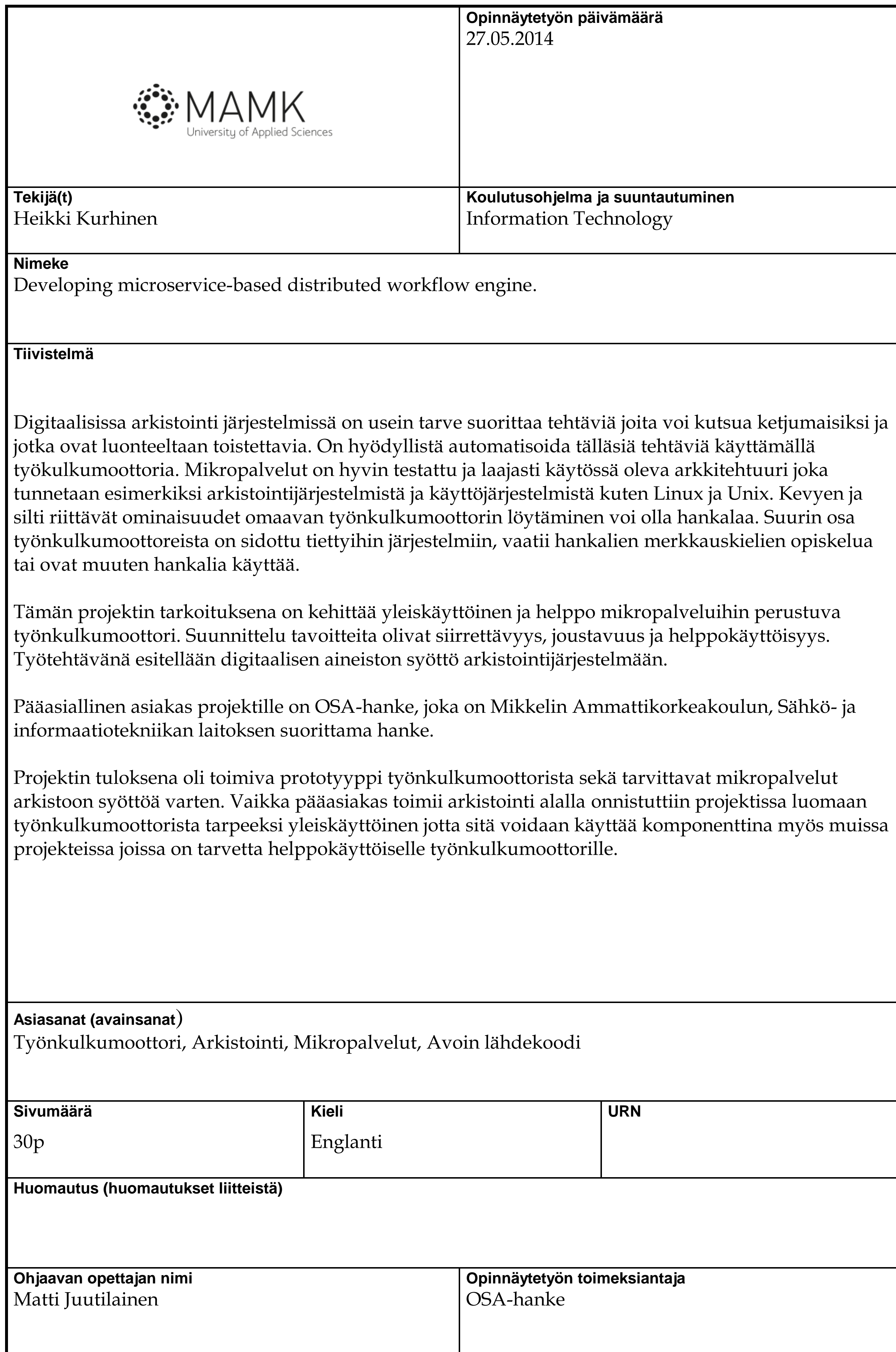
Heikki Kurhinen

DEVELOPING MICROSERVICE-BASED DISTRIBUTED WORKFLOW ENGINE

Bachelor's Thesis
Information Technology

May 2014

KUVAILULEHTI



MAMK University of Applied Sciences	**Opinnäytetyön päivämäärä** 27.05.2014
Tekijä(t) Heikki Kurhinen	**Koulutusohjelma ja suuntautuminen** Information Technology

Nimeke
Developing microservice-based distributed workflow engine.

Tiivistelmä

Digitaalisissa arkistointi järjestelmissä on usein tarve suorittaa tehtäviä joita voi kutsua ketjumaisiksi ja jotka ovat luonteeltaan toistettavia. On hyödyllistä automatisoida tälläsiä tehtäviä käyttämällä työkulkumoottoria. Mikropalvelut on hyvin testattu ja laajasti käytössä oleva arkkitehtuuri joka tunnetaan esimerkiksi arkistointijärjestelmistä ja käyttöjärjestelmistä kuten Linux ja Unix. Kevyen ja silti riittävät ominaisuudet omaavan työnkulkumoottorin löytäminen voi olla hankalaa. Suurin osa työnkulkumoottoreista on sidottu tiettyihin järjestelmiin, vaatii hankalien merkkauskielien opiskelua tai ovat muuten hankalia käyttää.

Tämän projektin tarkoituksena on kehittää yleiskäyttöinen ja helppo mikropalveluihin perustuva työnkulkumoottori. Suunnittelu tavoitteita olivat siirrettävyys, joustavuus ja helppokäyttöisyys. Työtehtävänä esitellään digitaalisen aineiston syöttö arkistointijärjestelmään.

Pääasiallinen asiakas projektille on OSA-hanke, joka on Mikkelin Ammattikorkeakoulun, Sähkö- ja informaatiotekniikan laitoksen suorittama hanke.

Projektin tuloksena oli toimiva prototyyppi työnkulkumoottorista sekä tarvittavat mikropalvelut arkistoon syöttöä varten. Vaikka pääasiakas toimii arkistointi alalla onnistuttiin projektissa luomaan työnkulkumoottorista tarpeeksi yleiskäyttöinen jotta sitä voidaan käyttää komponenttina myös muissa projekteissa joissa on tarvetta helppokäyttöiselle työnkulkumoottorille.

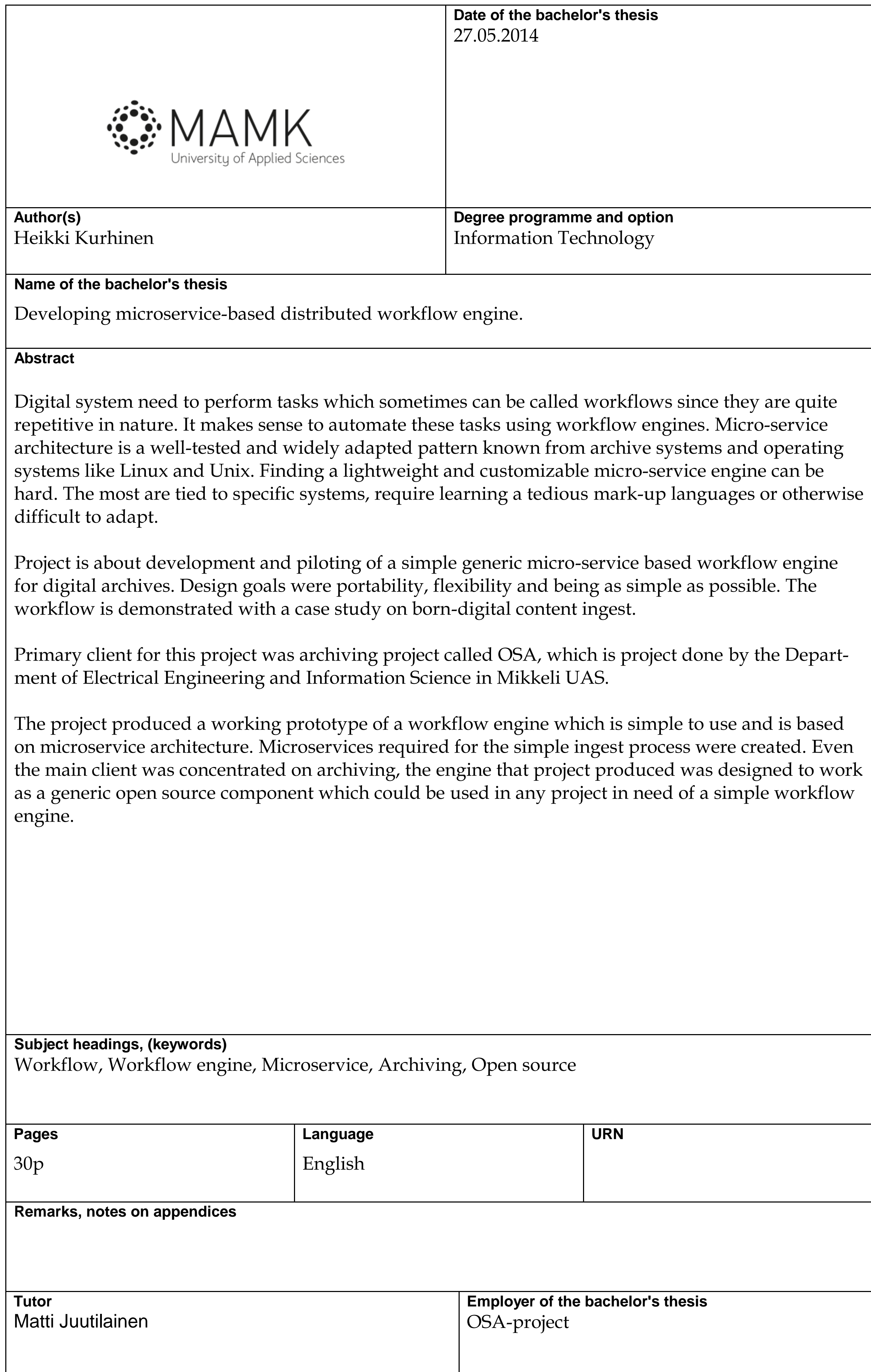
Asiasanat (avainsanat)
Työnkulkumoottori, Arkistointi, Mikropalvelut, Avoin lähdekoodi

