University of Bergen
Department of informatics

Western Norway University of Applied Sciences
Department of Computing, Mathematics and Physics

Prototyping and Evaluation of an Event-Driven Microservice Architecture for Tunnel Control Systems

Authors: Simon Indrebø Halvorsen and Øystein Hegglid Follo

Supervisor: Lars Michael Kristensen

September, 2019

Abstract

TrafSys AS has a 20-year-old monolithic system, called VegVokteren, for monitoring and controlling facilities such as tunnels and mountain passes. The system has become hard to maintain and it is difficult to extend with new components and functionality. The performance of the system also decreases when the workload is high. Research has shown that there are many good reasons for modernizing legacy software using microservices. We present a software architecture and prototype based on event-driven microservices that utilize message-oriented middleware for communication. This thesis investigates how the architecture aids in solving the challenges of the current solution.

The contributions of this thesis are: (1) Implementation of a representative prototype based on the core functionality of VegVokteren; (2) Reseach on event-driven microservices and how the architecture can be applied to a control and monitoring system that handles sensitive data; (3) Experiments to test the scalability, fault-tolerance, extendability, and performance of the architecture presented in this thesis.

The results indicate that an event-driven microservice architecture enables scalability of individual microservices. The decoupled nature of the architecture also increases the extendability and fault-tolerance of the system. The results of the perfomance tests that were conducted indicate that the performance of the prototype decreases at high workloads, but it is reason to think that this is a result of that the tests were conducted on limited hardware. Overall, the experiments indicate that an event-driven microservice architecture could be a good choice of technology for VegVokteren.

Acknowledgements

Foremost, we would like to thank our supervisor Lars Michael Kristensen for valuable feedback throughout the writing of this thesis. It would never have come close to the what it is today without him.

We would also like to thank Sveinung Stensletten and Anders Sandven at Trafsys for a thorough introduction to current version of VegVokteren.

Lastly, we would like to thank the whole of Trafsys for welcomming us to work at their office, for helping us with problems, and for providing feedback along the way.

`Simon Indrebø Halvorsen and Øystein Hegglid Follo`
11 September, 2019

Contents

Chapter 1

Introduction

Software architecture is a cornerstone in every enterprise application. It serves as an abstraction to help manage the complexity of a system, and it provides a blueprint for how the system components will interact with each other. And as technology is rapidly improving, we still see more complex systems running in a variety of environments such as in the cloud or on dedicated servers. This means that a software architecture that fits the domain, and the development team, are becoming increasingly important in the initial phases the development lifecycle. Other enterprises often find that legacy systems are hard to improve upon, and harder to maintain as time goes on. This can have many causes, some of them being new functional requirements to the system, or that knowledge on the core technologies of the system are rearer to find among younger developers.

1.1 3-Tier and Microservice Architecture

Traditionally, applications have been built in tiers, and the most common is the so-called 3-tier architecture. To explain it in simple terms, a 3 tier application contains a client tier, a place where the users interact with the application; A tier with business functionality, for processing and retrieval of data from the database tier. Such applications are typically highly coupled, which means that the different components in the system are hard to separate without breaking the entire system. This is also a drawback to this class of architectural structure: if one component breaks, the error is likely to cascade throughout the system and cause a total system breakdown.

The term microservice was coined by Martin Fowler in 2014 [16], but the idea of a service-oriented application is much older. In 1978, Bell System Technical Journal [29] published Doug McIlroy's article about the Unix Philosophy. This article listed four general principles that have gained currency among the Unix developers at the time:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown program. Do not clutter output with extraneous information. Avoid stringently columnar or binary input formats. Do not insist on interactive input.
3. Design and build software, even operating systems, to be tested early, ideally within weeks. Do not hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

These are the basics upon which the Service Oriented Architecture (SOA), and later the Microservice Architecture has built, to become more modern and useful today. Of these two maxims, the first two are the most important for understanding what a microservice architecture is. To make it oversimplified, a microservice application is a set of independent programs that can run on the same, or different servers. They communicate through other channels than in-memory references, which is the main communication method of the 3-tier application. In the case of services being hosted on different servers they often communicate through HTTP requests or a message broker.

Both SOA and microservice architecture styles are explained more in-depth in Section 2.1. The important take away from this section is that a 3-application can conceptually be looked at as a brick, or a monolith, of functionality and code. Microservices, on the other hand, are loosely connected components that perform specialized tasks and can be run individually.

1.2 Trafsys and VegVokteren

This thesis has been undertaken in cooperation with Trafsys. Trafsys is a company localized in Bergen and are developing a system called VegVokteren (VV). VV is an alarm and surveillance system used by Statens Vegvesen at their control stations. It enables the operator to view live information from different facilities. This information includes sensor data, live video feeds, the status of actuators such as signs and lights. The system has been in development since the late 1990, and are still based on the same architecture as back then. In tests, the system has performed poorly during high load. This combined with the fact that developers are struggling to add new features or equipment to the system has led Trafsys to conduct research on alternatives to this legacy architecture.

Previous to this thesis, Trafsys had already decided to gradually transit to a microservice architecture. When we started our discussions with them, they had already implemented a message broker, using the framework Apache Kafka framework. The goal was to enable all communication between system components to go through one scalable system for asynchronous message passing and to make a system that is easier to change and to add new features to.

1.3 Research Questions and Results

The main focus of this master's thesis is the microservice architectural pattern and the message broker technology which also allows the architecture to be event-driven. System development is becoming more complex as time goes on, and systems are becoming more distributed. Along with a bigger need for scalable components, due to varying system load either caused by usage patterns or messaging, the monolithic approach might not be the best solution anymore. There are a lot of enterprise applications that are already implemented as an event-driven microservice application. Companies such as Amazon and Netflix to mention a few, are companies that are using the microservice architecture in their applications. Implementation of a generic architecture that resembles VegVokteren, and verifies that this fulfills the requirements set by Trafsys and Statens Vegvesen is the goal of this thesis. The research questions that arise are:

1. Can an Event-Driven Microservice architecture make it easier to

    I Further extend the system with new functionality and services?

    II Meet the requirements for scalability, fault-tolerance, and security?

    III Meet the requirement for response time?

It is important to gain knowledge of how to correctly implement a microservice architecture, while still keep to principles of software engeneering. The microservice architecture is already used in many enterprise applications, but as always, there are never one answer to everything. In this thesis, we will provide research and results that indicates that an event-driven microservice architecture are the correct solution for Trafsys.

The main contributions of the work conducted in this thesis are:

- Implementation of a representative prototype of VegVokteren.
- Research in an event-driven microservice architecture.
- Analysis and testing of the prototype.

