CHALMERS
UNIVERSITY OF TECHNOLOGY

Event-Driven Data Collection and Processing

Master's thesis in Systems, Control and Mechatronics

JESPER JÄRNANKAR
JACOB SANDSTRÖM

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis EX073/2018

Event-Driven Data Collection and Processing

JESPER JÄRNANKAR
JACOB SANDSTRÖM

Department of Electrical Engineering
Division of Systems and Control
Automation research group
Chalmers University of Technology
Gothenburg, Sweden 2018

Event-Driven Data Collection and Processing
JESPER JÄRNANKAR
JACOB SANDSTRÖM

Supervisor: Kristofer Bengtsson, Department of Electrical Engineering
Examiner: Bengt Lennartson, Department of Electrical Engineering

Master's Thesis EX073/2018
Department of Electrical Engingeering
Division of Systems and Control
Automation research group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Event-Driven Data Collection and Processing
JESPER JÄRNANKAR
JACOB SANDSTRÖM
Department of Electrical Engineering
Chalmers University of Technology

Abstract

A modern manufacturing system typically consist of thousands of devices of different types from different vendors. Large amounts of data can be captured from these devices. However, to transform the data into valuable knowledge it is useful to have a common platform for data collection and processing.
In this thesis the open-source message broker and distributed streaming platform Apache Kafka has been integrated with Virtual Device, a framework developed by Volvo Cars for connecting shop floor equipment according to an event driven service oriented architecture. This integration will provide easier access to data and to develop, test and deploy services. In addition, it will enable researchers to test methods and tools using real data. For example the tool Sequence Planner, used for process modelling, optimisation, monitoring and control can be plugged in with little effort.
To test the capabilities of using a common platform for data collection and processing in combination with the Virtual Device framework two applications are proposed. The first application listens to events from various devices, enriches them with additional information and stores them in a combined database. This can later be used in optimisation, cycle time analysis or root cause analysis of process failures. The second application monitors an automatic maintenance process of spot welding electrodes called tip dressing. The application has two parts. One part predicts when the electrodes will need to be changed to be able to schedule manual maintenance and reduce wasted electrodes. A second part also detects when the tip dressing process is not performing as well as it should which will help reduce the time to rectify the error.

Keywords: Apache Kafka, Automation, Event-Driven Architecture, Manufacturing, Service Oriented Architecture.

Acknowledgements

We would like to thank Oskar Lindblad and Stefan Axelsson from Facility, Tooling and Equipment, Robots and Joining at Volvo Cars Torslanda for giving us the opportunity to write this master thesis.
We would also like to thank Håkan Pettersson, Subject Matter Expert in Information Technology, Interface and Network Technology at Volvo Cars Torslanda for your guidance and valuable information.
Lastly, we would like to thank our supervisor Dr. Kristofer Bengtsson at the department of Electrical Engineering at Chalmers University of Technology for guiding us through the project.

Jesper Järnankar and Jacob Sandström, Gothenburg, June 2018

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API application programming interface.
CPS cyber-physical systems.
EDA event-driven architecture.
EFA extended finite automaton.
EIE Equipment Information Exchange.
ESB enterprise service bus.
HMI human-machine interface.
IoT Internet of Things.
JMS Java message service.
JSON JavaScript object notation.
KPI key performance indicator.
LISA Line Information System Architecture.
NGPPS Next Generation Production Processes and Systems.
PLC Programmable Logic Controller.
PtP point-to-point.
RFID radio frequency identification.
SOA service oriented architecture.
SOP sequence of operations.
VD Virtual Device.
XML extensible markup language.

1 Introduction

In future manufacturing systems there will be an increased demand on operating in a flexible and efficient way with high quality, low cost and low emissions [1]. These demands can be met by integrating physical and digital systems through technologies such as Internet of Things (IoT) [2], cyber-physical systems (CPS) [3], big data [4] and cloud computing [5] which have seen great advances in recent years [1]. This integration is often referred to as Industry 4.0 or Smart factory [6, 7, 8]. A modern manufacturing system today is typically highly automated, very complex and consist of thousands of devices. The devices can often be connected to a factory network and produce large amounts of data. To meet the above mentioned demands the data must be collected in an efficient manner and be fully utilised by analysing and processing it in a proper way [9]. One of the challenges to overcome is to bring the different technologies together into a coherent platform for data collection and processing [9].

A coherent platform makes it possible to be very flexible in the analysis and processing. Since all data is gathered in one place several applications can perform different types of analysis on the same data. The result from an analysis of the data can further be put out to the platform for other processes to handle. Data can be combined and enriched with other information, from sources such as look-up tables or databases, to form higher level data. This means that data on the platform can span from bit values of sensors to key performance indicators (KPIs) or production specific events that can be analysed or presented. It would be very useful to have applications that can process the data and send it back to the platform for further analysis but it is important to focus on giving the possibilities to create the different processes that performs the analysis to get a truly flexible system. A platform where all data is gathered but where it is hard to create and deploy the applications for the analysis will not be flexible and adaptable for changes in the future.

Because of this, different approaches for integration and communication between equipment, control systems and business applications, such as service oriented architectures (SOAs) and event-driven architectures (EDAs), are receiving more and more interest within manufacturing. These revolve around services, defined as independent, self-describing and interchangeable code modules[10], that perform business tasks. A project run by researchers at Chalmers University of Technology, KTH

Royal Institute of Technology and Lund University in collaboration with partners from the automotive industry resulted in the Line Information System Architecture (LISA) [11]. LISA makes it possible to integrate a large number of different devices and process data in a flexible, efficient and event-driven way [11].

Today, at Volvo Cars, a lot of equipment are interacting and communicating. In the body shop at the factory in Torslanda there are at least 800 industrial robots performing tasks such as welding, gluing and moving parts. There are also other kinds of equipment such as Programmable Logic Controllers (PLCs) and radio frequency identification (RFID) tags.

A framework developed at Volvo Cars called Virtual Device (VD) that allows equipment to send data to a common platform is implemented today which makes it possible for devices to communicate with each other and with high level production controlling services. The VD framework is designed to operate in an event-driven manner similar to what is proposed in LISA. The common platform used today, delivered by IBM, is however not easily accessible for development of new services and hence not used as much as it could be. The consequence of this is that the capabilities of VD are not being fully utilised. Because of this there is a need for a more accessible and flexible way to collect and process data and to develop and test new services for analysis, optimisation, supervision and control.

1.1 Purpose

The purpose of this project is to investigate how data from equipment in the body shop at Volvo Cars Torslanda is gathered and analysed today. A second part is to present an easier way to access data, develop and deploy services that analyses data with an event-driven approach to get a flexible and adaptable platform.

1.2 Demarcations

The focus of the thesis is the integration of a messaging and data processing platform with the existing technologies at Volvo Cars. The thesis will hence not include advanced algorithms for analysing data, the presented services instead highlight the flow of information from raw data to valuable information.

The thesis will not present any evaluation of performance indicators such as message throughput for the developed integration.

The presented integration is developed to only receive data from the equipment in the factory, not to send data such as control signals to the equipment.

1.3 Our Contribution

This master thesis has contributed with a driver for the VD environment at Volvo Cars Torslanda, to send data out on a Apache Kafka cluster. This acts like an enterprise service bus (ESB) in parallel but separated from the existing one in the factory. This gives the possibilities to test and run services for logging and analysis of data in a convenient way on real data without interfering with the production.

Implementations of a `MessageReceiver` and a `Sink`, according to interfaces defined at Volvo Cars, for connection to the Kafka cluster has been created to be able to use existing tools at Volvo Cars for analysis. These makes it possible to take data from the cluster into existing services and also sending data back out on the cluster.

1.4 Thesis Outline

This report is divided into five chapters. Chapter 1 introduces the report. Chapter 2 present theory about integration architectures for communication between services and different ways to handle data. The Virtual Device framework, the architecture that is developed and used at Volvo Cars Torslanda, is presented in chapter 3. Chapter 4 presents the developed Virtual Device driver for Apache Kafka, adapters to use existing services at Volvo Cars on the Apache Kafka Cluster and two services for showing principles of the use of services in an event-driven architecture. Chapter 5 provides a summary and discussion of the content of the report.

2
Theory

This chapter presents theory about three different integration architectures for communication between services. First point-to-point integration is introduced which is the traditional way of communicating between devices and applications in a manufacturing system. Then the service-oriented and event-driven architectures are introduced. These are broad concepts that when applied provide more flexibility in data collection and processing. One implementation of such an architecture, the Line Information System Architecture is also introduced which is an event-driven approach to integration of manufacturing devices and services. Finally, some common ways of structuring and processing the data that is being transferred is presented.

2.1 Integration Architectures

In a highly automated manufacturing environment communication between devices and business applications is necessary. Data needs to be sent to devices for control and gathered from the devices for analysis and monitoring. There are several ways to establish such connections. The traditional approach to this is using point-to-point (PtP) integration while more recently companies and researchers have started focusing on more flexible methods such as SOA and EDA These three approaches are presented here and compared.

2.1.1 Point-to-Point Integration

In manufacturing the traditional method for communicating between shop floor equipment and applications has been to use PtP integration. The PtP approach works by establishing a connection directly between a client and a server that need to communicate with each other. This means that both the client and the server need to have knowledge about each other [10]. The main advantage of PtP is the simplicity, however this is also the great downside. PtP is meant to be used when there is a need for communication between only one source and one target application [12]. When involving more than two applications the complexity starts to grow very rapidly since every application needs to establish separate connections to all other applications that it communicates with. These connections may also need different interfaces to be able to communicate [10, 12]. This has lead to PtP being called "spaghetti integration" which figure 2.1 illustrates. The figure shows a PtP integration example with 8 applications. Every application is connected to all other applications which results in $n(n-1)/2$ connections, in this case 28 in total.

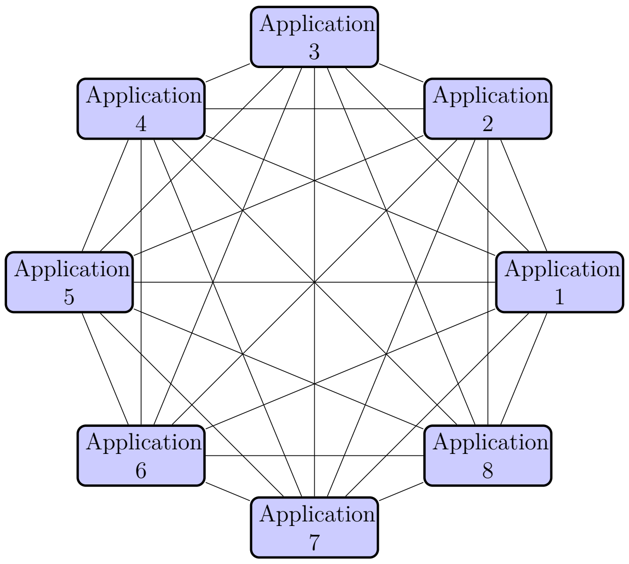


Figure 2.1: An example of a PtP integration between 8 applications. Every application is connected to all other applications which results in 28 connections in total.

This type of architecture becomes very difficult to maintain and make changes to when the number of connections grows. In a modern manufacturing setting PtP integration becomes unfeasible due to the increasing amount connected devices and difficulties in scaling up or down [10].

2.1.2 Service Oriented Architecture

SOA is a way of building and organising applications within a business [13, 10]. The principles of SOA are independent on hardware and software which makes it very flexible for integrating equipment and applications from different vendors [10]. As the name states, SOA is built around developing applications as services and combination of other services.

The major components of a SOA are: services, message bus, process choreography or orchestration and services registry [10].

A SOA applies a request/response type of communication. A user makes a request to a service and then waits for that service to respond. The first service may invoke other services in order to perform its task [10].

SOA is a good choice of architecture when the system handles data with high integrity and demands from the client to get a response. This is often the case of system with transactional nature [14].

Services in SOA

A service in the broadest general sense is something useful that a provider does for a consumer [13]. However, in terms of software, services are independent, self-describing and interchangeable code modules [10]. This means that a service should be able to operate alone without dependencies but also be used in combination with other services in a composite application [10].

Message Bus

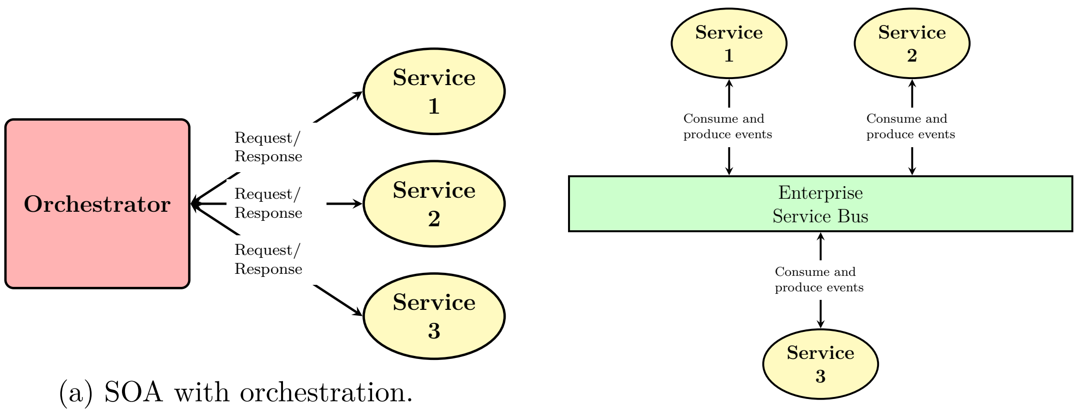
An ESB is a flexible integration infrastructure for connection between applications and services [10]. The ESB provides consumers with access to services and data as well as reducing the number and complexity of interfaces in the system [13, 10]. It acts as a central point for connection and thus removes the problems with PtP integration mentioned in section 2.1.1.