Sivumäärä	**Kieli**	**URN**
30p	Englanti	

Huomautus (huomautukset liitteistä)

Ohjaavan opettajan nimi	**Opinnäytetyön toimeksiantaja**
Matti Juutilainen	OSA-hanke

DESCRIPTION



Date of the bachelor's thesis
27.05.2014

MAMK
University of Applied Sciences

Author(s)
Heikki Kurhinen

Degree programme and option
Information Technology

Name of the bachelor's thesis
Developing microservice-based distributed workflow engine.

Abstract

Digital system need to perform tasks which sometimes can be called workflows since they are quite repetitive in nature. It makes sense to automate these tasks using workflow engines. Micro-service architecture is a well-tested and widely adapted pattern known from archive systems and operating systems like Linux and Unix. Finding a lightweight and customizable micro-service engine can be hard. The most are tied to specific systems, require learning a tedious mark-up languages or otherwise difficult to adapt.

Project is about development and piloting of a simple generic micro-service based workflow engine for digital archives. Design goals were portability, flexibility and being as simple as possible. The workflow is demonstrated with a case study on born-digital content ingest.

Primary client for this project was archiving project called OSA, which is project done by the Department of Electrical Engineering and Information Science in Mikkeli UAS.

The project produced a working prototype of a workflow engine which is simple to use and is based on microservice architecture. Microservices required for the simple ingest process were created. Even the main client was concentrated on archiving, the engine that project produced was designed to work as a generic open source component which could be used in any project in need of a simple workflow engine.

Subject headings, (keywords)
Workflow, Workflow engine, Microservice, Archiving, Open source

Pages	Language	URN
30p	English	

Remarks, notes on appendices

Tutor
Matti Juutilainen

Employer of the bachelor's thesis
OSA-project

CONTENTS

1 INTRODUCTION 1
2 WORKFLOW 1
2.1 History 2
2.2 Workflow components 3
2.3 Workflow engine 3
3 MICROSERVICE ARCHITECTURE 4
3.1 Microservice architectural style 4
3.2 Benefits of microservice architecture 6
4 DESIGN GOALS 10
4.1 Simple usage 10
4.2 Open source 10
4.3 Communication 11
4.4 Self-hosted 14
5 IMPLEMENTATION 14
5.1 Core 15
5.2 Application programming interface (API) 18
5.3 Timing/Triggers (QUARTZ) 20
5.4 Example workflow and microservices 20
5.5 Clustering 24
5.6 Testing 25
5.7 Client library 27
5.8 Results 28
5.9 Future development 28
6 CONCLUSIONS 29
BIBLIOGRAPHY 31

1 INTRODUCTION

In modern digital systems, it is important to be able to perform different repetitive tasks. This project is about creating engine to perform these tasks or workflows. The need for this engine rose from the fact that many current workflow engines are designed to work in large enterprise scale systems and configuring and creating workflows with them requires a great deal of effort. I noticed the need for this simpler approach while I was working for the Open-Source Archiving project called OSA.

OSA-project is open-source archiving project which is the primary client for this project.

"Project OSA is administrated and carried out by Department of Electrical Engineering and Information Science in Mikkeli UAS. Project partners are Central Archives of Finnish Business Records (ELKA), Brages Pressarkiv, Monikko Oy, Mikkelin Puhelin Oyj, MariaDB Services Ab and Disec Oy. We co-operate with the National Archives of Finland and Otavan Opisto as well as Digital Mikkeli – the umbrella organization for the memory organizations in the Mikkeli region. Our budget is 501.219 €; ERDF funding is granted by South Savo Regional Council, the other financiers are the City of Mikkeli, Mikkeli Region, Mikkeli UAS and our partners. The project started in May 2012 and will be closed at the end of June 2014."[1]

The goal of this project is not to create production ready workflow engine, but to explore possibilities and evaluate the idea and amount of work involved in creating own workflow engine. The goal is to build prototype which OSA-project can then continue working on and also to create open source component which could potentially develop into actual product by the contributions of the open source community.

2 WORKFLOW

Workflow in general level means tasks, steps of certain procedure, persons involved, input and output information and necessary tools for step in business procedure. Basically workflow can be used to describe a bunch of processes which can transform or process materials and data/information, or provide services. [2, 3, 4, 5]

2.1 History

The first examples of work which could be considered to be organized by the concept of workflow were found in 1920's. During that era workflows were mostly concentrated on manufacturing physical products. Good example from that era is the assembly line, which makes a good example of a workflow in general. A unfinished products is the "input" of the workflow which then goes thru steps where a worker assembles a single part of the product and then passes it onwards. [2, 3, 4, 5]

When copy-writer was invented, office work became more common and that way also workflow models were applied into office work, while trying to improve the efficiency for example in the fields of office automation and document producing. This way work optimization gained more attention and also some mathematical formulas were invented to help with optimization of work. [2, 3, 4, 5]

In 1980s workflow style optimization of work got some criticism. Some considered it to be in humanizing since it considers employees only steps in a process and eliminates the humans own creativity when organizing his work. Other sources of criticism were the quality of work since with workflow style organization it is not possible to adapt to unexpected things in the process. Good example of this is again the assembly line, if a single employee is supposed to tighten a nut and then pass the product onwards, he only pays attention to the nut that he is supposed to tighten and doesn't notice if previous employee forgot to tighten the previous nut. Since this project was about creating a workflow engine to manage digital work, the further cogitation of these problems was left out from this paper. [2, 3, 4, 5]

Currently workflows are used in many places, for example in machine shops, processing claims, software development etc. The primary use for this particular workflow engine is to manage workflows related to digital archiving systems. Such workflows are for example Ingesting files into the archive, migrating archived documents to different server and creating backups of archived documents. [2, 3, 4, 5]

2.2 Workflow components

Workflow can usually be presented with flow diagram which shows transitions between steps in the workflow. Single step of a workflow consists from three main components.

1. input description
2. transformation rules
3. output description

Input description is the information or material or the energy which will be the feed for the step in workflow. Transformation rules are algorithms which tell what needs to be done for the input. Output description is the information or material or energy which is processed by the step and what is provided as input for the following steps in workflow. [2, 3, 4, 5]

2.3 Workflow engine

Most software usually contains some workflows but the workflow engine means a piece of software which is designed to follow specification to execute a workflow, this basically means that person without coding skills can make changes to how the software works and what features there is available. Of course at least in this project developer needs to first create the required microservices and then they can be used. Basically workflow engine is software which is designed to manage and monitor the completion of the steps in a workflow. Workflow engine can makes decisions and also monitors the execution of specified tasks and if the task fails it can perform some actions to correct the situation or cancel next steps and report error. Workflow engine usually uses database to store information about workflows and steps in them. Usually workflow engines have three main functions:

- Checking that action is valid for the task to be executed.
- Access control, only users with permission can execute tasks.
- Execute task and determine if execution succeeded, after execution react accordingly.

Usually workflow engines contain business rules engine which is a piece of software that follows rules specified by something. In theory the rules might be laws or regulations etc. However usually in IT-world the rules are specified by modeling the process with some markup language, usually xml, which the business rules engine then follows and performs tasks speci-

fied in the configuration. This enables changing and customizing the process without rebuilding the application. Sometimes it may also enable the user to change the process without knowledge of a programming.[2, 6, 7]

3 MICROSERVICE ARCHITECTURE

Microservice architecture in itself is not a new thing. For example Unix has been using this for a long time.