The implementation is publicly available at [21].

1.4 Research Method

The focus of this thesis is to see if an event-driven microservice architecture will make the further implementation of VegVokteren easier, and still fulfill the requirements of the system. To answer the research questions we have implemented a generic, but a representative prototype of the system. VegVokteren is a large and complicated system that has been far too large for us to implement during the work on this thesis. We, therefore, chose to make major simplifications to the system while trying to keep the core functionality of the system intact. The prototype consists of the same type of components, such as a message broker, facilities that generate data, storage services, and processing services.

Our research has mainly been done through experimentation on the prototype, these results are summarized in section 6.1 through 6.4. The focus of the experiments is on scalability, fault tolerance, extendability and response time.

1.5 Related Work

The research field of going from a monolithic, or 3-tier architecture to a microservice-based architecture is vast. A common theme in much of the research on this field is primarily based around how to go from monolithic to microservices.

There are many reasons for the modernization of legacy software using microservices [25]. The primary driver for this is extendability, as a lot of enterprise applications have a large and complex structure. This makes the work required for implementing new features larger than they need to be. According to research, there are two main reasons for this low extendability: A deterioration of the internal structure, and a high number of entry points for client applications. This leads to a high amount of testing and reworks to ensure that all clients are working as they are indented to do. Secondary reasons for modernization is that technology, sooner or later, is going to be outdated. Many enterprise applications are written in old and outdated languages, where knowledge of the language or technology is difficult to obtain.

Another aspect of recent research is that as more and more applications are run in production in the cloud, a microservice architecture is something to consider [7]. This is due to how a microservice application is organized. You have a suite of services distributed across different servers, running as their own instance. A microservice architecture also enables DevOps. DevOps is a set of practices that aim to decrease the time between changing a system and transferring that change to a production environment. This, of course, is due to the services again, as they are smaller and easier to maintain than a monolithic application.

This thesis, on the other hand, is focused more on a case study for Trafsys. They have already decided on a microservice architecture as they want a more modular and easily extendable application than they got today. We are going to look more into if this could work for them with their requirements and functionality.

1.6 Thesis Outline

Below is a summary of the chapters that constitute this thesis. Due to there beeing two candidates on this master thesis, this summary also contains an overview of which candidate is responsible for each of the chapters

Chapter 1 - Introduction - Follo
Chapter 2 - Microservices and Event-Driven Software Architecture
Explains microservices and event-driven software architecture in detail, and what characterizes them.

2.1: Microservices - Follo
2.2: Event-Driven Architecture - Halvorsen
2.3: Apache Kafka - Halvorsen

Chapter 3 - Current Software Architecture and System Requirements
In depth introduction of the current version of VegVokteren, and a discussion about its limitations.

3.1: Current Solution - Halvorsen
3.2: Capacity and Performance of the Current Solution - Halvorsen
3.3: System Functionality and Requirements - Follo

Chapter 4 - Generic Architecture
Overview over the generic architecture that is implemented.

4.1: Prototype Description - Follo
4.2: Facility - Follo
4.3: Business Services - Follo
4.4: Message-Oriented Middleware - Halvorsen
4.5: Front-End CLI - Halvorsen

Chapter 5 - Implementation
Describes implementation details and the implementation aspects of the architecture.

5.1: Apache Kafka - Halvorsen
5.2: System Components - Follo
5.3: Alarm Interpreter Service - Halvorsen
5.4: Command-Line Interface - Halvorsen
6.6: Test Environment - Follo

Chapter 6 - Prototype Validation
Presents experiments performed on the implementation.

6.1: Scalability Analysis - Halvorsen
6.2: Fault Tolerance Analysis - Halvorsen
6.3: Extendability Discussion - Halvorsen

6.4: Response Time - Follo

6.5: Threats of Validity - Follo

Chapter 7 - Conclusion and Future Work

Summarizes the main findings and highlights some of the possible directions for further reasearch on event-driven microservice architecture.

7: Conclusion and Future Work - Halvorsen

Chapter 2

Microservices and Event-Driven Software Architecture

This chapter introduces microservice- and Event-Driven Architecture (EDA), by giving a brief overview of core the concepts for both of these architectures and how they can be used in an application. Section 2.1 describes common patterns in microservices, and section 2.2, we introduce the core concepts needed to understand EDA, and what types of applications EDA is relevant for.

2.1 Microservices

A microservice [16] is a software component that is a decomposition of a business capability. It is composed of a small code base that is independently deployable to either a dedicated server, a virtual machine, or a cloud service. If needed, microservices have their own storage, while other services are restricted to read-access to the information through a well-defined interface. This interface may allow requests to change business entities within the service itself, but it is the service that is responsible for the modifying the entities that handle the database transactions.

When talking about microservice architecture, it is useful to compare it to the monolithic style of architecture. A monolithic application is an application built as a single package of functionality or a single executable. Any change to the application requires building and deploying the entire code base again. A microservice architecture deploys each element of functionality into separate services and scales by distributing these services across servers, using replication if needed. When changes occur, the only thing needed to be rebuilt and deployed is the service where the change has happened. This reduces the time from the change is completed to when it is deployed to production servers. In the following sections, we review typical patterns one might find in a microservice based application.

Organized Around Business Capabilities

A monolithic enterprise application is usually split up into three tiers: a client tier, application server tier, and database tier. The client is what the users are exposed to, and can interact with. This client tier can be its a rich cient, or displayed through a web browser, also known as a thin client. An application server is where all business logic and functionality of the application lives. Finally, the database is where business entities are stored, typically a SQL database.

Development teams are often siloed into groups that specialize within what aspect of the application they are working on. This can result in poor communication between teams, and will usually result in every team forcing logic into whichever part of the application they have access to. Conway's Law [10] states that any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure. In Figure 2.1 we can see a typical monolithic application, as a result of siloed teams as described above. Teams are often specialized according to the three tiers of the application. The team working on the client tier are often user interface (UI) specialists, and are often refered to as front-end developers. The application server is developed by middleware specialists, while the database tier is traditionally developed and maintained by database administrators (DBAs).

