university of groningen

faculty of mathematics and natural sciences

SCALABLE AND FLEXIBLE MIDDLEWARE FOR DYNAMIC DATA FLOWS

STEPHAN BOOMKER

Master thesis Computing Science
Software Engineering and Distributed Systems

Primaray RuG supervisor : Prof. Dr. M. Aiello
Secondary RuG supervisor : Prof. Dr. P. Avgeriou
Primary TNO supervisor : MSc. E. Harmsma
Secondary TNO supervisor : MSc. E. Lazovik

August 16, 2016 – version 1.0

ABSTRACT

Due to the concepts of Internet of Things and Big data, the traditional client-server architecture is not sufficient any more. One of the main reasons is wide range of expanding heterogeneous applications, data sources and environments. New forms of data processing require new architectures and techniques in order to be scalable, flexible and able to handle fast dynamic data flows. The backbone of all those objects, applications and users is called the middleware.

This research goes about designing and implementing a middleware by taking into account different state of the art tools and techniques. To come up to a solution which is able to handle a flexible set of sources and models across organizational borders. At the same time it is de-centralized, distributed and, although de-central able to perform semantic based system integration centrally. This is accomplished by introducing of an architecture containing a combination of data integration patterns, semantic storage and stream processing patterns.

A reference implementation is presented of the proposed architecture based on Apache Camel framework. This prototype provides the ability to dynamically create and change flexible and distributed data flows during runtime. The implementation is evaluated in terms of scalability, fault tolerance and flexibility.

ACKNOWLEDGMENTS

I would like to use this opportunity to thank the people who supported me during this project.

I want to thank the team of TNO for providing the resources and time that we needed for finishing this project. Special thanks to MSc. E. Harmsma and MSc. E. Lazovik as daily supervisors for all the support, ideas and feedback between and during the weekly meetings.

I also wish to express my gratitude to my supervisors from the RuG, Prof. Dr. M. Aiello and Prof. Dr. P. Avgeriou for taking the time to provide me with detailed feedback on my progress.

Finally, I would like to thank my fellow students, friends and family for their support and presence. I would not have achieved this result without them.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API Application Programming Interface

BPM Business Process Management

CSV Comma Separated Values

DB Database

DSL Domain Specific Language

EIP Enterprise Integration Patterns

EOF End Of File

EOL End Of Line

ESB Enterprise Service Bus

GUI Graphical User Interface

HDFS HaDoop File System

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IoT Internet of Things

JEE Java platform Enterprise Edition

JMS Java Message Service

JNDI Java Naming and Directory Interface

JSON JavaScript Object Notation

JSON-LD JavaScript Object Notation for Linked Data

JVM Java Virtual Machine

LCIM Levels of Conceptual Interoperability Model

NAT Network Address Translation

NIO Non-blocking I/O

OS Operating System

OSGi Open Service Gateway initiative

OWL Web Ontology Language

RDF Resource Description Framework

RuG University of Groningen

SDK Software Development Kit

SLA Service Level Agreement

STOOP Sensortechnology applied to underground pipelineinfrastructures

TCP Transmission Control Protocol

TNO Dutch organization for applied scientific research

UI User Interface

URI Unified Resource Identifier

VO Virtual Object

XML Extensible Markup Language

1 INTRODUCTION

Nowadays, embedded intelligence has changed the way we live and work. For example, we like to create a heightened level of awareness about the world and monitor the reactions to the changing conditions that said awareness exposes us to. Almost all of the devices that contribute to such awareness are connected to the internet (or some network). Connecting and empowering these devices puts a lot of stress on the systems managing all the devices and data produced by the devices. This phenomenon of creating awareness using (smart) embedded devices is called the Internet of Things (IoT).

IoT is a concept that refers to transforming devices or machines (such as lights, signs, parking gates or even pacemakers) from ordinary to smart through the use of sensors, actuators and data communication technologies embedded into the physical objects themselves. IoT enables these smart devices to be virtually tracked, monitored and controlled across a wireless network. There are three fundamental functions of IoT applications: capturing data from the device, transmitting that information across a data network and taking the action based on the intelligence collected. From simple to sophisticated, there is unlimited potential for IoT applications [34].

The IoT may have been a sci-fi vision a couple of decades ago. It is a fast evolving reality of today and the future. Companies and research institutes work on large IoT projects to come to the solutions for, inter alia. Improving energy efficiency in buildings, by measuring presence and temperature, in order to control the heaters and lights more efficiently. Or the reduction of traffic congestions by measuring vehicles, air pollution, and noise levels in city centres and other crowded areas. To change the traffic flow by adjusting the digital traffic signs.

IoT related projects have to deal with a large amount of (embedded and heterogeneous) devices which produce a lot of data. This fast and diverse data is referred to as Big Data.

In general, Big Data is a term that describes the large volume of data, both structured and unstructured. However, it is more than just volume. Laney defined in his report [31] the challenges and opportunities of data growth as being three-dimensional, namely increasing volume, velocity and variety. His definition of Big Data is: "Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization."

The traditional client-server architecture can manage no more such a Big Data and the transactions over it due to a wide range of expand-

ing heterogeneous operating systems, applications, data sources and environments [5].

New forms of data processing require new architectures and techniques in order to be scalable, flexible and be able to handle fast (and dynamic) data flows. The layer of the system responsible for this is called the middleware layer, the backbone of all the objects, applications and users. The middleware is responsible for object integration, (de)coupling of objects, exchange of information, and management and support.

The middleware within IoT related systems have to deal with large, highly scalable and heterogeneous environments. Heterogeneous environments consist of a wide variety of different hard- and software components, each with their own properties. Creating a middleware platform that can integrate all the different inputs, that is flexible and is able to scale in a heterogeneous and distributed environment is not a trivial task.

Within this research we try to design and implement such middleware platform that is able to satisfy the needs in scalable, heterogeneous and dynamic (IoT related) environments.

In this research we contribute to the problem by looking at different state of the art tools and techniques to come up with a solution which is able to handle a flexible set of sources and models across organizational borders. At the same time it is de-centralized, distributed and, although de-central able to perform semantic based system integration centrally. To accomplish this, we look at data integration patterns, semantics and stream processing patterns all with their additional software tools.

1.1 TNO AND STOOP

This research is done at Dutch organization for applied scientific research (TNO). It aims at researching and implementing scalable and heterogeneous middleware for the dynamical sensor domain. This thesis research is (in the first case) created for a TNO project called Sensortechnology applied to underground pipelineinfrastructures (STOOP) [39].

TNO [46] is a Dutch non-profit organization with the goal to apply scientific knowledge in practice. TNO is involved in projects commissioned by governments and companies. Their mission is *connecting people and knowledge to create innovations that boost the competitive strength or industry and the well-being of society in a sustainable way [47].*

The goal of the STOOP project is to monitor changes in the layers of soil where underground pipelines are located. This information can be used to determine the stress rate on the pipelines. The pipeline operator is then able to improve the risk prioritization on when to replace the pipelines, and which old pipelines to replace first. The

data used in this project is distributed over different heterogeneous sources (located in different places). The raw data then needs to be transferred to different processing models in order to compute the chance of pipeline failure. STOOP contains multiple data sources and models which have to be put together as a chain to provide the desired output. Such flows of data from data source through models is a data flow. The creation and management of those data flows, and the ability to run the flow objects in parallel are important for this project.

TNO would like to create a system where having different data sources (ranging from sensors till distributed databases), helps to create an automatic data flow composed of different transformations and analysis phases regards to the actual needs of the user(s). Meanwhile the needs of a user could change through time: e.g. user would like to use a different algorithm to analyze the data, or use different data sources. The system could imply changes in data sources as well. One of the challenges to construct such a system lays in loose coupling of the large amounts of heterogeneous data sources. Another big challenge is to make it flexible. Flexibility includes dynamically adaptable to changes within the system during runtime. The middleware should serve as a unique point of coupling of any type of heterogeneous data source, data model and user.

1.2 RESEARCH QUESTIONS

We split our research in roughly three areas. First, the heterogeneity of the data sources providing sensor data imposes challenges regarding data integration and flexibility. Secondly, the data sources and the internal data flows could change, implying challenges related to flexibility and fault tolerance of the data flows. And lastly, the data should have a semantic meaning to be able to use it properly.

Regarding data integration, data integration frameworks are build on integration patterns. Those integration frameworks implement most of the Enterprise Integration Patterns (EIP)'s described in the book from Hohpe and Woolf [26]. Those patterns are used in integration frameworks as (static) building blocks to create an integration solution. Even though integration frameworks are useful for data integration, they do not have build-in functionality regarding scalability.

Concerning the change of data flows, there are techniques and patterns that allow processing of large volumes of data at high speed by supporting high level of scalability, fault tolerance and parallel processing. However, those techniques and patterns perform worse against integration.

Finally concerning semantics. Semantic interoperability is the ability of a system to exchange data with a shared meaning. It is a requirement to enable data federation between systems. There are tools

and techniques available that facilitate storing and adding metadata to the data from the data sources.

Within those three different areas we can see that there are multiple systems which are aimed at data integration, data processing and semantic interoperability. From this the main research question is:

How can middleware dynamically adapt to management of ever changing external heterogeneous data sources and processing models in a large-scale sensor data flow system?

In order to answer this question the following sub-questions are formulated:

1. *How can middleware provide internal and external data integration?* The data sources and processing models can be either local or external and need to be coupled to the middleware. Integration frameworks provide build-in components that facilitate integration. We need to look for the components and patterns that are needed to integrate the data sources and processing models within this research and which can be used to improve scalability, flexibility and fault tolerance.

2. *How to deal with the challenges imposed by scalability, fault tolerance and flexibility in a heterogeneous environment?* Using the frameworks providing data integration and processing functionalities, may result in additional challenges regarding scalability, fault tolerance and flexibility.

3. *How can the system guarantee to be fault tolerant, robust and persistent through the entire data processing flow?* We need to identify how the functionalities provided by stream processing and semantic interoperability systems can provide guarantees on the final result from the data processing flow, within the dynamic and heterogeneous sensor domain.

1.3 METHODOLOGY

This research project is split into roughly two parts of equal time length: a literature study and an implementation part.

The first half of the research consists of a study of state of the art of existing tools and techniques. The goal of this study is to find answers to (a part of) the research questions mentioned in the previous section, and to provide an overview of existing tools and techniques that can be used for the implementation. The results of this literature study can be found in Chapter 3 related work.

The second part of this research consists of implementation and evaluation. This includes formulating requirements, creating an architecture with additional diagrams (sequence- and class diagrams).

And based on the architecture, creating a prototype middleware based on the techniques selected from the literature study, including writing and testing code. And finally, an evaluation to validate the key-drivers, requirements and the feasibility of the solution.

1.4 CONTRIBUTION

As briefly mentioned before, many old solutions to integration problems are "static" and are not able to deal with dynamic, scalable and heterogeneous environments. We contribute to this problem, by making use of existing tools, patterns and techniques to create middleware that can integrate and manage different heterogeneous data sources and processing models. It is required for creating data flows that can provide the information the user asks for.

The academic novelty of the research is covered by the following four points:

1. *Handling a flexible set of sources and models, combined with ad-hoc user questions.*

   In large and complex systems with a lot of different sources and models working together, there is a need to manage and monitor all those sources and models. Most of the current (middleware) systems are able to couple different heterogeneous sources and models. However, when something changes within or outside of those components, the middleware is not able to adapt to those changes without user intervention or stopping (a part of) the flow or system.

2. *De-central system, completely distributed architecture: No central broker/Enterprise bus*

   Obtaining a scalable and flexible system that can handle a lot of different components working together requires a de-central system, where there is as little as possible centrally organized.

3. *Handling heterogeneous sources/models across organizational borders*

   Heterogeneous data sources and processing models consist of different hard- and software platforms producing and handling several types of data, following various protocols. Those sources and models are managed by different organizations, all those sources and models have to be make available to the middleware across those organizational borders.

4. *Although de-central, system integration is performed centrally (by semantic descriptions)*

   The pitfall of those large systems with a lot of different components working together, is the fact that it gets messy and unclear in time on what components do and the responsibilities

they have. Therefore, a central component is responsible for the system integration.

1.5 DOCUMENT STRUCTURE

The remainder of this document is as follows. Chapter 2 provides background information including the key drivers, quality attributes and the users. In Chapter 3, we describe the state of the art in the field of data integration, stream processing and semantic interoperability on which we base our approach. In Chapter 4, we formulate the functional and non-functional requirements and propose an architectural solution based on those requirements and the key drivers described in Chapter 2. In Chapter 5 we describe the implementation phase and the software used in our approach. In Chapter 6 we evaluate our solution in terms of scalability, fault tolerance, flexibility and the functional requirements and in Chapter 7 we present the conclusions and the future work that could be done from our research.

2 BACKGROUND

Based on the context of the system and the research questions, we specify in this chapter the key drivers, other relevant quality attributes and general categories of users.

2.1 KEY DRIVERS

Key drivers specify criteria that can be used to judge the quality of a system. They define how the system works instead of what the system should do. This section defines three key drivers that we think are the most important for this project, based on previous work from student of University of Groningen (RuG) Ruurtjan Pul [42]. With each key driver we give a definition of what we mean by this key driver and the reason we choose this key driver. Following the key drivers is a list containing our definitions of other useful quality attributes.

1. SCALABILITY The first and the most important key driver of this system is *scalability*. In general, scalability is the ability of an application to function well as it is changed in size or volume in order to meet a user need [4]. Bondi lists in [4] four general types of scalability, namely load scalability, space scalability, space-time scalability and structural scalability. In this research we focus on load scalability. Load scalability is the ability for distributed systems to grow and shrink its resources in order to be able to handle changes in load or number of inputs. We define scalability as:

   The ability of a system to change in order to handle growing usage

   Growing usage in this context means the increase in number of data sources, users, data flow components and the amount of internal communication.

   This is the most important key driver since we have to deal with large amounts of (sensor) data. We have to be able to respond to changes in the number of sensors, sensor data and users. This to increase the overall availability, performance, throughput and latency of the system and subsystems within the data flow.

2. FAULT TOLERANCE The second key driver for this research is *fault tolerance*. We define fault tolerance as follows:

   The property of a well functioning system that enables the system to remain properly operational when one or more components of the system fail or contain faults.

A well functioning system is a system that is available and responsive (see definitions in Section 2.2). Also if (external) components (sources and models) are unavailable.

Fault tolerance is an important key driver for the system. When the system is fault tolerant it can prevent data loss and increase the overall system availability. Based on the thesis from Pul [42] we have found the following shortcomings, related to fault tolerance. He mentioned those in his conclusion of this master thesis:

- When two flow graph modules are linked with different message types the system does not validate the compatibility of those modules. The user is responsible for composing valid flow graphs with the same message types and formats;
- Conversion from batches to streams (e.g., a Comma Separated Values (CSV) file to a message for each line) is also missed in that research;
- The chosen integration framework tries to send data to an external component until it is endlessly repeating the messages;
- Changes to the flow graph can not be made until the failing component recovers.

3. FLEXIBILITY The third important key driver is flexibility. The term flexible is a very broad term and can be interpreted in many ways. In the case of this research, it means building a flexible distributed system. The key to a highly flexible system is the loose coupling of its components based on a structured and modular design.

With flexibility we also mean adaptability. Adaptability is the process, in which a system adapts its behaviour to internal and external changes. In the context of this research this means changes in users, data sources and functional changes regarding the internal infrastructure see Figure 1. The internal infrastructure includes the internal configuration and the data flow. The system is adaptable if it can answer different user requests in (near) real time which requires the changes in data sources/-gathering and the data flow without external intervention.

Adaptability is important because this project has to deal with changing heterogeneous data sources, users and data flows. Flexibility improves scalability, changeability, development/engineering effort and maintainability. We define flexibility as follows:

A system is flexible if it is able to adapt to the functional and numerous changes in a heterogeneous and scalable environment.

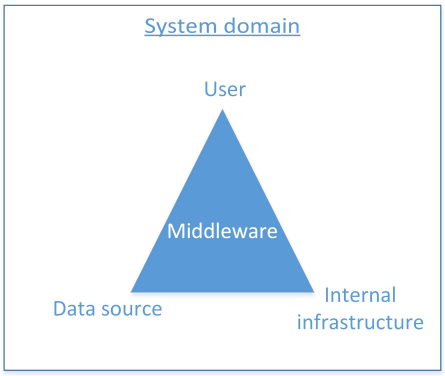

Figure 1: Flexibility context

2.2 QUALITY ATTRIBUTES

The three most important key drivers were mentioned in the previous section. This section covers other relevant quality attributes for this research.

In scope

In this research we focus, next to the key drivers, on the following quality attributes:

AVAILABILITY This quality attribute is related to fault tolerance. With this research we aim to have an availability rate of the middleware close to 100%, but the availability of the individual (internal) components is also important. The main objective regarding availability is to prevent data loss. Data that is submitted to an unavailable middleware system could be lost. Within this research availability has the following definition:

The middleware is available when it is able to accept new data or new requests from users

USABILITY Based on the system from the previous master student Pul [42] we classify usability as a relevant quality attribute. He did not mentioned usability as such, but he does mention some limitations related to usability, namely:

- Users need to understand Apache Camel's Domain specific language. Possible solution: Configuration template, so that the user can fill in a template without knowing the syntax of the Domain Specific Language (DSL)
- The user has to specify the type and format of the message when connecting two different components.
- Are there assumptions made on who or what a user is? What if the user is a computer?

We define usability as follows:

The property of how easy it is to use the system and how big is the learning curve to study the system behaviour.

Additionally, by using the system we also mean, starting with, maintaining, and monitoring such complex distributed system.

MESSAGE DRIVEN Message driven is defined as follows:

A system is message driven if it is able to handle messages asynchronously.

This quality attribute relates to highly available systems. The messages include all the messages transferred through the middleware, including sensor data, requests and transactions. If a system is message driven, the processes handling those messages are non-blocking. This improves responsiveness, flexibility and availability.

RESPONSIVENESS A system is responsive if it has the ability to respond to a task or request within a given time. On the other hand, a system is not responsive if the system is blocking or hanging while processing, or during an error or crash. We formulate responsiveness as follows:

The system is responsive if it is able to respond timely to (near) real time requests, whether the response is positive or negative.

A responsive system improves availability, flexibility and fault tolerance.

PERFORMANCE In the context of this project performance is defined as follows:

The detection and processing of changes related to throughput

We define throughput here as the rate at which data messages can be processed. In this research it is not important to strive for a high throughput, but instead detect and respond to changes related to throughput to increase flexibility, responsiveness and availability.

ROBUSTNESS In the context of distributed systems, robustness is the property that data and transactions will survive permanently. We define robustness as:

The system is robust if it is able to recover from unforeseen events.

Unforeseen events include power loss, restarts and crashes. The main reason to create a robust system is to prevent data and transaction loss and improve therewith the overall availability and fault tolerance of the system.

INTEROPERABILITY Within this research we define interoperability as:

The ability of a system to be able to operate on different Operating System (OS)'s and hardware

This quality attribute is not as important as the other mentioned before, but it can be a useful addition to the system when the system will actually be used. It improves the flexibility and usability.

Out of scope

In this research we are not focusing on the following quality attributes:

PRIVACY For this research we are not focusing on privacy. This research is not for one particular project, the use case for now is STOOP, but can be used in other projects in the future with different privacy demands. This makes it hard to define and implement privacy requirements. And because of the amount of work and time needed for this, we decided that this quality attribute does not have the same priority as the others for now.

SECURITY The second quality attribute that is out of scope of this research is security. Security is a very broad and large topic and since we are focusing on three main key drivers mentioned before, security is left our for this research. Security is still important, and will probably get attention in future work. However, some tools and techniques have build-in security features, so in that case, the security is handled by the used techniques.

2.3 USERS

Since this project is not specially focused towards a particular project, but planned to be used in many different domains, it is not possible to point out the exact users of the system. However, we can mention different general user categories, see Table 1.

USER	ROLE	DESCRIPTION
MODEL OWNER	Data/Domain specialist	The data/domain specialist is concerned about which data sources are available to be able to run the processing model(s)
DATA SOURCE OWNER	Company with data of interest, can also be organisations or governments (e.g. open data)	The data source owner wants to be able to connect, to change their data and to have insight into his sources through the middleware
FLOW DESIGNER	Manager, data analyst, data scientist	A flow designer is interested in the data from the data flows. This user wants to be able to see, create and change data flows, and request data through the UI.
SUPERUSER	Administrator, developer, maintainer	A superuser has extra rights and access to all the middleware related components. This user can manage all the functionalities and components of the middleware.

Table 1: Users

3 RELATED WORK

There has been a lot of work done in the field related to our research topic. Starting off in the field of data integration, which includes the state of the art on the EIP's, and an overview and comparison of the current available open source integration tools. The second part of this chapter goes into the theory and state of the art in the field of patterns of stream processing together with an overview on the current data processing tools. The last section contains the discussion about existing solutions and different levels of semantic interoperability. This related work helps in finding answers to the research questions presented in Section 1.2.

3.1 DATA INTEGRATION

Traditional integration systems typically contain specific code for each project to access data or systems. This results in static "point to point" integrations [8]. Meaning the implementation of a channel component between every combination of two systems that had to communicate. This "point to point" integration is feasible in case of a small system where only a few applications have to communicate. But when more heterogeneous applications need to communicate, the data flow between the systems quickly becomes messy and unclear.

Nowadays the data exchange between companies and systems increases a lot and due to the IoT the number of heterogeneous applications that have to be integrated increases as well. This means dealing with different technologies, interfaces, data formats and protocols [52].

Hohpe and Woolf presented a book [26] back in 2003 about Enterprise Integration Patterns (EIP's). A pattern is an advice about a general solution for frequently occurring integration problems. They present them as building blocks that can be combined and together make up for an integration solution. Those patterns can help understand the responsibilities and challenges of data integration. The first part of this section contains descriptions of relevant integration patterns.

After the EIP's, this section continues with an description on the different kinds of tools that use the EIP's to come to a solution for a particular integration problem, and this section finishes with comparison between four selected tools.

3.1.1 *Enterprise integration patterns*

When two or more applications want to connect to each other via an integration solution, a number of things have to happen to make those applications interact. Those things combined together make the middleware.

In [26], Hohpe and Woolf categorize an integration solution into a couple of basic elements, see Figure 2. The elements are used to categorize the 65 integration patterns [25].