3.1 Microservice architectural style

Defining what actually is a microservice is not that easy. That is because there is not clear outlines what microservice is or how to create applications using microservice style. The usual definition of a microservice is shortly that it is a piece of software which is supposed to do only one thing but do it well. Usually microservices are quite standalone, they have their own process, manage their own dependencies and possibly manage their own database connection. Microservices may also have their own API for communication however in this project it was decided that it will be better to wrap microservices inside a lightweight communication layer. [8, 9]

More traditional way to build applications is the monolithic style, monolithic style opposes microservice programming style in many ways and that is why it is good to do some comparison between them. Monolithic applications are contained within single executable which contains all the logic for the application. If there are changes made to some part of the application whole monolith needs to be built again as opposed to microservice architecture where only the single service which is modified needs to be rebuild, this can happen possibly while rest of the application is functional. [10, 11]

Both microservice and monolithic applications can scale. Monolithic applications are scaled simply by running many instances of the application behind a load balancer, this is decent way to scale your application but again applications which use microservice architecture provide something more, these applications can be scaled more intelligently in a way that only services which are resource intensive can be scaled, this way resources are not wasted on multiplying services which don't actually need more instances. The difference in scaling can be seen in figure 1. [11, 12, 13]

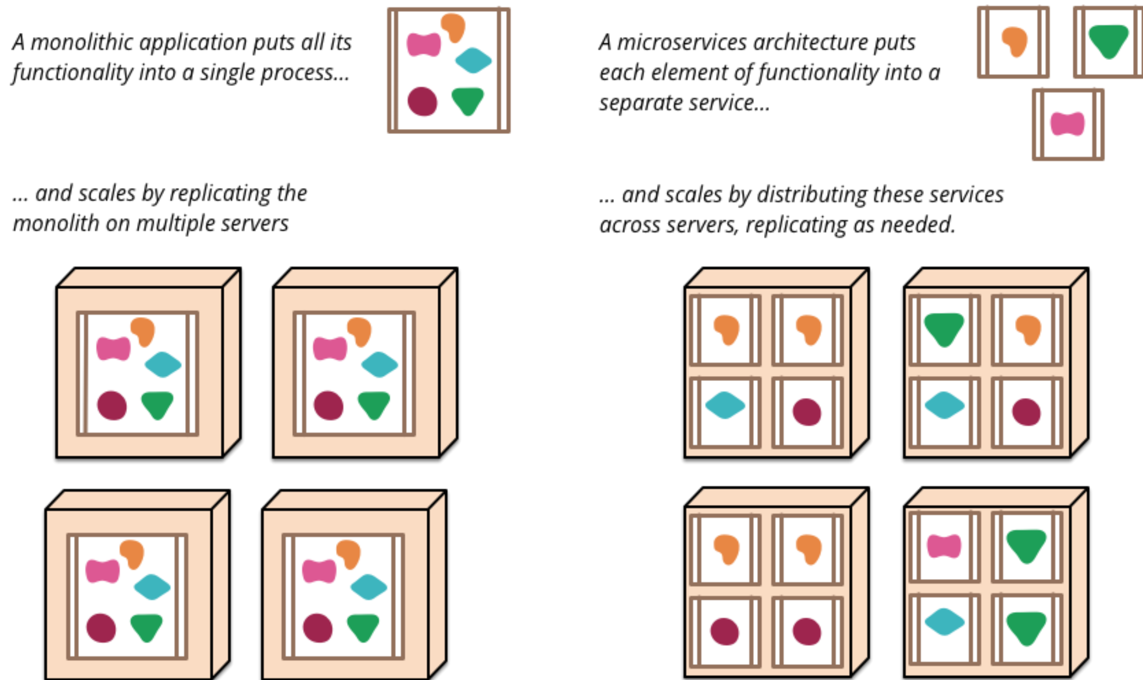


Figure 1 Scaling in monolithic and microservice based applications [11]

Another difference between monolithic and microservice based systems is database management, usually in monolithic systems there is single database which store all the data, this database uses single database technology like MySQL and all the data that needs to be stored, is stored there. In microservice based systems the situation is usually quite different since in these systems microservices usually take care of their own database connections, which means that one microservice may use for example MongoDB, other MySQL and the third one Cassandra. This can also be seen in figure 2.[10, 11, 12, 13]

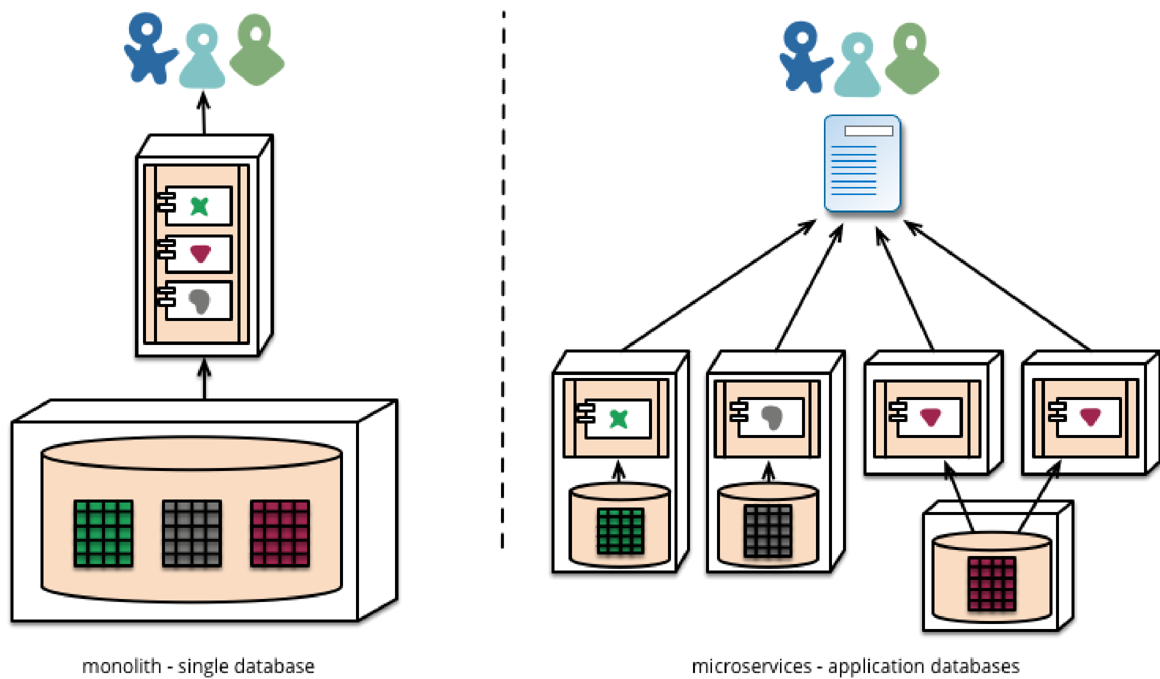


Figure 2 Database management in monolithic and microservice based applications [11]

This all brings certain benefits which make the life of a developer easier and also provide better services. Some examples are that with microservice architecture, services can be deployed independently; this means that if there is for example bug found in the system, the part of a system that has the bug can be taken down while the rest of it continues to operate. As stated before, application can be scaled smartly meaning that services which need more resources can be scaled independently. [11, 12, 13]

3.2 Benefits of microservice architecture

Microservice architecture brings quite many benefits, some of them affect straight into programs quality, while others do not necessarily affect that much to the end result, but make the life of the developer easier. Of course if developer is more interested while developing, it will affect also the end result and development costs. Conway's law says that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."[15] For example one such occasion would be that when team is building a monolithic application, teams come from technical skills like UI, middleware and databases, see figure 3.[11, 14, 15]

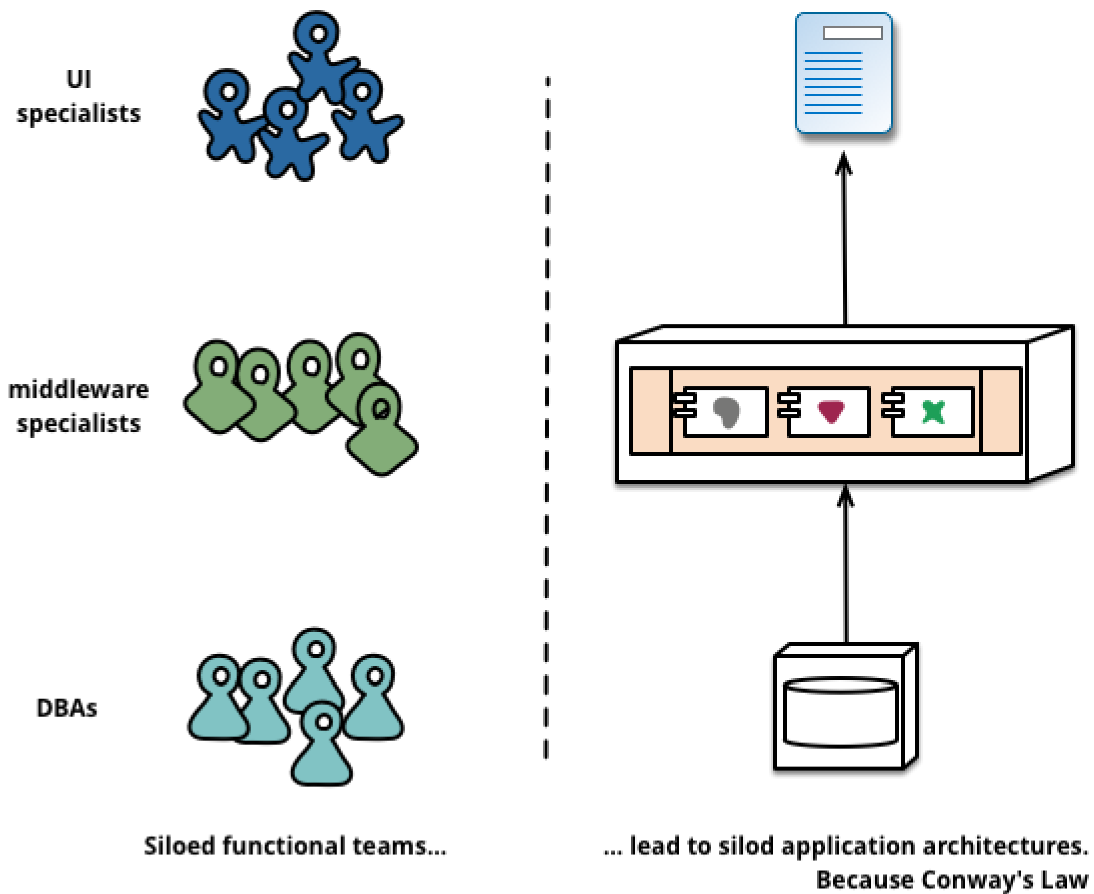


Figure 3 Teams in monolithic application development [11]

