. Available: http://www.ericsson.com/res/thecompany/docs/this-is-ericsson.pdf. [Accessed: 25-Mar-2015]

[3] Weixi Li, “Automatic Log Analysis using Machine Learning: Awesome Automatic Log Analysis version 2.0”, Master's thesis, Uppsala Universitet, Department of Information Technology, Report number IT 13 080, November 2013, [Online]. Available: http://uu.diva-portal.org/smash/get/diva2:667650/FULLTEXT01.pdf.

[4] “IBM - What is MapReduce.” [Online]. Available: http://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/. [Accessed: 19-May-2015]

[5] “Apache Spark™ - Lightning-Fast Cluster Computing.” [Online]. Available: https://spark.apache.org/. [Accessed: 19-May-2015]

[6] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: SQL and Rich Analytics at Scale,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2013, pp. 13–24 [Online]. Available: http://doi.acm.org/10.1145/2463676.2465288. [Accessed: 25-Mar-2015]

[7] Georgios Koutsoumpakis, “Spark-based Application for Abnormal Log Detection.”, Master's thesis, Uppsala Universitet, Department of Information Technology, Report number IT 14 057, September 2014, [Online]. Available: http://uu.diva-portal.org/smash/get/diva2:751988/FULLTEXT01.pdf.

[8] G. L. Geerts, “A design science research methodology and its application to accounting information systems research.”, *International Journal of Accounting Information Systems*, vol.12, no.2, pp. 142-151, June 2011. DOI: 10.1016/j.accinf.2011.02.004, [Online]. Available: http://www.sciencedirect.com.focus.lib.kth.se/science/article/pii/S1467089511000200. [Accessed: 14-May-2015]

[9] “What is the Agile Software Development Methodology?” [Online]. Available: http://agilemethodology.org/. [Accessed: 19-May-2015]

[10] “Waterfall Model.” [Online]. Available: http://www.waterfall-model.com/. [Accessed: 19-May-2015]

[11] Jaakko Hollmen, “Self-Organizing Map (SOM).”, Mar 8 13:44:34 EET 1996, [Online]. Available: http://users.ics.aalto.fi/jhollmen/dippa/node9.html. [Accessed: 19-May-2015]

[12] “Cross Validation.” [Online]. Available: http://www.cs.cmu.edu/~schneide/tut5/node42.html. [Accessed: 19-May-2015]

[13] “RabbitMQ - What can RabbitMQ do for you?” [Online]. Available: https://www.rabbitmq.com/features.html. [Accessed: 08-May-2015]

[14] “What is message broker? - Definition from WhatIs.com.” [Online]. Available: http://whatis.techtarget.com/definition/message-broker. [Accessed: 19-May-2015]

[15] “An Advanced Message Queuing Protocol (AMQP) Walkthrough,” *DigitalOcean*. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-protocol-amqp-walkthrough. [Accessed: 10-Apr-2015]

[16] “RabbitMQ - AMQP 0-9-1 Model Explained.” [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html. [Accessed: 10-Apr-2015]

[17] S. Torstendahl, “Open Telecom Platform.”, Ericsson Review No. 1, 1997, pp. 14-23, [Online]. Available: http://www.erlang.se/publications/ericsson_review_otp_1997012.pdf. [Accessed: 19-May-2015]

[18] “Using Erlang for an Open Telecommunications Platform.” [Online]. Available: http://www.methodsandtools.com/archive/erlang.php. [Accessed: 16-Apr-2015]

[19] J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues, and S. Ullah, “Performance evaluation of RESTful web services and AMQP protocol,” in *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*, 2013, pp. 810–815.
[20] S. Dawar, S. Van der Meer, E. Fallon, J. Keeney, and T. Bennett, “Building a Scalable Event Processing System with Messaging and Policies – Test and Evaluation of RabbitMQ and Drools Expert ” in proceedings of the 2013 Information Technology and Telecommunications conference (IT&T) 2013, [Online]. Available: http://www.researchgate.net/publication/255723410_Building_a_Scalable_Event_Processing_System_with_Messaging_and_Policies___Test_and_Evaluation_of_RabbitMQ_and_Drools_Expert. [Accessed: 21-May-2015]
[21] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge, “The Design Science Research Process: A Model for Producing and Presenting Information Systems Research,” in *In: 1st International Conference on Design Science in Information Systems and Technology (DESRIST*, 2006, pp. 83–106.
[22] “Maven – Welcome to Apache Maven.” [Online]. Available: https://maven.apache.org/. [Accessed: 30-May-2015]
[23] “Learn | The Scala Programming Language.” [Online]. Available: http://www.scala-lang.org/documentation/. [Accessed: 30-May-2015]
[24] “RabbitMQ » Blog Archive » RabbitMQ Performance Measurements, part 2 - Messaging that just works.” [Online]. Available: https://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/. [Accessed: 17-May-2015]

TRITA-ICT-EX-2015:55