Orchestration and Choreography

When a user makes a request to a service for a piece of information or an action to be performed the service that receives the request may not be the only service involved in the task of delivering a response to the user. The request may require multiple services to be executed in a sequence to get the desired response message. Since the services are independent there is a need for logic that routes the intermediary responses to the correct following service. This is typically done using two different methods: *orchestration* and *choreography* [10].

Orchestration is a centralised approach to controlling multiple services. An endpoint is created which has logic for invoking the correct services in the correct order. The individual services are only connected to the endpoint and not to each other [15]. This endpoint is also called a composite service or an orchestrator and an illustration is shown in figure 2.2a. The benefit of using this approach is that it gives a good way of controlling the work flow in synchronous processing. The tradeoffs are that it creates dependencies between services, if a single central orchestrator is used there is a single point of failure and the synchronous processing can create blockings of requests [15].

Choreography is a reactive approach that does not use a central process to control the services. Instead there are predefined rules for how individual services should interact with each other. It can be seen as a message exchange protocol. The services will execute their tasks asynchronously based on the rules and not by commands from a central process [16]. An illustration of choreography is shown in figure 2.2b. This is an approach that is adopted by the event-driven architecture which is presented in section 2.1.3. The main benefits of using choreography are that it provides loose coupling of services, faster end-to-end processing as the services can work in parallel and asynchronously and that there is no longer a single point of failure. The tradeoff is that the logic is shifted from a central point to being distributed across the individual services which might make it more difficult for developers to see how the process will execute.



(a) SOA with orchestration.

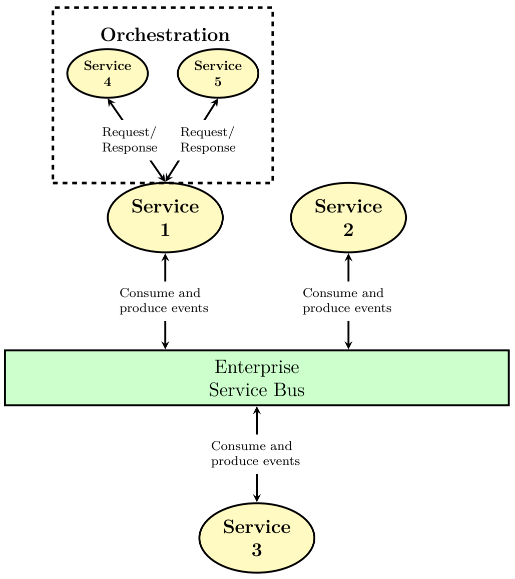
(b) SOA with choreography.

Figure 2.2: Two different approaches to organising and combining services, orchestration and choreography.

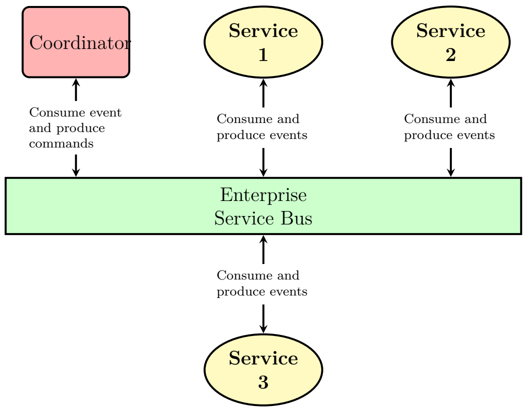
Two types of hybrid solution have been encountered that combines an orchestration with choreography approach. This will have some of the benefits from both of the approaches[16].

One way to combine choreography with orchestration is to use choreography in the outer layer of the services and internally, several microservices can work in a orchestration manner, see figure 2.3a. The internal orchestrator can both work synchronous and asynchronous but when it performed all actions it needs to perform it send out an event with the result. This will make services decoupled. One problem is that if a service that is run in a orchestration manner will be down, the service running it will also be blocked for usage [16].

Another way is to have a choreographic setup but having a coordinator that are sending and receiving events to services in the correct order to drive the flow, see figure 2.3b. This achieves the synchronous behaviour as an orchestrator has but the services are decoupled. This means that services that will be deployed through the



(a) Hybrid setup with choreography in an outer layer and orchestration internally.



Coordinator
Service 1
Service 2
Consume event and produce commands
Consume and produce events
Consume and produce events
Enterprise Service Bus
Consume and produce events
Service 3

(b) Hybrid setup with a coordinator driving the flow of execution.

Figure 2.3: Two types of hybrid SOA that combine both orchestration and choreography.

coordinator after another service has finished can be run from another process that doesn't need the first process. A tradeoff is that the coordinator needs to know the commands to call services in the correct way which is not needed in a pure reactive approach [16].

Service Registry

A SOA typically has a service registry containing descriptions of processes, services, events and important metadata of the system [13, 10]. A consumer can search the service registry and retrieve information such as status, owner and dependencies of a service [13].

2.1.3 Event-Driven Architecture

An EDA is similar to an SOA in many ways. It utilises services to perform actions and an ESB to send messages between services. The main difference is the focus of the architecture. As the name says, events are the focus of an EDA [14]. In an EDA the communication is based on publish/subscribe of events. Services are subscribing to events that will trigger an action and when a service has performed its task it may publish an event [17]. A user can request something by publishing an event on the bus. The user doesn't know and doesn't need to know who is listening to the event. The services that are subscribed to that type of event will get triggered, perform its action and publish a new event on the bus. Other services may be subscribed to those new events and they are then triggered. Eventually an event is published that contains the thing that was first requested by the user [18]. In this way the events that are published on the bus are the driving force of the whole system of services. This approach is very similar to the choreography of an SOA which provides a very loose coupling between different services [17]. One disadvantage of an EDA is that for large systems with a large amount of services the complexity grows and it can become very difficult to understand exactly what happens and what the effects are of a certain event.

2.2 Line Information System Architecture

LISA is the result of a research project that was carried out by researchers from Chalmers University of Technology, KTH Royal Institute of Technology and Lund University in collaboration with partners from the automotive industry. The project aimed to develop an information system for capturing raw data from production systems and transforming it into high-level knowledge that can be used in production management decisions, online monitoring, control, optimisation and reconfiguration. LISA is presented in [11] and this has been used as source for the information that will follow here.

LISA is a form of EDA that makes it possible to integrate a large number of different devices and services in a flexible and efficient way. There are three core components of LISA: the message bus, the LISA message format and communication and service endpoints.

2.2.1 Message Bus

LISA makes use of an ESB to send messages between applications. The user can choose any ESB they like as long as it fulfils a number of requirements that ensures loose coupling and avoids PtP connections. These requirements are:

- **Message**: The information or data are packaged into a message that can be transmitted on a message bus.
- **Messaging**: Messages are transferred immediately, frequently, reliably and asynchronously using customisable formats. Messaging is event based: when there is a new message, it is sent to the message bus.
- **Publish-subscribe channel**: When a message is sent on a publish–subscribe channel, a copy of the message is delivered to each channel subscriber.

- **Message filter**: If the content of an incoming message does not match the criteria specified by the message filter, the message is discarded. This pattern allows each application to further filter incoming messages.

2.2.2 LISA Message Format

In LISA there is no restriction to what can be considered an event, the user defines the events which makes the system flexible. The events are published on the ESB in the form of messages. The format of these messages is simple and gives the user much freedom to design the messages to suit their needs. A message in LISA consists of two parts, a header and a body. In the header, information related to sending and routing of the message is stored. The body of the message is what is actually interesting for the user. It contains an ordered set of key-value pairs where the values usually are primitive data types but can also be more complex types such as lists or maps. In this way it is possible to build hierarchical structures in a LISA message. The key-value pairs in the body can contain any data needed, the only requirement is that the keys event id and event timestamp should be included.

2.2.3 Endpoints

There are two types of endpoints in LISA, communication endpoints and service endpoints.

The communication endpoints connects devices on the ESB and acts as an adapter between the ESB and the device. Events sent from a device are converted to the LISA message format before being published. Likewise, incoming events are filtered and converted to the device specific format. In this way any number of different devices can be integrated into LISA and when a device or application is changed or replaced only the communication endpoints needs to be updated.

The service endpoints are responsible for transforming low-level events into higher level knowledge. There are three different basic transformations defined in LISA: Fill, Map and Fold. Both the Fill and Map transformations adds data to an event by appending key-value pairs to the message body. The Fill transformation does this using only static data meaning that the result of the transformation will always be the same if the input event is the same. The Map transformation is dependent on the current system state and takes both an event and current state as input. The result of the transformation may vary even if applied to the same event as the system state may have changed. Finally, the Fold transformation takes a sequence of events and produces a single new event based on some conditions.

A tool that has been developed to be used according to the concepts proposed in LISA is Sequence Planner. Sequence Planner is a tool developed at Chalmers University of Technology used for modelling, visualisation, optimisation, monitoring and control of discrete event systems [19, 20].

2.3 Message Formats

In general there are no restrictions of how the messages should be designed for an SOA or an EDA. However, since the applications will be interacting with each other a common format of the messages must be decided so each application knows how to handle the messages that it receives. There are many different formats for these types of messages where the two main ones are extensible markup language (XML) and JavaScript object notation (JSON).

2.3.1 XML - Extensible Markup Language

XML is a software- and hardware independent tool for storing and transporting data in the form of text documents. It is developed by the World Wide Web Consortium (W3C) and contains a set of rules for breaking up a document into parts which are used to structure the document in a hierarchical way [21, 22]. An XML document is built up using *elements* where an element consists of a start tag and an end tag with the option to have text and other elements in between. A tag should have a name which can be specified freely by the user and can also have attributes tied to them [22]. Listing 2.1 shows an example of an XML document. The example document contains information about two employees. The top level element of the document is `<employees>` which has two child elements that both have the tag `<employee>`. Each employee element has an attribute that describe the age of the employee and it has two child elements that contains the first and last name.

```xml
<employees>
    <employee age="25">
        <firstName>John</firstName>
        <lastName>Doe</lastName>
    </employee>
    <employee age="32">
        <firstName>Anna</firstName>
        <lastName>Smith</lastName>
    </employee>
</employees>
```

Listing 2.1: Example of data stored in XML format.

Each XML document has one single root element that is the parent of all other elements. For the example in listing 2.1 the `<employees>` element is the root element. The root element and its children can have an arbitrary number of child elements. In this way a tree structure is built up, as can be seen in figure 2.4 that shows the same information as the example in listing listing 2.1.

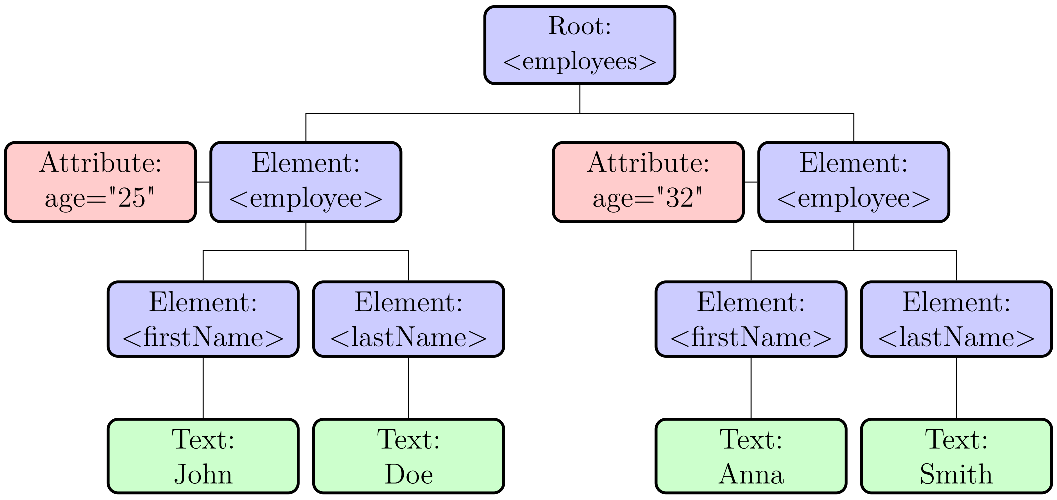


Figure 2.4: A tree showing the structure of an XML document.

2.3.2 JSON - JavaScript Object Notation

JSON is a language independent text-based syntax that is easy for humans to read. Due to it use of conventions and similarities from several programming languages it is suitable for defining data-interchange formats[23, 24]. According to [25] scalar variables, linear lists and key-value pairs are the most common structure used in programming and the fact that JSON represents these makes it convenient and efficient.