Because of this, a small modification to basically anything usually requires all the teams to work on the modification, at least in some point. This basically means that to complete the modification, multiple tasks need to be set, which then requires meetings and negotiations between teams to make it all happen which will take time. In micro-service based architecture this problem will go away because teams are constructed in a way that every team is responsible for their own service(s). Then if modification is required the team responsible of that service can perform all the modifications from UI to database, see figure 4. [11]

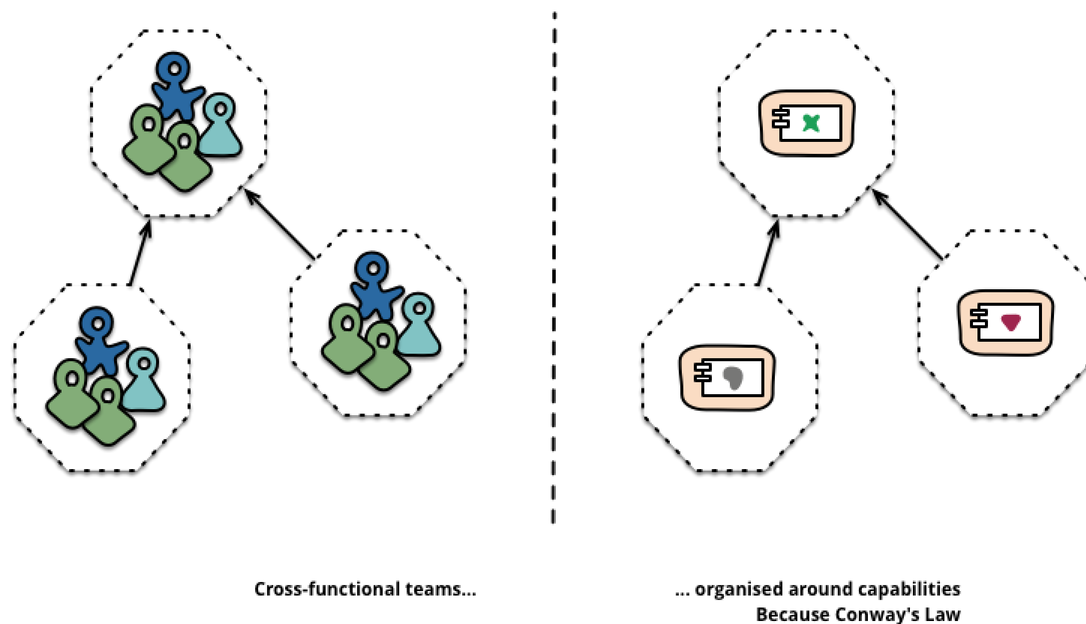


Figure 4 Development teams in micro-service based environments [11]

Micro-service based software is not a new invention, but it has just recently made its way into enterprise applications. However for example Linux and Unix have been using it already for some time. The micro-service architecture fulfills all the points in Unix philosophy which are presented in [16]:

- “Make each program do one thing well”
- “To do a new job, build a fresh rather than complicate old programs by adding new features”
- “Expect the output of every program to become theinput to another, as yet unknown, program”
- “Design and build software … to be tried early”
- “Don't hesitate to throw away the clumsy parts and rebuild them”

These design principles apply also for this project, so I will go thru each of them and explain why it is good to do it that way.

“Make each program do one thing well”, this is quite self-explanatory, but few points exist. For developer it is much easier to plan amount of work required if the goal is not gigantic software which can do “everything”. Usually this improved quality on the main point of the

software since the developer can really concentrate on building the main functionality. [16, 17]

“To do a new job, build afresh rather than complicate old programs by adding new features”, this principle helps software stay “recent”. In software development, tools, programming languages and techniques develop very quickly, this means that even quality software which is few years old, may not be as good as it could because some tools very not available while the program was developed. By using this principle new feature are always built using latest tools and knowledge. This also helps the developer because usually developers rather work with new tools and techniques rather than old ones. [16, 17]

“Expect the output of every program to become the input to another, as yet unknown, program”, this principle exists because without out it building software using microservice architecture would be very difficult. If developers follow this principle then building new softaware is easier because you can always expect that you are able to “pipe” data from another software into your software. [16, 17]

“Design and build software … to be tried early”, this principle is achieved easily using microservice architecture because microservices are lightweight and quick to build and after that they can be tried. By this so called agile development is achieved which enables developer quickly to change his plans if something doesn’t seem to work. [16, 17]

“Don't hesitate to throw away the clumsy parts and rebuild them”, using microservice architecture makes this easier because in traditional monolithic software replacing some feature usually means some refactoring also to other code but in microservice architecture if developer thinks that some feature is clumsy he can drop that microservice and develop a new one. When combined to open source this hugely improves quality of the code because everyone can easily replace “bad code”. [16, 17]

From these points it can concluded that microservice architecture makes life of a developer easier and brings large improvement to code quality. these points in mind this workflow engine was developed to work using microservices.

4 DESIGN GOALS

This chapter is about required features of a workflow engine, since the OSA-project was the primary client for the project the features are mostly planned to support the needs of digital archiving systems. Another goal of this project was to produce generic workflow engine component which can be used also in projects which are not related to archiving in any way.

There exists already quite many workflow engines, so what reason there is to develop a new one? Existing workflow engines are most meant for heavy enterprise level use. Which makes them quite hard to configure and they are generally too heavy. This workflow engine in the other hand provides only base for building workflows, with the plugin system developer can create new microservices which can then be used in a workflow.

4.1 Simple usage

Many existing workflow engines require modeling the workflows with complex xml or similar configurations and they are planned for large enterprise systems. This makes them very unattractive to use in smaller projects since the time and resources it takes to get the system running, even for testing and evaluating if it might be suitable for the project in hand, is quite much.

The main goal for this project was to create a lightweight workflow engine which would run with minimal configuration and that it would be possible to easily get the system up and running with default configuration. Also more advanced features, like clustering, should work with minimal configuration

4.2 Open source

It was decided that workflow engine would be open-source since especially when combined with micro-service architecture. It provides great benefits for systems using the engine. This is because if developer creates micro-service which is compatible with this workflow engine and publishes it as open source, everyone using the engine can use the same service in their workflows.

Another reasons to create open source workflow engine is that the main client for the project was the OSA archiving project, and in archiving, especially in long term preservation, a trust between the system provider and client is very important, and the best way to gain that trust is to say that “Our systems are open source, you can always check how we process your data and how it is preserved.” Another point is that with archiving it is very important to not be dependent on a single company offering the service with their closed source software. That is because in long term archiving, the time that objects spend in achieve can be hundreds of years, and after that time the company may not exist anymore, and if the data is preserved in the closed source archive it is not certain that you will get it back.

Because the goal of the project was to create only a prototype of workflow engine and only a single developer was working in the project, publishing the workflow engine as an open source components gives it a possibility to evolve into something which could be more widely used if enough developers contribute into making the engine better. To get enough developers to contribute to the workflow engine, it needs to be generic enough in order to interest people from different areas of expertise, so even the primary purpose was to create the workflow engine for OSA-project, for use with archiving and to work with Java programming language, the engine also needed to work with other areas and other programming languages.

4.3 Communication

One vital part of the workflow engine was going to be how the user would communicate with the workflow engine and how the engine would report back to user. Idea of building user interface integrated into the engine was scrapped because this software is probably going to be integrated into some other software so building UI would not be productive. Another point for this is emerging concept called API-driven development, which means that instead of design-

ing user interface for the backend software the API is designed first. This increases productivity since modern software has to work with different devices that have different screen sizes and so on. If the UI would be integrated into software, every device would need its own version. But when there is a good API implemented all frontend apps on different devices can communicate with the backend which saves time and eliminates repetitive work. [18]

Mainly two possible techniques on how this kind of API could be created were considered:

- Representational State Transfer (REST)
- Simple Object Access Protocol (SOAP)

To explain why Representational State Transfer or REST was chosen, I will shortly go thru these techniques.

SOAP is protocol which is designed for sending and receiving xml-based structured messages over networks. SOAP messages are always xml-based and they follow certain structure. Simplified soap message structure can be seen in Figure 5. [18, 19]

Figure 5 SOAP structure [20]

This brings one key advantage for SOAP over REST because the protocol is tightly typed and structured so processing messages can be more standardized and the overall feel is more rigid.