The microservice approach splits the application into services organized around business capabilities. A capability is a description of what a business does, with no regard to how or why this is done. When we are discussing business capabilities within the realm of system development, it is often thought of as a piece of functionality the business offers to its

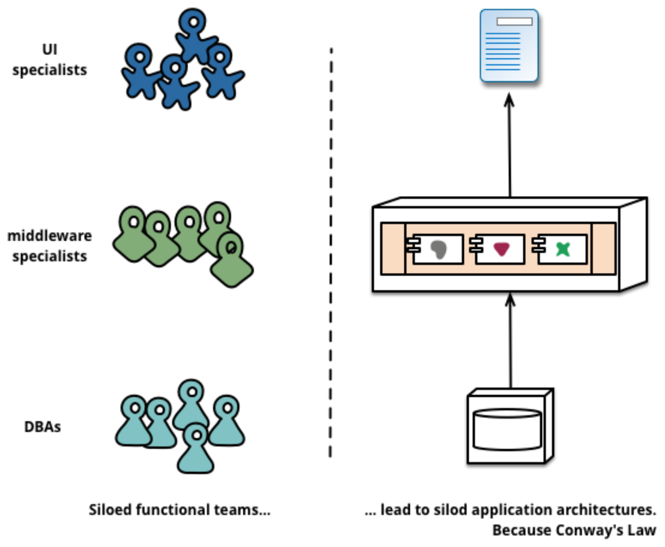

UI specialists
middleware specialists
DBAs
Siloed functional teams...
... lead to silod application architectures.
Because Conway's Law

Figure 2.1: Siloed teams leads to siloed applications [16]

customers, or to enable its employees to do their job. Services are full-stack implementations of those capabilities, including UI, persistent storage and business logic. Teams are for that reason cross-functional, including the entire range of skills required for the development of that service. Following Conway's Law, using cross-functional teams enables the microservice architecture shown in figure 2.2.

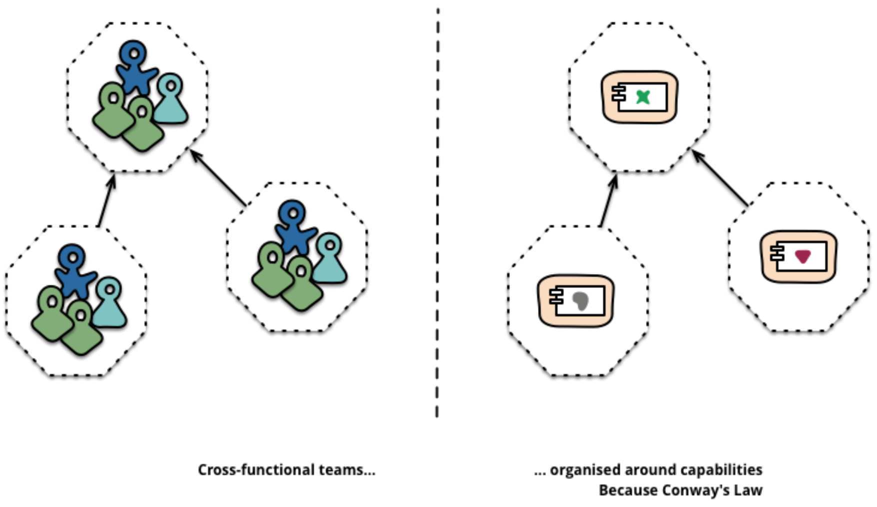


Figure 2.2: Microservices centered around business capabilties [16]

Product Centric

Traditionally when an application is completed and delivered, the code base is turned over to a maintenance team. This is usually the last time the development team touches the code. In a microservice environment, the typical approach is that a team should develop and own a product also after completion. A common inspiration for this is Amazon's notion of "you build, you run it" [19] where a development team takes full responsibility for the software in production. This can be harder to do as a consultant than as an in-house developer. When a product is done, a consultant is usually put on an entirely another project. The goal for this approach to maintenance is to bring developers into day-to-day contact with how their software behaves in production.

There is no reason this approach cannot be applied to a monolithic application, but the finer graned services make it easier to create a personal relationship between the service developers and their customers. There will also be a change in how the developer looks at the code. Instead of looking at the software as a set of functionality to be completed, there will be an on-going question of how the software can enhance business capabilities and continue improvements.

Smart Endpoints and Dumb Pipes

In a monolithic application, all components run within the same process, and communication between components are in-memory function calls. The same is not true for a microservice application, as there are hard borders between services. This is because in many cases services are distributed over a cluster of instances. Therefore the principle of smart endpoints and dumb pipes have been said to be the right way to go when developing a microservice application [16].

The two main forms of communication between services are [37]

- **Request-Response:** One service invokes another service by making an explicit request, usually to store or retrieve data. The service then expects a response: either a resource or an acknowledgment.

- **Observer:** Event-based, implicit invocation where one service publishes an event and one or more observers that were watching for that event respond to it by running logic asynchronously, outside of the awareness of the producer of the event.

2.1.1 Centralized Service Bus

A centralized service bus (CSB) that runs logic to route and manipulate messages is viewed as an architectural anti-pattern. The CSB can easily become a very complex monolithic component in your application, due to all the routing and message manipulation. It can become a bottleneck both for performance, but also from an engineering standpoint when new components are added to the application.

The microservice approach, therefore, benefits the most from decentralized communication, which uses dumb pipes to get messages from one service to another. With the term dumb pipes we refer to a communcation standard that do not require any form of logic to get the data from point A to point B. With this type of communication, there is not any difference between external and internal resources. This means that each microservice encapsulate its own logic for formatting its outgoing responses or supplementing its incoming requests. The decentralized approach lets developers add and modify functionality without altering a centralized bus.

Request-Response Model

The easiest way to implement the request-response pattern is by using HTTP requests. With this approach, a developer can add an intermediate load balancer. This results in the originating service making a request to the load balancer, and the load balancer then forwards the request to one of the instances of the backing service. This is a form of synchronous communication between services [37]. Figure 2.3 shows a typical request-response pipeline between services.