There are two structures that forms the JSON: A collection of name-value pairs and an ordered list of values[24]. A set of name-value pairs are called an object where name is separated from the value with a colon, pairs are separated with comma and all pairs are enclosed with curly brackets[23]. The lists, called arrays, begins with left square bracket, ends with right square bracket and separates the values with comma[23]. The name in the name-value pair is always a string and the value, for both objects and lists, can be a string, number, object, array or one of the three literal name tokens `true`, `false` or `null`. All strings are wrapped in double quotes[23]. Listing 2.2 shows an example of data stored in JSON format. This is the same data as the example displayed in section 2.3.1 and listing 2.1.

```
{
  "employees":[
    {
      "age":25,
      "firstName":"John",
      "lastName":"Doe"
    },
    {
      "age":32,
      "firstName":"Anna",
      "lastName":"Smith"
  }
  ]
}
```

Listing 2.2: Example of data stored in JSON format.

2.4 Batch and Stream Processing

Depending on the application area some methods are more suitable than others for processing large amounts of data. Typically there are two types of processing: batch and stream processing.

2.4.1 Batch Processing

In batch processing multiple jobs are collected in a batch before being processed together. The batches are typically stored in a database before being processed [26]. All the jobs in the batch are being processed in the same way by a program. Figure 2.5 shows the typical data flow in a batch processing setup. This is typically done for tasks that are not time critical and only need to be performed once in a certain time interval [27]. The size of the time interval can be everything from a matter of minutes to hours, days or weeks [28]. A batch computation can also be triggered when a certain batch size has been reached, for instance there might be a need to perform a task on 100 jobs at a time. Then the process will wait until 100 jobs have been collected no matter how long it will take and then perform the task on all 100 at the same time. A hybrid solution can also be implemented where the process has a maximum wait time before next execution but can also run if the batch size reaches a threshold. There are several benefits of running a process in batches. One is that it is possible to schedule tasks to a time when the computation resources are less busy. Another benefit is that because multiple jobs are processed at once there will be less overhead in the system [28].

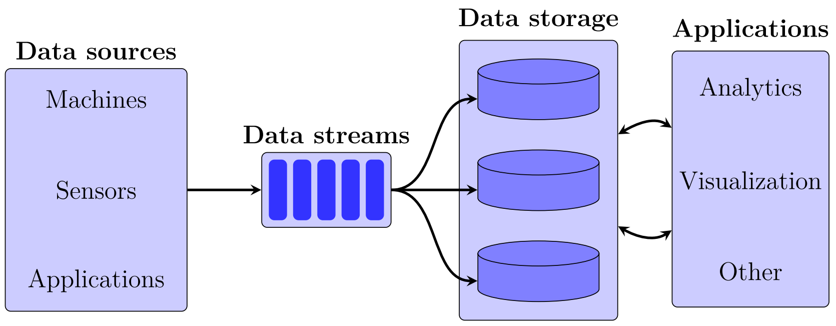


Figure 2.5: An example of how batch processing works. The data gathered from different sources is stored in batches before being processed all together by various applications.

2.4.2 Stream Processing

In stream processing data is processed continuously, either directly when it is created or when it is received by an application. A lot of data is created and transmitted as streams or series of events over time which makes this type of processing seem more natural than the traditional batch processing [29]. The applications receive a data stream and performs one or more computations on each of the individual records that is contained in the stream. After the computations are complete a new stream with modified data is created and sent from the application [26]. This type of processing is illustrated in figure 2.6. The streaming application can do simple tasks such as transform a value, update an aggregate or trigger some other action. To do more complicated tasks a stateful streaming application can be created that, as the name suggests, uses the state of a process to perform calculations. The state is information obtained from previous data that can be of importance. A stateful stream application combines data storage in the form of a table or database and the concepts of stream processing mentioned here [26].

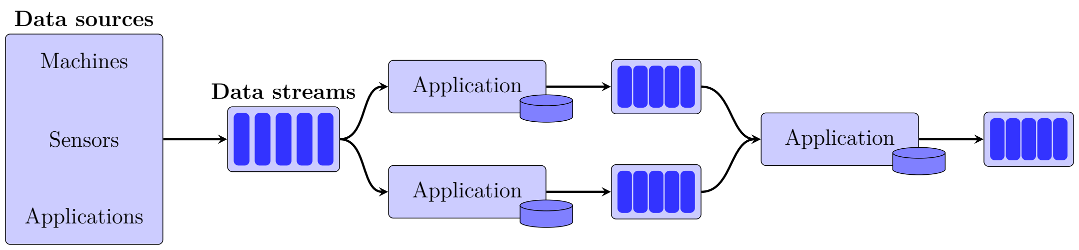


Figure 2.6: An illustration of how stream processing works. Data is processed continuously as it arrives in the application. The applications performs a task on the stream of data and sends out a modified version for other applications to use.

There are typically two different ways of performing stream processing: *record-by-record* and *micro-batching*.

Record-by-record processing is performed as described above. Each record of data is being processed individually directly when it arrives to the application. Although there is some network latency that cannot be completely eliminated this approach is still often considered to be real-time processing. The downside of processing each record one by one is that it introduces extra overhead [29].

If there is a need to decrease the overhead while maintaining the ability to process data in a stream one can apply a method called micro-batching. While the standard batch processing typically collects data for hours or days the micro-batch approach collects data in the range of milliseconds to seconds. A maximum batch size is also set to avoid creating too large batches. This size is often kept quite small to keep the response time of the process down. Many computations are also more efficient when performed in batches which is another benefit of applying micro-batching. Since each record is not processed directly when it arrives but still with a low latency compared to traditional batch processing this approach is considered to be near real-time [28].

3
Virtual Device Framework at Volvo Cars

This chapter provides a description of Virtual Device, a framework developed at Volvo Cars Torslanda for connecting shop floor equipment according to an event-driven service oriented architecture. The description includes a general overview of what Virtual Device is, its components and building blocks as well as how to configure and use it. Finally, the chapter includes a comparison between Virtual Device and the Line Information System Architecture where both similarities and differences are presented. The information presented in this chapter was provided by Håkan Pettersson, Subject Matter Expert in Information Technology, Interface and Network Technology at Volvo Cars Torslanda.

In 2004, Volvo Cars Torslanda conducted a study of the shop floor IT systems and found that the number of separate systems had become so many that they were considered a risk to the organisation. This started a program called Next Generation Production Processes and Systems (NGPPS) having the goal of replacing the many old IT-systems with a new site common solution. As a part of the NGPPS program a project called Equipment Information Exchange (EIE) was started which goals were to develop an integration platform for connecting shop floor devices that would replace the custom-made systems with a common, standard-compliant and future-proof system. The project concluded in 2010 and the result was the VD framework. The VD framework is designed to be used according to the principles of an SOA or an EDA where low-level data from shop floor equipment is published on an ESB and services connected to the bus process the data. The framework is written in Java which makes it easy to install on any operating system that supports a Java virtual machine and it is lightweight which makes it usable on low-cost computers. An instance of VD runs on one computer and connects several shop floor devices to a message broker that is compliant with Java message service (JMS). An overview of how VD is meant to be used can be seen in figure 3.1.

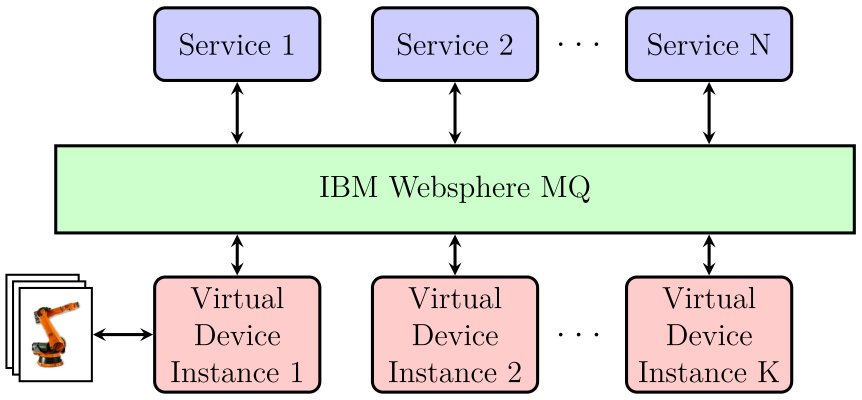

Service 1
Service 2
· · ·
Service N
IBM Websphere MQ
Virtual Device Instance 1
Virtual Device Instance 2
· · ·
Virtual Device Instance K

Figure 3.1: The figure is showing an overview of the virtual device framework with several instances of VD, with equipment connected to them, and several services connected to IBM Websphere MQ.

3.1 Device Drivers

Communication with equipment is enabled in VD through equipment specific plug-ins called *device drivers*. A device driver defines how the communication should be performed such as how to read or write data and if there is something that needs to be initialised before communication is enabled. An instance of VD has at most one device driver instance for each equipment type that is connected to that VD. A device driver for a specific type of equipment is only written once and are reused in all VD instances where that type of equipment is connected. Use case specific properties are set in a configuration file, see section 3.2.

There are three levels of a device driver: *driver*, *context* and *channel*. The bottom level of the driver structure is the channel. A channel is a connection to a device and it describes how data is sent from and to the device. Sending data from a device can be initiated either by a user making a request to read some data or by the device itself in the form of an event notification that is sent automatically when a certain event occurs. Multiple channels can be opened to separate different types of data or different sources. Each channel has three state variables: created/destroyed, enabled/disabled and opened/closed. The channels are created and destroyed when VD is started and shut down. When created the channel can be set to enabled or disabled which corresponds to allowing or disallowing the ability connect to the device. A channel can also be in either the opened or closed state. This state corresponds to having a connection established or not. In other words, enabled allows connection while opened is the actual connection.

Above the channels is the context level. All channels must belong to a context and there can be multiple contexts defined. This is currently not used, all current device drivers have one single context that holds all the channels for that driver. The context could however be used in the future if there is a need for another level of abstraction. There could be multiple contexts that hold a subset of the channels each.

The top level of the device driver is the driver level. There is only one driver per device driver which holds all the contexts. In the driver one-time actions that prepare the run environment can be performed. These actions will apply to all contexts and all channels. The structure of a device driver and a VD instance can be seen in figure 3.2.

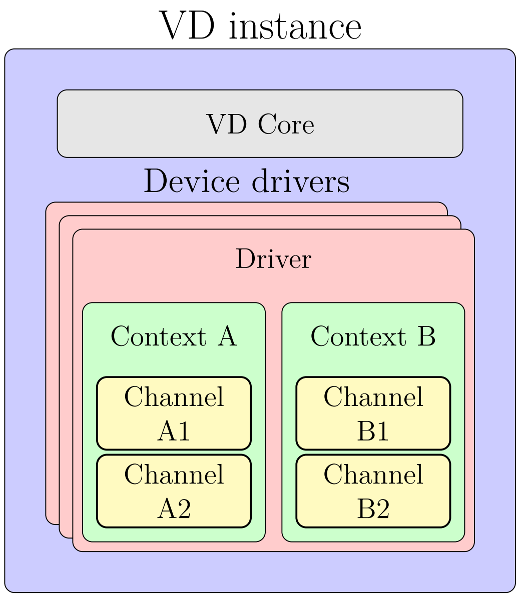


Figure 3.2: The figure shows the structure of an instance of a virtual device. A virtual device can contain several device drivers that each can contain several contexts and channels.

3.2 Configuration

Each VD instance is set up according to the specific needs of some user. It will contain only the device drivers needed and each device driver will be configured to suit the current use case. To do this a configuration file, written in XML, is supplied to VD on start up that contains all the information about how the instance is supposed to operate. Each instance has its own configuration file. The file can contain the following tags: `<DeviceDriver>`, `<Context>`, `<Channel>`, `<DataPoint>` and `<Transformer>`. The first three are the levels of the device driver described in section 3.1 while the last two concern the data that is read from and written to channels and how the data can be transformed internally in VD before being sent to a new destination.

Elements created with the `<DeviceDriver>` tag specifies what device drivers will be used by the VD instance, what Java classes are needed to run the driver and what options should be used for the driver. There can be at most one `<DeviceDriver>` element for each type of device driver in a configuration file. This corresponds to the driver level of the device driver as described in section 3.1. An example of a driver configuration is shown in listing 3.1. The configuration is for for a Siemens PLC.

```
<DeviceDriver>
  <dd:ID>SiemensVDComDriver</dd:ID>
  <dd:ClassName>devicedriver.siemens.vdcom.driver.DeviceDriver</dd:ClassName>
  <dd:ClassPath>../devicedrivers/SiemensVDCom-Driver-1.9.0.jar</dd:ClassPath>
  <dd:ClassPath>../devicedrivers/DeviceDriver-Common-1.9.0.jar</dd:ClassPath>
    <dd:FirstReopenDelayMilliSeconds>5000</dd:FirstReopenDelayMilliSeconds>
    <dd:ReopenDelayMilliSeconds>5000</dd:ReopenDelayMilliSeconds>
</DeviceDriver>
```

Listing 3.1: Example of configuration for a driver in VD.