Drawbacks of SOAP are that it is quite complex to setup and also heavier since there may be useless xml markup send with messages to conform with the structure. [19]

Representational state transfer or REST is a stateless, client-server communication protocol. REST uses http to perform create, read, update and delete (CRUD) operations. REST was interesting option because the REST tries to be as simple and lightweight as possible and still provide enough functionality to satisfy the needs of modern software API. To ensure interoperability between systems that use REST to communicate but may evolve independently there are some defined constraints which are:

- Client-server, this means that clients should be separated from servers. This makes both client and server simpler and allows them to be developed independently. [18]

- Stateless, even REST operates by changing states of objects the API itself doesn't have a state. This means that that every REST request is "self-sustaining" and contains all the information to complete the request. This makes REST API easily scalable because new nodes can be started and they don't have to exchange state information with existing nodes. [18]

- Cacheable, REST clients can cache responses from the REST server. This means that when creating an application with REST, the developer needs to take into account that REST methods need to tell the client whether it is or is not applicable to cache the response of that specific method. [18]

- Layered system, this means that client cannot tell if it is connected to the end point or to some intermediate server, for example a load balancer, on the way. This helps with system scalability since it enables the use of load balancing. [18]

- Uniform interface, this requires a couple of key points, REST resources should all have unique way to recognize each resource, in web applications this is usually the URI. Also each REST message should hold some information on how to process it. [18]

When application conforms with all these restraints it can be considered "RESTful" which means that the application conforms to the REST architectural-style.

From these points it can be concluded that even REST is not perfect and other protocols have some features that can be considered to be “better” than RESTs equivalents, it was decided that this project will use REST API since the primary goal was to create lightweight workflow engine which can be easily customized to fit in different projects and this is something that REST provides.

4.4 Self-hosted

Since the goal was to develop lightweight, microservice based workflow engine, it was clear that the engine should be completely standalone. It means that all required components in order to run the engine should be integrated into it. Another option would have been to run the workflow engine with some application server software like Apache Tomcat or JBOSS, however using the application server to host the engine makes installation more complex, since user then has to install the server first, then configure it and then deploy the software, which is the actual engine. With standalone option user only needs to start the workflow engine, as he would start any application and the engine will take care of starting the necessary services like web server.

To make the workflow engine work as standalone software, it needed to have integrated web server in order to provide the REST API and communicate with other systems, for this task lightweight webserver called Grizzly was chosen. Grizzly was used because it is lightweight, easy to use and open source web server. Other possibilities would have been for example Jetty which supports also Java servlets, however this project only needed the http server to host the REST API so servlet support would have only made the software heavier and more complex.

5 IMPLEMENTATION

This chapter is about the actual development work. Here is the technical information of different components of the workflow engine. Also different external libraries used are presented. The development work was done with Java programming language. There was couple of reasons behind this decision, because the primary client for this project was the OSA-project, Java was a good fit since the OSA-software is Java based so after this project they can continue

development with the knowledge they already got. Also the author of the project had good knowledge of Java so it was chosen.

5.1 Core

The core functions of workflow engine are the microservice interface which is used to introduce new microservices to the engine without rebuilding the engine itself and the functions used to create workflows from micro service classes and xml configurations. It was stated in the beginning that the workflow engine should not have complex xml configurations but it was decided that minimal configuration is necessary in order to make the workflow engine more customizable.

When workflow engine starts a workflow, it starts by reading workflows.xml file. In figure 6 there is workflows.xml file with 2 sample workflows configured.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <workflows>
        <workflow name="Ping-test" defaultAction="yes" description="Test workflow to ping localhost" >
            <task name="Ping" important="true" />
        </workflow>
        <workflow name="Batch-ingest" defaultAction="yes" description="Moving files to workspace" >
            <options>
                <mongoHost option="localhost" /><!-- If some of the microservices involved in the workflow, -->
                <mongoPort option="27017" />    <!-- need some static options like database information, -->
                <mongoDBName option="sgswe" />  <!-- it can be provided here. -->
            </options>
            <task name="CleanCheck" important="true" />
            <task name="ReadMetadata" important="true" />
            <task name="SaveToMongo" important="true" />
        </workflow>
    </workflows>
</root>
```

Figure 6 Workflows.xml file with 2 sample workflows

First one is a simple test workflow which has only one micro-service that only sends ping into localhost. All workflows must be contained in the `<workflows>` tag which does not have any arguments. A single workflow must be contained inside <workflow> tag, which has arguments: name, defaultAction and description. Value of name arguments is used to identify the workflow for example when starting it thru API. DefaultAction argument can have three possible values: “yes”, “ask” and “no”, this is used when micro-service needs to ask question from user. If default action is “yes”, the question is automatically answered yes, if “no” then no. If the value is “ask”, then the workflow is suspended to wait for user interaction. Descrip-

tion argument is used to store short, human-readable description on what this workflow does. Inside <workflow> tag there can optionally be <options> tag. All tags inside <options> are provided to each microservice in the workflow as tag name -> value of option attribute map. These options can be used to do static configuration for microservices like providing database configurations. Inside <workflow> tag there are also <task> tags. Each <task> tag represents the execution of microservice, <task> tags have two arguments, name and important. Name argument is used to identify the correct Java controller class for the microservice and it is very important that it is correct. Important attribute can have two values, true or false. If execution of this step in workflow fails, the important attribute defines if execution of the whole workflow should be cancelled.

When workflow engine has all the options and list of required microservices it continues by reading xml configuration file for each microservice involved in the workflow. Example of this file can be seen in figure 7.

```
<service name="ReadMetadata">
    <description>Read metadata from files using apache tika</description>
    <exec>java -jar {pathToMicroservices}/tika-app-1.4.jar -j</exec>
    <log>/var/log/sosWe/microservices/ReadMetadata/</log>
</service>
```

Figure 7 Example of microservices configuration file

All the tags are contained inside <service> tag, which has single argument, name. Name argument is used to identify the correct configuration file for each microservice, so it should be the same that is found from workflows.xml file. Inside <service> tag can be found: <description>, <exec> and <log> tags. The <description> tag defines human readable description on what this microservice does. The <exec> tag should hold the information on what command the microservice should execute, it can be also empty if the functionality has been programmed into the Java controller. The <exec> tag exists, because if there is a simpler few simpler microservices, which for example execute command, they can all use the same Java controller, just with different configuration files. The <log> tag is used to provide path for the microservice to save its log files, this can also be url.

After the workflow engine has read all the configuration files, it starts to load microservice controller classes. Because user should be able to add new microservices without rebuilding or even without restarting the workflow engine, microservice classes cannot be loaded by

classloader during startup like normal classes, instead they are loaded dynamically on runtime when the workflow is started. For that reason every microservice should have somekind of controller written with java, the controller can then start the actual microservice which can be written with any programming language. First of all, the Java controller should implement the interface designed for microservices. This interface can be seen in figure 8.

```
public interface MicroserviceInterface extends Plugin{
    public abstract boolean execute(String input,HashMap<String, Object> options) throws Exception;
    public void log();
    public String[] caps();
    public void buildService(String name, String Desc, String exec, String log, boolean important);
}
```

Figure 8 Microservice interface.

The interface has four methods, execute(), log(), caps() and buildService(). The execute method should perform the actual task of the microservice, it can be starting an external program or the functionality can be programmed straight into the method. As parameters the execute method gets the output from previous microservice which is called input and a hashmap which is called options. The hashmap contains all the options specified in the workflows.xml file and all the dynamical parameters (see chapter 5.2) given thru api in startup. Microservice can also add values into options map, if it wants to pass something on to the next service. The log() method should perform necessary logging for this microservice, for example write into text file. The buildService function takes all the settings found from the microservices configuration file and it should configure the microservice with correct values. I created this method instead of constructor because it was simpler to first create empty microservices and configure them later. The caps() function is used to identify to which microservice this controller belongs to. Example implementation of caps() method can be seen in figure 9.

```
@Capabilities
public String[] caps() {
    return new String[] {"name:ReadMetadata"};
}
```

Figure 9 Example implementation of caps() method.

Caps() method should return the name parameter in an array, and the name should be the same as it was in other configuration files. The return type is array because in future there could be also some other parameters than name which are used to choose the correct microservice, for example filetype. When creating new microservice, user can implement this interface straight but for convenience I also created an abstract helper class which user can extends and the base

class already takes care of the log() and buildService() functions as well as updating the state of microservice, so user only needs to implement the execute() and caps() methods.

5.2 Application programming interface (API)

All communication with the workflow engine is done thru REST-api, this means that there woun't be separate UI build to control the engine but everyone can create UI of theirs choosing or integrate the engine straight into their application using the REST-api.

Couple of open source libraries exists for building REST-api with java. Couple worth mentioning are:

- Jersey
- Restlet
- RESTEasy

In this case Jersey was chosen mainly because it provided reference implementation of JAX-RS (Java API for RESTful Web Services) in Java without other "useless" components. This made sure that project stayed as lightweight as possible and still provided all the necessary functions that user may need.

Implementing REST-method with Jersey is easy, sample method which responds to HTTP GET request in path "/helloworld" and returns string "Helloworld!" as plain text, can be seen in Figure 10.