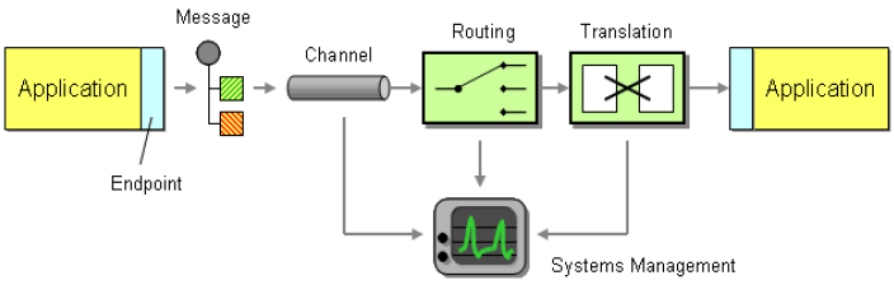

Figure 2: Basic elements of an integration solution [26]

In order to integrate multiple applications, data (*message*) has to be transported from one *application* to another. Each *application* needs an *endpoint* to be able to connect to the integration solution. When the *application* is connected to the integration solution, a *channel* is used to move *messages* between *applications*. If a number of *applications* are connected, the middleware has to take care of sending *messages* to the correct *application*. This is done by the *routing* component. Now that we can send *messages* from one *application* through a *channel* to the correct *application*, we have to convert the *message* into the data format of the other *application*. Because of the heterogeneous environments of most integration projects, one of the difficult tasks of an integration solution is the agreement upon a common data format. The *translation* component is responsible for this data conversion. Finally, in order to have control over a complex integration solution with multiple *applications*, data formats, *channels*, *routes* and *translations*, we should monitor data flows and make sure all components are available to each other. This can be accomplished by a *management system*.

Now that we have an overview of the basic elements of an integration solution, we describe the integration patterns within those elements. The patterns are used to define a more detailed description of frequently occurring problems in complex data integrations. Based on the patterns described in [26], we selected some patterns which are relevant to our integration problem.

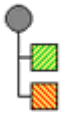

Figure 3: Message

Message. A data integration system is all about sending, receiving, routing, monitoring and transforming messages. A message is a packet of data that can be transmitted on a channel. To be able to transmit data, an application must split the data into one or more packets and wrap each packet as a message. The receiver of the message must extract the data from the message to process it [26]. A message can contain raw measurements, a command, a document, a request etc.

The data within a message can have different properties:

- Self descriptive: A message is self descriptive if the message contains data as well as the metadata that describes the format and the meaning of the data. Self descriptive data formats are Extensible Markup Language (XML) and JavaScript Object Notation (JSON).

- Structured/unstructured: Structured data is data which is organized, like data in a relational database. Structured data is easy to link, query and display in different ways. A structured data format is JavaScript Object Notation for Linked Data (JSON-LD), see Section 3.3.3. Unstructured data is not organized according to a predefined structure which makes it more difficult to understand.

- Defined/undefined: Defined data has a semantic meaning. This meaning can consist of handling rules, unittypes and a vocabulary (word meaning) stored in a data store, see Section 3.3.4. Unstructured data is the opposite, the semantic meaning is unknown, which makes it harder to understand and process.

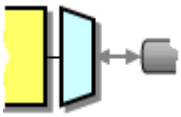

Figure 4: Channel adapter

Channel adapter. A part of an integration system is the ability to couple (heterogeneous) applications to the middleware. A channel adapter is categorized as a channel pattern. It can access the application's Application Programming Interface (API) or data and publish messages on a channel based on this data, and that likewise can receive messages and invoke functionality inside the application [26].

A channel adapter can behave as a message endpoint, which is developed for and integrated into an application, see Figure 2.

A channel adapter is often combined with a message translator (see Figure 9), to convert the application-specific message to a common format used within the channel of the middleware. Resulting in an abstraction between the middleware and the applications.

A variation on the channel adapter is the metadata adapter. The metadata adapter extracts data that describes the data formats of the application. This metadata can be used to configure message translators or detect changes in data formats [26].

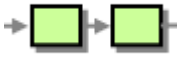

Figure 5: Pipes and filters

Pipes and filters. As stated before, an integration solution is typically a collection of heterogeneous systems. So it may occur that different processing steps need to execute on different (physical) machines. Resulting in a sequence of steps in a way that each processing component is dependant on other components. Pipes and filters is an architectural style categorized as a routing pattern to divide a larger processing task into a sequence of smaller, independent steps (filters) that are connected by channels (pipes) [26]. When using a common interface or adaptor for connecting the pipes with the filters, this pattern can be used to create chains of loosely coupled pipes and filters. This in order to develop independent, distributed and flexible processing flows. Many routing and transformation patterns are based on this pipes and filters architecture.

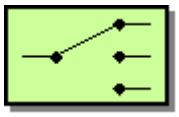

Figure 6: Message router

Message router. A message router is an addition to the pipes and filters architecture. It can be seen as a filter which consumes a message from one channel and republishes it to a different channel based on a set of conditions [26]. A message router differs from a filter in the fact that it has multiple output channels, see Figure 6. But the components surrounding the message router are unaware of the message router thanks to the decoupling property of the pipes and filters architecture. Message routers themselves are stateless, so they do not modify the message and only provide the routing to a destination. In most cases, message routers are combined with a message translator or a message adaptor.

A variation to the message router is the Content-based router, which routes the messages based on content. Such routers are commonly used to perform load balancing or fail-over strategies.

Additionally, some variants of the message router connect to a control bus, so the router can be controlled without changing code or interrupting the current flow, more about the control bus later, see Figure 14.

The following two patterns are variations to the message router.

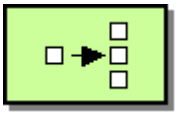

Figure 7: Splitter

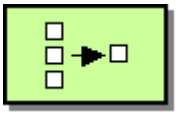

Figure 8: Aggregator

Splitter. A splitter breaks a composite message into a series of individual messages, each containing data related to one item. So multiple elements from one message can each be processed in a different way.

Aggregator. An aggregator does the opposite of the splitter. Which is collecting individual messages until a complete set of related messages has been received, then it publishes a single message distilled from those messages [26].

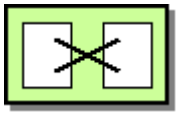

Figure 9: Message translator

Message translator. A message translator is a filter which translates one data format into another. The data need to be translated into a common format that all connected applications can understand. This is especially useful in an integration solution with multiple heterogeneous applications, all of which produce data in their own format. In this way the message translator offers decoupling and limited dependencies between applications. However Hohpe and Woolf [26] state that changing an application's data format is risky, difficult and requires a lot of changes to inherent business functionality. For instance when different applications produce data in a common format. It can still occur that they use different tag names. Or one application sends a CSV file with HyperText Transfer Protocol (HTTP) while another application uses XML files over TCP.

In order to overcome this problem the translation has to take place on different levels, namely transport, data representation, data types and data structures.

- Transport: The transport layer is responsible for transferring data between applications. It has to deal with the integrity of data while being transported across different communication protocols.
- Data representation: As the name implies, this layer defines the representation of the data. The transport layer transports characters or bytes and the data representation layer encrypts, decompresses and converts it into strings and eventually into common known formats like XML.
- Data types: This layer defines not only the data types, like strings and integers the application is based on, but also the representation of the data itself. For example, the notation of a date in Europe is different from the one used in America
- Data structures: The highest level of data translation, describing the data at the application level. It has to deal with entities and the relations associated with the entities.

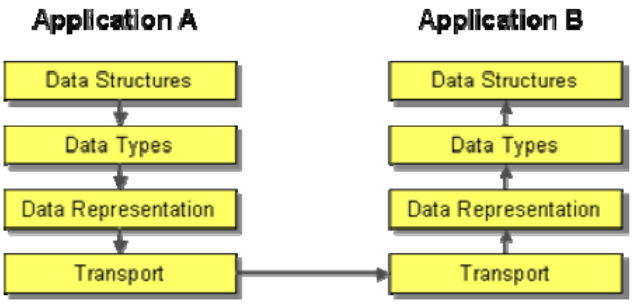

Figure 10: Levels of data translation [26]

Many integration and communication scenarios need more than one layer of data translation. The advantage of having layers of translation, is the fact that they can be used independent of each other. This way, you can choose to work at different levels of abstraction, see Figure 10. The next three patterns are variations to the translator pattern.

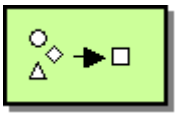

Figure 11: Normalizer

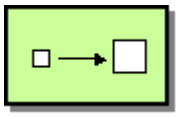

Figure 12: Enricher

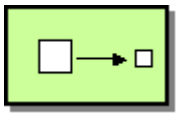

Figure 13: Content filter

Normalizer. The variety of incoming messages need to be translated into a common format. Those messages are of different types, so they need to be transformed by different translators. A normalizer routes each message type through a custom message translator, so

that the resulting messages match a common type [26].

Content enricher. A content enricher uses information from an incoming message to enrich it with missing data. The missing data can be obtained by computation, environment or another (external) system.

Content filter. A content filter does the opposite of the content enricher. It removes the unimportant data from a message to produce a message with only the desired items.

Figure 14: Control bus

Control bus. The three most important key drivers of this project are scalability, fault tolerance and flexibility, see Section 2.1. A distributed and loosely coupled architecture provides the scalability and the flexibility. But simultaneously poses some challenges regarding the management, control and therewith the fault tolerance of such distributed system. Next to the need to know if all components are running, the dynamic behaviour of the system needs to be monitored to make adjustments during runtime.

A control bus is used to manage an integration system. The control bus uses the same messaging mechanism used by the application data, but uses separate channels to transmit data that is relevant to the management of components involved in the message flow [26]. The components are able to subscribe to those channels, which are connected to a (central) management component.

A control bus can be used for the following types of messages:

- Configuration: Those messages are used to change the configurable parameters of each component. Examples of such parameters are channel addresses, data formats, time outs etc.
- Heartbeat: Heartbeat messages are send periodically to verify to the control bus that the component is available and functioning properly. A heartbeat message can contain additional information about the state and history of the component.
- Exceptions: Each component can send their exception messages and exception conditions to the control bus to be evaluated or processed.

- Statistics: A control bus can be used for collecting statistics about the components such as throughput or number of messages processed.
- Live console: The messages collected by the control bus can be aggregated to display in a console, which is used by operators or administrators of the system.

Conclusion

The EIP's from [26] are covered in the first part of research into the field of data integration. Those patterns define a detailed description about a solution to frequently occurring problems. Based on the basic elements of an integration solution, the patterns relevant to our research were selected.

However, those patterns still not answer the complete research question. Firstly, solutions regarding fault tolerance of the data are missing, the patterns are not able to deal with late and incomplete data. Secondly, knowledge about the metadata of a source is needed in order to integrate heterogeneous data sources. Finally, each pattern on its own solves a part of a specific problem. However, the combination of multiple integration patterns (statically) connected is not very flexible and dynamic.

The next section goes into the different integration tools currently available which implement the EIP's.

3.1.2 *Integration tools*

To be able to integrate different applications and data sources, a standardized architecture model or interface is needed for interaction and communication between mutually interacting software applications [52].

To come to a solution for a particular integration problem mainly depends on the complexity of the integration task, see Figure 15. When the task includes the integration of two or three different technologies, then writing a custom implementation is the simplest and fastest to do. However, when it gets more complicated, tools that are made to do those integration tasks are needed.

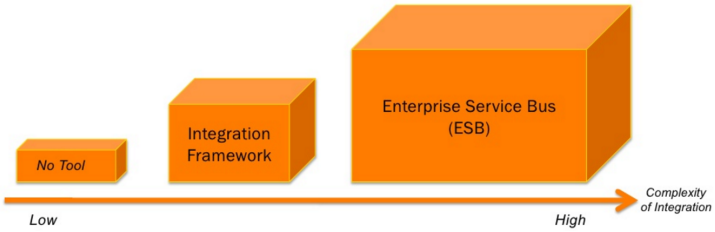

Figure 15: Complexity of the integration [52]

In very complex cases use an Enterprise Service Bus (ESB), or even more complex use an integration suite. They offer extended features like a registry, a rules engine, Business Process Management (BPM) and Business Activity Monitoring.

But if a sophisticated graphical designer, code generator and commercial support are not needed, an (lightweight) integration framework is a good choice.

Both ESB's and integration frameworks have their own properties. It is important to choose the right method and tool for the problem to reduce complexity and unnecessary work. The next subsections describe the two integration solutions.

Enterprise service bus

Mason [33] describes an ESB as an architecture with a set of rules and principals for integrating numerous applications over a bus-like infrastructure.

The concept of ESB was born out of the need to have a more flexible and manageable way to integrate multiple applications and getting off the static point-to-point integration principle. The basic principle of an ESB is to integrate different applications by putting a communication bus between them. So instead of applications talking to each other, each application talks to the bus. This decoupling of applications reduces the mutual dependencies.

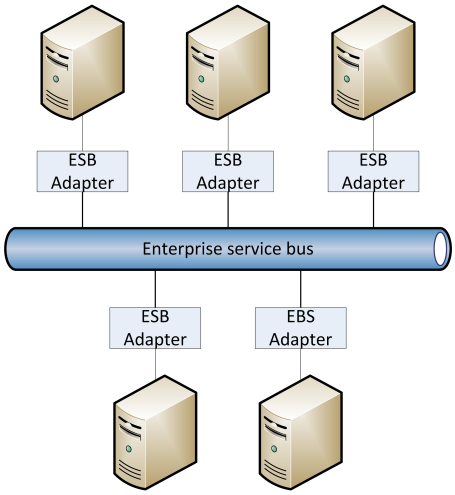

Figure 16: Enterprise service bus

Many of the EIP's described earlier are involved in an ESB. The ESB is responsible for routing messages to the correct destinations, so it relies on message routers. Each application (displayed as a desktop PC) is connected to the ESB through an adapter, as shown in Figure 16. Those adapters contain functionalities similar to the message transformer and channel adapter. For instance the communication with

the applications and converting the data into a common ESB data format.

One of the main advantages of an ESB is flexibility, the architecture is build to connect additional applications. It is also easy to connect new applications because the back end applications are abstracted by adapters. This makes an ESB suitable for systems with a lot of different applications. For example, for IoT related systems.

However there are some considerations to be taken into account before using an ESB [32]. First of all, the ESB is a single point of failure because there is only one ESB. This reduces scalability since all the data traffic has to go through the ESB. Having one ESB makes it also hard to manage, an ESB requires ongoing management and control over the flow of messages and routes to ensure the benefit of loose coupling. Incorrect, insufficient or incomplete management of messages and routes can result in tight coupling. Which makes an ESB not suited to create dynamic (streaming) data flows.

Secondly there is the fact that ESB software normally includes a lot of features which you probably do not all need and makes using it complex and hard to learn.

Finally there is the extra overhead. Every message has to go through the ESB which increases latency. And in addition, each application needs to have his own adapter. While you want to make an adapter as standard as possible so it can be stamped out quickly for new applications.

So, the lack of scalability and (management) flexibility makes an ESB not the ideal solution for our research.

Integration framework

In comparison to an ESB, an integration framework is not an architecture, it is a framework that implements the EIP's and is usually implemented as library. It can exist in any language. However most integration frameworks are realized by using Java Virtual Machine (JVM)'s, since almost all major software vendors use it.

With the use of an integration framework, the developer does not have to write a lot of glue code himself. Connectors, translators, DSL's and EIP's are already implemented in the framework.

Some benefits of integration frameworks are the fact that an integration framework is (in general) more lightweight than an ESB, can be added to a existing project as libraries, feature great flexibility and is open source[9]. Resulting in the fact that integration frameworks are widely supported by other (open source) tools, making it a powerful platform for a variety of integration tasks.

However, an integration framework is just a framework which means no Graphical User Interface (GUI) and more coding, debugging and analyzing is necessary. Furthermore, vendors usually do not offer commercial support for integration frameworks. Which makes it harder

to integrate commercial products into an integration framework related application. They typically support their own products, but this is also true for ESB's [9].

Conclusion

The decision on which tool to use depends on the complexity of the integration task. Based on this, two types of integration tools were selected, the ESB and the integration framework.

It turned out that the ESB is not the best solution to our problem, due to the lack of scalability and flexibility.

A better solution to our research question is an integration framework. It is implemented as an open source library, so it is more flexible and has more support for other tools.

However, an integration framework can not solve the whole research problem. We still need support for data integrity, flexible flows and semantics.

The next section contains a comparison between different integration tools to see which one them has the best fit for our research.

3.1.3 *Integration framework comparison*

Currently here are three integration frameworks available in the JVM environment, namely Spring Integration, MuleESB and Apache Camel [52]. In addition to those three is an integration toolkit called Openadapter included in this comparison, because Openadaptor uses the same integration principles as the integration frameworks and TNO uses Openadapater as well in projects and applications. All four are lightweight and open source tools which implement the EIP's [25] and provide support for connectivity, routing and data transformation.

The remaining of this section contains a comparison between the four mentioned integration tools.

Openadaptor

Openadaptor [38] is an open source lightweight integration toolkit. It provides a set of components and the means to use them to interconnect various systems and middleware solutions. As the name implies it is mainly focused on adaptors for decoupling of components/sources.

Pros and cons of Openadaptor

Openadaptor is completely open source, which makes it free to use and it has an open community that provides a lot of examples which makes it easy to learn.

The adapters of Openadaptor are assembled through XML configuration files based on the Spring framework, which reduces coding

effort and makes it easy to embed into Spring applications. But increases configuration overhead and reduces flexibilty due to a static configuration files. Furthermore, using the configuration of the Spring framework means it uses the same basic support for the most used technologies [30].

In contrast to the integration frameworks, Openadaptor does not mention support for the EIP's and their focus is less towards routing and data transformation. And adding to the lack of functionalities in order to solve our problem is that, as of now, the Openadaptor community is not very active any more. Their current production version dates back to November 2011 [37]. We can not use technology to solve our problems, that is not active at the market any more. So other choices should be made.

Spring integration

Spring integration [44] is part of the Spring framework, it provides an extension of the Spring programming model to support the EIP's. It is mainly used within Spring based applications to support integration with external systems.

Their goals are to provide a simple model for complex enterprise integration solutions, asynchronous message-driven behaviour within Spring applications and promote intuitive incremental adoption for existing Spring users. Their key drivers are loosely coupled, separation of concerns, reuse and portability [11].

Pros and cons of Spring integration

Spring integration is part of the Spring project, which makes it an useful addition to an already existing Spring project and makes it easy to learn by Spring developers. But the fact that it is heavily related to Spring, makes it less attractive to embed into other projects.

In comparison to the other two integration frameworks contains Spring integration the least adaptors. There is just the basic support for the common technologies, which is fine, until an unsupported one is needed.

Spring integration has a designer for Eclipse and IntelliJ, but are not as good as the other two. As far as coding goes are the integrations implemented in XML code, but recently there is support for Java and Scala.

Our research and implementation starts from scratch and is not flexible when it depends on a certain framework from the start. This makes it hard to embed into other projects in the future. So spring is not generic enough for this research.

MuleESB

MuleESB [36] is, as the name implies, is an ESB. This means as a full ESB it includes additional features over a standard integration framework. However MuleESB is included in this comparison because it can be used as a lightweight integration framework by just not adding and using those additional features.

Pros and cons of MuleESB

In comparison to the other two integration frameworks is MuleESB not completely open source. It has two versions. One called Community which is free and one more production focused version called Enterprise which is not free. They also have a commercial visual designer called Fuse Integrated Development Environment (IDE) and a free one as an Eclipse plugin. The fact that MuleESB is not completely open source, means that it has a more gated community than the other two. In this comparison we focus on the open source version of MuleESB.

The amount of adapters supported by MuleESB lies between Spring and Camel, but MuleESB offers support for some special ones which the other two do not support.

MuleESB only offers an XML DSL, but this one is easier to read than the one from Spring, which is useful if the integration gets more complex.

Finally, MuleESB has no Open Service Gateway initiative (OSGi)-support, Java's component model support.

To solve our problem we need to have a generic framework with wide support and flexibility. MuleESB does not provide us with the necessary flexibility due to the lack in supported DSL's and the fact that MuleESB is not fully open source.

Apache Camel

Apache Camel [16] is an open source Java Framework consisting of a small library with minimal dependencies for easy embedding in to Java related applications.

Pros and cons of Apache Camel

Apache Camel is completely open source project driven by an open community, so it has a large community and there are a lot of examples which makes Camel easy to learn.

In comparison to the other two contains Camel the most adaptors, and the developer is able to create his own with Camel archetype. It also has Java, Groovy, Scala and XML DSL's, which makes is easy to read when the integration code gets complex. The same visual

designers are available as with MuleESB, the commercial Fuse IDE and a free Eclipse plugin.

Camel can be deployed as a stand alone application, embedded in a web- or Spring container or in a Java platform Enterprise Edition (JEE)- or OSGi environment, which makes Camel very scalable. And with Camel you are able to test the code with a Camel extension of JUnit.

Key driver validation

Table 2 shows an overview of how well the tools performed against the three key drivers.

KEY DRIVER	OPENADAPTOR	SPRING	MULE ESB	CAMEL
SCALABILITY	×	✓	×	✓✓
FAULT TOLERANCE	×	✓	×	✓
FLEXIBILITY	×	✓	✓	✓✓

Table 2: Key driver validation

Apache Camel performed the best against scalability. Camel is able to distribute the load over different instances through load balancing and Camel can be deployed on different kinds of servers and in containers. A container is an isolated instance on a file system that contains everything to run your program, regardless of the environment. Spring can also be deployed in so called Spring containers. MuleESB provides a clustering/distribution model, which is not available in the free version. Since we are comparing open source tools, the free version of MuleESB is rated with a cross. Openadaptor however does not provide a lot of information about scalability apart from having a publish/subscribe architecture.

In terms of how fault tolerant each tool is, is hard to say. Fault tolerance of the tools can be related to software and hardware(high level) faults and failures.

All of the tools provide some form of error and exception handling regarding software failures. Openadaptor provides some simple exception handling while the other three provide more extensive information on how to be fault tolerant with exception strategies, error handling and routing patterns. Furthermore, all four tools contain functionality regarding persistence. Spring Integration and MuleESB have persistent stores for messages and objects. OpenAdapter and Camel have persistent delivery with Java Message Service (JMS) queus

and additionally, Camel also has options to use file systems and databases.

The three integration frameworks provide clustering, which allows them to prevent data loss and improve the availability when one or more (hardware) components fail. Camel provides re-routing, call-backs and onCompletion functions to deal with failing components. MuleESB has a backup mechanism and transaction roll-back functionalities, however they are not provided in the free version. Spring uses queues to prevent data and transaction loss.

Finally, Apache Camel performed the best against flexibility. Camel has the most adapters of the tools and the most choice in DSL's. Furthermore provides Camel integration support for a lot of different tools, like ESB's and data flow/processing systems. Finally it is able to change the route and adapt to changes during runtime by loading a new routing file. Spring can choose dynamically sources and destinations through the patterns that are used, but is not able to adapt to changes within the sources and destinations during runtime, the same is true for MuleESB.

In conclusion, an integration framework is useful when your integration problem is too complex to write your own and you do not want all the extra additions from a full ESB. An integration framework is easy to embed and use since it is a library with additional functions to create flexible flows between applications, containing different EIP's. We have compared four integration tools. They all provide support for connectivity, routing and data transformation. But there are some differences regarding their performance against the three main key drivers.