The `<Context>` elements describe what contexts are present for each device driver. The element must contain an id and information about which driver it belongs to. There can be an arbitrary number of `<Context>` elements for each driver. Listing 3.2 shows a simple configuration of a context that utilises the configured driver from listing 3.1.

```
<Context>
  <ctx:ID>VDComProtocol</ctx:ID>
  <ctx:DeviceDriverID>SiemensVDComDriver</ctx:DeviceDriverID>
</Context>
```

Listing 3.2: Example of configuration for a context in VD.

In the `<Channel>` elements the channels for the drivers are defined. In each `<Channel>` element there should be information about which context the channel belongs to, an id for the channel and options describing things like whether the channel should be set to enabled or not on start up and if automatic reconnection should be used. An example of this is shown in listing 3.3 where a PLC channel is configured to use the context from listing 3.2. This particular channel is just used for emulating alarms for testing purposes.

```
<Channel>
  <ch:ID>plemulated_alarm</ch:ID>
  <ch:ContextID>VDComProtocol</ch:ContextID>
  <ch:Type>
    <ch:Client>
      <ch:DurableConnection>
        <ch:ReopenAtConnectionFailure>true</ch:ReopenAtConnectionFailure>
      </ch:DurableConnection>
    </ch:Client>
  </ch:Type>
  <ch:Mode>
    <ch:Disabled/>
  </ch:Mode>
  </ch:Option><ch:Option key="tcp.remoteHost">
    <co:Value>gotsvw1255</co:Value>
  </ch:Option>
</Channel>
```

Listing 3.3: Example of configuration for a channel in VD.

Internally in VD the data is represented by a so called datapoint. A datapoint correspond to a piece of memory in the equipment that can be read from or written to by making requests or automatically read from when an event occurs. An example of a datapoint is a variable in a robot. The variable can describe some state in the robot that a remote user can be interested in reading, in which case the user would either actively requesting to read the datapoint to get the value of the variable or an event could be emitted when the variable is updated and the user can then subscribe to the event notification. The datapoint could also describe some parameter that a user might want to adjust, then a write request can be made on that datapoint which allows the user to change its value. It is also possible to initiate a write request automatically on datapoints by configuring a subscription to event notifications of other datapoints. When a datapoint is configured to listen to events from another datapoint, VD picks up the data from the event and automatically makes a write request on the subscribed datapoints using that data. The events can also be configured to be sent to the ESB. To define datapoints the tag `<DataPoint>` is used. It should include an id for the datapoint, information about which channel it belongs to and an address that describes where the data is located in the device. If events are emitted by the datapoint, the subscribers of those events should also be configured. These subscribers can, as mentioned, be either another datapoint or the ESB. A continuation of the example with the PLC alarm emulator from listing 3.3 is shown in listing 3.4. Here a datapoint is configured for an emulated alarm. The datapoint can emit events that are in this case being published to the ESB. This is stated by the element `<ESBSubscriber>`. They could have been sent to another datapoint by instead using an element called `<DataPointSubscriber>` where the id of the subscribing datapoint would be included.

```xml
<DataPoint>
  <dp:ID>vcta.123001e.alarm</dp:ID>
  <dp:ChannelID>plemulated_alarm</dp:ChannelID>
  <dp:Address>*</dp:Address>
  <dp:EventNotification>
    <dp:ESBSubscriber>
      <dp:TransformationID>AlarmEventTransformation</dp:TransformationID>
    </dp:ESBSubscriber>
  </dp:EventNotification>
</DataPoint>
```

Listing 3.4: Example of configuration for a datapoint in VD.

For the four tags described above there is also the possibility to define custom options based on the type of driver and how it is used. These options can be used to configure anything such as IP address and port number that should be used to establish a connection or some other device specific settings.

The `<Transformer>` tag can be used to supply the VD instance with Java classes for intermediate transformation of data before being published to the ESB or sent to another driver. In the `<Transformer>` elements the class paths are just listed and then to use the transformations the datapoints must be configured to do so. Listing 3.4 that shows a datapoint configuration contains a transformation. The alarm events that the datapoint emits are being sent to the ESB but before that happens they are being transformed by a transformation called AlarmEventTransformation.

3.3 Message Format

Any data that is to be published on the ESB is encapsulated in an XML message. The messages must follow a certain structure that has been defined by Volvo Cars. The root element states what type of message it is, for example an event notification or a write request. Every message must include a header tag that contains a unique message id and a timestamp. Besides the header and the root element the message structure depends on the specific message type but most messages has a data tag that contains an XML representations of the data objects being sent. These XML representations can be built using a number of tags that represent different data types that are supported by the VD framework. These data types can be primitive ones such as booleans or integers but also complex ones such as structs and arrays. These complex data types give the possibility of nesting data. A list of supported data types can be seen in table 3.1.

Table 3.1: List of supported data types in the virtual device framework.

Data type	**Description**
Array	Homogeneous list of supported data types
BitString	String of bits
Boolean	Boolean
Integer	Long integer
OctetString	Byte array
Real	Double precision floating point number
Struct	Heterogeneous list of supported data types
UTF8String	String of UTF8 characters

3.4 Comparing VD and LISA

The VD framework is built to provide an infrastructure that operates in a service oriented manner that also supports event based messaging. This means that Volvo Cars architecture shares many similarities with LISA, described in detail in section 2.2.

The VD drivers provides an adapter for communicating with different device types. This corresponds to the communication endpoints in LISA. The service endpoints of LISA do not have a direct equivalent in VD although there are similarities since it is possible to perform simple transformations of data in the VD instances.

The LISA message format do not specify which language to use for the messages but only what structure they should have. The messages should contain key-value pairs that makes it possible to build hierarchical structures and there should be an id and a timestamp included. Volvo Cars has chosen to use XML for their messages which supports this type of key-value paired structure. In addition Volvo Cars has defined a more specific schema to follow in order to keep a common data format for the organisation.

A central part of LISA is the message bus. It connects all applications and is responsible for sending the messages. As mentioned above the VD drivers and instances correspond to endpoints and are thus responsible for the communication with shop floor devices. To connect different VD instances and services that utilise data from the devices Volvo Cars uses IBM Websphere MQ as their ESB.

4 Developed Integration and Services

This chapter presents an integration between the open-source distributed streaming platform Apache Kafka [30] and the Virtual Device framework. Two examples of how the developed integration can be used are also presented. The first example takes low-level events from different sources, attaches additional details to the event messages and creates a combined event log. The second monitors an automatic maintenance process of spot welding electrodes called tip dressing. The integration is developed in the form of a device driver for Virtual Device and software components that makes it possible to reuse previously developed Volvo Cars services. An introduction to Kafka is first given that describes how the platform works and what its core components are before the developed Kafka device driver is presented.

4.1 Integration of Apache Kafka into Volvo Cars

The VD framework developed by Volvo Cars provides possibilities to collect data from different device types and to transform it to a standardised format. To make use of the data it is distributed to different services via an ESB. The already implemented IBM Websphere MQ takes care of some of the information and has some services but it has been very difficult for the engineers to make changes to what data is sent and to add, remove or make changes to the services. Many of the existing service are also very large monolithic applications that only consume data from the ESB without publishing any results back for further processing. They are hence not built according to the event-driven approach that provides flexibility to the system. This has resulted in the current architecture not being utilised as well as it could be. When new data is needed today the typical method of collecting it has been through PtP connections. To remediate this problem an integration between the open-source platform Apache Kafka and the VD framework has been implemented. Kafka is a platform that is easy to scale up to handle large amount of data[30]. It has the ability to scale the processing by dividing the work load to several services. Unlike a message queue where message are gone when they are consumed the messages published on Kafka can be consumed by many services. This means that it is easy to work in an event-driven approach where all services that are interested in a message can process that message.

The integration of Kafka will provide a common platform for collection and processing of data in an event-driven manner. It will be easier to access and manage than previously. The Kafka implementation will not replace the existing Websphere MQ but will instead be available to use in parallel as shown in figure figure 4.1.

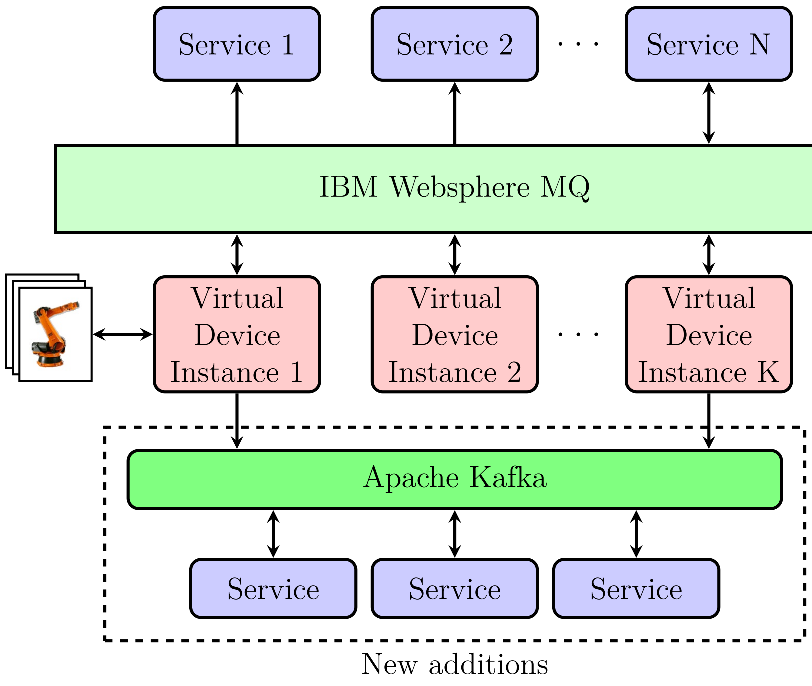


Figure 4.1: Overview of the integration of Apache Kafka in parallel to the existing platform at Volvo Cars.

4.1.1 Apache Kafka - A distributed Streaming Platform

Apache Kafka is a distributed streaming platform where applications can publish and subscribe on data[30]. The data can be stored in a fault-tolerant way and be processed in streams as they occur[30]. This makes it possible for applications to communicate with systems in real-time and transform or react to the data being sent. An overview of the internal components of Kafka is shown in figure 4.2. These components will be described here.

The Apache Kafka platform is run on a cluster of one or more brokers running on one or more servers. The data is stored in records that consists of a key, value and timestamp and streams of records are divided into categories called topics[30].

The data in each topic is stored in the cluster as an immutable log that can be divided to several partitions[31]. Each partition stores records in an ordered sequence where every record has an offset to uniquely be identified in a partition. All records are appended last in the partition when written to the topic. Unlike in a message queue, such as Websphere MQ, where the brokers have logic to keep track of which record should be sent to which consumer Kafka leaves this part to the consumers themselves. This means that the brokers do not get much more work to do when the number of consumers grow. The records are stored in the cluster persistently even if they are read by an application and are only removed if the retention policy says so[31, 30]. The retention policy describes for how long time a record is stored or how much the total memory of the partition can be before the oldest records are discarded. If both a time and a max size are defined a record will be discarded when the first of the policies triggers. It is possible to have a retention policy that stores the records permanently on the topic[32].

Because of the persistently stored immutable log, the possibility to share the load of processing the messages using partitions and the fact that the brokers do not need to keep track of which messages a consumer has process makes the Kafka platform scalable, efficient and suitable for an event-driven approach.

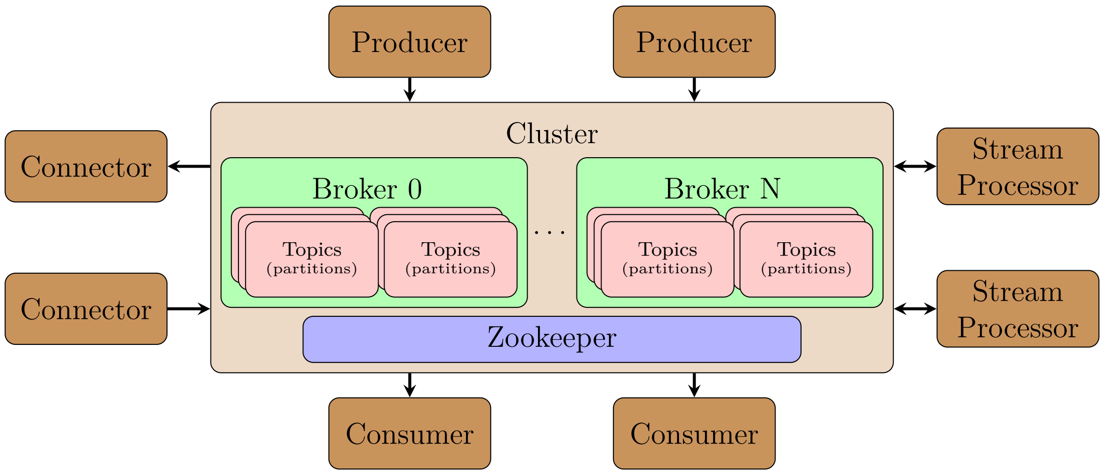


Figure 4.2: An overview of the Apache Kafka cluster and clients connected to it. The cluster can consist of several brokers that each can have several topics that clients can publish and subscribe to. Producers can publish messages to the cluster, consumers subscribe to messages, connectors can connect topics to other systems and stream processors operates on every message in a stream. Zookeeper is used to store and manage information about the cluster.