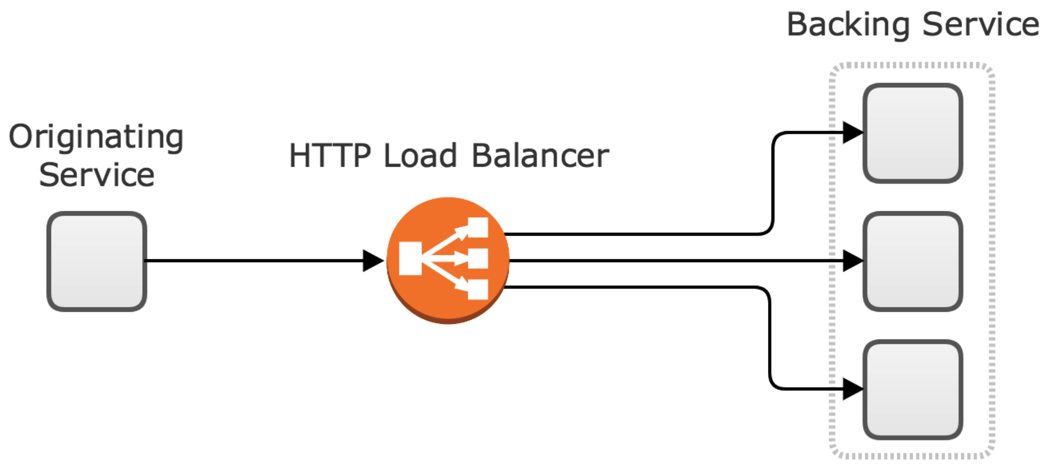


Figure 2.3: HTTP/REST pipeline [37]

Observer Model

Services that do not need an immediate response, either in the form of an acknowledgment or a resource, can use the observer model for communication. A service can publish messages through a message broker, which then places the message in a topic. Other microservices can then subscribe to this topic, and asynchronously receive messages. This allows one-to-many communication, where the publisher does not need to know how many subscribers the topic has, or how subscribers are responding to this message. In the case of a subscriber failure, most implementations of queuing systems have a redelivery feature that ensures that the message eventually is delivered. Figure 2.4 shows a simple pipeline of the described features and model.

Decentralized Governance

When we are talking about decentralized governance, we mean that the decisions on what technology stack to use have been moved from the central project management to the development teams [36]. This enables the teams to work independently and choose the technology stack that is best suited for the task at hand. In other words, project management does not need to be intimately familiar with all technologies that all the services use for their implementation, but the team working with a given stack needs to be. A contributor to this freedom is what we discussed in the previous section, on smart endpoints and dumb pipes.

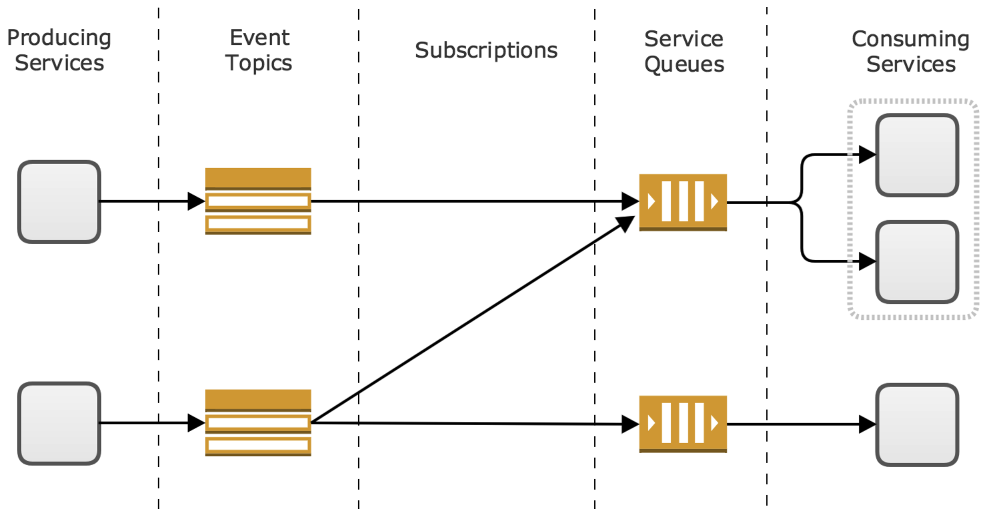


Figure 2.4: Observer pipeline [37]

All services communicate with each other through well-defined interfaces over the web. By using technologies such as JSON or XML it does not matter for a given service if the data is genereated in another programming language than itself. The service will take the values that are in the body of the request, or the data of the message that is passed to it and performs its logic accordingly.

The goal of decentralized governance is to give freedom to development teams to solve problems faster and more efficiently. This opposes the traditional "forced" technology stack of a monolithic application, where a system architect, along with project management, have come to an agreement on what technologies to use for the entire application. By enforcing decentralized governance into a development process, all the developers are able to contribute to the system architecture.

Decentralized Data Management

In a monolithic architecture, it is traditional to have one large SQL database with many tables as illustrated in figure 2.5. This central database is often used as an engine for data persistence, and in many cases stores, its own procedures and have some application logic within it [35].
Microservices on the other hand, focuses on decentralization in all aspects of an application, also in data persistence. Each service that needs storage has its own database, and

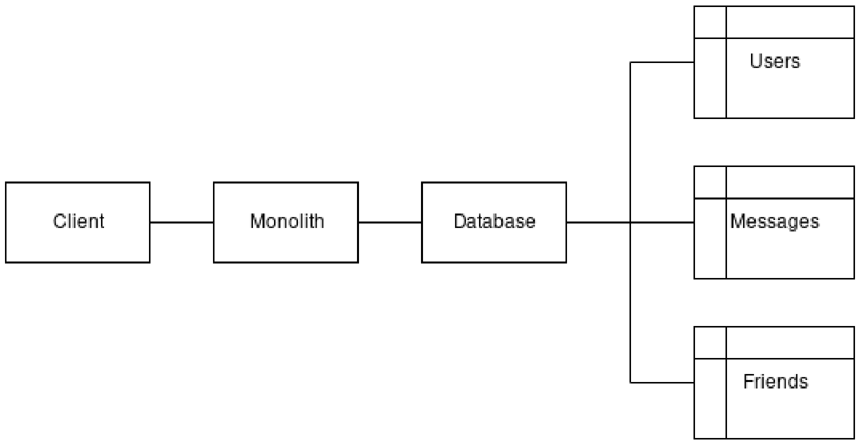


Figure 2.5: Monolithic database architecture [35]

what type of database depends on the needs and requirements for each service. This introduces a term often used when talking about microservices; polyglot persistence. Polyglot means to be bilingual and polyglot persistence means that multiple database technologies can be used within the same application [15]. Polyglot persistence enables an application to have a mixture of SQL and NoSQL databases based on the need of each individual service.

A microservice deployment that enforces decentralized data management, with the same example as in figure 2.5, is shown in 2.6 Here we can see that all the three services have their own database. It does not mean that there must be three independent database servers. They can be hosted on the same physical server if they have the logical distinction of three separate databases [35]. By having this logical distinction, it is ensured that the databases can be scaled up easily, as this can be done by moving each database to its own server.