```
@GET
@Path("/helloworld")
@Produces(MediaType.TEXT_PLAIN)
public String Helloworld(){
    return "Helloworld!";
}
```

Figure 10 Simple REST-method with Jersey.

Structure of REST-method is as follows. First @GET annotation tells that this method should answer to HTTP GET request. Other possibilities are @POST, @PUT, @DELETE and @HEAD. Next annotation @Path("/path") tells the application which path should be mapped to this method. Last annotation is @Produces() which tells the application what mimetype this method should return. Jersey provides automatic encoders for example if it is specified that method should return JSON, method can return any serializable java-object which is then automatically serialized into JSON. [21]

Currently implemented REST-methods are:

- `GET: /nodes`

This method was mainly developed for debugging purposes while testing the clustering. It returns nodes currently involved in the cluster as plain text.

- `GET: /status`

This method was developed for logging and debugging purposes as well as just for following what the workflow engine is doing. It returns last 20 lines of workflow engines output log as plain text.

- `GET: /list/xml or json`

This method is for use with some kind of user interface, so the interface will know which kind of actions can be performed by the workflow engine. This method accepts one parameter which defines if the returned data should be in xml or json format. This method returns list of currently configured workflows in format specified by the parameter.

- `GET: /currentjobs`

This method was created for monitoring the operation of the workflow engine. It returns all workflows that are currently scheduled to run sometime in the future in XML format. Data also returns the information about the time when the each workflow will be executed next time.

- `GET: /{workflowname}/desc`

This method doesn't have any use in actual operation of the workflow engine but it exists as a convenience, since it is highly likely that person creating microservices and/or workflows is not the same that uses them. This method takes a single parameter, which is the name of the workflow. It then returns the description specified in configuration. With this method developers can provide clear text description of each workflow and this description is available thru REST-api in JSON format.

- `GET: /start/{workflowname}`

This method takes a single parameter which is the name of the workflow to start. It also accepts the N number of other parameters in format

`/start/testworkflow?input=/var/files&ouput=/temp` these parameters are forwarded to the workflow in a form of a key -> value map. I call these parameters dynamic parameters since they are provided when the workflow is started and not statically configured. Also if parameter with name “schedule” is present workflow is not started instantly but scheduled according to the cron expression which is expected to be the value of this “schedule” parameter.

5.3 Timing/Triggers (QUARTZ)

One of the main features of any workflow engine would be possibility to schedule the execution of workflows. It was so also in this case. Because of the limited time and resources I decided not to reinvent the wheel and used Open-Source scheduling library called Quartz to implement the scheduling features. [22]

Quartz provides many features and in many places it is considered to be the de-facto-standard library when it comes to scheduling. Usually Quartz executes jobs which are defined in standard java classes. In this workflow engine Quartz has been integrated into the workflow engine in a way that when the workflow is launched a Quartz job that contains all these microservices with settings specified in xml configuration as well as launch parameters, is generated. This Quartz job is then executed immediately or stored and scheduled according to scheduling plan. Some of Quartz's clustering functions were used in creating clustering for this workflow engine but more about that can be found at chapter 5.5 [22]

5.4 Example workflow and microservices

As part of the project I decided also to create simple ingest workflow and required microservices. This could be used for testing and as a proof-of-concept use case to prove that the engine is suitable for use in the OSA-project.

The first thing was to plan the suitable workflow for ingesting files into an archive. When the workflow with all the necessary steps was done, I created a model which represents the process. This model can be seen in the figure 11.

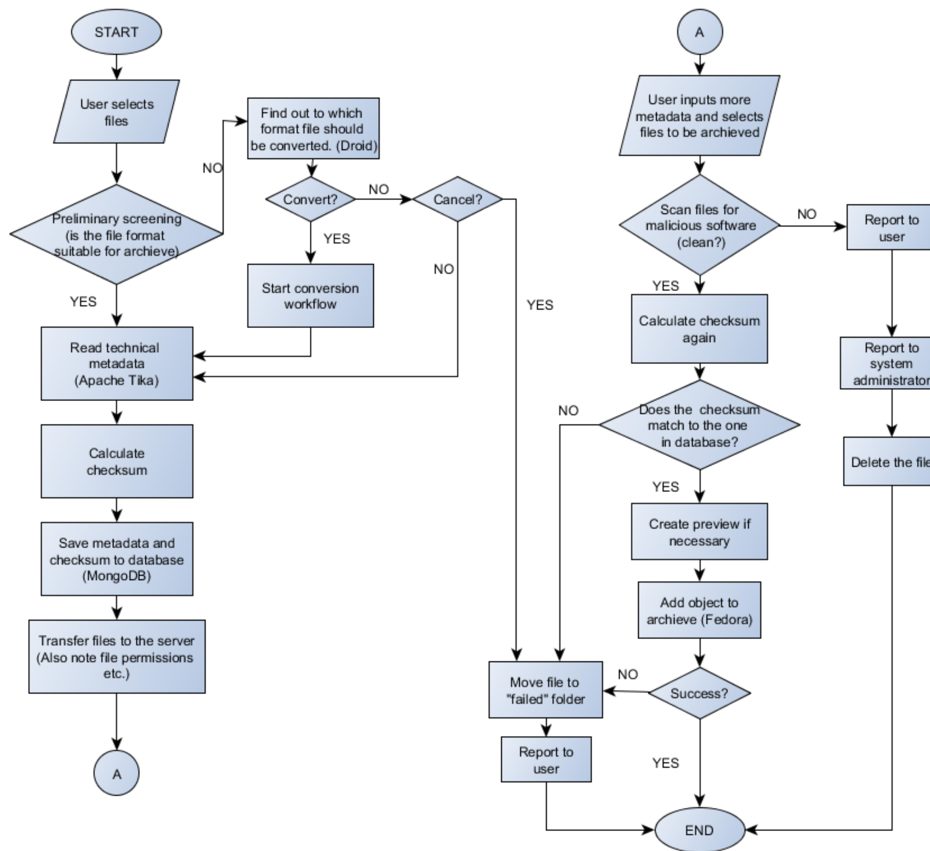


Figure 11 Ingest workflow model

When transforming this process into microservices each blue box represents a single microservice, which would need to be implemented. From figure 11, you can see that the number of microservices required for ingest process is quite high. Since the goal of the project was to create the workflow engine and not the ingest process, a simplified versions of this workflow was actually implemented. Workflow that was implemented can be seen in figure 12.

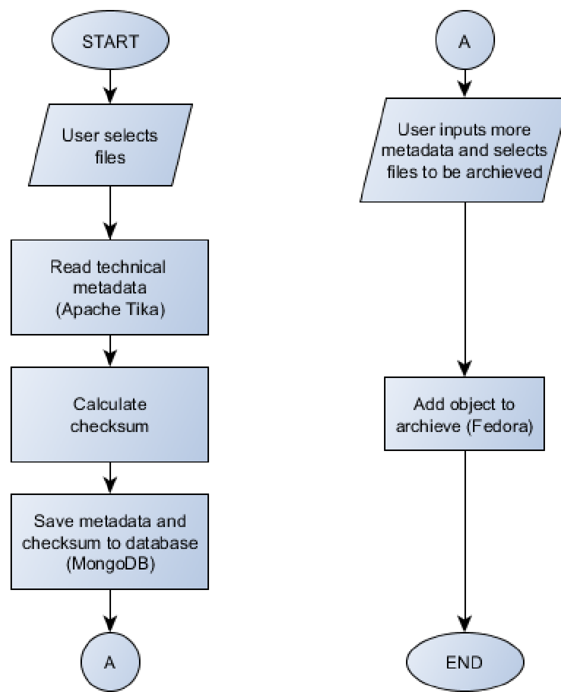


Figure 12 Implemented ingest workflow

As stated earlier all microservices required for actual ingest process in real production environment are not there. However this was not the goal of the project and with this minimal version will suffice for testing and as a proof-of-concept. Rest of the services will be done later by the OSA-project.

As seen on figure 12 the required process consisted actually from 2 separate workflows. One is used to read files into temporary workspace with automatically extracted metadata. When files are in the workspace, user can interact with them, for example add more metadata, modify metadata and delete metadata. When user has done all the required modifications, he can trigger ingest workflow which actually reads the file + metadata and saves it into Fedora Commons archive. Reason why files are not automatically archived into Fedora is that all metadata which is important may not be automatically available but it needs to be inserted by human. Here is short description on what each of the microservices does:

- Read technical metadata automatically

  This micro-service is pretty straight forward. It uses Apache Tika to read technical metadata from files automatically and then passes files on with the metadata. Apache Tika is a software which can read available metadata from almost all types of files. Tika is available as a programming language library, so it can be integrated into software, howev-

er in this case it was better to use standalone .jar file and create a small handler which will execute the jar-file.

- Calculate checksum

This micro-service calculates checksums for files so it can be guaranteed that files have not been modified while they are waiting for possible user to input more metadata. The main functionality of this microservice was to implement method which would calculate MD5-cheksum for file. This method can be seen in figure 13.