Storing the records in a topic in different partitions makes it possible to store data on different hard drives and therefore possible to have a topic store an infinite amount of records. Each topic needs to fit on one hard drive but you can always add a new partition on a new drive. The partitions also makes it possible to read from the same topic in parallel[30].

There are four core application programming interfaces (APIs) that are used to create applications to run on the cluster[30]. The Producer API is used to publish records to one or more topics. The consumer API is used to subscribe to one or more topics and process the records that are published to them. The Streams API is used to create stream processing applications that takes streams from one or more topics and perform some operations and then produces output streams to one or more topics. The Connector API is used to create reusable producers or consumers that connects topics to already existing applications or data systems, like databases.

To keep track of statuses of brokers or detect when brokers are added to or taken from the cluster Apache Zookeeper[33] is used[31]. Zookeeper is also used to, amongst other things, to store information about consumers like: which consumer group it belongs to, which topics it subscribes to and the last offset that it has processed the data on[31].

For this project the producer and the consumer are the APIs that will be used.

Producer Applications

A Producer application is an application that contains and uses a KafkaProducer object from the Producer API. This application can write data to the cluster by creating a record containing key and value and then send it to one or more topics[34].

To create a KafkaProducer properties can/needs to be passed when creating the object. There are three properties that are mandatory and the rest that are available have default values. The mandatory properties are: a list with one or more address/port pairs to brokers for initial connection to cluster; a serialiser class for the key of the record; and a serialiser class for the value of the record.

Consumer Applications

A Consumer application implements the Consumer API and creates a KafkaConsumer object. This object can subscribe on one or more topics and get topics from them by running the *poll*() method. The API keeps track of the offset you where last polling so each time the *poll*() method is called the new messages are being fetched. It is possible to manually set the offset to read from to anywhere in the partition. A consumer can be set to belong to a consumer group. Consumers in the same group share the load of reading the records of a topic[30].

Much like the Kafkaproducer, in section 4.1.1, the KafkaConsumer demands three mandatory options: a list with one or more address/port pairs to brokers for initial connection to cluster; a deserialiser class for the key of the record; and a deserialiser class for the value of the record. There are other properties that are avaliable to set but that otherwise have default values set[34].

4.1.2 Virtual Device Driver for Apache Kafka

In order for the services on the Apache Kafka cluster to access and process real data from the factory an integration with the existing systems is needed. This section describes an integration between Kafka and the VD framework described in chapter 3.

A device driver for VD has been developed that acts as a bridge between the factory and Apache Kafka by simply forwarding the desired messages. The device driver is built according to the structure described in section 3.1 with one driver, one context and the possibility to have multiple channels. From Kafkas point of view the device driver will be seen as a standard producer, see section 4.1.1, that just publishes messages on different topics. This is done by using the Producer API of Kafka to create a Producer that lives in the channel level of the device driver. In case that there are multiple Kafka clusters, for example one cluster for testing and one that is running in production, more channels can be created with new Producers. The context level is not used in this device driver and in the driver level no specific actions are needed since all that need to be configured are done in the channel level.

The datapoints in configuration are used for separating different types of data. One example of a separation can be that data from robots are sent on one datapoint and data from PLCs are sent on a different datapoint. Another example is separating data by location so that data from different parts of the factory are sent on different datapoints. For the Producer this means sending messages to the Kafka cluster on different topics. Each datapoint is tied to a topic specified in the configuration file of the device driver. Data is sent to Kafka when a write request is made by VD on one of these datapoints and the write function in the channel is executed. To choose what data should be sent to certain topics the user configures a subscription to events from datapoints of other device drivers. Figure 4.3 shows an example of this where a robot can send alarm messages. The alarm message has a corresponding datapoint in the robot device driver. When an alarm is triggered the datapoint corresponding to that alarm emits an event notification and a write is invoked on all other datapoints that are subscribed to that event. In this example there is just one

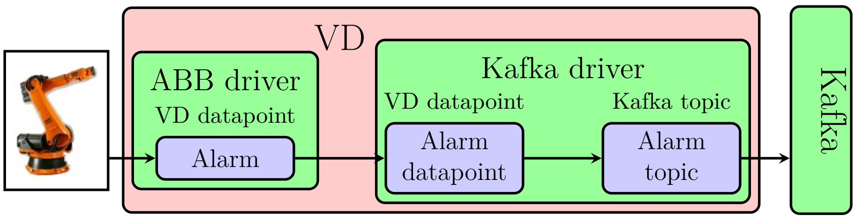


Figure 4.3: Flow of an event triggered in robot to the Kafka cluster. When an event is triggered in the robot the ABB driver emitts an event message from the alarm datapoint which a datapoint in the Kafka driver subscribes to. When this happens the Kafka driver writes to the Alarm topic and the message is published on the Kafka cluster.

datapoint in the Kafka driver that subscribe to the alarm datapoint of the robot. Since a write will be invoked on the Kafka datapoint when an alarm is emitted the Producer of the Kafka driver will publish the alarm message on a Kafka topic.

In short, it is possible to enable forwarding of data from one datapoint to another through a subscription. An overview can be seen in figure 4.4 of the general case where datapoints in other device drivers are connected to datapoints in the Kafka driver that have a one-to-one relationship with topics on the Kafka cluster.

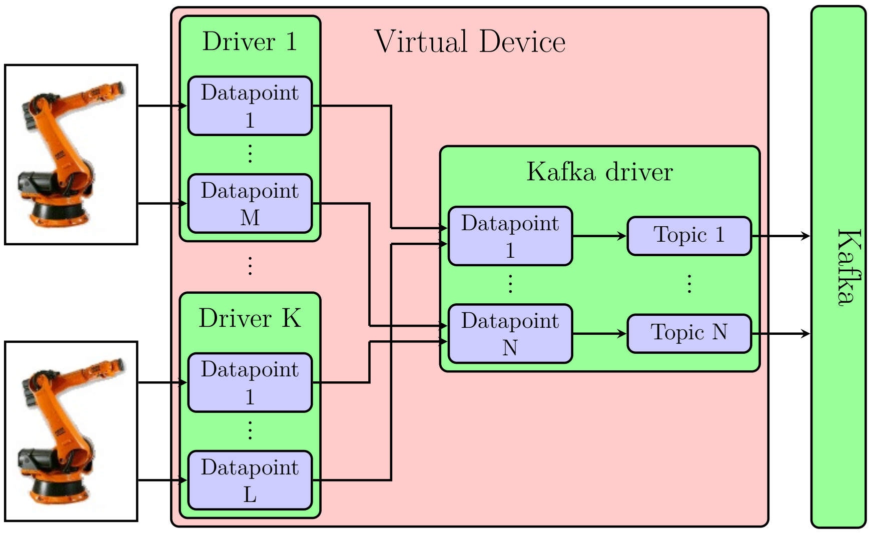


Figure 4.4: Overview of how data flows from equipment, through VD and finally to the Apache Kafka cluster. One datapoint in the Kafka driver can subscribe to several datapoints in different other drivers. This makes it possible to publish events from different devices to the same topic.

Since VD wants to know if the connections between the channels and the devices that are up and running the messages are published in a synchronous manner to Kafka that checks that it was successfully sent. To be able to know if there is a connection when no messages are sent a thread that sends a heartbeat, to a for this purpose dedicated topic, is created when the device driver is initialised. This thread sends a message to Kafka periodically and checks if it were published with success and on failure it aborts the channel.

4.1.3 Adapters for previously developed services

To be able to use all the already created services and tools that exists at Volvo Cars an integration for the Apache Kafka cluster is needed. This implementation should be able to fetch records on the cluster and forward them to a Volvo Cars service and also be able to take messages from the services and publish them to the cluster.

Volvo Cars has a way of building their services using dependency injection with

Spring framework[35]. With dependency injection it is possible to separate objects from other objects that depend on them by using an assembler that creates and injects objects where they are needed[36]. In this way classes can be written separately and be sent and used in many different applications. With Spring the dependencies are defined in a configuration file, an XML file, where the properties of the objects also are set.

Volvo Cars has several Java interfaces that structures the way they build applications, such as `Sink`, `MessageReceiver` and `MsgFlowApplication`, see appendix A. The `Sink` implementations are tools that can receive messages. The `MessageReceiver` implementations fetches messages from some source and puts them in a `Sink` and the `MsgFlowApplication` implementations operates on the messages they gets and sends them to the next application in the message flow. The `MsgFlowApplications` usually are the place were the service specific calculations are done while the `Sinks` and `MessageReceivers` are used to fetch and send the messages. There are also bridges between interfaces to be able to communicate between different types of objects. `Sink2MsgFlowApp` is a `Sink` implementation that receives messsages and forwards them to a `MsgFlowApplication` and `MsgFlowApp2Sink` is a `MsgFlowApplication` that puts messages from a message flow into a `Sink`. Figure 4.5 shows the parts in an example of a service that takes messages and process them sequentially with several `MsgFlowApplication`.

These tools are objects created that can perform different operations. The first tool in the flow will create a thread that gather data and performs the operations in that order they are defined. When all operations are performed the thread will return and get the next data and performs the operations again.

With this approach a variety of applications that operates or analyses messages can be created and used in different combinations of tools in different applications. If data needs to be gathered from a new environment it is only necessary to create a `MessageReceiver` that can communicate with this environment and puts it in to a `Sink`. Then all the other applications can be used regardless of how the source operates. The same applies if data needs to be published to a new environment, just implement a `Sink` that sends data according to this specific environment.

Due to this structure the only things needed to use the existing tools at Volvo Cars is adapters between Kafka and the existing tools.

To fetch records from one or more topics on the cluster a `MessageReceiver` is created. In this `MessageReceiver` a thread, that creates a consumer, is created that loops in a while loop and polls records from the cluster. It takes the first record and call the write function of the `Sink`. This `Sink` can be, for example, a `Sink2MsgFlowApp` that forwards the record to a message flow where it can be processed. When the thread performed all the operations defined by the Spring configuration the thread takes the next record and repeats the process.

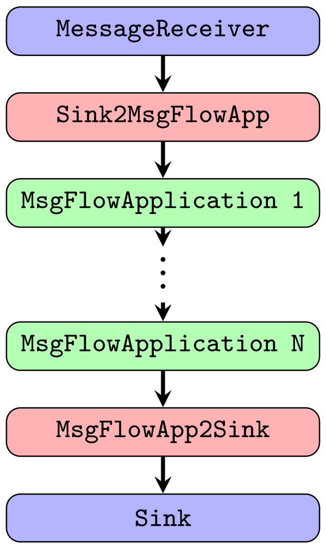


Figure 4.5: An example of the flow in a service built with Spring. The MessageReceiver and Sink fetches and sends data to and from the service. Sink2MsgFlowApp and MsgFlowfApp2Sink connects the sink interface to MSgFlowApplication interface and vice versa. The MsgFlowApplications operates on the messages.

A producer is created in an application that implements the `Sink` interface. When a message is written to this sink the producer publish it to one or more topics. Which topics the consumer and producer should read from and write to is set as properties for these objects in the Spring configuration.

One examples of a service that is built this way is a database append service that takes events and depending on what type of event it is puts it in the correct database. Another example is a alarm resolve service that takes alarm events, reads the alarm code and look up what type of alarm it is and possible sends an event for an action that needs to be done such as sending a SMS to maintenance staff.

4.1.4 Testing the Developed Integration

To test the created `MessageReceiver`, `Sink` and VD driver for Apache Kafka, described in sections 4.1.2 and 4.1.3, a VD instance is started on a computer. In the configuration file two device drivers are defined and set up. One is the Kafka driver that should be tested and the other is a driver for a service that send information over socket. This service is used to send messages to a datapoint in the socket driver that are forwarded to a datapoint in the Kafka driver. On the computer a Kafka broker is started and two topics are created. The messages from the VD driver are published on the first topic.

A test service is built, with spring dependencies and some of the existing tools from Volvo Cars. This service first has the `MessageReceiver` implemented for Kafka that consumes records from the first topic. Then a `Sink2MsgFlowApp` forwards the message to a dummy `MsgFlowApplication` that only appends a string to the message and sends it to a `MsgFlowApp2Sink` that bridges to the `Sink` implemented for Kafka. This publishes the message to the second topic on the Kafka cluster.

When downloading Kafka a so called *consoleConsumer* is included amongst the files. It is a simple consumer for testing that only print all records that are published on a topic. Two instances of consoleConsumer, one for each of the topics, are also running on the computer for monitoring the messages.

With this setup the functionality of the implemented `MessageReceiver`, `Sink` and VD driver are tested with successful results. The data flows from a source, the Socket service, through VD to the Kafka cluster and can be accessed by a service, using the `MessageReceiver` tool, that adds a string to the message and sends it back to the Kafka cluster with the `Sink` tool.

Since it is only to configure in the device driver configuration what data that should be sent to a datapoint the service for sending messages over socket to VD represents any data being sent to the Kafka driver. The test service is built like the example in figure 4.5 but with only one `MsgFlowApplication` but the implemented `MessageReceiver` and `Sink` are used as it should be for all services using this structure.