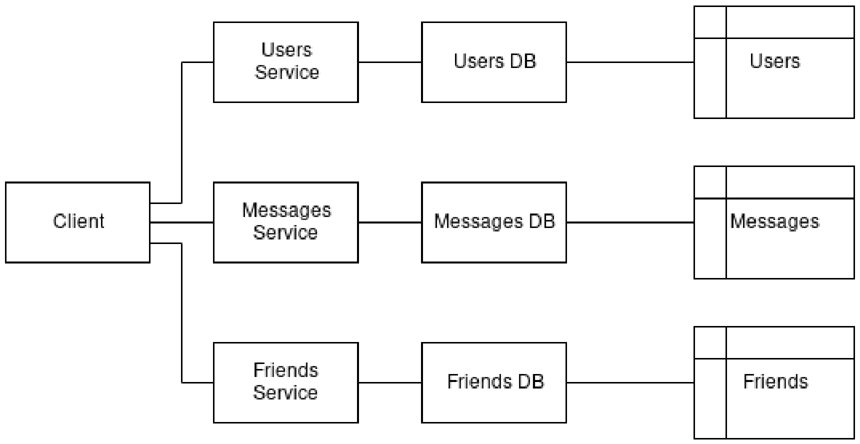


Figure 2.6: Microservice database architecture [35]

The microservice pattern also causes some challanges, and one of them is how to combine data from different services for presentation in a client. With a single database, this is easily done through built-in functions in SQL. One way to overcome this challenge is to add another service to the application that is able to combine data from different services. In figure 2.7, we can see that the new service lives closer to the client and serves as a gateway between the client and the core servers. The gateway acts as a central component for the combination logic and for request handling.

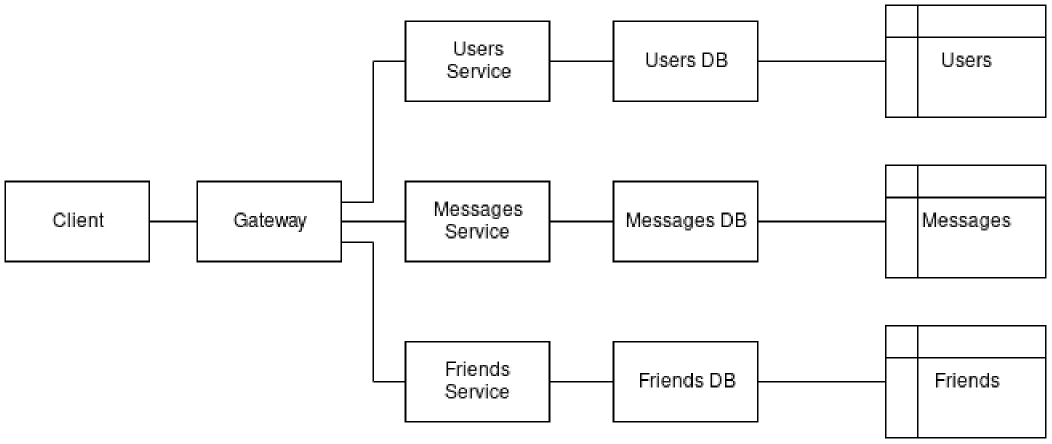


Figure 2.7: Microservice database architecture with gateway [35]

2.2 Event-Driven Architecture

Many software systems need to be able to constantly react and adapt to changing circumstances in an unstable environment. These characteristics are referred to as dynamic capabilities [48] and are some of the capabilities that event-driven architecture (EDA) provide. EDA is a popular architectural pattern for developing systems that demand high responsiveness, the ability to scale well and to communicate asynchronously. These demands are met by implementing decoupled, single-purpose components that receive and process events asynchronously. It is an architecture well-suited for applications that are dependent on real-time transactions because monitoring and processing of events are conducted as they occur. EDA is sometimes referred to as a messaging system that notifies all interested parties when an event occurs [4], and revolves around a publish/subscribe model. In this section, we look at the components of the event-driven architecture and how they interoperate.

Components of Event-Driven Architecture

As mentioned, EDA is essentially an asynchronous messaging system that revolves around the publish/subscribe model. EDA consists of components that make it comply with the decoupling principle. An event producer produces an event and publishes it to an event channel for the event consumers to process. These components are mutually independent [13]. In simple terms, EDA consists of one or more event producers that produce an event containing an event message (information about the event) and sends it to an event channel that translates the message to the messaging protocol of the event consumer. The consumer(s) processes the message and decides if an action or service should be triggered in response. Many software systems rely on the delivery of events. Publish/subscribe models utilize durable subscriptions and persistent storage to meet this requirement. A durable subscription ensures that the consumer eventually receives the events published on an event channel, even if events are published when the consumer is unavailable. If the consumer(s) of an event channel is unavailable, the event will be stored until the consumer(s) are up and running again. In a reliable publish/subscribe model, illustrated in listing 2.8, the producer receives an acknowledgment from the messaging middleware when the event has been persisted. By acknowledging the event, the messaging middleware guarantees the delivery of the

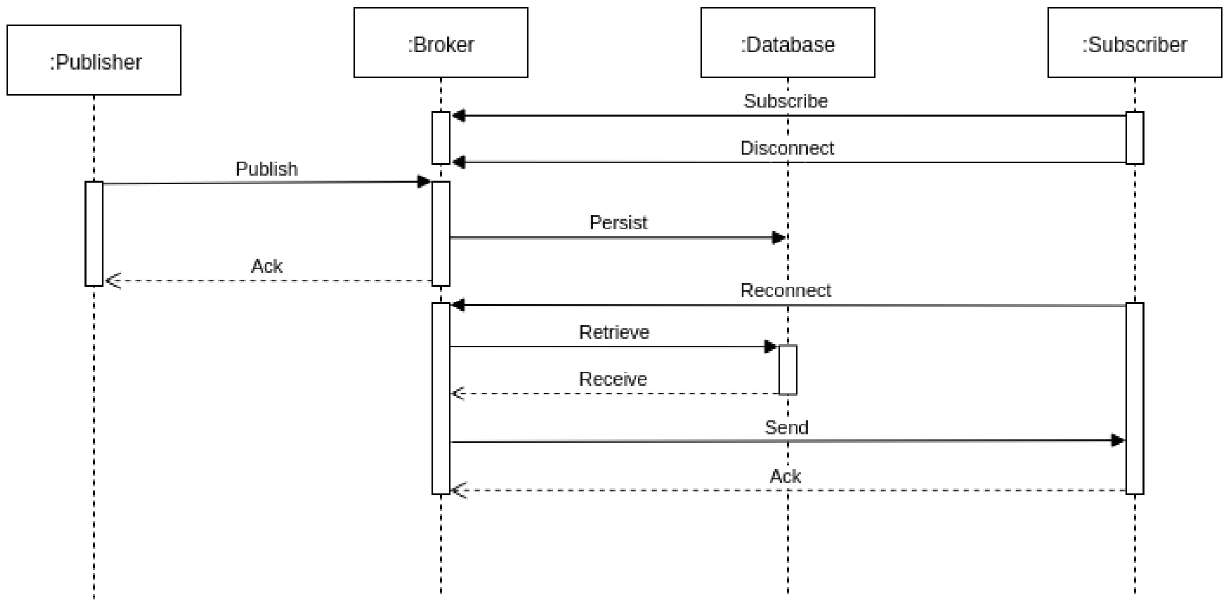


Figure 2.8: Reliable publish/subscribe message flow using durable subscriptions

event to the intended consumer. Likewise, the messaging middleware receives an acknowledgment from the subscriber as a confirmation of the event being successfully delivered, and that it can be deleted from the persistent store. We now take a closer look at the different components of the event-driven architecture.