```
private String calculateChecksum(File  file) throws NoSuchAlgorithmException, IOException{
    MessageDigest md = MessageDigest.getInstance("MD5");
    FileInputStream fis = new FileInputStream(file);
    byte[] buffer = new byte[1024];
    int readLength = 0;
    while ((readLength = fis.read(buffer)) != -1) {
      md.update(buffer, 0, readLength);
    }
    fis.close();
    byte[] checksumBytes = md.digest();
    StringBuffer sb = new StringBuffer("");
    for (int i = 0; i < checksumBytes.length; i++) {
        sb.append(Integer.toString((checksumBytes[i] & 0xff) + 0x100, 16).substring(1));
    }
    return sb.toString();
}
```

Figure 13 Method to calculate checksum for file.

- Save metadata into MongoDB

This micro-service stores the metadata which has been automatically read by the previous services into database (MongoDB). This phase exists because in this point metadata may not be correct and complete so it cannot be saved into the archive, however it has to be saved somewhere and MongoDB provides excellent support for unstructured data.

- Archive document with metadata into Fedora Commons

This micro-service stores documents with collected and user-inputted metadata into Fedora Commons. The current implementation of this service is not final since currently it is using its own Fedora Commons library and sending the file + metadata using it. However OSA-project needs the archived object to be in certain format and that functionality has already been implemented into the OSA-software, but there is no API yet. In the future, when OSA-software has working API, this method will just send the file + metadata thru the API and the OSA-system will do the archiving.

With these microservices it was possible to create simple test workflow which would archive documents with some metadata.

5.5 Clustering

Because of limited time and resources for development, clustering was specified as more of an optional feature, since properly testing and solving all problems related to clustering, is quite an extensive task. Also one major problem was to make clustering work "out-of-the-box" without any xml configuration since this was one of the goals specified in the beginning.

First thing in developing clustering was to create some way to form the cluster. This was done by having a background thread listening to specified port for UDP traffic and once it receives packet which contains the cluster name and node name, it checks if that cluster name matches its own and if it does the node is added with the address of the packets sender. Then the thread responds to the sender with its own information so both nodes know about each other. This can be seen in Figure 14.

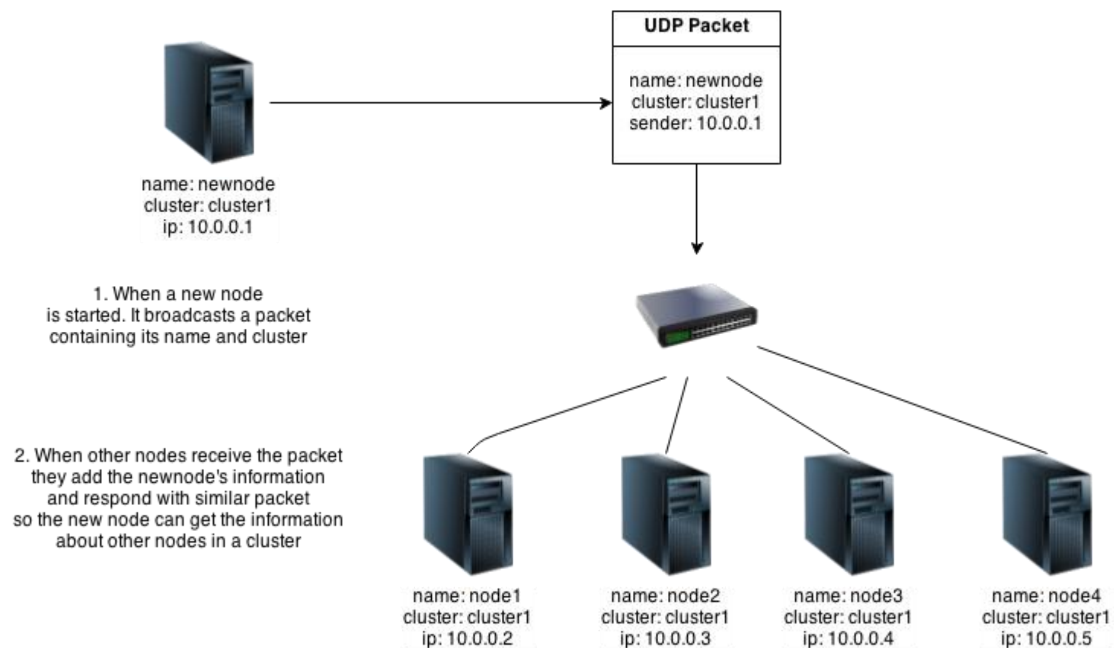


Figure 14 Forming the cluster

If the cluster doesn't match the nodes cluster the packet is simply ignored, also if the node already knows a node with the same name, or if the node name equals the receivers name the packet is ignored. The cluster name is included because this way multiple clusters can operate

within the same local area network. System currently supports clustering only within the same local area network but this is something which could be improved in a future.

Sharing the work within the cluster works by using Quartz schedulers built-in clustering functions. It works in a way that all the nodes in a cluster must share common database which acts as a storage for the workflows. The system can be seen in Figure 15.

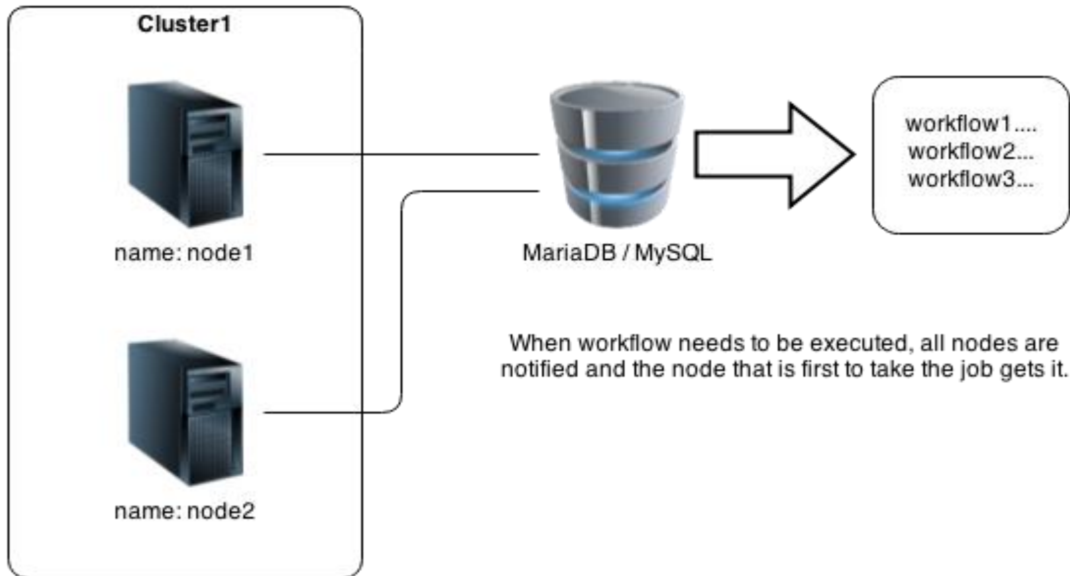


Figure 15 Workflow sharing

As seen in the figure 15, system currently supports clustering only on workflow level which is not ideal since it would be beneficial to have for example CPU intensive microservices run on node which has a powerful CPU. However because this was not the main goal of the project, this was left for the future development.

5.6 Testing

First test were of course done locally with the development machine. Once the workflow engine had gotten stable enough, I wanted to test also the clustering features, and running workflows in the clustered environment.

I created two virtual machines with Oracle VirtualBox to do some testing. These virtual machines custom Arch Linux operating systems with only minimal amount of applications in-

stalled in order to run Java console applications, see figure 16.

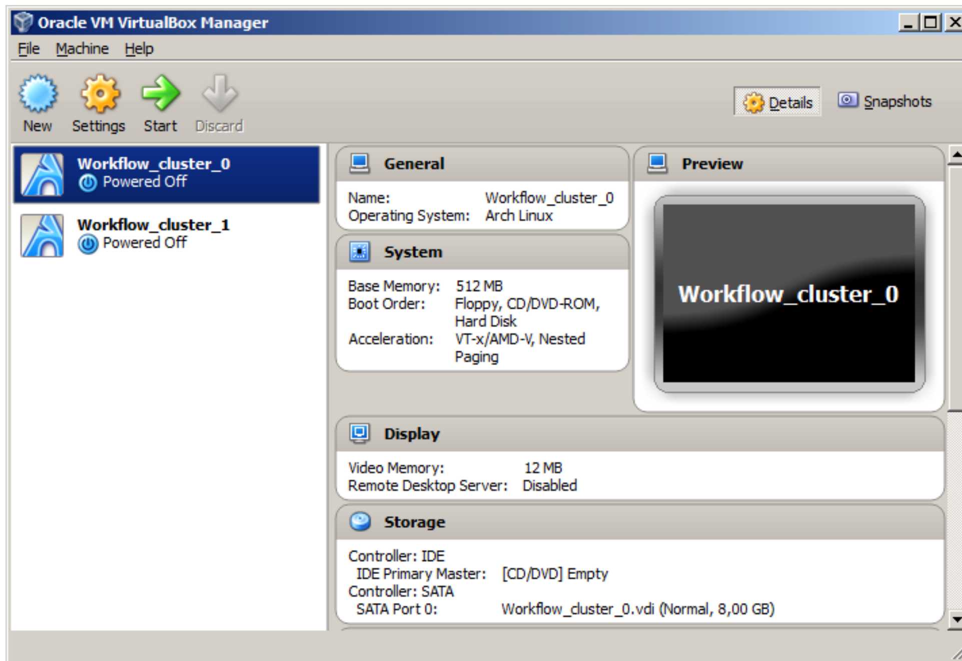


Figure 16 Testing environment.

Even it took more effort to create the testing environment using custom Arch Linux it was decided worth it since this way it would be certain that any external service or software does not interfere with the testing and cause false results. You can see workflow node that has just been started on figure 17.