3.1.4 *Summary*

There has been a lot of work done around the topic of data integration. As starting point, we looked into the theory of the EIP's from [26]. Those patterns define a detailed description about a solution to frequently occurring problems. Based on the basic elements of an integration solution, we selected the patterns which are relevant to our integration problem. Then we looked into two types of integration tools, namely the ESB and the integration framework. Based on the advantages and disadvantages we compared the four main open source data integration related frameworks available. It turned out that there are some differences in the tools regarding their performance against the three main key drivers, the results where in favour of Apache Camel.

However, by itself Apache Camel can not solve all the problems to deal with our research questions, because Camel lacks in support for the data itself, like data integrity, completeness and additional seman-

tics. We decided to use Camel as one of the parts of our solution, but we need to enrich it with stream processing and semantics.

3.2 STREAM PROCESSING

The amounts of unstructured and automatic generated data from sensors, networks and devices led to an increase in volume of data [45]. However the traditional processing architectures where not able to scale and provide real time response for those big data applications, this is where stream processing comes in.

There are three forms of data processing systems:

- **Batch processing systems.** The data is collected into sets or *batches* and each batch is processed as an unit.
- **Stream processing systems.** In contrast to batch processing, real-time processing involves continuous processing of data.
- **Lambda architecture.** Takes the advantages of both batch- and real-time processing. The streaming part of a Lambda architecture gives low-latency and inaccurate results, due to continuous and fast processing. The batch part provides the correct output, because of accurate batch processing [2].

A good and efficient way of handling streams of data is important for this project. The middleware needs to handle the data from the different sources and able to scale accordingly. Furthermore correct handling (without loss) of the data adds to the fault tolerance of the system.

Akidau recently added a two part blog [2][1] about streaming patterns on the basis of the Lambda architecture. Those streaming patterns can help us fulfil a part of our problem statement together with the EIP's. The integration patterns add flexibility and scalability to the project. The streaming patterns give more guarantees to be fault tolerant and robust on the final result from the data processing flow.

This section continues first with a short into on stream processing, then the patterns from the blog from Akidau and ends with a couple of data processing tools which implement those patterns.

Akidau defined the term streaming as: *A type of data processing engine that is designed with an infinite data set in mind [2].* This definition includes both true streaming and micro-batch implementations.

However there are some other definitions of "streaming" which are commonly used:

- **Unbounded data.** Is essentially an infinite "streaming" data set. Where as a bounded data set is usually referred to as a finite "batch" data set.

- **Unbounded data processing.** Is an ongoing mode of data processing applied to unbounded data. This also include repeated runs of batch engines that process unbounded data.

An important part of unbounded data processing is time. It requires a clear understanding of the two main time domains involved:

- **Event time.** Is the time at which events actually occur.
- **Processing time.** Is the time at which event are observed in the system.

Ideally the event time and the processing time are the same, but in reality this is normally not the case. The time difference between the event time and the processing time is called the skew, see Figure 17.

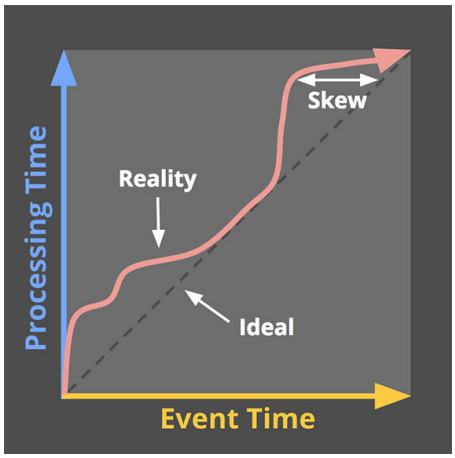

Figure 17: Time domain mapping [2]

The black dashed line represents the ideal case where the processing time and the event time are the same. The red line represents reality. The horizontal difference between reality and ideal is the skew. The skew is the latency that can be caused by resource, software or data limitations.

The skew varies over time, which means that the relation between the processing time and the event time is not static. So the data can not be analyzed only relying on the processing times, when also caring about the event times. However most existing systems build for unbounded data do operate like this. To cope with this problem, systems use some form of windowing. Windowing means cutting the data up into finite pieces along temporal boundaries. However, when here is no correlation between processing time and event time, data may end up in the wrong window. And how does the system know when it has observed all the data of a certain window? This problem is discussed in the next section.

3.2.1 *Data processing patterns*

There are different data processing patterns suited for processing of unbounded datasets that can be used for streaming and micro-batch engines.

Windowing

Windowing is a common approach to cope with the fact that unbounded data sources technically may never end. As stated before, windowing is chopping up a data source along temporal boundaries into finite chunks for processing. Figure 18 shows the three different windowing strategies.

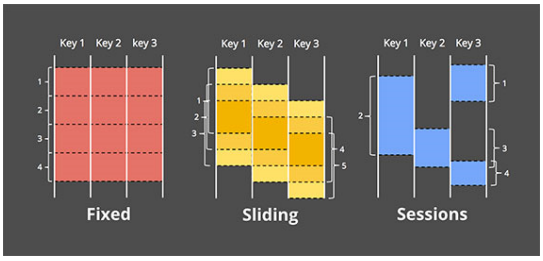

Figure 18: Windowing strategies [2]

- **Fixed.** Fixed windows cut the time into parts with a fixed temporal length. In general the windows are applied over the entire dataset (over all keys), called aligned windows. Windows for different subsets of the data are called unaligned windows.
- **Sliding.** Sliding windows are a variation to the fixed windows. The windows have a fixed length and period. Sliding windows are used when the period of the window is less than the length. Sliding windows are typically aligned.
- **Sessions.** Session windows are dynamic windows, created on the basis of related events. The length of the windows depend on the data involved.

The two domains of time, processing time and event time were discussed in Section 3.2. Next, more details on how those two domains look in terms of windowing. Starting of with windowing by processing time.

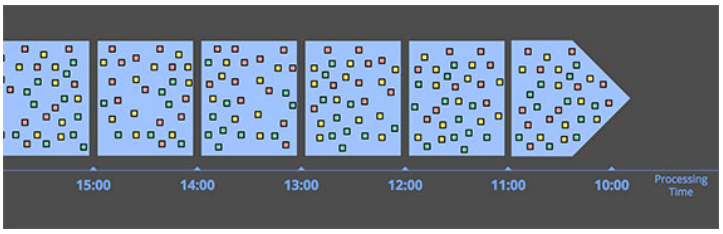

Figure 19: Fixed windows by processing time [2]

Figure 19 shows how windowing by processing time works. The system buffers data for a certain amount of time (time of a window). After the window time has passed, the system treats the data observed in that time as a window. The window can than be used for further processing.

The benefits of this is that it is simple, window completeness is straightforward and it is useful when collecting information about the source as it is observed by the system. However, this type of windowing is not very useful if the data has event times associated with it. In that case the windows do not reflect reality because usually the observed data is not ordered with respect to event times.

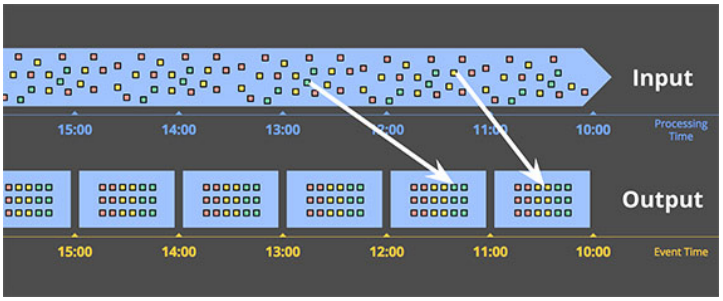

Figure 20: Fixed windows by event time [2]

As the name implies, event time windowing is used when observing data that reflects the times at which those events actually happened, see Figure 20. The white arrows point out two pieces of data that did not arrive in the same window as the event window they belong to. Which, when caring about event times, resulted into incorrect windows if processing time windowing was used. In addition, event time windowing can also be combined with dynamically sized windows, such as sessions.

Drawbacks of event time windowing are the fact that it requires more buffering of data, due to the extended lifetime of windows. And in case of dynamically sized windows there is often no good way of knowing when a window is complete. There is a method to estimate when windows are ready to materialize, called watermarks.

Watermarks

Akidau [1] defines watermarks as, *the way the system measures progress and completeness relative to the event times of the records being processed in a stream of events.* A watermark can be seen as the red line in Figure 17. The point on the red line in event time(E), is the point to which the system believes all inputs with event times less than (E) have been observed. However this depends on the type of watermark, there are two:

- **Perfect.** With perfect watermarks, the system has complete knowledge of all the the input data. There is no late data.
- **Heuristic.** In practice it is hard to have complete knowledge of the input data. Heuristic watermarks estimate the progress of the data based on available information from the inputs. So it can occur that the watermark is wrong, which can lead to late data.

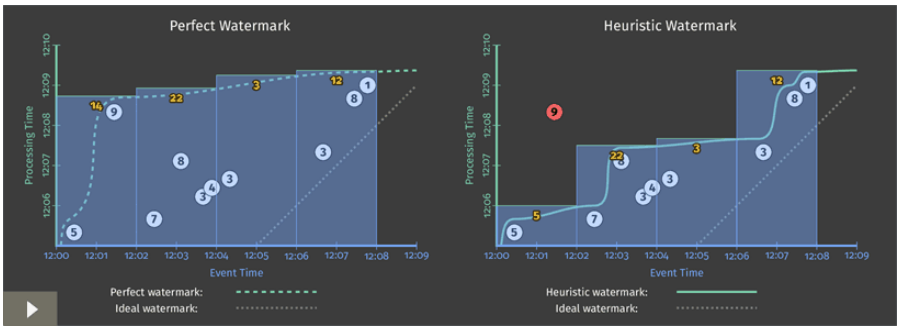

Figure 21: Perfect and heuristic watermarks [1]

Figure 21 illustrates both types of watermarks. The event time on the x-axis is set against the processing time on the y-axis. This example shows four windows, each window covers two minutes of event time. A message with a number is placed in the window corresponding with the event time of that message. This is shown in the diagram against the processing time, this means that the time proceeds from the bottom to the top of the diagram.

Each window adds up numbers when they are observed by the system (processing time). A window is materialized when the watermark passes the end of the window. The right diagram in the figure shows that the heuristic watermark fails to take the 9 in to account, which results in a different watermark compared to the perfect watermark on the left.

The perfect watermark is slow in this case because it knows all the data, this causes delayed results for all the following windows. Waiting on completion of data is not always ideal, especially in real time systems.

On the other hand, the heuristic watermark produces quick results. But it passed the first window before all the data was observed, resulting in an incorrect output (5 instead of 14).

The two cases of perfect and heuristic watermarks illustrate the balancing between correctness and latency. The next section about triggers explains how to deal with late data.

Triggers

Triggers declare when output for a window should happen in processing time. Each specific output for a window is referred to as a pane of the window [1]. Signals that can be used for triggers are:

- **Watermark progress.** The output is materialized when the watermark passed the end of the window or a trigger when the lifetime of a window exceeds some horizon.
- **Processing progress.** Used for periodic updates.
- **Element count.** Trigger when a certain amount of elements are observed in a window.
- **Punctuations.** Trigger when data with a certain feature has passed (e.g. End Of File (EOF) or End Of Line (EOL)).

Those triggers fire based on concrete events, but can also trigger based on less concrete events. This requires more sophisticated logic. Examples of such triggers are, repetition, sequence and con- and disjunction triggers.

Figure 21 showed two scenarios of windowed summation, one case where the watermark was to slow and one where the watermark was to fast. Both can be improved with the use of triggers, shown in Figure 22.

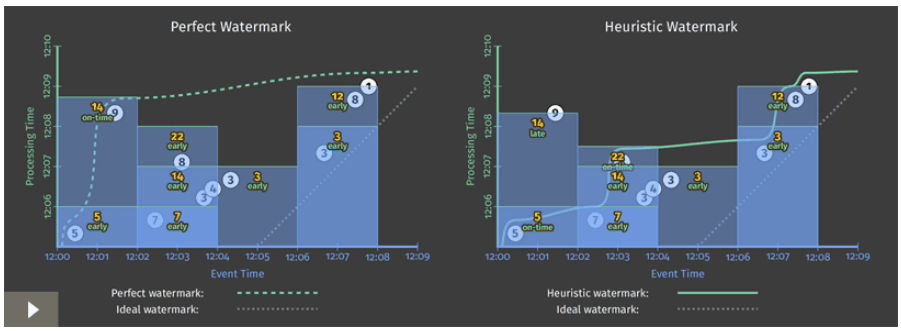

Figure 22: Early and late triggers [1]

In the cases where the watermark was too slow, the trigger fired early updates once every minute. This improves the latency between

the window being complete and the final output pane being materialized. Especially in case of the second and third window of the perfect watermark case.

In the one case when the heuristic watermark was too fast (when number 9 was not taken into account), a new corrected pane is created with the value 14.

Due to the use of triggers both the perfect watermark and the heuristic watermark look quite the same. The only difference is that in the case of the perfect watermark, the system knows that it will not see any more data after the watermark, where in the heuristic case it might. In that case the system needs to know for how long to keep the state of the window. A solution to this is by defining a horizon on the allowed *lateness* within the system, see next section.

Lateness

The definition of lateness is the placement of a horizon on how late any given event may be for the system to bother processing it. The data arriving after the horizon is dropped. The state of the window must be kept around until the watermark exceeds the lateness horizon (expressed in time), starting from the end of the window.

Lateness horizons are not necessary when perfect watermarks are available.

Accumulation

Lateness horizons are not needed when computing global aggregates over all time. For instance in the IoT domain, computing the average value of sensors grouped by area. In this case, the number of windows is bounded by the number of areas (keys). As long as the number of keys remain manageable, the lifetime of the windows is not important. The relation of the panes within each window is.

Figure 3.2.1 about triggers described how triggers are used to create multiple panes within one window over time. Accumulation modes are able to express the way the panes (sub-results) relate, there are three:

- **Discarding.** Is a mode where any time a pane is materialized, the previous stored state is discarded. This is used when numbers have to be summed to produce a final count.
- **Accumulating.** In accumulating mode, any previously stored state is retained and future inputs are accumulated into the current state
- **Accumulating and retracting.** This mode looks like accumulating, but it produces two values. One value being the accumulation value and the other value is the retraction of the previous pane.

	Discarding	Accumulating	Accumulating & Retracting
Pane 1: [7]	7	7	7
Pane 2: [3, 4]	7	14	14, -7
Pane 3: [8]	8	22	22, -14
Last Value Observed	8	22	22
Total Sum	22	51	22

Figure 23: Accumulation modes [1]

Figure 23 shows the three modes applied on the three planes from the second window of Figure 22.

Conclusion

First, we looked into the basic principles of stream processing, mentioning bounded and unbounded data and the difference between event time and processing time. Then we continued with the data processing patterns, which helped dealing with response times and late data.

For this research we have to deal with large amounts of continuous data from other systems and sensors. The data processing patterns help in building a system which is robust and is able to provide accurate data within time. This adds to the overall fault tolerance of the system because based on the type of trigger and watermark used, the output produced is always complete and/or on time. Additionally, this also has an added value for the flexibility of the system, since the data does not have to be complete in order to be processed reliably.

3.2.2 *Data processing tools*

This research does not go into the inner working of the different processing tools, this is already done by Ruurtjan Pul, the student from RuG which has exploited the first steps in this field in his thesis [42]. Since this research focuses on an architecture where the different components and layers are loosely coupled, the focus lies on the interfaces of the selected tools. For this research are the interfaces of the tools more important than the actual inner working of the data processing tools, because of the focus on the coupling and the formats of the data rather than performance and throughput.

The remaining of this section contains an overview of the most common and active data processing tools with regards to their interfaces.

Apache Hadoop

Apache Hadoop [12] is an open source framework for reliable and scalable distributed processing of large datasets. This processing of

large amounts of (batches of) data is accomplished by using MapReduce. Their interface has push and pull support and is able to process unstructured data. Custom in- and output formats can be configured with Hadoop in order to process unstructured data [7]. However Hadoop is heavily focused on their on file system called HaDoop File System (HDFS)

Apache Spark

Apache Spark [17] is a general engine for large-scale data processing consisting of data source libraries. Spark can be used for batch and stream processing, interactive queries and machine learning workload. In contrast to Hadoop does it not provide its own storage system. Spark is implemented in Scala and it provides API's in other languages like Java, Python and R. Windowing is limited in Spark due to the nature of mini batches. The batches can only be windowed based on the process time.

IBM InfoSphere

IBM InfoSphere [27] provides foundational building blocks for inter alia, near real-time data integration. However IBM InfoSphere is a commercial product and not for free.

Akka Streams

Akka Streams [28] is build on reactive streams, which provides inter alia, stream processing, back-pressure and distribution functionalities. Akka streams is a library which can be integrated with Apache Camel, which makes a powerful combination of a data integration framework and a stream processing framework. A source is called an outlet in Akka, this can be a queue, a file, a database, an API, and so on.

Apache Flink

Apache Flink [14] is open source and build upon Akka Streams (and therewith reactive streams). Flink was an Apache top level project in December 2014, so it is relatively new. It supports both batch and stream processing and relies heavily upon the map/reduce format to do the data source integration. Flink is the first open source system that gives control over event time, which allows Flink to use windowing and watermarks. It has support for session (unaligned) windows and triggers. These triggers define conditions when the windows should be evaluated [10]. Flink is implemented in Java, but is does provide a Scala API.

Apache Storm

Apache Storm [15] is a free and open source distributed, both batch and real time computation system. They say that "Storm makes it easy to reliably process unbounded streams of data, doing for real time processing what Hadoop did for batch processing". Storm contains three abstractions within a flow, spouts, bolts and topologies. A spout is a source of streams in a computation, a spout reads from any source. A bolt is a processor, which processes any number of input streams and produces any number of new output streams and a topology is a network of spouts and bolts. Storm does not have control over event time, so windows and triggers are not present in Storm.

Apache Beam

Apache Beam [18] is an open source model and a set of DSL's Software Development Kit (SDK)'s for defining and executing data processing workflows and supporting EIP's. Furthermore, it supports both batch and stream processing and can run on a number of runtimes like Flink and Spark. They provide a concept of four levels namely, pipelines, PCollections, PTransforms and I/O sources and Sinks. A pipeline is a data processing job made of a series of computations including in- and output. PCollection is a bounded or unbounded dataset. PTransforms is a step in a pipeline and I/O sources and sinks are the roots and endpoints of the pipeline.

Conclusion

This overview was focused on the interfaces of the processing tools and thereby the differences according to stream- and batch processing, API's and support for control over the event times. The streaming patterns, mentioned before, are relatively new. That means that the majority of the tools do not have support for both stream and batch processing at the same time, and thereby control over the event times. Apache Spark supports windowing during batch processing based on process time, but not combined with event times. The only open source tool, that we found that has support for controlling event times is Apache Flink. Apache Flink is also the only one supporting watermarks and triggers. So from the different processing tools we looked at, is Apache Flink the most useful to use for implementing the data processing patterns. So Apache Flink can become a part of a flexible data flow systems and therewith our solution.

3.2.3 *Summary*

Stream processing appears in three forms, namely batch processing, stream processing and the lambda architecture. This last form is used

by Akidau in this blogs [2][1] on data streaming and streaming patterns. Those blogs are used as the theoretical background for the first two parts of this section where we did go over the difference between event time and processing time and the streaming patterns: windowing, watermarks, triggers, lateness and accumulation. We ended with an overview on the interfaces of the current state of the art data processing tools and concluded that Apache Flink has the best support for the integration patterns and therefore can become a part of our solution.

3.3 SEMANTIC INTEROPERABILITY

In general, semantic interoperability stands for the ability of a system to work with other systems and devices. The goal of such a system is the ability to exchange data and interpret that shared data, in order to produce useful results as defined by the user. This can be accomplished by adding information about the data, called metadata.

Semantic interoperability is an evident problem in a system containing heterogeneous devices, data sources, networks and services [53]. In this research we have to deal with those different sources, which use different techniques and protocols to communicate. Having a common knowledge base about the meaning (and format) of the data helps us improving the robustness, flexibility and fault tolerance of the system.

First we cover the levels of conceptual interoperability. Then continue on with semantic interoperability formats and how to store and reason about data with additional semantic languages. The three discussed semantic formats are: ontologies, linked data, and semantic web. We complete this section with a conclusion on how we can use this field of computing science to solve the subproblem about how the functionalities provided by semantic interoperability systems can provide guarantees on the final result from the data processing flow, within the dynamic and heterogeneous sensor domain, see Section 1.2.

3.3.1 *Conceptual interoperability*

The (Levels of Conceptual Interoperability Model (LCIM)) consists of six levels of interoperability. The focus lies on the data to be interchanged and the interface documentation. The levels are defined as follows [48].

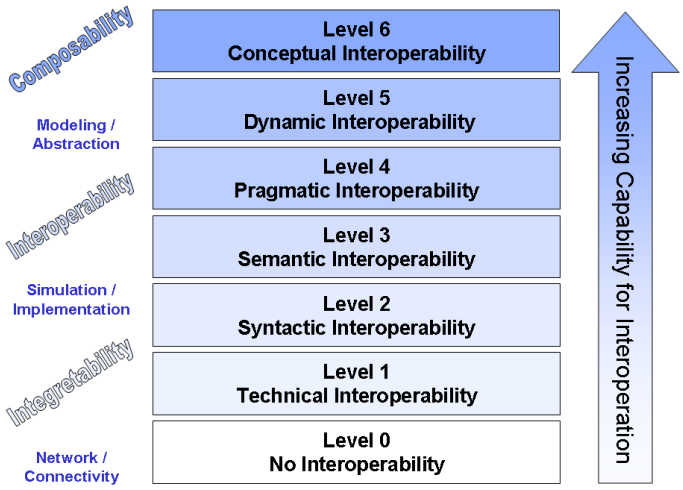

Figure 24: Levels of conceptual interoperability model(LCIM) [48]

LEVEL 0 Stand alone systems have no interoperability

LEVEL 1 Technical interoperability, a communication protocol exists for exchanging data between participating systems. On this level, a communication infrastructure is established allowing it to exchange bits and bytes, and the underlying networks and protocols are unambiguously defined.

LEVEL 2 The syntactic interoperability level introduces a common structure to exchange information, i.e., a common data format is applied. On this level, a common protocol to structure the data is used. The format of the information exchange is unambiguously defined.

LEVEL 3 If a common information exchange reference model is used, the level of semantic interoperability is reached. On this level, the meaning of the data is shared. The content of the information exchange requests is defined. This layer defines (word) meaning.

LEVEL 4 Pragmatic interoperability is reached when the use of the data, or the context of its application, is understood by the participating systems. The context in which the information is exchanged is unambiguously defined. In addition to the layer above, this layer puts the (word) meaning into context.