An event producer is a component that produces events and sends them through an event channel to the event consumers. Applications, data stores, services, business processes, and sensors can all be event producers. Looking at EDA as a graph, an event producer can be seen as a source node with only outgoing connections to the event channels. The number of connections corresponds to the number of event channels involved in the transportation of events from producer to consumer. The event producers know nothing about the destination of the event and do not expect a response. As a result, the event producers might produce an event that is not in the required format [30]. It might, therefore, be necessary to transform the event to the enterprise standard format before being delivered to the event channel. In the other end, the event consumers do not care about the origin of the event or its original form. Producers and consumers are decoupled and know nothing about each other [12].

In EDA, an event is a fact and a notification that represent a change in state that is relevant for a business process [44]. This means that an event is an action or occurrence that is recognized and dealt with by software. An event may contain information about a problem (or imminent problem), a threshold, a business opportunity, an alarm, or a detected anomaly [30]. An example is a reading of sensor data. The reading event could be used to decide whether a service invocation should be triggered, or a business process instantiated [12]. This makes the event a significant part of both the system and the business process. An event consists of an event header and an event body. The event header contains the metadata such as event ID and event type, while the body contains a description of the event. Events are often referred to as fire and forget [8] messages which means that events are sent in only one direction and that the producer does not expect a response. The body does not carry any expectation of what action the state change should trigger, and the event producer has no knowledge about the processing of the event, and who processes it. This leads to loose coupling and enables different components to change independently and without affecting each other.

After an event has been generated, it is sent through an event channel to the event consumers. An event channel can be either a queue or a topic. A topic forwards events to all active subscribers. Subscribers that had an active subscription at the time the topic received the event will receive the event. An event queue forwards an event to only one consumer. If no consumers are available when the queue receives the event, it will be stored until a consumer is ready to process it. Event channels can receive and send events to one, or multiple clients depending on the type, and the events can be of different types [12]. The event transmission in EDA revolves around the publisher-subscriber model. The event producers correspond to publishers, and the event consumers correspond to the subscribers. An event channel forwards events asynchronously to the event consumer.

Event queues offer one-to-one message distribution. The number of consumers may be more than one, but they have to compete for the published events. Event queues store the published events durably, meaning that the event producer and consumer do not have to be sending and receiving simultaneously for the event to reach a consumer. The size of the queue varies with the varying system load, while the time it takes to process an event remains fairly consistent. More event consumers can be added when the system load peaks. Every consumer pulls and processes events at its maximum rate, which leads to high utilization of resources [6].

Event topics typically accept events from one or several event producers and distribute them to all active subscriptions (one-to-many distribution). A subscription is like a virtual queue which receives the events from the topic, and the subscribers/event consumers receive a copy of the event message from the subscription. If there are no active subscriptions when the topic receives the event, the message can be either stored or dropped. Subscriptions can support the same patterns and have the same message receiving functionality as event queues (competing consumer). It is useful to use topics when an application demands the ability to scale to a large number of event consumers for parallel processing [6].

Mediator and Broker Topologies

There are two main topologies when it comes to event-driven architecture. The first topology is the mediator topology. It uses a mediator to coordinate the steps of an initial event. The second topology is the broker topology. This topology is used for chaining events together. Since the characteristics of the two topologies differ, it is important to understand them both so that the topology best suited to meet the requirements of a specific system is implemented. We now go into more detail about these two topologies [41].

When system events consist of multiple steps, being able to orchestrate these steps is important. For this, an event mediator can be used. The event flow using a mediator is illustrated in figure 2.9. After an initial event is generated, it is sent to an event queue. The event is then transported from the queue to the event mediator. Upon the arrival of an initial event, the mediator generates processing events and asynchronously sends them to different event channels to execute the steps of the initial event separately. Each processing event corresponds to a step in the initial event. The event channels, which can be either an event queue or an event topic, send the processing events to the corresponding event processors where the business logic is applied. Some processing events can be processed concurrently [41].

In the broker topology, there are only two types of components: a broker and processors. As illustrated in figure 2.10, events are chained together and distributed over several event processors. This way of processing is useful when the event processing flow is simple and there is no need for orchestration. Event channels in the broker topology can be either an event queue, an event topic, or both, and are contained within the broker component. All