4.2 Creating Valuable Information from Low Level data

Most of the devices in the factory can send low level events before and during the execution of an activity. These events can describe for example a robot reaching a certain physical position or changing the program pointer position, a sensor reading an RFID tag or a PLC sending a program number to a robot. These types of events describe on a detailed level what each device is doing but individually they do not provide an overall context of what is happening. To get knowledge of the behaviour of a system and be able to perform various types of analyses it is therefore valuable to enrich the data with more information and to combine streams of events from different sources to get the overall behaviour of larger parts of the system.

The services presented here collects event data from both robots and a PLC and enriches these with additional information about the tasks being performed. The enriched events will then be appended to a database where it will be possible to see very detailed information about the behaviour of a full station in the factory. This information can for example be useful for manual analysis of cycle times or finding the cause of failures. An overview of the event flow of the services can be seen in figure 4.6.

4.2.1 Data Collection

The data being collected from robots consists of the program pointer, the alarms and a signal indicating if the robot enters or leaves the home position. The program pointer event data contains information about what module and routine is being executed by a robot together with a row and a column number that corresponds to a specific instruction within the currently executing routine. The alarms are events that are emitted when something goes wrong in the robot or work station. The home position event contains a value of the new state with the value `1` when the robot reaches the home position and `0` as soon as it leaves. An example of a home position event can be seen in listing 4.1.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<EventNotification dataPointID="vcta.1739070.r01.r8245.homeposition">
  <Header>
    <Timestamp>2018-05-16 19:04:12.948</Timestamp>
      <UniqueID>69e8c694-0a78-2668-1eab-3412366814f8</UniqueID>
  </Header>
  <Data>
    <Struct id="signalChanged">
      <UTF8String id="controllerName">17-39-070R01_R8245</UTF8String>
      <Struct id="newSignalState">
        <Real id="value">1.0</Real>
        <Boolean id="simulated">false</Boolean>
        <Struct id="quality">
          <Integer id="value__">1</Integer>
        </Struct>
      </Struct>
    </Struct>
  </Data>
</EventNotification>
```

Listing 4.1: Event for changed value of the homeposition variable in a robot.

All robot events are being sent from the robots to separately configured datapoints in a VD driver for ABB robots [37]. A subscription to these event notifications should be configured for a datapoint in the developed Kafka VD driver. Listing 4.2 shows a configuration of this type of subscription for the program pointer datapoint. As described in section 4.1.2 the datapoints in the Kafka driver corresponds to a topic on the Kafka cluster. This setup will publish events emitted by a robot to a topic on the Kafka cluster and become available to any service that needs them.

```xml
<DataPoint>
  <dp:ID>vcta.1739070.r04.r8248.programposition</dp:ID>
  <dp:ChannelID>r1739070r04_r8248</dp:ChannelID>
  <dp:Address>rapid.programPointer.T_ROB1</dp:Address>
  <dp:EventNotification>
    <dp:DataPointSubscriber>
      <dp:DataPointID>vcta.kafka.robot</dp:DataPointID>
    </dp:DataPointSubscriber>
  </dp:EventNotification>
</DataPoint>

<DataPoint>
  <dp:ID>vcta.kafka.robot</dp:ID>
  <dp:ChannelID>vcta.kafka</dp:ChannelID>
  <dp:Address>vcta.kafka.robot</dp:Address>
</DataPoint>
```

Listing 4.2: Configuration of two datapoints: vcta.1739070.r04.r8248 and vcta.kafka.robot. The first datapoint is configured to have the second as subscriber of its events.

The data from the PLC consists of the alarms and a set of values on variables describing things such as when different cycles start and stop, when a program is running on a robot, what product type is currently being worked on. All variables that should be published via VD are gathered in the same data block in the PLC. Most of the variables are simple booleans and when any of the variables change value an event is emitted which contains the entire data block, including the unchanged values. The events are only emitted once per scan cycle which means that multiple values can change but only one event is emitted. Because of this all of the changed values will receive the same time stamp even though they may in reality have occurred with a small time difference. The emitted events are, just like for the robots, sent to a configured datapoint in a VD driver for the PLC and should then forwarded to a datapoint in the Kafka driver before finally being published on a topic in the Kafka cluster. An overview of the data flow and services that will be described here can be seen in figure 4.6.

As described here collecting data from different sources and publishing it on the Kafka cluster can be done in a very similar manner for all equipment types.

4.2.2 Event Processing

The data contained in the PLC events includes, as described above, the entire data block of the values of all variables including changed and unchanged values. Because of this it is not obvious which variables has changed their value just by looking at the event data. Therefore, a small help service is needed for the raw PLC events that detects which variables have changed their values and emits a new filtered event that contain only those that have changed values. It should also be possible to configure the service to emit multiple events if there would be a need for an early separation between different values.

The final step of this application is to create a combined event log filled with detailed information about the actions being performed by a full production cell. Before that will be possible the collected data must be enriched with additional details that is not contained in the original events. The variables in the PLC data comes in the form of a memory offset and the value stored in that offset. It does not contain a description of what signal the variable describes and is hence quite uninformative unless that information is added. The robots program pointer has a similar problem. The program pointer events contain which routine is being executed but the only information about each individual instruction is a row and a column number that corresponds to the instruction that the program pointer is currently pointing at. An example of this is shown in listing 4.3 where some of the more important parts of a robot program pointer event can be seen. The struct "position" contains information that the routine D930Seg1 is being executed in module LD930R8234. The only information about the specific instruction is the "row" and "column" integers within the "begin" and "end" structs.

```xml
<Struct id="position">
  <UTF8String id="module">LD930R8234</UTF8String>
  <UTF8String id="routine">D934SchDefault</UTF8String>
  <Struct id="range">
    <Struct id="begin">
      <Integer id="column">6</Integer>
      <Integer id="row">130</Integer>
    </Struct>
    <Struct id="end">
      <Integer id="column">30</Integer>
      <Integer id="row">130</Integer>
    </Struct>
  </Struct>
</Struct>
```

Listing 4.3: Example of parts of a robot program pointer event before adding additional details.

This problem can be remediated by a service that subscribes to program pointer events and filtered PLC events and adds these types of additional information. Depending on the event type the service would perform a lookup operation in different databases to find the specific instruction for the program pointer or the informative description for the PLC values. When the additional information has been retrieved from the database a new element containing that detailed information is added to the data part of the event message before the event is published to Kafka again. This service performs tasks very similar to the Fill transformation of the LISA described in section 2.2. For the example in listing 4.3 the row and column integers would be used to find the name of the executed instruction which would be added to the existing event message to produce listing 4.4.