LEVEL 5 When a system operates on data over time, the state of that system will change, and this includes the assumptions and constraints that affect its data interchange. If systems have attained dynamic interoperability, they are able to understand the state changes that occur in the assumptions and constraints that each

is making over time, and they are able to take advantage of those changes.

LEVEL 6 The highest level of interoperability is conceptual interoperability. This requires that conceptual models are documented based on engineering methods enabling their interpretation and evaluation by other engineers. In essence, this requires a fully specified, but implementation independent model.

We choose level 3 of the LCIM to focus on. Within this research, the meaning of the data is important in order to connect data sources and process the data along the data flow. It is useful to know what the meaning of the data is from a source, and before and after processing.

Within this project, the semantic interoperability is described as the meaning of the data sources, the data itself and the relations between them. This can be either in natural language or in computer language, and this information needs to be stored. There are different techniques and formats on how to store and reason about this information.

3.3.2 *Ontologies*

A conceptualization is an abstract and simplified view of the world that we wish to represent for some purpose. Every knowledge base, knowledge-based system, or knowledge-level agent is committed to some conceptualization, explicitly or implicitly. An ontology is an explicit specification of a conceptualization [24].

In computer science, this means that an ontology is a formal naming and definition of concepts like, types, properties and interrelationships of entities within a certain set or domain. Ontologies are used in large systems to reduce the complexity and organize information. When an application is based on heterogeneous subsystems with different hardware, programming languages and protocols it can help to specify a shared vocabulary.

The two main languages used to build such shared vocabulary of ontologies are Resource Description Framework (RDF) and Web Ontology Language (OWL).

RDF. RDF [22] was designed as a metadata model. It is used as a general method for conceptual descriptions or modelling of information that is implemented in web resources by using a variety of syntax notations and data serialization formats.

RDF extends the linking structure of the Web to use Unified Resource Identifier (URI)s to name the relationship between things, as well as the two ends of the link. This linking structure forms a directed, labelled graph. The edges in the graph represent the named link between two resources, represented by the graph nodes, called

triples(more about triples later). The query based language used to search within a RDF model is SPARQL.

OWL. OWL [20] is a Semantic Web (see Section 3.3.4) language designed to represent rich and complex knowledge about things, groups of things, and relations between things.

The engineering tool for ontologies used by TNO is called TopBraid [49]. The standard edition (which is free) offers a RDF editor, a SPARQL generator, automated data source converter, ontology mapping and triples view.

3.3.3 *Linked data*

Linked data [3] is a digital method to publish structured data in way that it is free available on the internet and therewith becomes more useful. By publishing this structured data as linked data, it can be interlinked and become more usable through semantic queries. The data and data sources can be highly heterogeneous, which makes it useful within this research to be able to search in heterogeneous data from different sources and models.

Linked data is build on web technologies as HTTP, RDF and URI's, to extend them to share computer readable information and enable heterogeneous sources to be connected and queried. An example of a linked data language is JSON-LD.

JSON-LD. JSON-LD [21] is a JSON-based format to serialize linked data. JSON-LD is designed to integrate into systems that already use JSON, and provide a smooth upgrade from JSON. There are also comparable languages available for XML.

3.3.4 *Semantic web*

The semantic web can be seen as a web of linked data. This data is available in a standard format, reachable and manageable by semantic tools. The standards promote common data formats and exchange protocols on the web, most fundamentally the RDF. Semantic web technologies enable people to create data stores on the web, build vocabularies, and write rules for handling data [51]. An example of a data store is `schema.org`.

Triples. Triples [23] are used to make statements about the relation between resources in the semantic web, in the form of subject-predicate-object expressions. The language used for triples is called Terse RDF Triple Language (Turtle).

3.3.5 *Conclusion*

In the previous sections, we looked into three different techniques within level 3 of LCIM, namely ontologies, linked data and semantic web. We can use those techniques to add metadata and relations to the data.

Linked data is build on common web technologies and the linked data languages are based on widely used computer languages. Furthermore linked data can be interlinked which makes it searchable. This is useful for coupling different components to create dynamic flows, based on requests. However the central knowledge base with a shared vocabulary is also useful for storing metadata. So, linked data and ontologies can both help to improve flexibility and fault tolerance of the overall system.

3.3.6 *Summary*

Semantic interoperability is the ability of a system to work with other systems and devices in order to exchange and interpret shared data. There are six levels of interoperability. Within this research we focused on level three, which defines the (word) meaning of the shared data. Then we continued on about techniques used within this level, namely ontologies, linked data and semantic web. With each technique we mentioned the most common formats. Finally we concluded that linked data and ontologies both add to the overall flexibility and fault tolerance of our problem.

3.4 SUMMARY

The three topics of this related work are used to find answers to the research questions presented in Section 1.2.

The first topic was about data integration. We looked into the theory of the EIP's. Those patterns define a detailed description about a solution to frequently occurring problems. We selected the patterns which are relevant to our integration problem and looked into two types of integration tools, namely the ESB and the integration framework. Based on a comparison of integration frameworks against the three main key drivers, we selected Apache Camel as best suited for our solution.

Secondly, we described the theory behind the streaming patterns: windowing, watermarks, triggers, lateness and accumulation. We ended this subject with an overview on the interfaces of the current state of the art data processing tools, and concluded that Apache Flink has the best support for the integration patterns.

The final subject of this related work was about semantic interoperability. Semantic interoperability is the ability of a system to

work with other systems and devices in order to exchange and interpret shared data. We focused on the third level of interoperability, which defines the (word) meaning of the shared data. Techniques used within this level are: ontologies, linked data, and semantic web.

4 ARCHITECTURE

Based on the key drivers and related work, we propose our architectural solution. We define the functional and non-functional requirements for our system, give an high level overview of our architectural solution and finally describe parts of the architecture in more detail on the basis of different views.

4.1 REQUIREMENTS

To make a working operational system, one must start with the requirements for that system. We have decided on specific set of requirements for the system we develop.

There are two types of requirements, functional and non-functional requirements. Functional requirements describe the functionality or behaviour of the system. Non-functional requirements describe how the system works, they specify the criteria that can be used to judge the operation of a system.

The requirements were collected based on the overall context of this research. The requirements must contribute to the three key drivers and help in finding answers to (a part of) the research questions. Furthermore we have taken into account the results of the related work, the future users of the system and the wishes of TNO and the STOOP use case.

The requirements are prioritized, by following the MoSCoW method [35]. The MoSCoW method uses the following prioritization categories.

MUST Requirements that have to be included into the final deliverable in order for it to be successful.

SHOULD Describes requirements that have a fair change of being delivered within the defined time box, but they are not necessary.

COULD The requirements that could be delivered, if everything went extraordinarily well. They are nice to have but not essential. Usually those requirements could improve the user experience for little development cost.

WON'T The least important requirements, which are not planned for the current deliverable.

4.1.1 *Functional requirements*

Table 3 demonstrates the functional requirements with a reference ID and priority.

REQ ID	REQUIREMENT	PRIORITY
FR-01	The system must be able to connect with external heterogeneous data sources	Must
FR-02	The system must be able to translate data into other formats	Must
FR-03	The system must treat process models as blackboxes	Must
FR-04	The system should store semantic descriptions of data sources, and enable semantic interoperability between sources and models	Should
FR-05	The system should store semantic descriptions of processing models, and enable semantic interoperability between sources and models	Should
FR-06	The system must be able to monitor the current status of the system	Must
FR-07	The system must be able to report faults and unexpected behaviour of users, models and sources	Must
FR-08	The system must be able to handle instructions from data flow system	Must
FR-09	The system could link ontologies to the source/model semantics	Could
FR-10	The system must support push and pull methods for intake of source data	Must
FR-11	The system must be able to create and remove instances of objects dynamically	Must
FR-12	The system should register new objects	Should

FR-13	The elements of the system should expose a clearly defined interface which describes syntax and semantics in such a way that people and other (sub)systems can easily understand it	Should
FR-14	The system should support data processing patterns to maintain consistency and data quality in stream processing	Should

Table 3: Functional requirements

4.1.2 *Non-Functional requirements*

Table 4 shows the non-functional requirements with reference ID and priority.

REQ ID	REQUIREMENT	PRIORITY
NFR-01	The system must scale according to the number of external data sources	Must
NFR-02	The system must scale according to the number of models	Must
NFR-03	The system must allow integration with different types of heterogeneous data sources	Must
NFR-04	The system must allow integration with different heterogeneous models	Must
NFR-05	The system must not loose incoming source data in time of processing	Must
NFR-06	The system must be tolerant to middleware communication faults	Must
NFR-07	The system should be easy to deploy, requiring minimum setup and configuration to get up and running	Should
NFR-08	The system should be user friendly	Should
NFR-09	The system should have an availability rate close to 100%	Should
NFR-10	The system must provide distributed processing capabilities	Must

NFR-11	The system must provide fault tolerance mechanisms to perform automatic fail-over	Must
NFR-12	The system should be easy to maintain	Should

Table 4: Non-Functional requirements

4.2 SYSTEM ARCHITECTURE

From the information gathered in Chapter 3 we noticed that that an integration solution can be build on the EIP's. In addition, we found out that an integration framework is currently the best solution for our integration problem. An integration framework is in general a library which provides build-in functionality for the EIP's, and can be combined with other tools and techniques. This makes it possible for other tools to fulfil the quality attributes an integration framework lacks in.

The current integration frameworks fall short on build-in capabilities in terms of fault tolerance, scalability and adaptability. In order to obtain those qualities the developer need to add those themselves, by additional configurations, patterns and techniques.

Therefore we propose a solution that relies on an integration framework, the stream processing patterns and a semantic interoperability (database) tool. Each solving a part of the problem and combined make up for the middleware. The integration framework serves for the purpose of decoupling and routing data from the data sources through processing models. That adds to the overall flexibility. The stream processing framework supports robustness and guarantees regarding the data. The semantic interoperability tool adds to the usability, maintainability and overall management of the middleware.

The remaining of this section continues with a description of our high level layered architecture. After that we go in more detail through the first three views of the 4+1 model [29]. Namely, they are the logical view, the development view and the process view.

In this chapter we do not go into the physical view. The physical view refers to the software components in the physical layer and the physical connections between them. Our physical view would not always be the same since our solution does not always run specifically on certain physical components. The physical view depends on specific use cases and is shown in a deployment diagram in the evaluation.

The "+1" view of the 4+1 model are the scenarios. This view has some overlap with the process view. Use cases and the interactions are covered by sequence diagrams in the process view, so we do not go over the scenarios as well.

4.2.1 *High level architecture*

As mentioned in the introduction, this research is based on IoT applications. This is the basis of our architecture. For this high level architecture we used the layering principle from an IoT application platform mentioned in [43]. From this, the following high level layered architecture is created, see Figure 25.

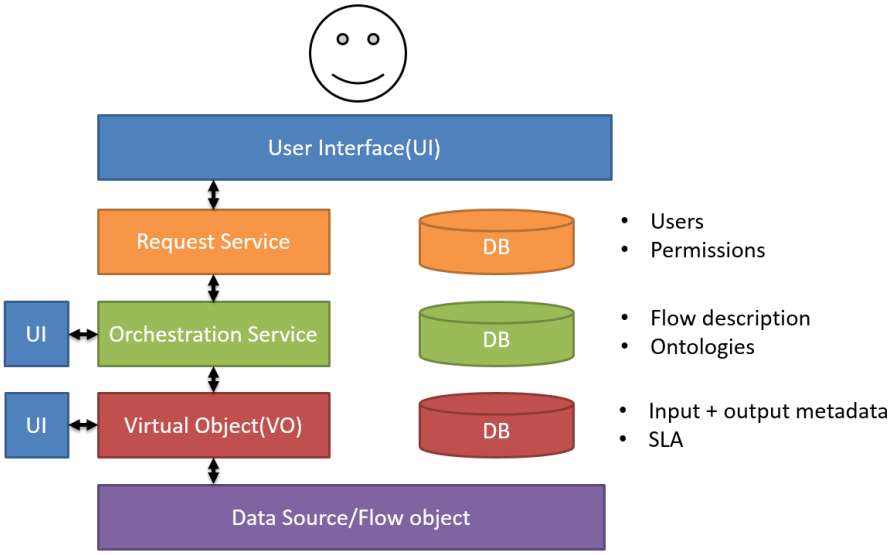

Figure 25: High level architecture

The bottom layer represent two types of objects. The first are the heterogeneous (often external) data sources, ranging from distributed databases to sensors to web servers. They contain or produce data. The other type of object is a flow object. These are a processing models within a data flow, where every processing model requires data input(s), then it performs an operation on the data and produces new data output(s).

One layer higher is the VO (for easiness of reading) layer. This VO principle is responsible for the virtualization of the data sources and flow objects. A virtual object is connected to either a data source or a flow object and has an input and an output. The connection is established through a Service Level Agreement (SLA). Furthermore, the VO has functionalities to translate, route, manage and monitor data. We go into details further in Section 4.2.2. The VO's with their properties and their additional in- and output metadata are stored in a Database (DB).

The layer above the VO layer is the orchestration layer. As the name implies, this layer is responsible for the orchestration and management of data flows. A data flow consists of multiple VO's constructing a chain together to couple data sources and flow objects in order

to produce the desired output. The DB of the orchestration service contains information about flows and relations (ontologies).

The requests for data are derived from the request service. This layer contains information about different users and their permissions, and this layer handles the requests from the users coming through the UI.

The UI is the layer where the end-user interacts with. This is an interface with a collection of all outside functionalities of the system the user is allowed to use. The UI has connections to the orchestration service and the VO's, in order to provide information about the requests he/she can execute (shown as the blue squares next to the orchestration and VO layer).

The continuation of this chapter consists of the different views of the system according to the first three views of the 4+1 model [29], namely the logical view, the development view and the process view. Starting with the logical view in the next section.

4.2.2 *Logical view*

The logical architecture primarily supports the functional requirements and is concerned with the functionality that the system provides to the end-users. The logical view in Figure 26 follows the same corresponding color schema as the high level architecture in Figure 25. The square with the black dashed line contains the the components of the flow including data sources, models and VO's.

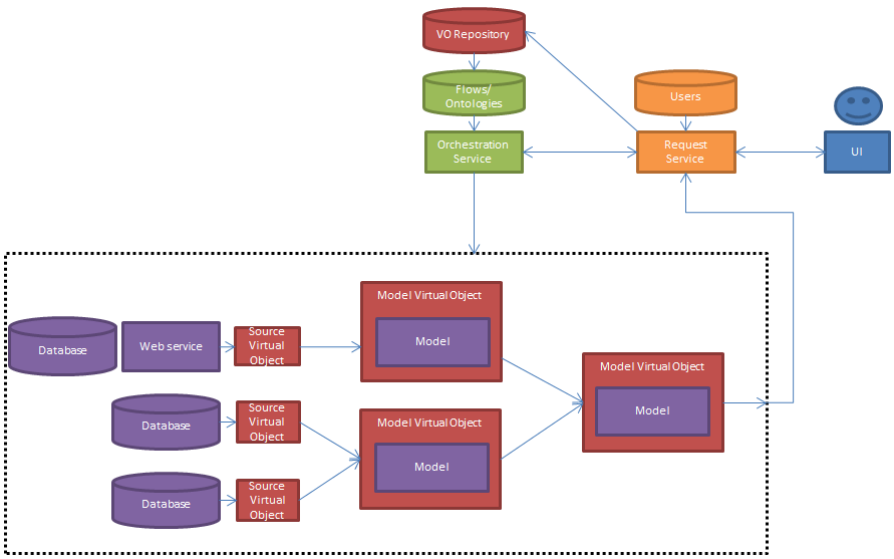

Figure 26: Logical view

VIRTUAL OBJECT The main focus of this research lays towards the concept of VO's. A VO is the building block of the data flow and

it is responsible for the coupling of the data sources and models together. Those responsibilities include:

- Transport protocol connection
- Data format/protocol
- Push/pull conversion

A source VO is able to connect a data source to the system and a model type VO provides a wrapper for a processing model. For this research, our concept of a VO has three interfaces, one input, one output and one for instructions from the orchestration service. More detailed information on the two different VO concepts later in Section 4.2.3.

VO REPOSITORY Descriptions (and additional metadata) about the VO's is stored in a central (semantic) DB.

DATA SOURCE As stated in the previous section, a data source can appear in different types and formats, shown in purple on the left side of Figure 25. In general those data sources are managed by third parties, which means we do not have full control over them and they are only accessible through an interface (i.e. IP address + port or path). Depending on the type of data source, they contain or provide data. Which means the VO of the connected source must be able to support both push and pull actions.

PROCESSING MODEL A processing model is enclosed into a model VO. A processing model can, just like a data source, be managed by external parties on external/internal servers. For this research we look at a processing model as a blackbox with an address and in- and output.

ORCHESTRATION SERVICE The orchestration service is responsible for management of the VO's and the orchestration of flows, consisting of one or more VO's. The orchestration service creates and instructs the VO's within the data flow based on the requests from the request service and the information on the flows/ontologies from the DB.

ORCHESTRATION REPOSITORY The repository used by the orchestration service contains information to be able to create data flows.

REQUEST SERVICE The request service is the service between the UI and the orchestration service. The first responsibility of the request service is to translate the instructions from the UI to the orchestration service. Secondly, the request service is used for the management of the users. And finally, it is able to pull information from the VO repository about the current VO's and to

send it to the GUI. This way the users know the current active VO's and the sources in order to formulate their requests.

REQUEST SERVICE REPOSITORY The request service repository contains data about the users and the permissions they have.

USER INTERFACE As stated in the previous section, this is the place were the user can perform their actions on the system. The UI gets information from the VO repository through the request service. The allowed actions of a user are based on their permissions, which are listed in the user DB.

4.2.3 *Logical view virtual objects*

As mentioned before, we use the concept of VO's in our architecture. There are two types of VO's. A source VO connected to a data source and a model VO as a wrapper for a processing model. In this section we take a closer look at those two VO's.

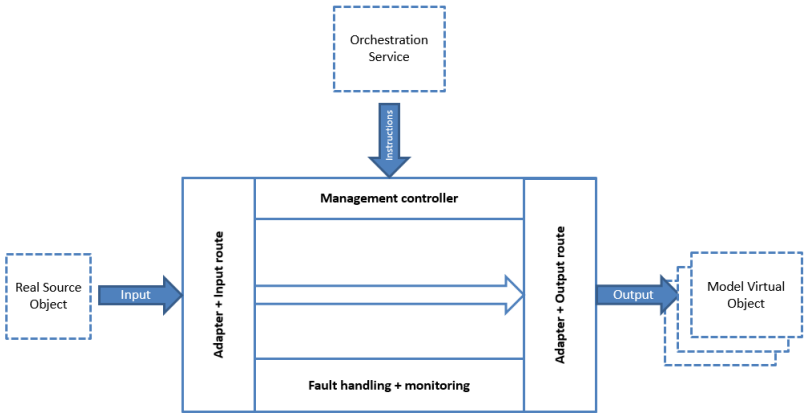

Figure 27: Logical view of source VO

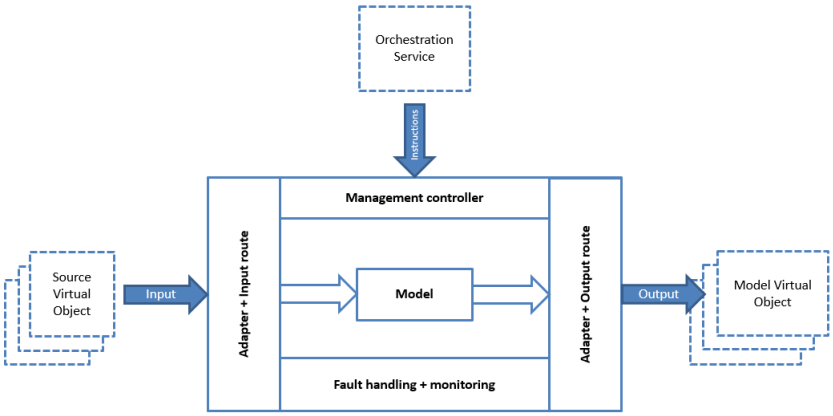

Figure 28: Logical view of model VO

A logical view of a source VO is shown in Figure 27. Our concept of a source VO has, when in a flow, its input connected to a data source.

And its output connected through adapters to one or more model VO's or to some endpoint.

Both endpoints of a route consist of adapters, an input adapter and an output adapter. An adapter provides support for different source and transport protocols and data formats. There is only one in- and output adapter for each VO. But there can be multiple in- and output VO's.

The adapters are used to chain multiple VO's together to create a data flow. The models are used to analyze the data, therefore a data flow with models can be used for data analysis.

The arrow between the two adapters represents different operations in the form a synchronous sequence of operations based on the EIP's. We discuss it more in details in Chapter 5.

Each VO contains a third interface, called the management controller. This interface is responsible for receiving instructions from the orchestration service during runtime. Those instructions include, inter alia, start, stop and update instructions. More about the instructions in Chapter 5. This interface is also used to send metadata or ontology information to the orchestration service for storage.

Lastly, our concept of a VO contains a local fault handling and a monitoring component. A VO has build-in functionalities to handle faults and failures by using different fail-over strategies and exception handlers, and is able to report those faults and its state through the management controller to the orchestration service.

The concept of a model VO, shown in Figure 28 is quite similar to the concept of a source VO apart from two properties. Firstly, a source VO is connected to an actual data source, so the URI and the input format of the source must be known. Secondly, a model VO contains wrapper functionalities for a processing model. For this research we consider a model as a blackbox, which means that the internal structure of the model is not known to the model VO. The only information provided by the model is the address/location, description (metadata) and the in- and outputs.

4.2.4 *Development view*

The development view shows the system from a programmers perspective. It gives a more detailed overview of the architecture regarding the implementation of the system. Figure 29 is a class diagram showing the most important classes of the system and the connections between those classes. The bolt horizontal dashed line divides the diagram in two parts. For this research we focus on the part of the diagram below the dashed line. We try to solve basic flexibility problem of uncoupling data sources and users. When that is solved we can start to deal with on-the-fly orchestration of services, models

etc. The top part of the diagram is out of our scope, but it is included in the diagram to give a complete view of the system.