```
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
com.belvain.soswe.cluster.DetectionThread>>>Ready to receive broadcast packets!
Soswe node[Captain Exuberance] started in 3163 ms
*******************************************
Clustering: true
Microservice configuration folder: microservice/microservice_config
Microservice executable folder microservice/msexecutables
*******************************************
Starting grizzly...
May 25, 2014 8:19:02 PM org.glassfish.jersey.server.ApplicationHandler initialize
INFO: Initiating Jersey application, version Jersey: 2.4.1 2013-11-08 12:08:47...
May 25, 2014 8:19:02 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [localhost:9300]
May 25, 2014 8:19:02 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
```

Figure 17 Workflow engine running on test server.

The basic system without clustering was found functional and usable, more testing should be done in the future because during this project time used to testing was very limited. With clustered environment the system was found mostly functional. Sometimes automatic cluster

forming fails, probably due to packet loss in network. This could be fixed by improving the clusters self-monitoring.

Also a lot more testing should be done regarding performance testing, meaning testing with some large file, testing with large amount of small files and testing with large amount of large files. Also testing with environments which need high fault tolerance was not done.

5.7 Client library

Even the communication with the workflow engine is done with the REST-api and it is quite easy to develop a REST-client using for example Jersey library, it was decided that a client library is necessary. Since there was not enough time to develop client library for multiple programming languages, it was decided that client library will be done for Java programming language.

The client library was based on Jersey library because I was already familiar with it since API was done using Jersey. Library was basically simple REST-client which has methods that make interacting with workflow engine easier. Client library works by first creating a client object with host, and port parameters. The client object has following methods:

- `startWorkflow(String workflowname, Map<String, Object> opts)`

  Starts the workflow specified by the workflowname – parameters, and provides dynamic parameters in form of a map.

- `getNodes()`

  Return all nodes currently connected to the cluster.

- `getCurrentJobs()`

  Return currently scheduled workflows.

- `getStatus()`

  Returns last 20 lines of output from workflow engine.

- `getWorkflows(String type)`

  Return all workflows configured in type which is specified by type – parameter.

All the methods return the response from workflow engine as string. Example of using the client library can be seen in figure 18.

```
public static void main(String[] args) {
    Map<String, Object> opts = new HashMap<String, Object>(); // Create some options
    opts.put("input", "/var/data");
    opts.put("output", "/temp");

    SosweClient client = new SosweClient("localhost", 9090); // Create a client that will connect to localhost:9090

    client.startWorkflow("testWorkflow", opts); // Start workflow named "testWorkflow" with specified options
                                                // Basically client library makes http GET request into address:
}                                               // http://localhost:9090/start/testWorkflow/?input=/var/data&output=/temp
```

Figure 18 Starting a workflow using client library

5.8 Results

The results of the project were a working prototype of a workflow engine, which can run workflows built from multiple microservices. Workflows can be scheduled to run during some time in the future and also they can be specified to repetitive tasks which will run with certain interval. Clustering features also work at least on proof-of-concept level, some bugs exist, and for example automatic cluster forming sometimes misses a node due to packet loss in network. Workflow engine is completely standalone, so it does not require any external server or such components.

As a side product there was produced documentation, part of which was automatically generated, so called Javadoc, and other part was “getting started” style quick guide on how to use workflow engine. Another side product was a client library for Java programming language to make interacting with the workflow engine easier.

Also necessary microservices that are needed for ingest process in digital archive. More complete process was also planned and modeled.

5.9 Future development

There is few potential areas of development which could be improved. Clustering features could be improved in few areas, at the moment clustering is done with Quartz library if there

is a problem with connection between nodes the cluster cannot do anything to fix it, in future nodes could communicate with each other and pass also other messages like alternate routes to some node if a part of network is down. This would of course make cluster more fault tolerant.

Another area of development would be task sharing in cluster, currently tasks are shared only on workflow level but it would bring some performance benefits of tasks could be shared on microservice level, then cluster could be formed in a way that each node would have performance in different areas and the overall costs for the cluster would be lower. Also microservices, that are not dependent on previous services, could run simultaneously. This way the workflow could be completed faster.

Also variable and function names should be more descriptive. Documentation should be a little better and more accurate and client library should cover all the functionality that workflow engine provides. Also client libraries should be available for other programming languages than Java.

6 CONCLUSIONS

After the development period there was working prototype of a workflow engine, with few example services. The design goals specified in the beginning were fulfilled quite well, some compromises had to made because the lack of time and resources. For example in the beginning it was said that this should have no extensive XML-configuration in order to operate, and the goal during the project was to make it have none, however in the end there was very small XML configuration added mostly because this way the workflow engine is more configurable and can be more flexible.

Even the primary client was the OSA-project and their needs are in the archiving field, the workflow engine remained generic enough in order to serve also other areas. This was very important and also difficult sometimes to keep the workflow engine generic, but it had to be done in order to make this interesting for other open source developers.

Clearly most of the development work went into developing the distribution and clustering features and it was still the area which had the most problems. From this it can be concluded

that distributed high-performance systems are complex and it is even more complex to develop and even more complex to make them easy and simple.

Interesting point was that even simple documentation takes quite a lot of time and effort. Also creating these convenience tools like client libraries, takes some time but is still very important in order to make software interesting for others.

The development will be actively continued by the OSA-project to make the engine more production ready and customize it more for their needs. Overall the client which was the OSA-project was satisfied with the results. Source code will also be shared as open source, so it will be interesting to see what comes from this in the future.

BIBLIOGRAPHY

[1] Osa project. Info. Website. 2014. http://osarchive.wordpress.com/about/.

[2] Dragos-Anton Manolescu. Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development. PDF-document. 2001. http://micro-workflow.com/PDF/phdthesis.pdf

[3] CEITON technologies Inc. Case Studies. Website. http://ceiton.com/CMS/EN/workflow/casestudies.html#Sector_Independent

[4] Alan R. Simon, William Marion. Workgroup Computing: Workflow, Groupware, and Messaging. 1996. McGraw-Hill.

[5] Michael Chatfield. The History of Accounting (RLE Accounting): An International Encylopedia. 2014. Routledge.

[6] Oracle Corporation. Overview of the Workflow Engine. Website. 2003. http://docs.oracle.com/cd/B13789_01/workflow.101/b10286/wfapi.htm

[7] NocSmart, Workflow Automation. Website. http://www.nocsmart.com/index.php?option=com_content&view=article&id=17&Itemid=135

[8] James Hughes. Micro Service Architecture. Website. 2013. http://yobriefca.se/blog/2013/04/28/micro-service-architecture/

[9] Chris Richardson. Pattern: Microservices Architecture. Website. 2014. http://microservices.io/patterns/microservices.html

[10] Chris Richardson. Pattern: Monolithic Architecture. Website. 2014. http://microservices.io/patterns/monolithic.html

[11] Martin Fowler & James Lewis. Microservices. Website. 2014. http://martinfowler.com/articles/microservices.html

[12] Mark Little. Microservices and SOA. Website. 2014 http://www.infoq.com/news/2014/03/microservices-soa

[13] Petri Kainulainen. The Microservice Architecture Sounds Like Service-Oriented Architecture. Website. 2014. http://www.petrikainulainen.net/software-development/design/the-microservice-architecture-sounds-like-service-oriented-architecture/

[14] Benjamin Wootton. Microservices - Not a free lunch!. Website. 2014. http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

[15] Eric S. Raymond. Conway's Law. Website.1996. http://catb.org/~esr/jargon/html/C/Conways-Law.html

[16] Stephen Abrams, Patricia Hswe, Delphine Khanna and Katherine Kott. Curation. Micro-Services. PDF-document. 2011. http://www.diglib.org/wp-content/uploads/2011/01/06micro-services.pdf

[17] Heikki Kurhinen, Mikko Lampi. Micro-services based distributable workflow for digital archives. IS&T Archiving Conference Manuscript. 2014.

[18] Leonard Richardson, Sam Ruby, RESTful Web Services. 2007. O'Reilly Media

[19] Max Ivak. REST vs XML-RPC vs SOAP – pros and cons. Website. 2011. http://maxivak.com/rest-vs-xml-rpc-vs-soap/

[20] Damien Foggon, Chris Ullman, Daniel Maharry, Karli Watson. Programming Microsoft(r) .Net XML Web Services. 2003. Microsoft Press.

[21] Oracle Corporation. Jersey 2.7 User Guide. Website. 2014. https://jersey.java.net/documentation/latest/user-guide.html

[22] Terracotta, Inc. Quartz Documentation. Website. 2014. http://quartz-scheduler.org/documentation