```xml
<Struct id="position">
  <UTF8String id="module">LD930R8234</UTF8String>
  <UTF8String id="routine">D934SchDefault</UTF8String>
  <UTF8String id="instruction">WaitSignal AllocateZone2</UTF8String>
  <Struct id="range">
    <Struct id="begin">
      <Integer id="column">6</Integer>
      <Integer id="row">130</Integer>
    </Struct>
    <Struct id="end">
      <Integer id="column">30</Integer>
      <Integer id="row">130</Integer>
    </Struct>
  </Struct>
</Struct>
```

Listing 4.4: Example of parts of a robot program pointer event after adding instruction name.

The enriched events contain detailed information about the actions of robots and PLCs that can be converted to highly valuable information through additional services. One of the simpler things to do with the events is to create a combined event log that shows the joint behavior of stations with multiple robots and PLCs. This can be done by gathering the different events in the same database. Tools for appending records to a database has been developed previously as it is a quite common thing to do. Using the Kafka `MessageReceiver`, described in section 4.1.3, the existing database appender can be used.

This combined event log can be of value for both engineers working with analysing and improving the process and maintenance workers trying to find the cause of errors. To make the data available on the shop floor and in the office there are existing tools that can be utilised. There is a factory intranet which has a web interface currently used for things such as showing a high level overview of the status of the factory and displaying different KPIs such number of units manufactured. This web interface could also be used to display the combined event log so that anyone can see details about a specific station. It would be possible to display updates of the log in real time but also to look at historical data. For the shop floor there are human-machine interfaces (HMIs) that currently can show for example information about an alarm that occurred. These can be used to display the same information as the web interface.

4.2.3 Other Use Cases for Enriched Events

There are many other applications for the detailed events apart from the simple combined event log described above. The events can be used to create formal mathematical models of the system which then can be used for many different types of analyses, optimisations or for control. One way to model a system is to use define tasks as *operations* and combine multiple operations into sequences of operations (SOPs) [19, 20]. An operation can be specified by a state model containing three

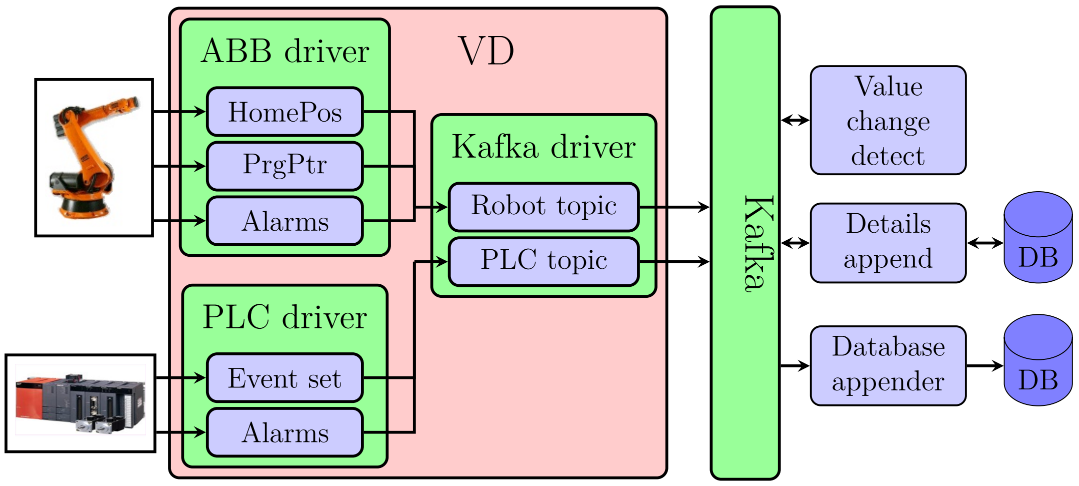


Figure 4.6: An overview of the event flow through VD to services on the Kafka cluster.

states: *initial*, *executing* and *finished*. The transitions between these states are called start and stop events. The start event is enabled when a *precondition* is fulfilled and similarly the stop event is enabled when a *postcondition* is fulfilled [19, 20]. Multiple operations can be combined to form SOPs.

These operations and SOPs can be translated to other types of mathematical models such as automaton, Petri net and extended finite automaton (EFA) [19, 20, 38]. These models typically describe discrete event systems however they can be extended to include continuous dynamics. One such model is the *hybrid Petri net including explicit differential equations* which has successfully been used for energy optimisation of robot stations [39, 38].

There are many potential use cases for these types of mathematical models of a system. The tool Sequence Planner, currently being developed at Chalmers university of technology [19, 20], has an implementation of the above mentioned energy optimisation. Sequence Planner collects hybrid operations from a message bus and performs optimisation on the robot trajectories and the SOPs [39, 38]. Other potential use cases include for example cycle time analysis or root cause analysis. The PLC events include multiple signals that for example mark the start and end of work cycles and loading and unloading of a new car body in the station. These signals are designed to be used for measuring cycle time and can thus easily be used for this purpose. The robot events can give a more detailed cycle time analysis by measuring the time between the different program pointer events. They can also be used for analysing how efficiently the robots are operating. The instruction names included in the detailed program pointer events can describe instructions that are dedicated to waiting for things such as permission to enter a shared zone. The time it takes to get past these wait instructions can be calculated and then analysed to see if the robots could do their work differently to reduce wait times or in other ways work more efficiently. One simple way to perform such analysis would be to create a Gantt chart from the sequences of events. This would provide a visual overview of the bahavior of the system and it would be easy to see when robots are waiting. These Gantt charts would also be useful in root-cause analysis as they could be used to look back in time and see precisely what happened.

4.2.4 Testing the Combined Event Log Services

The setup described in section 4.2 consists of multiple services that receive event messages and perform small tasks before re-publishing the events to the Kafka cluster. This is the way services should interact and process data according to LISA and it is also the way VD was developed to be used. However, to test the services on real data a different setup had to be used. The test setup collects data from PLCs and robots just as described in section 4.2.1 but instead of forwarding the events to the Kafka cluster through the Kafka VD driver all of the processing steps are done in a sequence internally in VD. The components that are put together in a sequence can however be deployed as individual services on the Kafka cluster using the in- and outbound components presented in section 4.1.3. If that would be done the setup from section 4.2 would be obtained.

A sample from the combined event log can be seen in table 4.1. The data in the table was created by running the collection and enrichment of event messages described above and in section 4.2. The sample shows parts of the execution of one work cycle of a full station consisting of four robots and one PLC. It can be seen in the top part of the sample that the PLC sends a number of events corresponding to a car body entering the station and events stating that the process is started and that the robots are running some programs. This is followed by robot events showing that the robots are leaving their home positions and are initiating their respective tasks. In reality there are over 100 robot events describing in detail every instruction executed by the robots but just a few are included in table 4.1. The robots keep executing instructions and when they are done they return to their respective home positions and PLC receives an event stating that no program is running on the robot. When all robots have finished executing their tasks a final event is sent by the PLC saying that the process is complete.

4.3 Tip Dressing Services

One of the most used processes in the body shop at Volvo Cars is spot welding. There are approximately 750 spot welding robots in the factory. Spot welding is used to join two sheets of metal and this is done by pressing the metal together with a clamping tool that has two electrodes, also called tips where one is called the fixed electrode and the other one is called the moving electrode. An electric current is then run through the electrodes until the point on the metal sheets that are pressed together by the electrodes have melted. This will join the two pieces in a small

Table 4.1: Sample of a combined event log containing event from robots and a PLC.

LogTime	Alias	Workcell	Event	State	Description
2018-05-24 12:59:43.495	pl173931	1739010	CTBU	0	Unload
2018-05-24 12:59:44.888	pl173931	1739010	CTBT	0	Transport
2018-05-24 12:59:44.888	pl173931	1739010	CTRS	0	Ready To Start
2018-05-24 12:59:44.888	pl173931	1739010	CTSP	1	Process
2018-05-24 12:59:47.052	pl173931	1739010	CTProdR02	1	Production PrgNbr Running
2018-05-24 12:59:47.052	pl173931	1739010	CTProdR03	1	Production PrgNbr Running
2018-05-24 12:59:47.115	pl173931	1739010	CTProdR01	1	Production PrgNbr Running
2018-05-24 12:59:47.115	pl173931	1739010	CTProdR04	1	Production PrgNbr Running
2018-05-24 12:59:47.130	r8232	1739010		0	IO@homeposition
2018-05-24 12:59:47.232	r8232	1739010	WaitSignal AllocateStation		LB944R8232@B944SchDefault
2018-05-24 12:59:47.238	r8234	1739010		0	IO@homeposition
2018-05-24 12:59:47.238	r8234	1739010	WaitSignal AllocateStation		LB944R8234@B944SchDefault
⋮	⋮	⋮	⋮	⋮	⋮
2018-05-24 13:00:36.708	pl173931	1739010	CTProdR01	0	Production PrgNbr Running
2018-05-24 13:00:36.879	r8231	1739010		1	IO@homeposition
2018-05-24 13:00:36.879	r8231	1739010	ExecEngine		R8231@main
⋮	⋮	⋮	⋮	⋮	⋮
2018-05-24 13:00:38.996	pl173931	1739010	CTProdR04	0	Production PrgNbr Running
2018-05-24 13:00:38.997	r8234	1739010		1	IO@homeposition
⋮	⋮	⋮	⋮	⋮	⋮
2018-05-24 13:00:42.186	pl173931	1739010	CTProdR02	0	Production PrgNbr Running
2018-05-24 13:00:42.198	r8232	1739010		1	IO@homeposition
⋮	⋮	⋮	⋮	⋮	⋮
2018-05-24 13:00:43.810	r8233	1739010		1	IO@homeposition
2018-05-24 13:00:43.819	pl173931	1739010	CTProdR03	0	Production PrgNbr Running
2018-05-24 13:00:43.911	r8233	1739010	ExecEngine		R8233@main
2018-05-24 13:00:44.208	pl173931	1739010	CTSP	0	Process

spot, hence the name spot welding. To ensure that the welds keep good quality the electrodes on the tool must be cut once in a while to remove wear and impurities on the surface and to restore the geometric shape. This is done automatically by a maintenance process called tip dressing. Tip dressing is performed by pushing the electrodes into a rotating cutter where the pressure, the rotational speed of the cutter and the time is set as input parameters. The result of the cut is measured to see if enough material has been removed and a resistance test is made to see if the electrodes have the correct conductivity. If either of these tests fail the tip dress process can be run again until the tests pass or the robot can move to a service position where it can receive maintenance. The retry can either be initiated manually by an operator or automatically by the equipment if it has been configured to do so.

4.3.1 Tip Dress Measurement Data

Data from the tip dressing equipment is available through a configured datapoint in a VD driver for ABB robots [37]. When the tip dressing has been performed the measurements are recorded internally in the robot and an event is emitted on the configured datapoint. The measurements recorded in the robot contains a number of different values such as measurements of the amount of material that has been removed from the electrodes since they were changed, counters of number of dresses

Table 4.2: Part of the data from a tip measurement event. These are the events that are relevant for the tip dress services.

Data type	**Name**
Real	WearFixed
Real	OldWearFixed
Real	WearMove
Real	OldWearMove
Real	MillLengthFix
Real	MillLengthMove
Real	TipWornOut

since the last electrode change and number of welds since the last dress. There are also boolean values stating if the resistance test and length measurements are successful. In addition there a number of values corresponding to input parameters to the tip dressing process. Unfortunately not all of these values are included in the event message that is sent to the datapoint in VD. The counter values and boolean values are currently not included as well as the input parameters. However, many of the values are not interesting for this application but some of the more important ones that are available can be seen in table 4.2.

The first four values in table 4.2, named WearFixed, OldWearFixed, WearMove and OldWearMove describe the new and previous measurements of the amount of material removed from the electrodes. The two values named MillLengthFix and MillLengthMove are the original lengths of the electrodes and TipWornOut describe the length at which the electrodes are worn out and need to be replaced. With these values it is possible to make a prediction of when the electrodes need to be changed and to detect when the process is not removing enough material.

4.3.2 Overview of Tip Dressing Services

Two services are here proposed concerning tip dressing. The first should predict when the electrodes will need to be changed and the second should detect if the process is not performing well. These services should utilise the VD driver for Apache Kafka developed in this thesis together with a previously developed VD driver for ABB robots [37] used to extract the tip dress measurement data. An overview of the data flow for the two services can be seen in figure 4.7.

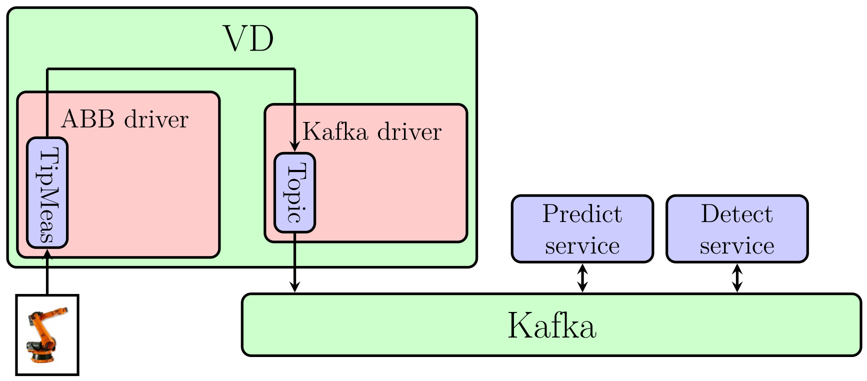


Figure 4.7: Data flow for tip measurement data from a robot through VD to the Apache Kafka cluster.

As illustrated in figure 4.7 the tip dress measurements are sent as events from the robot to a datapoint in a VD driver. A subscription to these events are then configured for the Kafka driver. The events are then forwarded from the datapoint in the ABB driver to a datapoint in the Kafka driver which corresponds to publishing on a topic on the Kafka cluster. Both the prediction service and the detection service then subscribe to messages on this topic.

4.3.3 Predicting Wear on Electrodes in Spot Welding

The tip dressing process automates some of the maintenance of spot welding robots but there are still parts that need to be performed manually such as changing the electrodes when they are worn out. Changing the electrodes requires a worker to enter the robot cell which means that production has to stop for a short amount of time. It is therefore desirable to schedule these changes to occur on breaks or when the cell has a low work load. To be able to schedule this manual maintenance it is necessary to predict when the electrodes will become worn out and in need of replacement.

The process is meant to remove the same amount of material every time until the electrode needs to be changed. The ideal behavior is that the wear of both electrodes follow the same straight line until a maximum wear threshold has been reached. This threshold is set by the TipWornOut value that can be seen in the event data. At this point the electrodes are changed and the wear is set to zero again. An example of real measurements can be seen in figure 4.8 where the wear of the two electrodes is shown on the y-axis and the x-axis shows the number of dresses that have been made since the electrodes were changed. As the figure shows, the ideal behavior is not always followed. Figure 4.8 was created with data from an old log where the counter for number of dresses where available but as mentioned in section 4.3.1 this is currently not available in the data that is sent to the datapoint in VD.

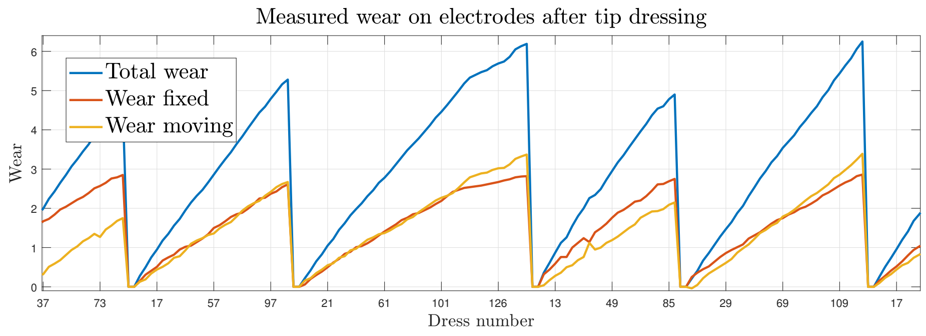


Figure 4.8: A plot of the wear of the individual electrodes and the total wear of both added together. When the wear reach high enough the electrodes are changed, which can be seen by the sudden drops down to zero.

Since the focus of this thesis is not to evaluate advanced algorithms for analysing data the method suggested here for predicting when the electrodes will become worn out is very simple. The following description is for a general electrode but should be applied on both the fixed and moving electrode. The amount of material removed in a single tip dress operation is calculated by taking the difference between Wear and OldWear. This difference can be called DW as seen in equation (4.1).

$$DW = Wear - OldWear \tag{4.1}$$

Based on what is known about the tip dress procedure these differences are expected to remain at a constant value across every new measurement. There can, of course, be some variations over time but locally the values are expected to stay approximately the same. A moving average is therefore used on the calculations of DW according to equation (4.2). Here k is the current time step and n is the window size for the moving average.

$$\overline{DW} = \frac{1}{n}\sum_{i=0}^{n-1} DW_{k-i} \tag{4.2}$$

This average gives a rate that describes how fast the tip will become worn out. Using this rate along with MillLength, Wear and TipWornOut one can calculate an estimation of how many more dresses can be made before the tip length reaches the threshold set by TipWornOut. This can be described by equation (4.3) where N_{dress} is the estimated number of dresses left.

$$N_{dress} = \frac{MillLength - Wear - TipWornOut}{\overline{DW}} \tag{4.3}$$

As mentioned this calculation gives an estimate of how many more dresses can be made. This number in itself is however not very helpful for the maintenance workers as it is hard to know how often the robot performs tip dressing. What would be helpful is to see how many car bodies can be welded before the tips need to be replaced. For this, additional data is however needed that keeps track of how many welds have been made between each tip dress and information about how many welds are made by a robot on each car. This data is currently not configured to be sent by the robot so unfortunately this can not been done as of today. If this would be done the service would make the calculations described above for both electrodes and emit an event containing the prediction of how many car bodies that can be welded before maintenance is needed. This event could then be converted to actual time if data of current manufacturing rate is included.

4.3.4 Detecting Bad Performance Trend of Tipdress Process