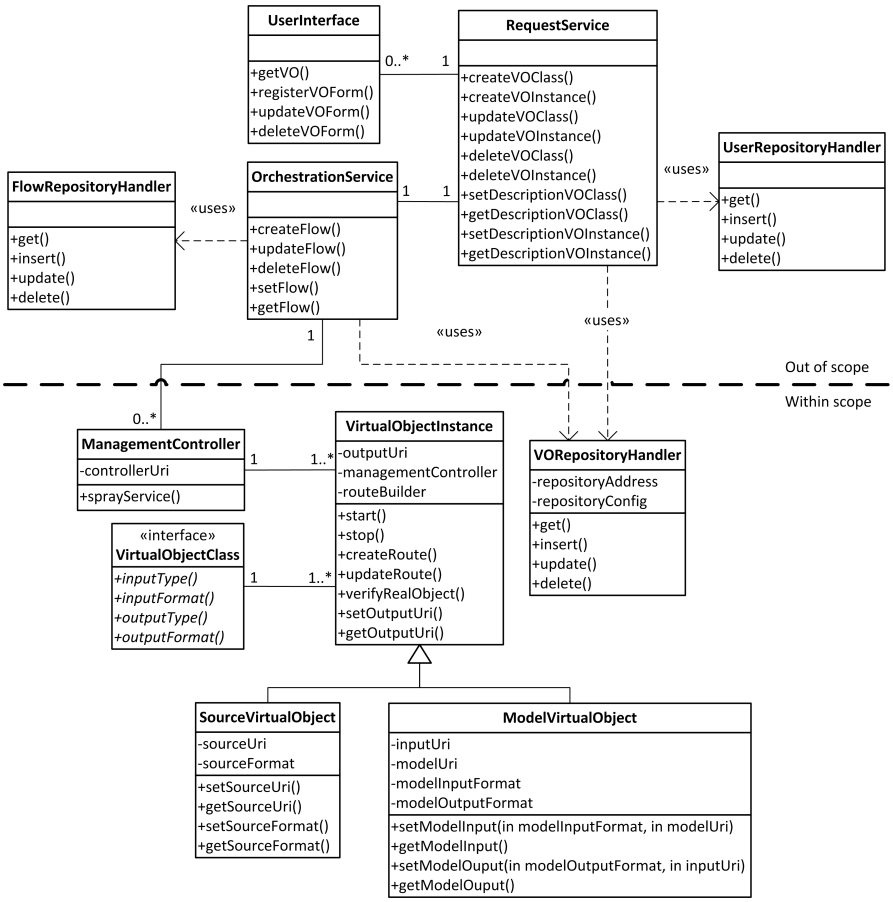

Figure 29: Class diagram

The top part of the class diagram shows the orchestration part of the system, including the UI, request service, orchestration service and the handlers for the user and flow repositories.

The UI contains functionalities and forms for the user to register, update and delete VO's. The UI is connected to one request service. The request service uses both the handlers for the user and the VO repositories. This is developed in order to provide information to the user about running VO's and their own information and permissions. The orchestration service is also connected to the request service and uses the flow repository and the VO repository handlers to create flows based on the requests from the request service.

The bottom part of the diagram contains the classes regarding the implementation and storage of the VO's. The VO is divided into a VO class and a VO instance. The VO class is the general description of a type of VO. This description is stored in the central VO repository. The VO instance is the actual VO that is running in the system and does the work. The VO instance can exist in two types, the source VO and

model VO with their own properties as mentioned before. Since the VO class is a general description of a VO, multiple VO instances can be created from the same VO class. The instances of a class are also registered in the VO repository, so that the orchestration service can manage and monitor them.

Next to the VO instance's in- and output, there is a third interface, the management controller. A VO instance always has one management controller interface to receive its instructions from. However, one management controller is able to manage multiple VO instances. For example, when running multiple VO instances on the same machine, the machine only needs one management controller to control the local VO instances.

4.2.5 *Process view*

A sequence diagram deals with the dynamic aspect of the system, explains the system components/parts/modules and how they communicate over time. It shows the components involved in the different scenarios and the sequence of messages exchanged between them to accomplish the function, or chain of the functions in the scenario.

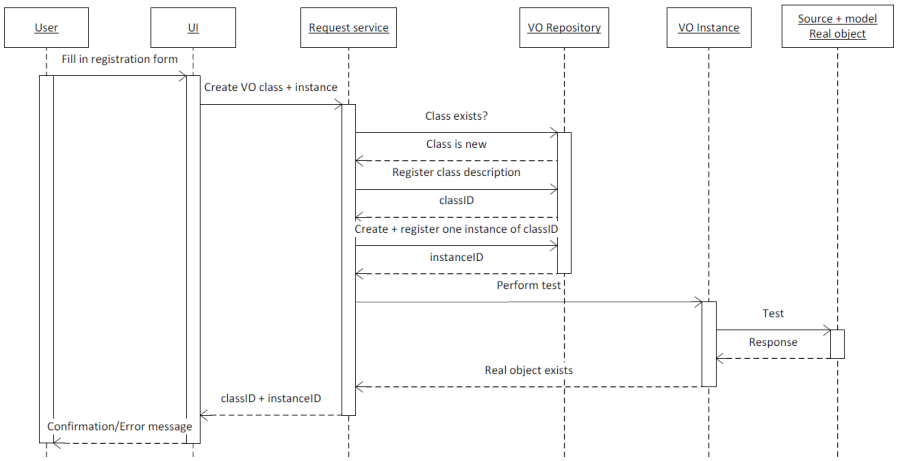

Figure 30: Registration of real (source + model) object

Figure 30 shows the sequence of messages during the registration of a new VO class type. After the creation by the user, the request service checks the VO repository whether the VO class does not already exist. If the class is a new type of class, then the new class description is generated and is stored in the repository. Then a new instance VO is created out of the class description. Lastly, the VO instance tests whether the real model or source exists, and if yes, a confirmation/error message is send back to the user. Once a VO instance is created, it is running and available to use.

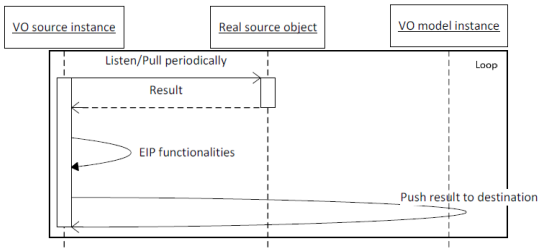

Figure 31: Running source VO instance

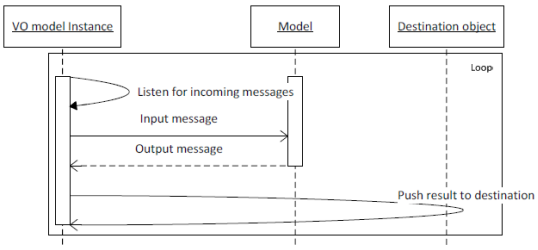

Figure 32: Running model VO instance

Both Figure 31 and 32 depict the loops of running VO instances. We made the assumption that a real object is running before the creation/connection of the VO.

A source VO instance listens or pulls periodically from its connected source. It means that a source VO is able to support both push and pull sources and uses the one it needs. Once a source VO receives a result from the source it is able to perform some EIP related actions on the data from the source and finally pushes the outcome to the next VO.

The running sequence of a model VO starts by listening for incoming messages from other VO's. Those messages are pushed towards a model and the result from the model is then pushed to the next destination.

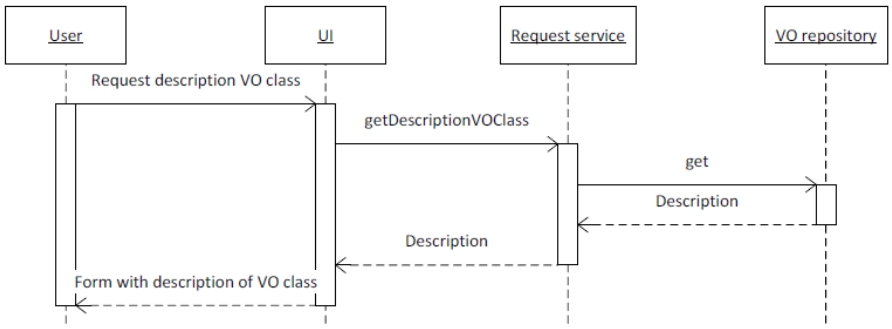

Figure 33: Getting (source + model) VO class

The sequence of messages exchanged in order for the user to get the description of a VO class is shown in Figure 33. The user sends a request for a certain VO class description by using the UI. UI sends the request through to the request service, which in its turn queries the VO repository. The result is send back the same path, back to the user.

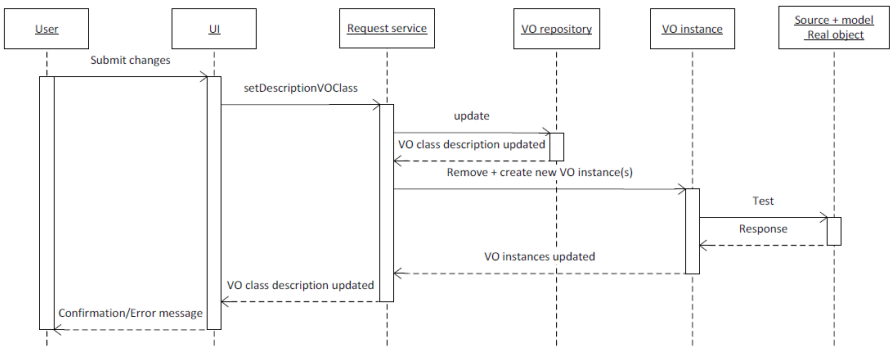

Figure 34: Updating (source + model) VO class

Figure 34 describes the sequence of actions to update a VO class. This is the most important functionality, since it allows to have flexibility.

After user got the description of a class (see Figure 33), he/she is able to change the description of a class and submit it. When user does this, all the current VO instances of the VO class are removed and created again according to the new class description. Each VO instance validates whether the real source is accessible and the user receives a confirmation message if the class is successfully updated.

Figure 35: Stopping (source + model) VO instance

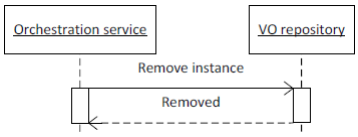

Figure 36: Removing (source + model) VO instance

As stated before the orchestration service is able to control the VO instances. Two of the control functions of the orchestration service are stopping and removing a VO instance, see Figure 35 and Figure 36. More details about the commands to control the VO instances are discussed in Chapter 5.

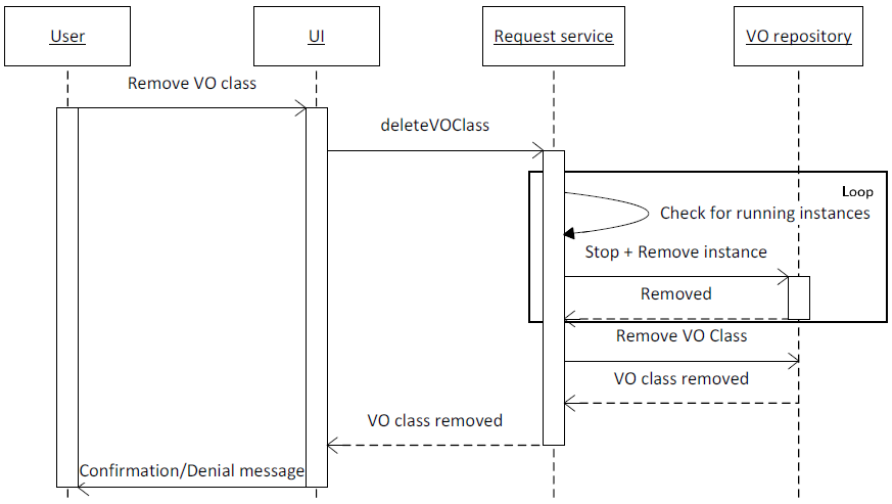

Figure 37: Removing VO class

Figure 37 shows the message exchange between components when a VO class is removed from the VO repository. After user got the description of a class (see Figure 33), he/she is able to remove the description of a class and submit it. When user does this, all the current VO instances of the VO class are removed. After all the VO instances are removed, the VO class is removed and the user receives a confirmation message if the class is successfully removed.

4.2.6 *Summary*

The functional requirements describe the functionality of the system and thee non-functional requirements describe how the system works. Based on the formulated functional and non-functional requirements, we showed the architecture of our solution by means of a high level architectural view and the first three views of the 4+1 model. A logical view, a development view and a process view. Those views provide an overview of our solution to show the architecture and the components within the architecture from different perspectives.

5 IMPLEMENTATION

In Section 1.2 we described the problem we wanted to solve during this research. In Section 2.1 we presented our three most important key drivers. Based on the problem statement and the key drivers we showed in Chapter 4 the architecture of our solution.

In order to meet the three key drivers and to answer the research questions in practice, we created a prototype. This prototype is based on our architectural solution described in the previous chapter.

As a basis our implementation, we have used Apache Camel to build on. From this we have created our own processors, streaming patterns, dynamic data flows, distributed routes and made them work on different machines. This in order to reach scalability, fault tolerance and flexibility.

This chapter is split into four sections, each covering a part of our implementation. The first step shows how we create a dynamic data flow, then we demonstrate how we distribute those flows over different machines. In the third step we add our own processors and stream processing patterns, and in the last step we show how we can control the data flows and objects through an interface.

5.1 DYNAMIC ROUTES

Based on the review of state of the art, described in one of previous chapters we chose to use Apache Camel as basis for the implementation of the VO instances.

Camel uses so called routes to integrate external systems. A route has two endpoints, when a route is started data flows through the route from endpoint to endpoint, see Figure 38. Camel has support for different endpoint components, where each component can integrate a different technology. The EIP's can be used between the two endpoints. So a Camel route consists of an endpoint, then a sequence of (optional) EIP operations (shown in the figure as short arrows) and then an other endpoint.

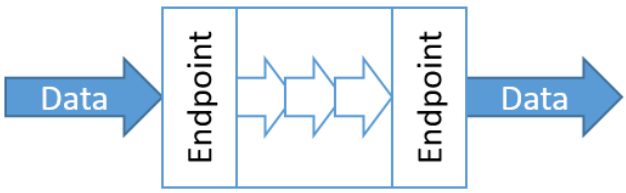

Figure 38: Route with two endpoints

We show how multiple Camel routes can form a data flow, and how the flow can be fault tolerant and able to change dynamically in Figure 39.

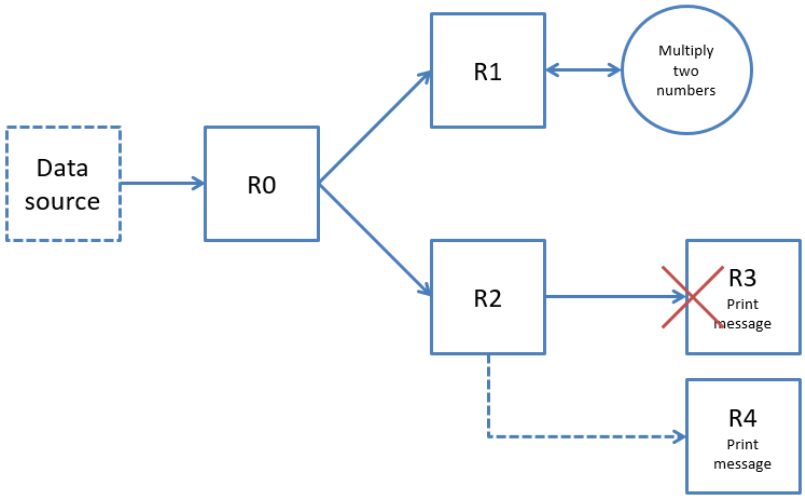

Figure 39: Dynamic data flow

Every route is displayed as a square block with an "R" number. The circle represents a processor, more about our implementation of those processors later in this chapter.

The data source hereby is a client-server socket setup. The client creates a message every n seconds with two random numbers, and sends this message to R0. R0 is a Camel route which performs a parallel multicast to both R1 and R2. R1 contains a processor, multiplying the two number from the message. We created our own processor linked to Camel's bean component. The bean component binds beans to Camel message exchanges, where a bean is a method. More about the working of Camel beans in Section 5.3. Finally, R2 sends the message to R3 which prints the original message.

As already shown in the figure by the red cross at R3. We want to demonstrate how to dynamically change the flow when a data flow component (R3) fails or becomes unavailable. And what it means to change a Camel route.

5.1.1 *Failover*

Failover means automatic switching to another (backup) component when the regular component fails or becomes unavailable. Camel has a failover loadbalancer which can be configured in a way that one can add routes to fall back on.

In the case of the scenario in Figure 40, R3 and R4 are added to the failover loadbalancer of R2. So the default route is R3 and the fallback route is R4. Failover happens locally inside Camel, therefore it

is a decentralized decision at runtime inside this process. The central repositories do not make this decision.

When we kill/remove R3 after a certain amount of time, an exception is thrown and handled by the failover loadbalancer. Then it tries the next route, which is R4.

The failover can be configured in roughly two ways. One way is that the failover always start from the first route (R3) when a new message is to be processed. The other option is that the failover keeps the state and will continue with the last known available route. This last option is desirable when a lot of routes are added to the list of failover destinations.

This failover mechanism allows us the keep the system available even when a route or more routes become unavailable, until at least one path exists which can reach the endpoint of the dynamic flow. How our implementation deals with scenarios when there is no available path at all, is described in Section 5.1.3.

5.1.2 *Camel routes in Scala*

Listing 1 shows the implementation of the scenario demonstrated in Figure 39. For the implementation of the Camel routes we have used Scala DSL from Camel.

Listing 1: Camel routes in Scala

```
// Route R0
from("direct:R0").routeId("R0")
  // Print statement
  .process {exchange => println("R0: " + exchange.getIn.getBody)}
  // Error handler, try 3 times with delays of 200 ms
  .errorHandler(deadLetterChannel("direct:deadLetter").
      maximumRedeliveries(3).redeliveryDelay(200))
  .multicast.parallelProcessing // Parallel multicast
  .to("direct:R1", "direct:R2") // Destinations of multicast

// Route R1
from("direct:R1").routeId("R1")
  .process {exchange => println("R1: " + exchange.getIn.getBody)}
  .bean(Multiply) // Processor bean

// Route R2
from("direct:R2").routeId("R2")
  .process {exchange => println("R2: " + exchange.getIn.getBody)}
  .errorHandler(deadLetterChannel("direct:deadLetter").
      maximumRedeliveries(3).redeliveryDelay(200))
  // Failover loadbalancer
  .loadbalance.failover(3, true, false, false)
  .to("direct:R3", "direct:R4") // Routes the fall back on

// Route R3
```

```
from("direct:R3").routeId("R3")
  .process {exchange => println("R3: " + exchange.getIn.getBody)}

// Route R4
from("direct:R4").routeId("R4")
  .process {exchange => println("R4: " + exchange.getIn.getBody)}
```

The scala code shows the five routes, route R0 to route R4.

The "from" command is the starting point and listening URI of each Camel route with a given endpoint component. The endpoint can be from one of the many Camel components. Since this example runs on one machine and has one Camel instance (called Camel Context) we use the component "direct". "Direct" is a call to another endpoint within the same Camel Context.

The first command of each Camel route is "from", this is the starting point, it provides the URI of the Camel route, with a given endpoint component. The endpoint can be one of many Camel components. Our scenario from Figure 39 runs on one machine and has one Camel instance (called Camel Context). Therefore we use the component "direct". "Direct" is a call to another endpoint within the same Camel Context.

The first step within all of the routes is a print statement with the exchange information. The exchange contains the message, more details on the exchange pattern in Section 5.3. In this case we print the content (Body) of the incoming messages in to each Camel route.

The errorHandlers for both R0 and R2 are explained in details in the next section.

The statement after the errorHandler in R0 is the multicast. This multicast is executed in parallel, so sending of the messages to the two endpoints occurs concurrently. The two endpoints of the multicast are given as parameters to the "to" command. The function "to" is the end point of a Camel route with a given endpoint component. "to" command pushes the message to the specified endpoints URI.

While looking on the second route in the Listing 1, you can see method "bean". The bean specified in route R1 is a so called bean processor. The bean processor invokes a method. The message exchange is automatically converted into the parameters of the method. The details of the Camel bean processor method is described in Section 5.3.

The final method covered by this scenario is the failover loadbalancer in route R2, which is described in Section 5.1.1. The first parameter of the failover represents the number of failover attempts before giving up. The other parameters indicate the use of an specified errorhandler (the dead letter channel), no use of Round Robin, and always start from the first endpoint. So the message is always sent first to the first endpoint, in this case R3. When R3 is unavailable, the message is directed to R4.

5.1.3 *Dead letter channel*

Both routes R0 and R2 in Listing 1 contain a dead letter channel errorHandler. The dead letter channel errorHandler improves the fault tolerance of a flow, it handles the messages in cases when they could not be delivered. So when R1 or R2 are unavailable from route R0 the messages are sent to a dead letter channel. In route R2 the dead letter channel is an addition to the failover loadbalancer. So when both R3 and R4 are unavailable, the message is sent to the dead letter channel.

We specify the properties of the errorHandler in a way that the errorHandler tries to sent the message 3 times with interval periods of 200 milliseconds. When the errorHandler is not able to sent the message after those 3 attempts, the message is sent to the dead letter channel.

Listing 2: Definition of dead letter channel

```
// Error handler, try 3 times with delays of 200 ms
errorHandler(deadLetterChannel("direct:deadLetter").
    maximumRedeliveries(3).redeliveryDelay(200))
```

A dead letter channel can be defined for specific routes or for all the routes within a Camel Context. When using it for all the routes the errorHandler for the dead letter channel has to be declared before the definition of the routes, see Listing 2.

Listing 3: Dead letter channel channel

```
from("direct:deadLetter").routeId("DeadLetter")
  // Print statement
  .process { exchange =>
    println("Dead letter channel: " + exchange.getIn.getBody
    + "from: " + exchange.getFromRouteId + " is not delivered")
    }
```

The implementation of the dead letter channel is similar to a Camel route code. The dead letter channel contains in our scenario a print statement with the message and the source route (which is in our case route R0), see Listing 3.

5.1.4 *Summary*

We chose to implement the VO instances in Apache Camel with Camel routes. We demonstrated that the Camel routes are suited to create a dataflow with additional functionalities regarding fault tolerance and flexibility.

With our demonstration we did not cover the other key driver, which is scalability. demonstration does run on one machine and is able to scale when more Camel routes are created, but the load is

not distributed over different machines. However, the distribution of routes over multiple machines is possible as well, and it is described further in the next section.

5.2 DISTRIBUTED ROUTES

As mentioned before, a Camel route always consists of two endpoints. Camel can maintain around 200 different components for endpoints [13] within a Camel Context. We covered the "direct" component which allows communication between Camel routes on the same machine and within the same Camel Context. In this section we look at other Camel endpoint components that allow communication over different Camel Contexts and machines. This allows us to build a more scalable system.