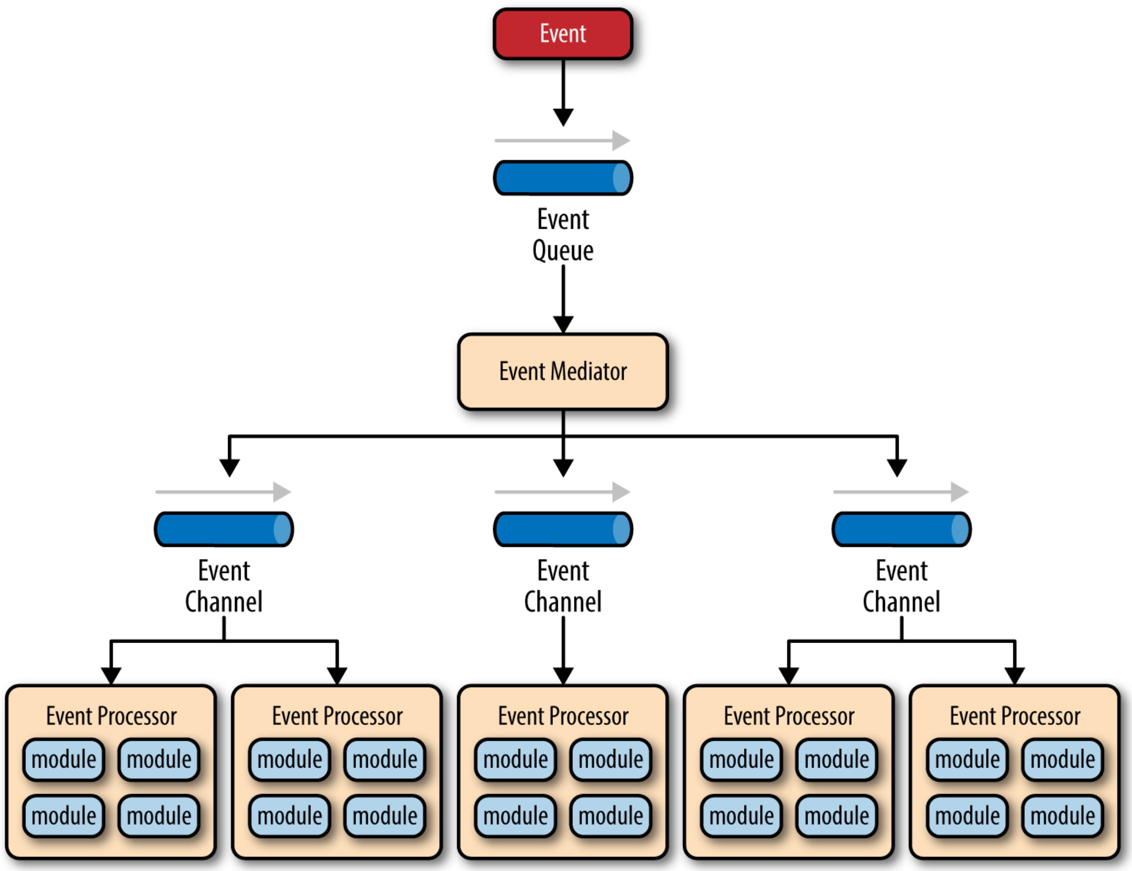


Figure 2.9: Event flow in EDA Mediator Topology [34]

operations involving a message broker is inherently asynchronous, meaning that provided that an event processor is available, the messages will be processed [31]. Events are sent to an event channel and from there distributed to the subscribing event processors which process the event and produces one or more new events that indicate the conducted action. The new events are then published to another event channel to be processed by another event processor. This continues until the steps in the initial event is completed and no processing events are published [41]. Most message broker technologies ships with exactly-once delivery semantics and idempotent producer capabilities which ensures the delivery of messages and that duplicate messages is not published on an event channel.

The last two components of EDA are event processors and event consumers. An event contains information that is relevant for a business process or system. The components responsible for processing the events are called event processors. These components show

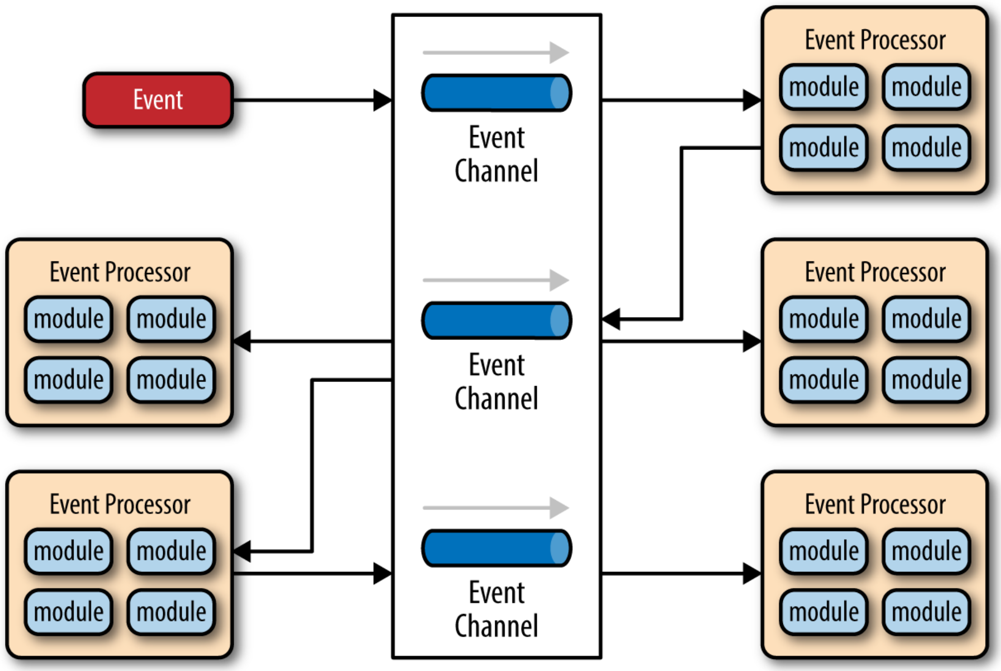


Figure 2.10: Event flow in EDA Broker Topology [33]

their interest in events by subscribing to event channels. When an event is published to an event channel, the event is sent out to one or more subscribing consumers depending on the type of the event channel. An event processor contains all the event processing logic and is responsible for performing specific tasks associated with the event it receives. After the processing of an event is complete, the processor reacts to the processed event. The reaction could be either simple or complex. Simple actions could be database operations like create, read, update, delete (CRUD-operations), update a UI component, or publish a new event. More complex operations could be an aggregation of events to get new higher-level events.

Event processing in EDA

Event processing can be of several types: simple processing, stream processing, or complex processing. They are commonly used together in event-driven architecture.

In simple event processing, there is no modification of events before they are filtered and routed. Simple processing includes translation, splitting, and merging with other events. Even though these events are called simple, it does not mean that they do not provide value to the system. They provide both value and information that is significant to the process or system [12]. Examples of simple event processing are changing event schema from the schema of the event producer to the schema of the event consumer, or adding metadata (such as a timestamp) to the event header. Simple event processing also includes the generation of new events and redirecting those events to a new event channel.

Event stream processing revolves around detecting meaningful patterns in continuous streams of events. This includes identifying relationships and correlation between events, event hierarchies, and event memberships. Event stream processing is useful if the events are happening frequently, and there is a need to detect and respond to events quickly. In financial trading, for instance, a business opportunity could be lost if a pattern in an event stream is not detected quickly. In fraud detection, the response to suspicious behavior should be immediate. Event stream processing happens in real-time and enables a business to initiate proactive measures when an anomaly is detected in an event stream. The output rate must be higher than, or equal to, the input rate in the long run. If the system cannot handle the number of inputs, storage and memory issues will arise [43]. This could lead to lower throughput and higher latency.