As mentioned in section 4.3 the tip dressing equipment can perform retries when it fails, either manually or automatically. This function exists as a quick solution to temporary problems that may resolve themselves just by running the process again. For example some chips of metal can get stuck either in the cutter or on the electrode. Then the chips may come loose the second time the tip dressing is run. When the process fails multiple times in a row the operators in the factory will get an alarm notification and hopefully send the robot to service.

A problem exists when the failures occur quite often but not so often that it becomes obvious to the operators. For example if a cutter has become dull the tip dresser may need to perform two dresses every time to get a satisfying result. If the equipment is configured to allow one retry then these failures will never be seen since the retry is made automatically. The process may then underperform for a very long time without anyone noticing until the cutter is completely broken and the failure becomes obvious. Another example is if there are a few alarms every other hour but ongoing for a whole day or multiple days each individual operator will not be able to see the pattern since they swap stations during their work day. Each operator will think it is a temporary problem when in reality there is a more severe problem hidden beneath the surface.

These types of problems can be detected quite easily just by looking at a plot of the measured wear, like the plot shown in figure 4.8. However, since there are over 750 spot welding robots manual analysis of such plots is not a feasible solution.

This service receives the same data as the prediction service but it is now interesting to compare the current behavior with the normal behavior. The normal behavior can be pre-calculated on historical data from the tip dress process. Each robot has an internally stored log of the latest 1000 tip dress measurements. This log contains entries with measurements from both good and bad performance. If all the entries with bad performance is removed from the log the mean and standard deviation for the difference between current wear and the old wear, DW, can be calculated. These two values would describe the normal behavior of the robots tip dress process. It is then possible to see if the calculated DW for new measurements deviate from this normal behavior. If the new DW is further away from the mean than a threshold it is considered to be a failure. If the failure frequency is too high, meaning that too many failures occur within a certain number of tip dresses, the service should emit an event message that contains information about which robot is not performing well. This event can then be picked up by a previously implemented service for sending text messages. A text message containing information about the problem would then be sent to a maintenance worker.

4.3.5 Testing the Tip Dress Services

To test the data flow from robot, through VD and finally out to the services on the Kafka cluster the same socket driver will be used as in section 4.1.4. This time it will be used with previously collected tip dress measurement events to simulate that they occur.

Proper testing of the accuracy of the two services has unfortunately not been made as that would be quite time consuming. However, whether these service perform well or not is of less importance for this thesis as the proposed services are meant to show that it is possible to use the Kafka driver in combination with VD to perform calculations for more advanced forms of analyses.

5 Summary and Discussion

This thesis has explored different approaches to integration between devices and software applications in a manufacturing environment. The broad concepts of service oriented architecture (SOA) and event-driven architecture (EDA) have been introduced as well as the Line Information System Architecture (LISA). Volvo Cars implementation of these architectures, called Virtual Device (VD), is designed to support both the service oriented and the event-driven approaches and has many similarities with LISA. The VD framework provides very flexible ways to connect to devices even though they may be using very different interfaces for communication. This enables flexible and efficient collection of data and is supposed to also provide easy access to the data through an enterprise service bus (ESB) that services can use to perform various types of analysis.

The currently implemented solution at Volvo Cars is however not easy to work with and thus VD is not reaching its full potential. It is a hard and slow process to deploy new services to analyse data from equipment in the factory. This thesis has provided an implementation of Apache Kafka as an additional platform to the existing IBM Websphere MQ where new services can be developed and tested without interfering with the system that is running in production. The additional platform is more accessible for engineers and properties of the Kafka platform are more suitable for an event-driven approach which gives a more flexible platform for analysing data from the factory.

Integration with Kafka has been developed as a device driver for VD so that Kafka operates as any other device in the system using VD as its communication endpoint towards other devices. Components for reusing previously developed services have also been developed. Two applications have been proposed to show how the integration can be used. One receives low level data from events emitted by PLCs and robots and attaches additional details to these events to create more valuable information that can be used in higher level processing. Some ways to use these events are for example in cycle time analysis, optimisation or root-cause analysis. The second application monitors the tip dress process by using data from measurements to predict when the electrodes need to be replaced and to detect when the process is not performing as well as it should.

5.1 Discussion and Future Work

When a new type of analysis is to be made the collection and processing of data in the factory is mainly done using PtP connections which is not very flexible or efficient. This is done even though the infrastructure for Industry 4.0 is for the most part already in place with the existing solution based on IBM Websphere MQ and VD. In this project it has been seen that VD can potentially be a highly valuable tool for utilising the large amounts of data from the factory. As mentioned, it is not easy to develop and test new services and publish or subscribe to new data using the existing solution. This problem is caused by the way the organisation is built but results in a very slow development of new services and VD not being fully utilised. There is a lot of experience and knowledge among the engineers who work with the equipment at Volvo Cars Torslanda and they have many great ideas about what data is relevant to collect and what analyses could be necessary to perform. What is lacking is knowledge about how to work with the infrastructure that is in place. What VD is, what it is good for or even that it exists is unfortunately not common knowledge at Volvo Cars Torslanda. A hope is that the platform using Kafka that has been developed will make it easier to try new things which in turn will help people to learn about VD and how services can be developed according to an event-driven approach.

Although the main part of the integration between Kafka and VD has been developed there remains some details to make the solution work in the real system. These details include things such as choosing hardware to use as Kafka brokers and writing configurations for VD instances. These configurations need to include options of how to connect to the Kafka cluster as well as which topics to use and what events should be published on these topics.

Before using the developed integration on a larger scale some tests on performance and reliability of the system need to be performed. This is needed to ensure that a failure in Kafka does not affect the manufacturing system in any way other than that the services working on data from the factory stops executing.

There are also many interesting services that could be developed using Kafka and VD. In this thesis no advanced algorithms were used to perform analysis, however given the amount of data that exists there is the possibility to use much more sophisticated methods for example from the fields of machine learning, data mining or complex event processing. More advanced methods are currently being explored for analysing the tip dressing process at Volvo Cars. It would also be beneficial to develop an endpoint for connecting Sequence Planner to the Kafka platform as this would help researchers test and further develop their tools using real data.

One aspect to consider when creating services is whether to use a batch or stream processing approach. If the service can perform calculations after the events happened or if it needs many consecutive events a batch processing approach may be suitable where the service only runs once a day or week and performs the calculation with the data gathered. On the other hand, if the data only is relevant for a short time or something needs to react to the results a stream processing service can be better. If the events comes with high frequency and the analysis needs to be as close to real time as possible it can be necessary to analyse if the service works fast enough and maybe use tools such as the Kafka Streams API that is built for high throughput and low latency.

As the number of services grow it will also be necessary to design how events should be organised and filtered in the system. In Kafka services can subscribe to topics, which is one level of filtering. Each event message also contains a key that can be used for further filtering. Another alternative is to look inside the body of the messages for keywords to filter on. What is needed is to define a common structure to use for filtering and routing of messages. Another important thing regarding the messages is that it can be very beneficial to have a defined schema of how messages should be structured and what they can contain. This will make the filtering of messages easier since each service will know what a message looks like and can have common methods for parsing the messages.

Bibliography

[1] S. Wang, J. Wan, D. Li, and C. Zhang, "Implementing smart factory of industrie 4.0: An outlook," *International Journal of Distributed Sensor Networks*, 2016. [Online]. Available: http://proxy.lib.chalmers.se/login?url=https://search-proquest-com.proxy.lib.chalmers.se/docview/1761401265?accountid=10041

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568

[3] S. K. Khaitan and J. D. McCalley, "Design techniques and applications of cyberphysical systems: A survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, June 2015.

[4] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," vol. 19, 04 2014.

[5] X. Xu, "From cloud computing to cloud manufacturing," *Robotics and Computer-Integrated Manufacturing*, vol. 28, no. 1, pp. 75 – 86, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0736584511000949

[6] Y. Liao, F. Deschamps, E. de Freitas Rocha Loures, and L. F. P. Ramos, "Past, present and future of industry 4.0 - a systematic literature review and research agenda proposal," *International Journal of Production Research*, vol. 55, no. 12, pp. 3609–3629, 2017. [Online]. Available: https://doi.org/10.1080/00207543.2017.1308576

[7] A. D. Maynard, "Navigating the fourth industrial revolution," *Nature Nanotechnology*, vol. 10, no. 12, pp. 1005–1006, 12 2015, copyright - Copyright Nature Publishing Group Dec 2015; Last updated - 2015-12-08. [Online]. Available: http://proxy.lib.chalmers.se/login?url=https://search-proquest-com.proxy.lib.chalmers.se/docview/1739096677?accountid=10041

[8] H. Kagermann, J. Helbig, A. Hellinger, and W. Wahlster, *Recommendations for Implementing the Strategic Initiative INDUSTRIE 4.0: Securing*

the Future of German Manufacturing Industry ; Final Report of the Industrie 4.0 Working Group. Forschungsunion, 2013. [Online]. Available: https://books.google.se/books?id=AsfOoAEACAAJ

[9] T. Dawson, "Industry 4.0 - opportunities and challenges for smart manufacturing," https://ihsmarkit.com/research-analysis/q13-industry-40-opportunities-and-challenges-for-smart-manufacturing.html, accessed: 2018-04-26.

[10] A. Boyd, D. Noller, P. Peters, D. Salkeld, T. Thomasma, C. Gifford, S. Pike, and A. Smith, *SOA in Manufacturing Guidebook.* MESA International, IBM Corporation and Capgemini Co-Branded White Paper, 2008.

[11] A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, and B. Lennartson, "An event-driven manufacturing information system architecture for industry 4.0," *International Journal of Production Research*, vol. 55, no. 5, pp. 1297–1311, 2017. [Online]. Available: https://doi.org/10.1080/00207543.2016.1201604

[12] D. S. Linthicum, *Next Generation Application Integration: From Simple Information to Web Services.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[13] L. Dikmans and R. van Luttikhuizen, *SOA Made Simple.* Packt Publishing, 2012.

[14] Z. A. Bukhsh, M. van Sinderen, and P. M. Singh, "Soa and eda: A comparative study: Similarities, differences and conceptual guidelines on their usage," in *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, vol. 02, July 2015, pp. 213–220.

[15] F. Jammes, H. Smit, J. L. M. Lastra, and I. M. Delamer, "Orchestration of service-oriented manufacturing processes," in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 1, Sept 2005, pp. 8 pp.–624.

[16] A. Bonham, "Microservices-when to react vs orchestrate," https://medium.com/capital-one-developers/microservices-when-to-react-vs-orchestrate-c6b18308a14c, 2017, accessed: 2018-03-23.

[17] B. M Michelson, "Event-driven architecture overview: Event-driven soa is just part of the eda story," January 2006.

[18] U. Dahan, "Eda: Soa through the looking glass," http://udidahan.com/2009/09/29/article-eda-soa-through-the-looking-glass/, 2009, accessed: 2018-03-23.

[19] B. Lennartson, K. Bengtsson, C. Yuan, K. Andersson, M. Fabian, P. Falkman, and K. Akesson, "Sequence planning for integrated product, process and automation design," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 4, pp. 791–802, Oct 2010.

[20] K. Bengtsson and B. Lennartson, "Flexible specification of operation behavior using multiple projections," *IEEE Transactions on Automation Science and Engineering*, vol. 11, no. 2, pp. 504–515, April 2014.

[21] "Extensible markup language (xml) 1.0 (fifth edition)," accessed: 2018-03-27. [Online]. Available: https://www.w3.org/TR/REC-xml/

[22] E. Harold, *XML: Extensible Markup Language.* Wiley, 1998. [Online]. Available: https://books.google.se/books?id=RVG_AQAACAAJ

[23] E. International, "The json data interchange format," "http://www.ecma-international.org/publications/standards/Ecma-404.htm"0, standard ECMA-404.

[24] "Introducing json," http://json.org/, accessed: 2018-03-23.

[25] C. Severance, "Discovering javascript object notation," *Computer*, vol. 45, no. 4, pp. 6–8, April 2012.

[26] "What is stream processing?" https://data-artisans.com/what-is-stream-processing, accessed: 2018-03-27.

[27] G. Vaseekaran, "Big data battle : Batch processing vs stream processing," https://medium.com/@gowthamy/big-data-battle-batch-processing-vs-stream-processing-5d94600d8103, accessed: 2018-03-27.

[28] S. Shahrivari, "Beyond batch processing: Towards real-time and streaming big data," *Computers*, vol. 3, no. 4, pp. 117–129, 2014. [Online]. Available: http://www.mdpi.com/2073-431X/3/4/117

[29] M. Garofalakis, J. Gehrke, and R. Rastogi, *Data Stream Management: Processing High-Speed Data Streams.* Springer Berlin Heidelberg, 2016.

[30] "Introduction," https://kafka.apache.org/intro, accessed: 2018-03-27.

[31] B. Konieczny, "Introduction to apache kafka," http://www.waitingforcode.com/apache-kafka/introduction-to-apache-kafka/read, Mar 2016, accessed: 2018-05-29.

[32] J. Kreps, "It's okay to store data in kafka," https://www.confluent.io/blog/

okay-store-data-apache-kafka/, Sep 2017, accessed: 2018-03-27.

[33] "Welcome to apache zookeeper™," https://zookeeper.apache.org/, accessed: 2018-05-29.

[34] "Documentation," http://kafka.apache.org/documentation/, accessed: 2018-05-29.

[35] "Spring: the source for modern java," https://spring.io/, accessed: 2018-05-16.

[36] M. Fowler, "Inversion of control containers and the dependency injection pattern," Jan 2004, accessed: 2018-05-16.

[37] D. Nord and H. Wahlqvist, "The tweeting robot - collection and processing of data from industrial robots," Master's thesis, 2016.

[38] B. Lennartson, K. Bengtsson, O. Wigström, and S. Riazi, "Modeling and optimization of hybrid systems for the tweeting factory," *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 1, pp. 191–205, Jan 2016.

[39] S. Riazi, K. Bengtsson, O. Wigström, E. Vidarsson, and B. Lennartson, "Energy optimization of multi-robot systems," in *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, Aug 2015, pp. 1345–1350.

A

Appendix 1

```
public interface MessageReceiver extends Channel, SinkReceiver {
}


public interface Channel {
  void open() throws IOException;

  void close() throws IOException;
}


public interface SinkReceiver {
  void setReceivingSink(Sink arg0);
}
```

Listing A.1: Code for the MessageReceiver, Channel and SinkReceiver interface. MessageReciever extends the interfaces Channel and SinkReciever.

```
public interface Sink {
  void write(byte[] arg0) throws IOException;

  void write(byte[] arg0, int arg1, int arg2) throws IOException;
}
```

Listing A.2: Code for the Sink interface.

```
public interface MsgFlowApplication {
  void handleMsg(Object arg0) throws MessageHandlingException;
}
```

Listing A.3: Code for the MsgFlowApplication interface.