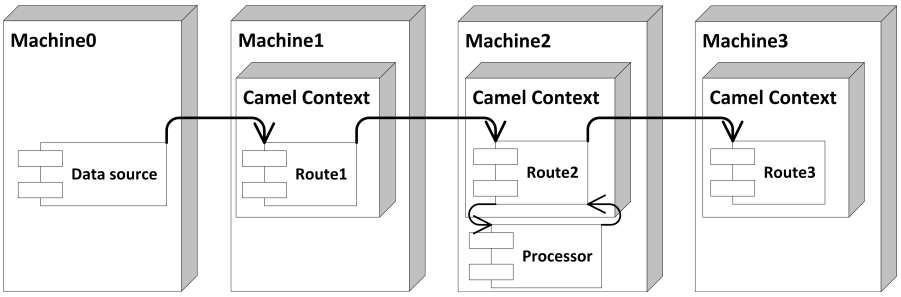

Figure 40: Deployment diagram of distributed routes

A data flow of Camel routes distributed over different machines is shown in Figure 40. Machine0 contains a data source with a web API. Apache Camel is running on other three machines. Each machine runs one Camel Context with one Camel route.

Listing 4: Websocket and TCP endpoints of Camel routes

```
// Route1 with websocket endpoint
from("websocket://hostname:port/pathofdatasource")
  .routeId("Route1")
  // Send data over TCP connection to machine2
  .to("netty4:tcp://IPMachine2:port")
```

The implementation of Route1 on Machine1 consists of the steps shown in Listing 4. This route has "from" and "to" methods with two different Camel endpoints components. "from" for this route is a websocket component. The websocket component provides an endpoint for communicating with clients using websocket. It uses the Eclipse Jetty Server [19]. The client in this case is the data source on machine0.

The data gathered through the websocket is then sent to machine2 over a TCP socket connection. The component used for the implementation of the TCP socket is Netty4 [41]. Netty project of version 4 is a

Non-blocking I/O (NIO) client server framework which enables quick and easy development of network applications, such as TCP socket servers. Route1 is able to sent data over TCP to the IP address and the port defined by user of machine2 where the begin point of route2 is listening on.

Listing 5: TCP connection between Camel routes

```
// Route2 listening for incoming TCP connections
from("netty4:tcp://"
  + InetAddress.getLocalHost().getHostAddress + ":port")
  .routeId("Route2")
  .bean(Processor) // Processor bean
  .to("netty4:tcp://IPMachine3:port")
```

The last part of our implementation of distributed routes includes the code of route2 on machine2, see Listing 5. This route contains the same Netty4 TCP component as route1, only this time route2 is listening on its own IP address and a specific port for incoming messages. Machine2 is also running a processor method which can be accessed with the bean component as briefly stated before, more details in Section 5.3. The result of the bean processor is then sent to route3 on machine3 over a TCP connection. Route3 has the same "from" Netty4 TCP component endpoint as route2.

5.2.1 *Summary*

We have proved that the distribution of Camel Routes between different machines is also possible. Therefore, our solution is fully scalable. The only limit is the 200 supported components from Camel. We have accomplished scalability by using additional Camel endpoint components. This way our system is not bounded by one machine. This is useful to not only distribute the load, but also to share resources and information in cases where systems with multiple different users and different parties need to collaborate.

5.3 PROCESSING MODELS AND STREAM PROCESSING

Previously we mentioned that we divide VO to tow different types: a data source VO and a processing VO. A processing model VO contains an additional wrapper for the specific functionalities of a processing model. The processing model is in that case a black box with an input and an output. All processing models are expected to be implemented as Camel beans. At first we should describe the functionality of a processing bean.

5.3.1 *Processing Bean*

A bean is a Camel component, which binds beans to Camel message exchanges.

The name of the bean can be any string which is used to look up the bean in the registry. Camel has support for three types of registries. One of those types is the Java Naming and Directory Interface (JNDI). JNDI allows looking up data and objects via a name, including external directory services or objects in a network. As a consequence a bean processor method can also live on other machines or servers. That improves the scalability and flexibility. An other type of registry is the Application Context Registry which is used in combination with Spring. And the third registry is called SimpleRegistry, which is a Java Map registry. In our implementation we use this last registry, since our processor bean lives on the same machine as the Camel Context.

We have implemented the bean as a Scala method. The bean processor invokes the method. The message exchange is automatically converted into the parameters of the method. The message exchange is a message container holding the message during the entire routing of a message.

Listing 6: Multiply processing bean

```
// Function compute within Multiply object
def compute(exchange: Exchange) {
  // Get body from message exchange container
  // Split body on commas
  val messageFields =
        exchange.getIn.getBody.toString().split(",|\r\n")
  val number1 = messageFields(0)
  val number2 = messageFields(1)

  // Multiply two numbers and set result
  // as new body of message exchange container
  exchange.getIn.setBody(
        number1+"*"+number2+"="+number1.toInt*number2.toInt)
}
```

Listing 6 shows Scala function representing multiplication functionality of the bean from the dynamic routes scenario in Listing 1. The function within the class object "Multiply" is called "compute" and the parameter of the function is the message exchange. The message contains two numbers separated by commas. The code *"exchange.getIn.getBody"* gets the body of the incoming message. This message is split to retrieve the two numbers. Those numbers are multiplied and set as the new body of the message. The result can then be used in the Camel Route to print or to sent to the next destination. Here is an example

of bean. The users can define completely different beans according to their needs.

5.3.2 *Stream processing*

The event time is the time that events actually occur. We found out from the state of the art that the majority of the current processing tools have no support for event times. This means that they have no support for the processing patterns that we described in related work.

For this first implementation we decided to use two of the processing patterns in our Camel bean function. The two patterns that are implemented are: windowing and triggers.

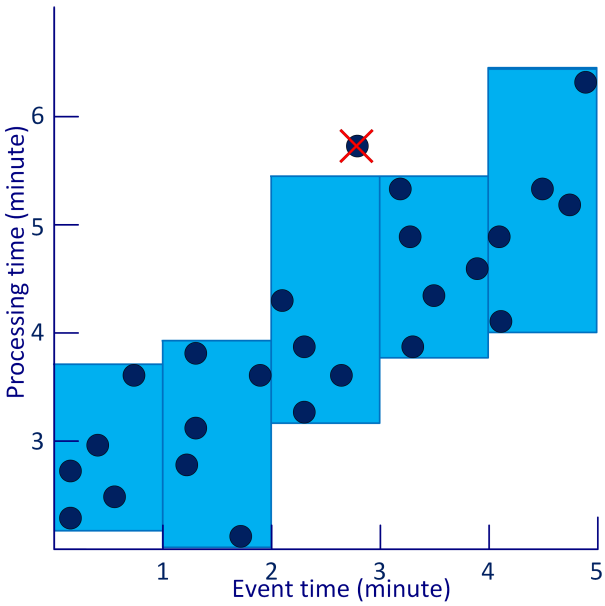

Figure 41: Implementation of streaming patterns

Within our implementation we simulate delayed messages based on event- and processing times. The event times of the messages are attached to the message. The processing times are registered when a message arrives at the Camel bean, see Figure 41.

The windows are created based on event times. The size of each window is one minute. This means that, when a message arrives (processing time), it is placed in the window corresponding with the same minute as the minute of the event time of the message.

The used trigger fires based on element count. So when a window contains a certain amount of messages, the window is materialized. Figure 41 demonstrates our implementation with a element count of 5. So when 5 messages (small dark blue circles) are observed within a window, the windows is materialized. However when a window is materialized before the previous window, the remaining messages are discarded, this is late data. This can be seen in the third and fourth

window. The fourth window is materialized because it has received 5 messages, but the third window has only received 4 messages. For our implementation we discard all messages in the windows before a materialized window.

This can be used in practice when the algorithm of the next processor has to have a certain amount of messages to be able to run an algorithm.

5.3.3 *Summary*

We have looked more deeply into the possibilities of the VO processing models. Camel beans are typically used for the implementation of the processing models. Those processing bean functions can be linked through registries, so they can in theory be located outside of the Camel Context. This improves the overall scalability and flexibility of the system. In the demonstrated scenario, we used the SimpleRegistry since the bean was located on the same machine as the Camel Context. We have also used a bean function to implement two of the streaming patterns.

Our processing model and data source VO's are implemented with Camel routes. Multiple Camel routes can be chained together to form a data flow.

In the next section we extend our implementation so that each Camel route can be stored, and dynamically controlled through an interface.

5.4 STORAGE AND MANAGEMENT CONTROLLER

Coming back to the logical view of our architecture Section 4.2.3, we see that a VO has three interfaces: an input, an output, and a management controller to receive instructions from the orchestration service. Since we have not discussed the usage of the third interface yet, we should describe its implementation in details.

Moreover, we need to describe the implementation of how to store the camel routes, so that they can be interpreted and reused.

5.4.1 *Management controller*

The management controller is an interface to control the VO's. The user is able to control a specific VO by sending requests to the management controller. Each Camel Context has a management controller, so that VO's (implemented as Camel Routes) can be added, changes and removed.

Camel does not provide us with support for a management controller interface, so we have to use additional tools and techniques for

our implementation. As mentioned in our architectural solution, we strive for a loosely coupled solution, so we chose to use REST.

REST demands the use of HTTP, which scales very well since the client (user) and the server (management controller) are very loosely coupled. With REST the management controller interface is free to change and there is no fixed API above what REST itself defines. The user needs only to know the initial URI, and thereafter chooses from management controller supplied choices to navigate or perform actions.

For the implementation of the management controller we use an open source toolkit for REST/HTTP based integration on top of Scala, called Spray [50]. With Spray you can set up a REST webservice with different Spray routes. From within the Spray routes we are able to call functions, which control the Camel routes of a Camel context.

Listing 7: REST interface get one Camel route

```
// Path definition: /route/{routeid}
path("route" / Rest) { routeid =>
  get { // GET request
    dynamic {
      // If the route exists within the Camel Context
      if (getContext.getRoute(routeid) != null) {
        // complete request with description of
        // route with routeid
        complete(getContext.getRoute(routeid))
      } else {
        // else complete request with 409 Conflict
        complete(StatusCodes.Conflict)
      }
    }
  }
```

Listing 7 shows the reference implementation of a Spray route. This Spray route is called when the user performs a GET request on /route/routeid, where routeid is the id of a Camel route. When the routeid does not exists within the Camel Context, statuscode 409 Conflict is returned. When the route does exist, the user receives a description back of the route with the specified routeid.

As you can see from the implementation, the Spray service is not coupled to one Camel route but to one Camel Context. This is different from the logical views from Chapter 4. So one Camel Context has one management controller, through this interface is the user able to access all the routes within the Camel Context.

Listing 8: REST interface post new Camel route

```
// Path definition: /route/{routeid}
path("route" / Rest) { routeid =>
post { // POST request
  dynamic {
```

```
    // If the route exists within the Camel Context
    if (getContext.getRoute(routeid) != null) {
      // complete request with 409 Conflict
      complete(StatusCodes.Conflict)
    } else {
      // Mandatory and option headers
      headerValueByName("fromuri") { fromuri =>
      headerValueByName("touri") { touri =>
      optionalHeaderValueByName("process") { process =>
      optionalHeaderValueByName("aggregate") { aggregate =>
      optionalHeaderValueByName("failover") { failover =>
        // Function which creates a new route with
        // the parameters from the headers
        createDynamicRoute(routeid, fromuri, touri, process,
        aggregate, failover)
        // Write new route descriptions to file
        FileWriter.writeFile(FileWriter.getFile)
        // Completion message
        complete("Route " + routeid + " added")
 }}}}}}}
}
```

As stated before each Camel Context has a management controller interface to control the Camel routes. In order to be flexible and scalable the user must be able to dynamically create a new Camel route.

Listing 8 shows our Spray route where the user is able to dynamically create a new Camel route during runtime through a POST request. The user has to add two mandatory headers to the request, one named "fromuri" and one named "touri". Those two specify the two endpoints of the new route. The other headers are optional, and they are used only if user want to add extra functions to the route, for example a bean processor or a failover. The function *"createDynamicRoute"* creates the new route within the Camel Context. All the available GET, PUT, POST and DELETE user requests needed for management controller are described in Appendix A.

5.4.2 *Semantic storage*

Semantic storage includes storing VO's and additional metadata about meaning and relations of the data in repositories. The information in the repositories is used by the request service to provide information to the user, and is used by the orchestration service to construct and control dynamic data flows.

In our related work we looked at different tools and techniques which are able to store information with metadata and relations. In our architectural solution we described how the semantic repositories fit in our solution. However, due to the lack of time. They were not implemented but we dived into exploring different strategies.

We have implemented the basis of how Camel routes can be stored. Every time something about a Camel route changes within a Camel Context at the local node, a JSON file with the Camel route descriptions is updated. So in the future the routes can be stored with additional metadata, for example as linked data in JSON-LD format.

The user is able to get a description in JSON of all current (or one specific) Camel routes within the Camel Context through the management controller, see Listing 9.

Listing 9: REST interface get all routes in json format

```
// Path definition: /route
path("route") {
  get { // GET request
    dynamic {
      // Mapping each route to Json format
      complete(getContext.getRoutes.map { x => x.toJson.
          prettyPrint }.toString())
    }
  }
}
```

Figure 42 shows the result of a GET request for all the Camel routes. In this case there are three routes: Deadletter, A and B each with two endpoints.


```
  "id": "DeadLetter",
  "route": "EventDrivenConsumerRoute[Endpoint[direct://deadLetter] -> Channel[DelegateSync[org.apache.camel.scala
      .ScalaProcessor@5bda80bf]]]"
}, {
  "id": "A",
  "route": "EventDrivenConsumerRoute[Endpoint[direct://A] -> Pipeline[[Channel[DelegateSync[org.apache.camel.scala
      .ScalaProcessor@5f3b9c57]], Channel[sendTo(Endpoint[direct://B])]]]]"
}, {
  "id": "B",
  "route": "EventDrivenConsumerRoute[Endpoint[direct://B] -> Pipeline[[Channel[DelegateSync[org.apache.camel.scala
      .ScalaProcessor@1e044120]], Channel[sendTo(Endpoint[direct://C])]]]]"
```


Figure 42: Current routes in Json format

5.4.3 *Summary*

By our implementation of REST server we have shown how the user is able to get information from the Camel routes, how to control them and create new ones. This is all performed through the management controller, which allows the user to control all the routes within the same Camel Context. We chose to implement the management controller as a HTTP service in Spray, because of the flexibility and loose coupling between the client and the server components. We demonstrated that the user is able to get route information from the Camel routes of a Camel context through Spray routes. Furthermore we have shown that the user is able to create a new route through a POST request with addition header information.

Due to the lack of time, we were not able to implement semantic storage functionalities. So we implemented basic storage for Camel routes. Descriptions of Camel routes are stored in JSON format. When additional semantic information is available in the future, the Camel routes can easily be reproduced and stored as linked data in JSON-LD.

6 EXPERIMENTS & EVALUATION

Beforehand we were concentrated on the description of the architecture and the reference implementation, to fulfil our proposed solution. This implementation does not have added value without the thorough evaluation of it. We validate our solution with two experimental runs against the key drivers and requirements. Those two experiments are used to prove the feasibility of having scalable, heterogeneous and dynamic (adaptable) middleware.

6.1 EXPERIMENT 1: DYNAMIC DATA FLOWS

The goal of our first experiment was to validate whether our implementation works in practice. In the test runs of the experiment we have tried to validate whether our reference implementation meets our goal we described earlier. In the experimental runs we try to cover all aspects of the implementation and explain the fulfilment of all key drivers and requirements.

The first experiment proves the possibility of coupling different data sources and the flexibility of data flows during runtime.

6.1.1 *Setup*

Table 5 shows the software/hardware units of our test setup. This experiment is performed on one local Windows machine running one Camel Context with one management controller and multiple Camel routes.

COMPONENT	DESCRIPTION
OS	Windows 7 Enterprise (64 bit) SP 1
Processor	i5-3470 @ 3.2 GHz
Memory(RAM)	4,00 GB
Eclipse IDE	Mars.2 (4.5.2)
Maven	4.0.0
Scala	2.11.8
Apache Camel	2.17.0
Spray	1.3.1

Table 5: Setup test 1

The experimental run is divided into nine small steps. In each step we control the behaviour of a part of the implementation. In this scenario we use data from a data source which pushes data to the data flow, and from a data source which contains data, meaning that the system has to pull the data. This data is processed in parallel by two different processors. The results of those processors are aggregated back together into one message.

6.1.2 *Execution*

Figure 43 shows the logical view of the first experimental run. The blue squares with a dashed line represent data sources, squares with a solid line are Camel routes, and the ovals represent processors. This experiment is split into 9 small steps each covering a part of the implementation.

Step 1: Description of route A and B

Step 2: Description of route C with link to one bean processor

Step 3: Connection of route B with route C

Step 4: Change to other bean processor of route C

Step 5: Creation of route D

Step 6: Connection of route D with bean processor

Step 7: Adding multicast from route B to route C and route D

Step 8: Creation of route E

Step 9: Adding aggregator to route E and changing destinations of route C and route D

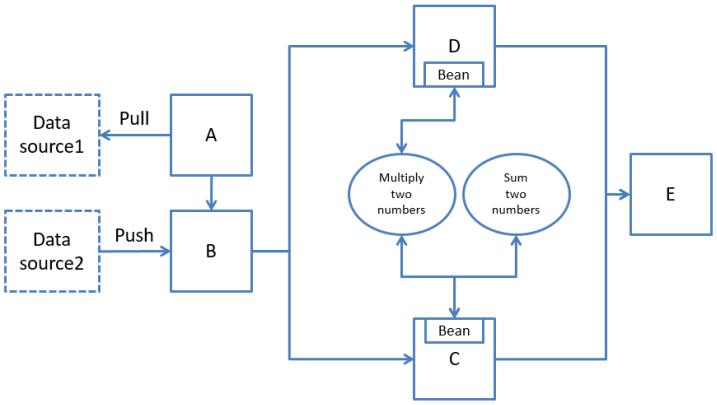

Figure 43: Overview of experiment 1

Figure 44 shows the logical view of the first four steps of the experiment. The steps are highlighted with the text "step + number" and black dashed squares. In step 1 we decided on having two Camel routes for connecting push and pull data sources according to the figure. In step 2 we describe a Camel route which uses a processor written by us. In step 3 we connect the routes from step 1 and step 2. And in step 4 we change the processor of route C.

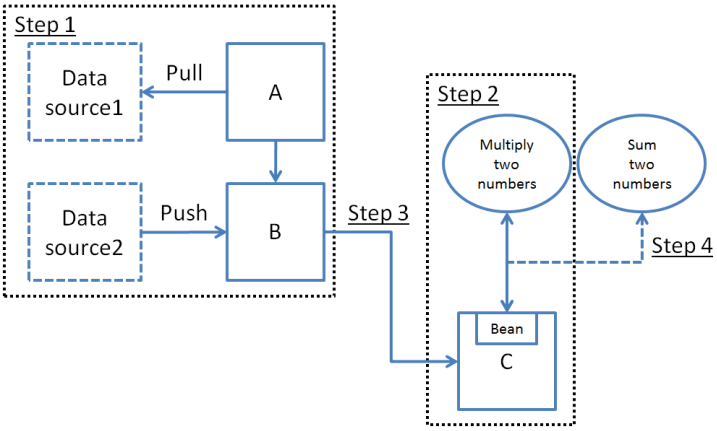

Figure 44: Overview of test steps 1 to 4

Step 1

The content of the first step includes two Camel routes, route A and route B. Route A uses Camel's file system endpoint component to pull data from a file system folder. When there is a file in the folder, the file is pulled from the folder and the content (body) of the file is converted into a string. This string is then sent as a message to route B. For implementation see Listing 10.

Data source2 is a web service which generates and pushes data to route B, which contains Camel's web-socket endpoint component. Both data sources contain or generate typical sensor data consisting of two random numbers separated by a comma. The data through route B is not sent to an other Camel route for now. Therefore we use Camel's mock endpoint component. The mock endpoint is designed for testing, it catches the message exchange to allow for later validation. We use it when we define or create a Camel route without a specific destination.

Listing 10: Routes A and B in Scala

```
from("file:data/inbox").routeId("A")
  .convertBodyTo(classOf[String])
  .process {exchange => println("A: " + exchange.getIn.getBody)}
  .to("direct:B")

from("direct:B").routeId("B")
  .process {exchange => println("B: " + exchange.getIn.getBody)}
  .to("mock:result")
```

Step 2

Route C is created to link our processor to the flow. Therefore route C defined the function which has a link to a bean processor. In this bean processor we multiply the two numbers from the message and return the answer back to route C.

Listing 11: Route C in Scala

```
from("direct:C").routeId("C")
  .bean(Multiply)
  .process {exchange => println("C: " + exchange.getIn.getBody)}
  .to("mock:result")
```

The descriptions of the current active routes can be collected by sending a GET request to the address where the management controller of the Camel Context is running. The management controller is an interface to manage and control the Camel routes. The user is able to control or request data of a specific Camel route by sending requests to the management controller. Each Camel Context has a management controller.

By sending a GET request to the path */route*, the user is able to receive a list in JSON format with the descriptions of the routes. Figure 45 shows the descriptions of the routes at the end of step 2 of our experimental test. Those descriptions are also stored in a JSON file, and are updated every time something about the routes change.

GET localhost:8080/route Params Send Save

Authorization Headers Body Pre-request Script Tests Generate Code

key value Bulk Edit Presets

Body Cookies Headers (4) Tests Status: 200 OK Time: 937 ms

Pretty Raw Preview HTML

```
ArrayBuffer({
  "id": "DeadLetter",
  "route": "EventDrivenConsumerRoute[Endpoint[direct://deadLetter] ->
      Channel[DelegateSync[org.apache.camel.scala.ScalaProcessor@4204541c]]]"
}, {
  "id": "A",
  "route": "EventDrivenConsumerRoute[Endpoint[file://data/inbox] ->
      Pipeline[[Channel[convertBodyTo[java.lang.String]], Channel[DelegateSync[org.apache
      .camel.scala.ScalaProcessor@2484f433]], Channel[sendTo(Endpoint[direct://B])]]]]"
}, {
  "id": "B",
  "route": "EventDrivenConsumerRoute[Endpoint[direct://B] ->
      Pipeline[[Channel[DelegateSync[org.apache.camel.scala.ScalaProcessor@60b71e8f]],
      Channel[sendTo(Endpoint[direct://C])]]]]"
}, {
  "id": "C",
  "route": "EventDrivenConsumerRoute[Endpoint[direct://C] ->
      Pipeline[[Channel[BeanProcessor[net.alchim31.maven.scala_archetype_simple.Multiply$
      (0x1255b1d1)]], Channel[DelegateSync[org.apache.camel.scala.ScalaProcessor@464649c]],
      Channel[sendTo(Endpoint[mock://result])]]]]"
})
```

Figure 45: Overview of active routes

Step 3

To get the data from the data sources to route C and perform the multiplication of the two numbers. We have to connect route B with route C. This can be done dynamically during runtime, by changing the destination of route B, see Figure 46. The figure shows a PUT request to the management controller on *localhost:8080*. This is were the management controller runs on our local machine. The PUT request sets the new destination of route B to the address of route C, *direct:C*.

PUT localhost:8080/route/B

Authorization Headers (1) Body Pre-request Script

newdestination direct:C

Figure 46: Change route B with PUT request

The data is sent to route C as soon as the PUT request is completed. Figure 47 shows the output of route A, B and C. Route B receives *73,18* from data source2, those numbers are then multiplied inside route C. The message *20,10* from data source1 flows from A to C through B. And *19,74* is from data source2 again.

```
B: 73,18

C: 73*18=1314
A: 20,10
B: 20,10
C: 20*10=200
B: 19,74

C: 19*74=1406
```

Figure 47: Output of routes A, B and C

Step 4

Figure 44 shows two processors. One processor multiplies the two numbers and another processor sums two numbers. In step 4 we decided that we want to change the functionality within our route, and we want now to sum the two numbers instead of multiplying them. Figure 48 shows the change request, so that route C uses the *Sum* processor in place of *Multiply* processor. When this PUT request is completed, the messages are sent to the *Sum* processor and the result is printed by route C.

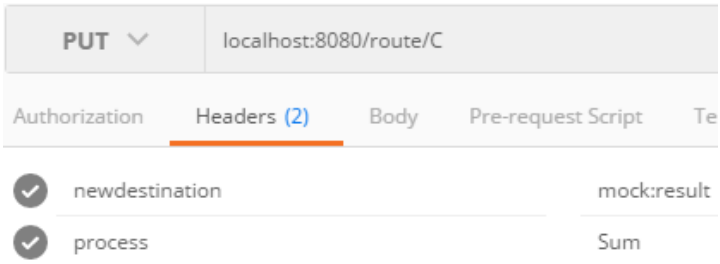

Figure 48: Change processor of route C

The next three test steps are introduces in Figure 49. Now, when we can dynamically couple routes and change processors, we want to have both processors at the same time, i.e., we want to run both *Sum* and *Multiply* processors together at the same time. In order to do this, we create a new route, called route D. This route has a link to the multiply processor. In order to perform both actions at the same time we add a parallel multicast to route B.

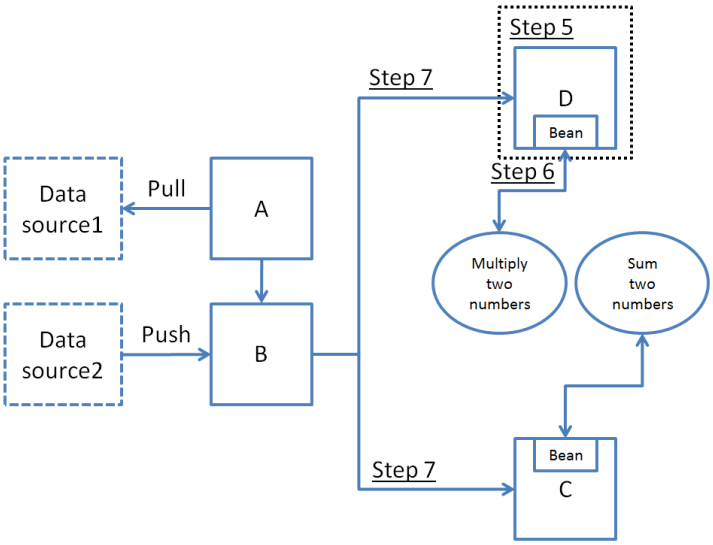

Figure 49: Overview of test steps 5 to 7

Step 5

Figure 50 shows a POST request to the management controller to create a new route, called D. This route listens to *direct:D* and does not have a specified destination for now.

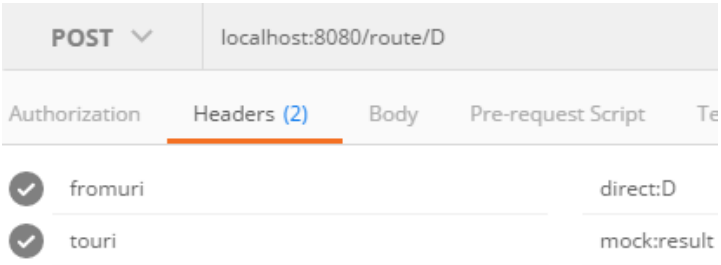

Figure 50: Create new route D

Step 6

Then we add a bean processor to route D with a PUT request, similar to step 4 in Figure 48.

Step 7

In step 7 we add both route C and route D as a destination to B. This is done by sending a PUT request to the management controller with a header *newdestination* with the value *direct:C,direct:B*. The messages from B are then sent to route C and route D in parallel. The output of a message from data source2 is shown in Figure 51.

```
B: 87,76

C: 87+76=163
D: 87*76=6612
```

Figure 51: Output of routes B, C and D

In the last two steps of this test, we want to aggregate both the messages from route C and route D into one message. Therefore, we create a new route with aggregator functionality, called route E.

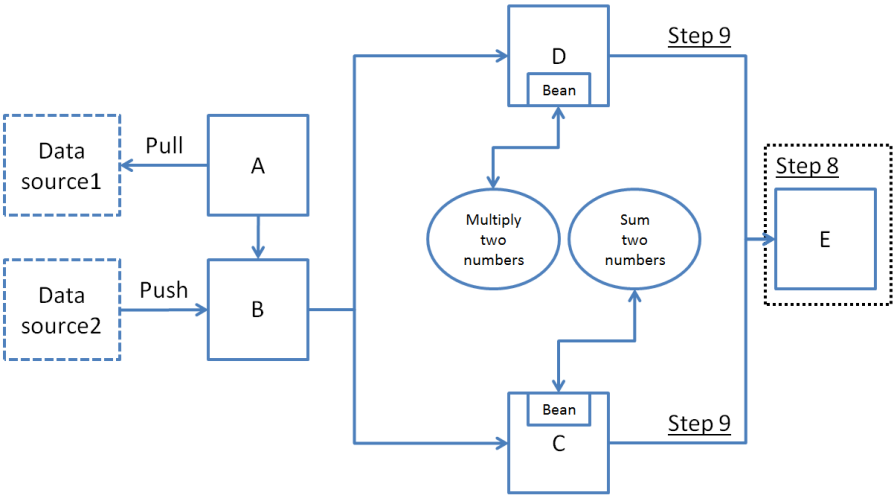

Figure 52: Overview of test steps 8 and 9

Step 8

We create a new route, similar to route D in Figure 50. Only this time a route called E has a header *fromuri* with value *direct:E*, and an aggregator. The preconfigured aggregator merges all messages within 500 milliseconds together into one message.

Step 9

In the last step we change both the destinations of route C and route D to *direct:E*. The result of the flow from data source2 is shown in Figure 53. Both the messages from route C and route D are sent to route E, which merges the two messages separated by a comma.

```
B: 99,71

C: 99+71=170
D: 99*71=7029
E: 99+71=170,99*71=7029
```

Figure 53: Output of routes B, C, D and E

6.1.3 *Evaluation*

While running this experiment, we validate our architecture and implementation against the requirements and key drivers chosen beforehand.

Requirements

In Section 4.1 we specified thirteen functional requirements that the software should meet in order to solve the problem statement. This section lists the functional requirements, and how they are covered by the experiment.

FR-03 - The system must treat process models as blackboxes. For the implementation of the process models we used Camel's processing beans. A bean processor invokes a method. The user must know the name of the method, and if the method is accessible form the current Camel Context. The inside of those methods is not known to the bean and the system. Therefore the methods can be treated as blackboxes with an in- out output. The experiment showed this functionality with the multiply and the sum beans, where the computation is not known to be bean; only a message with the two numbers is sent to the bean function.

FR-08 - The system must be able to handle instructions from data flow system. Each Camel Context has a management controller interface. In our case we use a Spray HTTP service. The user is able to send various instructions in the form of GET, PUT, POST and DELETE requests during runtime. An overview of all possible instructions is shown in Appendix A. The instructions can be used to get information about the routes within the Camel Context, or to add or change Camel routes dynamically.

FR-10 - The system must support push and pull methods for intake of source data. This requirement is an addition to the requirement *FR-01*. Our implementation supports both push and pull methods for intake of incoming data, owing to the wide range of supported endpoint components of Camel. Camel contains endpoints that can pull data from a database or file from a folder, and endpoints that are able to deal with stream of data from a web service. This is demonstrated in the experiment with two data sources with a websocket connection which pushes data to route B, and a folder where a file can be pulled from, by route A.

FR-11 - The system must be able to create and remove instances of objects dynamically. This requirement is an addition to requirement *FR-08*. As stated before, each Camel Context has its own management controller

interface. The user is then able to send instructions to this interface. Two of those instructions include creating a new Camel route with a POST request and removing a Camel route with a DELETE request. This can all be done during runtime. We have demonstrated the POST request in step 5 of our experiment.

FR-12 - The system should register new objects. Information about the current Camel routes within a Camel Context is registered in JSON format in a file. Every time a Camel route is changed, created or removed, this file is updated according to the current situation. This is demonstrated in the second step of the experiment when we received the descriptions of all the current routes in response to a GET request.

FR-13 - The elements of the system should expose a clearly defined interface which describes syntax and semantics in such a way that people and other (sub)systems can easily understand it. The interface on how to control the Camel routes within a Camel Context with HTTP request is shown in Appendix A. However, the additional semantics is not described.

Key drivers

Key drivers are criteria that used to judge the quality of a system. In Section 2.1 we specified three key drivers that we decided as the most important for this research. In this section we discuss the aspects of the key drivers which are covered by the experiment.

Scalability. Within this research we focused on load scalability, and we defined scalability as: *The ability of a system to change in order to handle growing usage.* Here the growing usage means the increase in number of data sources, users, data flow components and the amount of internal communication between them.

In order to be scalable, we created a modular architecture. Within this modular architecture we focused on the VO module. This module consists of loosely coupled VO's that can be chained together to form a data flow. Each VO has three interfaces, their own adapters and additional support for processing models and the EIP's.

With this experiment we have shown that our system is in theory scalable due to the principle of VO's, which are implemented with Apache Camel routes. We showed that the system is able to handle a growing usage by adding new routes and creating new flows. As a consequence, the load can be spread over multiple flows in parallel in order to handle more data.

Flexibility. With flexibility we mean a system which adapts its behaviour to internal and external changes. Those changes include changes in user requests and needs, data sources and internal infras-

tructure. We defined flexibility as: *A system is flexible if it is able to adapt to the functional and numerous changes in a heterogeneous and scalable environment.*

In order for our system to be flexible, we have added functionalities that allow the user and the system to make changes to the Camel routes and the data flows during runtime. We have demonstrated those functionalities in this experiment.

The user can perform requests to create, change and remove Camel routes during runtime by sending HTTP requests to the management controllers. Therefore, user can dynamically change and connect existing Camel routes and data flows or create new ones.

The user is by changing the Camel routes next to the two endpoints, also able to change the EIP operations and the processing beans during runtime. What can be seen in steps 4, 6 and 7.

6.1.4 *Summary*

In this experiment we have demonstrated that our implementation is flexible and scalable in theory. Firstly, we are able to connect data sources using both push and pull methods to a dataflow. This dataflow consists of multiple Camel routes and processors which can be changed dynamically during runtime. In this way, the data flow remains available during changes and the flow does not have to be stopped or completely rebuilt in order to apply changes. Secondly, this test scenario also shows that our implementation is theoretically quite scalable. We can add more Camel routes to the flow and are able to process data in parallel.

However, this test was performed on one single machine with one Camel Context. In the next experiment we show how the Camel routes can be distributed over multiple machines to scale even more.

6.2 experiment 2: distributed data flows

In our second experiment we validate our reference implementation against the requirements and the key drivers fault tolerance and scalability. We proof the validity by using components of Camel and distributing the Camel routes over multiple machines.

6.2.1 *Setup*

The setup of this experiment consists of multiple machines running their own Camel Context with management controller respectively. One of the machines is the local Windows machine from test 1 (Table 5), the other two are Linux virtual machines running on a server with the specifications shown in Table 6. The software versions are the same as the ones mentioned in experiment 1 in Table 5.

COMPONENT	DESCRIPTION
OS	CentOS Linux 7.2.1511
Processor	AMD Opteron Processor 6328 @ 3.2 GHz
Memory(RAM)	8,00 GB

Table 6: Setup test 2

6.2.2 *Execution*

This experimental run is divided into five small steps. In each step we cover a part of the implementation relevant for scalability and heterogeneity. We use data from one source which pushes its data to our data flow. This data flow is distributed over multiple machines. We created an experiment where there is a backup component which can be used when the main component becomes unavailable. The final result is pushed to a third machine, which we see as an external party server which receives relevant data, see Figure 54. The message contains three numbers, separated by commas. The first number is the minute of the event time, the other two are random numbers, which is typical sensor data.

The blue square with the dashed line is a data source. The other squares with a solid line are Camel routes. The Camel routes are displayed in different colors to show that they are located on different machines. The blue Camel routes run on a local machine, and the green and purple squares are Camel routes on external machines. The purple oval represents a processing bean on the purple machine.

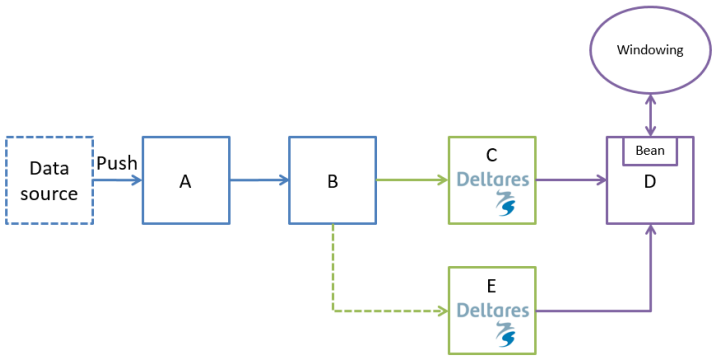

Figure 54: Overview of experiment 2

Figure 55 shows the first two steps of this run. In this scenario we simulate that route C is running at an external server of a STOOP parter called Deltares [6]. Deltares is an research institute in the field of water and subsurface, developing software models and algorithms.

It need data about the changes of soil and the locations of pipelines. The black squares with the dashed line highlight the content of a step.

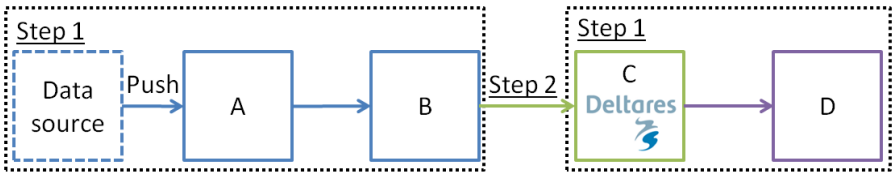

Figure 55: Overview of test steps 1 and 2

Step 1

This step is quite similar to the first step of the previous test, consisting of the declaration of four routes. Route A and B communicate via Camel's *direct* endpoint component and route C and D communicate via a TCP connection with Camel's *Netty4* endpoint component. Which is an embedded web-server component in Camel.

Step 2

In order to let the data flow through the Camel routes on the other machines, we have to connect route B with route C. This is accomplished by changing the destination of route B to the address of route C. Since route C is on an other machine we cannot do this with the *direct* component. In this case we connect to route C with a *Netty4* TCP connection, see Figure 56 where the successful connection is demonstrated.

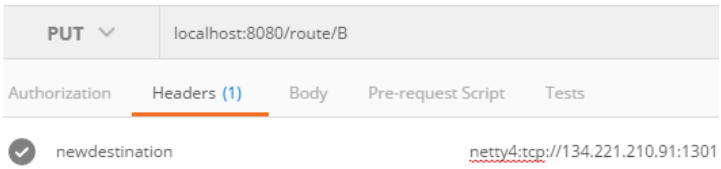

Figure 56: Connect route B with C

In the next two steps we create a failover scenario, see Figure 57. Route D does not receive data when route C crashes or stops working. When this happens, it is desirable that the user knows about this. That is why we test this is step 3. To overcome the problem of route D not receiving data, we add a "backup" route in step 4 to fall back on when route C becomes unavailable.

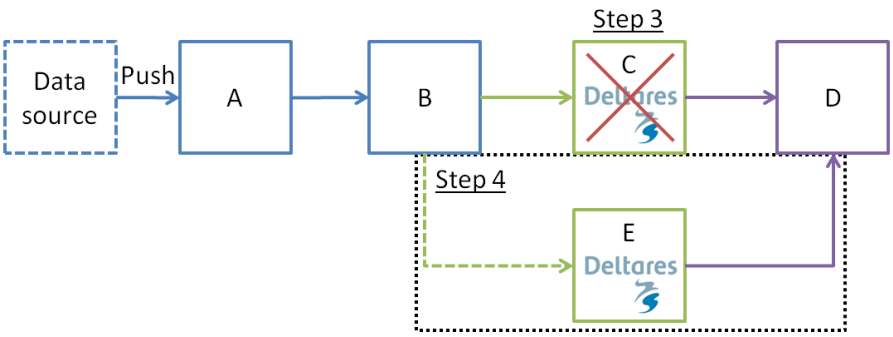

Figure 57: Overview of test steps 3 and 4

Step 3

Each Camel context in this test scenario contains a dead letter channel, so when the message cannot be delivered to the next route, the message is transferred to the dead letter channel. In this experiment we kill the Camel Context containing route C on the green machine. This means that route B is not able to deliver the message to route C any more. The result is shown in Figure 58. Route A and B both print the message and the message is then sent to the dead letter channel.

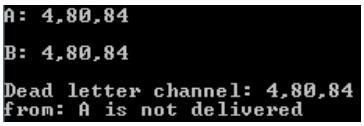

Figure 58: Dead letter channel from route B

Additionally, instead of killing the complete Camel Context within one machine, we are able to delete, or to change the status of a particular Camel route. Changing the status can be accomplished by sending a PUT request to */route/routeid?state=status* through REST server, where the *routeid* is the id of the route and the *status* is a desired state. A route can be running or suspended, and a suspended route can be resumed. The current status of a route can be requested through a GET request using REST server. More details on the methods and paths of the management controller are in Appendix A.

Step 4

In order to create a failover scenario we declare a new route called Route E. Route E is connected to route B and D, just as route C.

The goal here is to fall back on route E when route C is unavailable. Therefore we add Camel's failover loadbalancer component to route B, with route C and E as destinations, see Figure 59. Now, when route C is still not available after few tries, route B sends the message to route E.

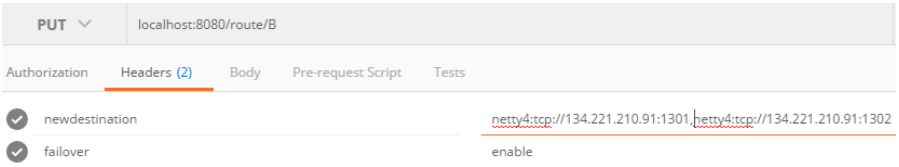

Figure 59: Add failover to route B

Step 5

As the final step of this experiment, we add a streaming pattern to route D. Therefore, we can send a PUT request similar to the one in step 4 of the previous test, but in this case, the process is called *Windowing*.

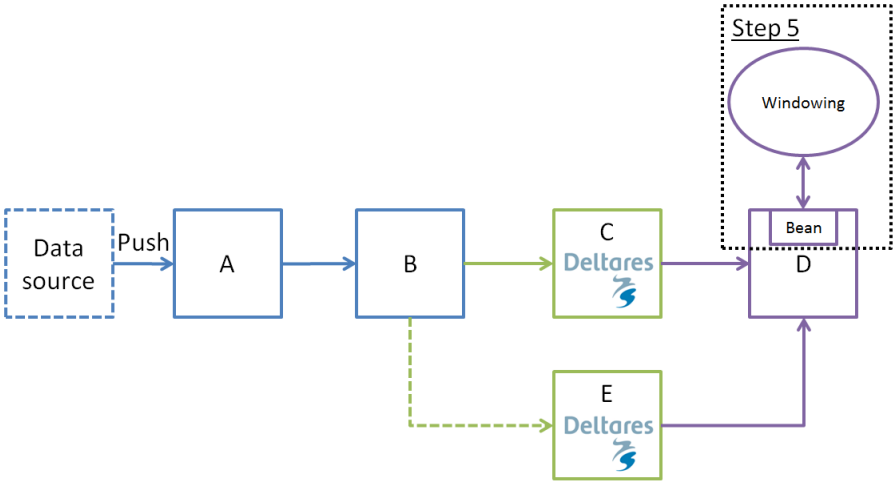

Figure 60: Overview of test step 5

The first number within the message is the minute of the event time of each message. The bean processor has windows of one minute and a trigger based on counting of five elements. So, when the *Windowing* processor has observed five messages with the same event time, a window is materialized. This means that the window contains five messages and a new message is generated. The result is shown in Figure 61.

This could be useful when an algorithm needs a batch of five measurements in order to perform a calculation.

```
Minutelist: ListBuffer(
40,92,60, proctime: 41,
41,40,70, proctime: 42,
41,85,36, proctime: 42,
41,84,99, proctime: 42,
41,53,49, proctime: 42,
42,35,65, proctime: 42,
41,57,42, proctime: 42)

Materialize window number 41! Window is: ListBuffer(
41,40,70, proctime: 42,
41,85,36, proctime: 42,
41,84,99, proctime: 42,
41,53,49, proctime: 42,
41,57,42, proctime: 42)
```

Figure 61: Windowing in route D

6.2.3 *Evaluation*

With this second experiment, we validate again our architecture against the requirements and aspects of the key drivers chosen beforehand, which are not validated at first experiment.

Requirements

FR-01 - The system must be able to connect with external heterogeneous data sources. The system uses Camel endpoint components to connect with the external data sources and other Camel routes. Camel support around 200 of those endpoint components, ranging from TCP connections, to web services and databases. In the first experiment we have demonstrated the "direct" component. In this experiment we have shown that it is possible to distribute the data sources endpoints and the Camel routes over different machines.

FR-06 - The system must be able to monitor the current status of the system. In our architecture we defined an orchestrator component as the component monitoring and instructing the VO's. The orchestration service is aware of the current active VO's, their state and runtime errors. Within the implementation of the prototype we are able to monitor the current status of the Camel routes through the management controller interfaces. The status of a Camel route can be: running or suspended. The user receives a conflict status code through the interface when the route is removed or does not exist. The implementation of the orchestration service is out of our scope, but we have implemented a part of the interface with status messages. Therefore, when the orchestrator is implemented, we can directly test this functionality.

FR-07 - The system must be able to report faults and unexpected behaviour of users, models and sources. In the architecture we stated that each VO reports their faults to the orchestration service. In the implementation we have used the dead letter channel for reporting faults back to the user. Those faults are faults regarding the connections between

the Camel routes. So, when the user wants to sent a message to an unavailable destination, the dead letter channel reports this.

Each Camel Context has its own management controller interface. The user is able to connect to it and sent HTTP requests to it. When user tries to send an unsupported instruction or a request for data about an unknown/non-existing component, the management controller sends an appropriate status code back to the user. It means that the prototype is able to report faults during runtime and is able to react to an unexpected behaviour of user.

FR-08 - The system must be able to handle instructions from data flow system. In this second experimental run we have demonstrated that our system is still able to handle instructions from the data flow system, when the data flow is distributed over different machines. And therewith we fulfil also requirement *FR-11* and *FR-12*.

FR-14 - The system should support data processing patterns to maintain consistency and data quality in stream processing. In related work we have covered the theoretical patterns and the state of the art in the field of the tools of stream processing. With this second experiment we have demonstrated the incorporation of windowing and trigger pattern within a Camel bean. With those two patterns we have proved that we are able to deliver consistent batches of five messages based on event time. However, this is only a small part of the potential of the streaming patterns, since not a lot of tools support them at the moment. More details about the streaming patterns can be found in future work.

6.2.3.1 *Key drivers*

In this experiment we have validated the two key drivers: fault tolerance and flexibility.

Fault tolerance. The second key driver for this research is fault tolerance. We defined fault tolerance as: *The property of a well functioning system that enables the system to remain properly operational when one or more components of the system fail or contain faults*.

In order to make system fault tolerant, we have used fault tolerant components from Camel and Spray. We found out that Camel itself has the build in functions to be able to fulfil our definition of fault tolerance. We have used the failover loadbalancer in our implementation to be able to switch automatically to another component when the regular component fails or becomes unavailable. Additionally, we have used the dead letter channel to monitor failing components. When a Camel route sends a message to an unavailable destination, a status message is sent back to the user. Those two fault tolerance components are both local within the Camel routes.

Next, we mentioned in key driver Section 2.1, that we found four fault tolerance related shortcomings based on future work from the previous student Pul [42] from RuG. Those four shortcomings are evaluated as well:

- *No validation when to flow graph modules are linked with different message types.* Camel has a so called content based router, which allows routing messages to the correct destination based on the content from the message exchange. So the router is able to direct the message to the correct flow graph model, which fits the message type. Camel's content based router is tested during development, however, it is not included into the prototype due to the lack of time. This happened because of the priorities of other functionalities.

- *Missing conversion from batches to streams.* It turned out that this was not a requirement for this research. For this research we have only looked at unbounded streams of data. However, with windowing and triggers we have shown that is it possible to reliably create batches of messages based on event time.

- *Camel tries to send data to an external component until it is successful.* Both the failover loadbalancer and the dead letter channel can be configured is such a manner that after a number of tries Camel gives up, and the specific additional actions are taken.

- *Changes to the flow graph can not be made until the failing component recovers.* When a component in a data flow fails, stops or suspends, the remaining flow components continue to work, and the changes through the management controller can still be made.

Furthermore, we have used Spray's status codes to be fault tolerant against unexpected behaviour from the users. Those status codes are returned back to the user when the user sends faulty requests or requests with non existing variables to the management controller.

Finally, the concept and implementation of the stream processing patterns also supports the overall fault tolerance of the system. The stream processing patterns help the delivery of correct data within a certain time span, making the system resistant to fault and late data.

Scalability. The first experiment fulfilled a scale-up part of our definition of scalability. The system was able to handle growing usage by adding new routes and creating new flows on a local machine. In this second scenario, we showed that we are able to spread the load over multiple flows in parallel over multiple machines. Therefore, we covered scale-out part of our definition of scalability.

Furthermore, each Camel route has its own local functionalities regarding fault handling and monitoring. Therefore, there is no central

monitoring or fault handling component. It covers some scalability aspects namely, because there is no extra stress on a central component when the number of Camel routes and data flows increase.

Finally, since we are able to create, change and remove Camel routes dynamically, we are able to scale and balance the load dynamically during runtime. Therefore, theoretically there are not short comings when we would need to scale our system further.

6.2.4 *Summary*

In the first experiment we were able to dynamically change the data flow during runtime, and we showed that our prototype is able to scale because of the parallel multicast. However, it happened only on one machine within one Camel Context.

In this scenario we demonstrated the additional features of our prototype regarding scalability and fault tolerance. We were able to create a data flow distributed over multiple machines, each accessible through their own management controller and running their own Camel Context.

Furthermore, we have shown two of Camel's fault tolerance components: the failover loadbalancer and the dead letter channel. Both of those components contribute to overall fault tolerance and availability of the system.

Lastly, we added the windowing and trigger streaming patterns. With those patterns we were able to convert single messages into batches of (five) messages.

6.3 CASE STUDY: STOOP

In Section 1.1 we stated that this research is (in the first case) created for a TNO project called STOOP. The goal of STOOP is to monitor changes in layers of soil where underground pipelines are located, in order to determine the stress rate on those pipes. The raw data from the used heterogeneous sensors needs to be transferred to different models. Those sensors and models form a data flow.

In this first prototype, we have created a solution were we can couple different data sources (provided that they are supported by the Camel endpoint components). Additionally, we are able to create a flow composed of different transformations and computations regarding the needs of the user. Those data flows can in theory scale and be changed dynamically during runtime. So we have created a flexible solution with loosely coupled components serving as the middleware for any type of data source, model and user.

This would work for pipes in STOOP, since the sensors monitoring the underground pipelines can be coupled, managed and dynamically changed by our middleware. Furthermore, the models which

compute the stress rate on those pipes can also be coupled, managed and dynamically changed by our middleware. By loosely coupling of both sensors and models we can create a dynamic and distributed data flow.

Therefore, we can say that our solution is suitable for project STOOP.

7 CONCLUSION & FUTURE WORK

In our research we have tried to answer the question: ***How can middleware dynamically adapt to management of ever changing external heterogeneous data sources and processing models in a large-scale sensor data flow system?*** According to that research question we have chosen three key drivers that we think are the most important for such a solution to have. The three key drivers are scalability, fault tolerance and flexibility.

Based on the research question and the key drivers, we presented a state of the art study. It has helped us to understand the challenges of data integration, the state of the art of streaming patterns and the applicability of semantic tooling.

From the related work we have found out that in order to answer the research question and provide an implementation that can meet all three key drivers, we have to design a solution based on the concepts on which the integration frameworks are built. Moreover, we should integrate them with the additional features from the stream processing patterns and semantic interoperability tools.

We created an solution which relies on an integration framework, namely Apache Camel, with the support for the EIP's, semantic storage and state of the art streaming patterns. An EIP is an advice about a general solution for frequently occurring integration problems. Multiple EIP's can be combined together as architectural building blocks to make up for a loosely coupled integration solution. Their combination altogether makes it possible for our solution to reach scalability, flexibility and fault tolerance, are in one system. Semantic storage provides functionalities regarding the description, understandability and meaning of the data. And lastly, the streaming patterns add features to our solution regarding data timing and data integrity.

We have created a reference implementation out of our solution using Apache Camel as a basis. We have implemented some additional features for flexible dynamic data integration. Moreover, we have added other functionalities to be able to handle data related specifics, like windowing and triggers. Furthermore, we have used REST Spray's HTTP service to enable automatic control over flow distribution.

Finally, we have performed two experiments to validate the feasibility of our solution in practice. Those experiments were used to evaluate our solution against the three key drivers and the functional requirements. Our solution fulfils our definitions of the three key

drivers and the functional requirements. The two experimental runs demonstrate that our solution is:

- Flexible due to the ability to loosely couple the system with the different data sources and models, and to change data flows, and delete data flow components during runtime.
- Fault tolerant, because of the including of the failover, dead letter channel and monitoring functions.
- Scalable, due to the ability to ad more flow components distributed over multiple machines.

The experiment scenario's were taken from the STOOP use-case, so we also have validated our system against the proposed use-case.

Therefore, we can say that the main research question is answered. Our solution can dynamically adapt to ever changing management of external heterogeneous data sources and processing models in a sensor data flow system.

7.1 FUTURE WORK

One of the major limitations during this research project was time. Therefore, there are some topics which have not received as much attention as others. In this section we cover those topics. Starting with our work regarding the requirements.

The fulfilment for requirement *FR-02* is not completely implemented. However, Apache Camel supports data translation to a wide variety of data formats, and with Camel you are able to specify your own translator with Camel beans. Since Apache Camel supports data translation, it helps us to reach data translation automatically. Within our prototype we are able to convert Camel routes into JSON format, so they can be interpreted by software and reproduced. However, data from data sources cannot be translated into other formats within our implementation for now. This is not implemented because of time constraints. Furthermore, Camel has support for data translation, so it is not innovatory which resulted in a lower priority for this first prototype.

Secondly, we have mentioned how the semantic storage of our solution fits in our architecture. We described a VO class as the general description of a VO which is stored in a database, and a VO instance being the actual running implementation of a VO class. However, a semantic modelling is not included into our first implementation. We have made a start by storing the Camel routes (which are the VO's) in JSON format. Due to time constraints and other higher priority requirements we have not investigated further in semantic storage part and therefore, are requirements *FR-05*, *FR-06* and *FR-09* are included

in the architecture as must, but are not in our first prototype. This might be a nice addition to the next version of our solution.

Thirdly, with two experiments we have proven that our solution is in theory scalable, due to the dynamic creation of new Camel routes and data flows over multiple machines. However, because of the time constraints we were not able to create a test scenario to proof the real ability to scale in practice.

One of the subjects of our research was also the concept of streaming patterns. Since those streaming patterns are relatively new, and not many tools provide the support for them, we decided to implement windowing and triggers ourselves. But this is only a small piece of what we have covered in the theory. We hope that the streaming patterns get more support by tools and libraries in the future, so that the they can be widely used in software products and the next version of our reference implementation.

Within our prototype we have management controllers which can be used through HTTP requests to control the Camel routes. It means that we provided handles for the usage of an orchestration service. So, the interesting component for future research is the orchestration service. The task of this service is to automatically create and control flows, and/or balance load between distributed flows based on the information from the semantic database and actual load of physical machines.

Furthermore, we have encountered some limitations during the development process which we think are worth to take a look at in the future. The first limitation we found applies to Apache Camel. Within this research we wanted to create flexible Camel routes. However, the Scala code responsible for the flexible routes is not flexible enough. Within Camel you are only allowed to use Camel's route supported functions between *from* and *to*. So it is not allowed to use traditional Scala code within a route, like *if else* for example. Therefore, the current code for the flexible Camel routes can probably become more efficient, which is desirable when more functionalities are added in the future.

The other limitation applies to the distribution of the Camel routes. In the final stage of this research we found out that *from-to* construction of Camel works in a way that *from* pushes and *to* listens. This is not desirable when a route on a external server should send data to a machine located behind a Network Address Translation (NAT) or a Firewall. The external server has no connections open to the outside world, but can be accessed from the outside. This limitation could be accessed by using SSH tunnels and by port forwarding on the server. Due to the time constraints we were not able to switch this push/pull principle in time by ourselves.

Finally, we have theoretical proven that our solution is feasible for a use-case like the STOOP project. The next step for future work is to

actually integrate our solution to the real calculation scenario within the STOOP project.

A

REST INTERFACE

Table 7 shows an overview on the paths of the Spray REST interface. To access the REST interface of the program, is to pick a REST Client (We have used the Chrome extension called Postman [40]) and go to the IP address + Port number where the program runs. Then add the paths and parameters/headers according to Table 7.

METHOD	PATH	PARAMETERS / HEADERS	DESCRIPTION
GET	/route		Returns information about all the routes **Status response:** 200 OK
GET	/route /status /{routeid}		Returns the current status of the route **Status response:** 200 OK, 409 Conflict, when a route with routeid does not exist
GET	/route/ {routeid}		Returns information about the route with routeid **Status response:** 200 OK, 409 Conflict, when a route with routeid does not exist
POST	/route/ {routeid}		Create new route with routeid **Status response:** 200 OK, 409 Conflict, when a route with routeid already exists
		fromuri	Mandatory header, specifies the input URI of the new route
		touri	Mandatory header, specifies (separated by commas) the output URI('s) of the new route
		process	Optional header, specifies the processor bean, this class must already be in the project
		aggregate	Optional header, specifies an aggregator, which combines every three messages into one new message. Fill this header to activate.
		failover	Optional header, specifies a failover strategy. When multiple output URI's are specified in the *touri* header, this failover tries sending the message to the next URI when the first URI becomes unavailable, and so on. Fill this header to activate.

PUT	/route/ {routeid}		Changes the route with routeid **Status response:** 200 OK, 409 Conflict, when a route with routeid does not exist
		?state=suspend	Suspends the route with routeid
		?state=resume	Resumes the suspended route with routeid **Status response:** 200 OK, 409 Conflict, in case of an unknown state. *When neither suspend nor resume are used the following headers can be used:*
		newdestination	Mandatory header, specifies (separated by commas) the new output URI('s) of the route
		process	Optional header, adds a processor bean to the route, this class must already be in the project
		aggregate	Optional header, specifies an aggregator, which combines every three messages into one new message. Fill this header to activate
		failover	Optional header, specifies a failover strategy. When multiple output URI's are specified in the *newdestination* header, this failover tries sending the message to the next URI when the first URI becomes unavailable, and so on. Fill this header to activate.
DELETE	/route/ {routeid}		Stops and removes route with routeid **Status response:** 200 OK, 409 Conflict, when the route with routeid does not exist

Table 7: REST interface

BIBLIOGRAPHY

[1] Tyler Akidau. *The world beyond batch: Streaming 102. The what, where, when, and how of unbounded data processing*. Jan. 20. URL: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102 (visited on 04/05/2016).

[2] Tyler Akidau. *The world beyond batch: Streaming 101. A high-level tour of modern data-processing concepts*. Aug. 5. URL: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101 (visited on 04/05/2016).

[3] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked Data - The Story So Far." In: *International Journal on Semantic Web and Information Systems* 5.3 (2009), pp. 1–22. URL: http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf (visited on 05/30/2016).

[4] André B. Bondi. "Characteristics of Scalability and Their Impact on Performance." In: *Proceedings of the 2nd international workshop on Software and performance* (2000), pp. 195–203.

[5] Qiyang Chen, James Yao, and Rubin Xing. "Middleware Components for E-commerce Infrastructure: An Analytical Review." In: *Issues in Informing Science and Information Technology* 3 (2006), pp. 137–146.

[6] Deltares. *Deltares - Enabling Delta Life*. 2016. URL: https://www.deltares.nl/en/ (visited on 07/24/2016).

[7] Chris Deptula. *Hadoop File Formats: It's not just CSV anymore*. Nov. 12. URL: http://www.inquidia.com/news-and-info/hadoop-file-formats-its-not-just-csv-anymore (visited on 05/31/2016).

[8] Glenn Donovan. *The History of Middleware*. July 31. URL: http://salescoachforstartups.com/the-history-of-middleware/ (visited on 03/08/2016).

[9] Alan R. Earls. *Deciding why and when to use an integration framework*. Dec. 2013. URL: http://searchsoa.techtarget.com/feature/Deciding-why-and-when-to-use-an-integration-framework (visited on 04/05/2016).

[10] Stephen Ewen and Kostas Tzoumas. *How Apache Flink enables new streaming applications. Part I: The power of event time and out of order stream processing*. Dec. 9. URL: http://data-artisans.com/how-apache-flink-enables-new-streaming-applications-part-1/ (visited on 05/31/2016).

[11] Mark Fisher et al. *Spring Integration Overview. Part III. Overview of Spring Integration Framework*. 2015. URL: http://docs.spring.io/spring-integration/reference/html/overview.html (visited on 02/23/2016).

[12] Apache Software Foundation. *Welcome to Apache Hadoop*. 2014. URL: http://hadoop.apache.org/ (visited on 04/11/2016).

[13] Apache Software Foundation. *Apache Camel: Components*. 2015. URL: camel.apache.org/components.html (visited on 06/08/2016).

[14] Apache Software Foundation. *Apache Flink: Scalable Batch and Stream Data Processing*. 2015. URL: https://flink.apache.org (visited on 04/19/2016).

[15] Apache Software Foundation. *Apache Storm*. 2015. URL: http://storm.apache.org/ (visited on 04/19/2016).

[16] Apache Software Foundation. *Apache Camel: Index*. 2016. URL: http://camel.apache.org (visited on 02/16/2016).

[17] Apache Software Foundation. *Apache Spark - Lightning-Fast Cluster Computing*. 2016. URL: http://spark.apache.org/ (visited on 04/11/2016).

[18] Apache Software Foundation. *Beam Incubation Status - Apache Incubator*. 2016. URL: http://incubator.apache.org/projects/beam.html (visited on 04/19/2016).

[19] The Eclipse Foundation. *Jetty - Servlet Engine and Http Server*. 2016. URL: www.eclipse.org/jetty/ (visited on 06/08/2016).

[20] OWL Working Group. *OWL - Semantic Web Standards*. 2012. URL: https://www.w3.org/2001/sw/wiki/OWL (visited on 05/09/2016).

[21] RDF Working Group. *JSON-LD 1.0*. 2014. URL: https://www.w3.org/TR/json-ld/ (visited on 05/12/2016).

[22] RDF Working Group. *RDF - Semantic Web Standards*. 2014. URL: https://www.w3.org/RDF/ (visited on 05/09/2016).

[23] RDF Working Group. *RDF 1.1 Turtle*. 2014. URL: https://www.w3.org/TR/turtle/ (visited on 05/17/2016).

[24] Thomas R. Gruber. "A Translation Approach to Portable Ontology Specifications." In: *Knowledge Acquisition* (1993), pp. 199–220. URL: http://tomgruber.org/writing/ontolingua-kaj-1993.pdf.

[25] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. URL: http://www.enterpriseintegrationpatterns.com/ (visited on 02/17/2016).

[26] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions*. 1st Edition. Pearson Education Inc., 2003.

[27] IBM. *IBM InfoSphere Platform - big data, information integration, data warehousing, master data management, lifecycle management and data security*. 2016. URL: www.ibm.com/software/data/infosphere/ (visited on 04/14/2016).

[28] Lightbend Inc. *Akka*. 2016. URL: http://akka.io/ (visited on 04/14/2016).

[29] Philippe Kruchten. "Architectural Blueprints — The "4+1" View Model of Software Architecture." In: *IEEE Software 12* (1995), pp. 42–50. URL: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf (visited on 05/11/2016).

[30] Kris Lachor. *Systems Integration with Openadaptor. Business system integration with little or no custom programming*. 2008. URL: http://java.sys-con.com/node/535350 (visited on 03/10/2016).

[31] Doug Laney. "3D Data Mangement: Controlling Data Volume, Velocity, and Variaty." In: *META Group Inc.* (2001). URL: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf (visited on 04/29/2016).

[32] Ross Mason. *ESB or not to ESB Revisited - Part 2*. June 28. URL: http://blogs.mulesoft.com/dev/tech-ramblings/esb-or-not-to-esb-revisited-%E2%80%93-part/ (visited on 04/04/2016).

[33] Ross Mason. *ESB or not to ESB Revisited - Part 1*. June 8. URL: http://blogs.mulesoft.com/dev/news-dev/esb-or-not-to-esb-revisited-part-1/ (visited on 04/04/2016).

[34] MeshSystems. *INTERNET OF THINGS (IOT) COMPANIES GEAR UP FOR $6 TRILLION MARKET DEMAND*. Jan. 17. URL: http://www.mesh-systems.com/company/newsroom/internet-things-iot-companies-gear-6-trillion-market-demand (visited on 04/29/2016).

[35] Eduardo Miranda. *Time Boxing Planning: Buffered Moscow Rules*. Oktober 2011. URL: http://www.stickyminds.com/article/time-boxing-planning-buffered-moscow-rules?page=0%2C0 (visited on 02/11/2016).

[36] MuleSoft. *Mule ESB | Enterprise Service Bus | Open Source ESB | MuleSoft*. 2016. URL: www.mulesoft.com/platform/soa/mule-esb-open-source-esb (visited on 02/23/2016).

[37] OpenAdaptor. *Openadaptor builds*. 2016. URL: www.openadaptor.org/downloads.html (visited on 03/10/2016).

[38] OpenAdaptor. *Welcome to openadaptor.org*. 2016. URL: www.openadaptor.org (visited on 03/10/2016).

[39] Wim van der Poel. *Waarom monitoren?* 2016. URL: http://www.ijknet.nl/stoop/monitoren (visited on 02/15/2016).

[40] Postman. *Postman | Supercharge your API workflow*. 2016. URL: http://www.getpostman.com/ (visited on 06/27/2016).

[41] The Netty Project. *Netty: Home*. 2016. URL: http://netty.io/ (visited on 06/08/2016).

[42] Ruurtjan Pul. "Flexible data flow management in data processing systems." Master thesis. Rijksuniversiteit Groningen, 2015.

[43] Chayan Sarkar, Akshay Uttama Nambi S. N., R. Venkatesha Prasad, Abdur Rahim, Ricardo Neisse, and Gianmarco Baldini. "DIAT: A Scalable Distributed Architecture for IoT." In: *IEEE Internet of Things Journal* (2014), pp. 230–239.

[44] Spring. *Spring Integration*. 2016. URL: http://projects.spring.io/spring-integration (visited on 02/23/2016).

[45] *Stream processing explained*. URL: http://www.sqlstream.com/stream-processing/ (visited on 04/11/2016).

[46] TNO. *Innovation for life*. 2016. URL: https://www.tno.nl/en (visited on 05/15/2016).

[47] TNO. *Mission and Strategy*. 2016. URL: https://www.tno.nl/en/about-tno/mission-and-strategy/ (visited on 04/22/2016).

[48] Andreas Tolk, Saikou Diallo, and Charles Turnitsa. "Applying the Levels of Conceptual Interoperability Model in Support of Integratability, Interoperability, and Composability for System-of-Systems Engineering." In: *Journal on Systemics, Cybernetics and Informatics (JSCI)* (2007), pp. 65–74. URL: http://www.iiisci.org/journal/cv$/sci/pdfs/p468106.pdf.

[49] Inc. TopQuadrant. *TopBraid Composer Standard Edition | TopQuadrant, Inc.* 2016. URL: http://www.topquadrant.com/tools/modeling-topbraid-composer-standard-edition/ (visited on 05/12/2016).

[50] Inc. Typesafe. *Spray | REST/HTTP for your Akka/Scala Actors*. 2014. URL: http://spray.io (visited on 06/02/2016).

[51] W3C. *Semantic Web*. 2015. URL: https://www.w3.org/standards/semanticweb/ (visited on 05/30/2016).

[52] Kay Wahner. *Spoilt for Choice: Which Integration Framework to use - Spring Integration, Mule ESB or Apache Camel?* 2012. URL: http://www.kai-waehner.de/blog/2012/01/10/spoilt-for-choice-which-integration-framework-to-use-spring-integration-mule-esb-or-apache-camel/ (visited on 02/16/2016).

[53] Alfred Ka Yiu Wong, Pradeep Ray, N. Parameswaran, and John Strassner. "Ontology mapping for the interoperability problem in network management." In: *IEEE Journal on Selected Areas in Communications* 23 (10 2005), pp. 2058 –2068.