Complex event processing revolves around generating new, high-level events by aggregating, filtering, and matching a set of simple events [51]. The set can contain events of different types, and the process involves applying the processing logic and constraints to the set [12]. Events can occur over a long period and event processors apply event interpreters and correlation techniques and match them against defined event patterns. The objectives of complex event processing are, like for stream processing, to detect anomalies, threats, and opportunities.

Discussion

There are several benefits to EDA. By implementing single-purpose, decoupled event processor components, EDA systems can respond quickly to an environment in constant change. Due to the decoupled nature of the architecture, changes are normally isolated to a few event processors. Changes to event processors or other decoupled EDA components can be made without affecting other components. Another result of the decoupled nature of EDA is that it makes the deployment of components relatively easy. It is possible to deploy single components as soon as they are developed and tested. In the mediator typology, the event mediator splits an initial event into several processing events and sends them to corresponding event processors. This leads to a somewhat tight coupling between the event mediator and the event processors. If a change is made in the event mediator, it might be necessary to make a change in one or more of the event processors and vice versa. All components involved in the change will have to be redeployed. In the event broker typology, there is loose coupling between the broker and the processors. It is, therefore, easier to deploy. The ability to scale up or down when needed is another direct result of the decoupled nature of the architecture. It is possible to scale individual components, or a set of components, whenever necessary. It is also possible to start more instances of a specific processor or allocate more system resources if the throughput is lower than the size of the workload. Since communication in EDA is asynchronous, non-blocking and provides the ability to scale and to perform concurrent operations between decoupled components, the architecture can achieve high performance for any workload size [41].

EDA also has its drawbacks. One is that event-driven systems can be hard to test. It is hard to specify and analyze the asynchronous and concurrent behavior of EDA. Testing individual components is not hard, but generating events requires specialized tools. Formal description techniques that specify the interaction between components/processes will have to be applied [24]. The asynchronous, decoupled nature of EDA makes it highly scalable but makes development difficult. Implementing communication between the decoupled components takes time and effort. There is also a need for advanced error handling to deal with failing event processor modules, event mediators, or event brokers [41].

2.3 Apache Kafka

Message-oriented middleware (MOM) are messaging systems used to transmit messages between system components. In this thesis, further discussed in section 5.1, we chose Apache Kafka as our MOM. One of the reasons for choosing Kafka is that TrafSys are utilizing Kafka in the current system. Other good reasons are that Apache Kafka is a high throughput, distributed messaging system that provides stable performance even at very high workloads. It uses the publish/subscribe messaging model and brokers to collect and distribute streams of messages which makes it a good fit for VegVokteren. Kafka also performs replication

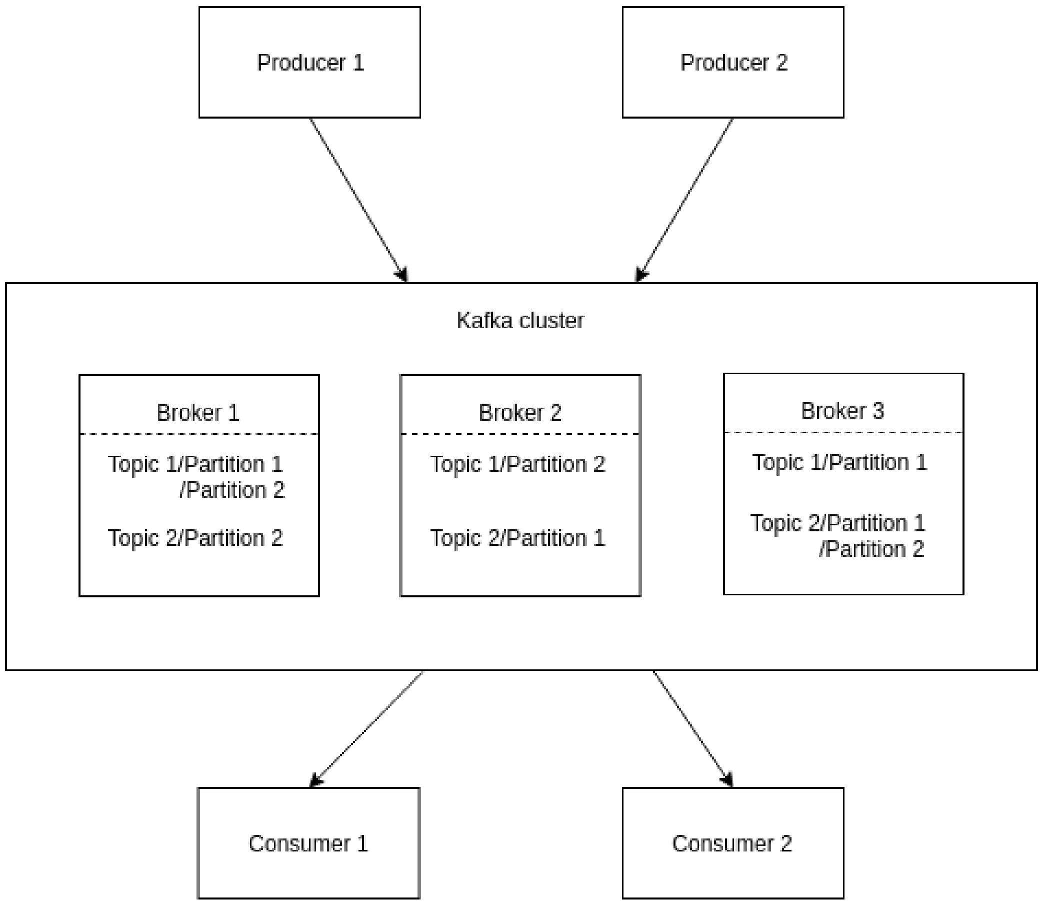


Figure 2.11: Kafka Architecture

between brokers in a cluster to increase availabilty and fault-tolerance. Figure 2.11 shows an example Kafka architecture. The Kafka brokers are spread across different nodes (machines) for better resilience. One or more Kafka brokers (servers) constitute the Kafka cluster, and connecting to any broker connects the user to the entire cluster [50]. Topics in the Kafka brokers receive messages from the producers and stores them in a persistent store for a defined amount of time. A topic holds a particular stream of data and is identified by its name. The topics can be divided into partitions, which is an ordered log of messages, and each message within a partition is given an incremental identifier called offset. The offsets only have a meaning within a specific partition. Messages are immutable which means that when they are successfully transmitted to a topic, they cannot be changed once they are persisted in the broker. Data is assigned to a partition using a round-robin algorithm unless a key is provided. Messages with the same key are guaranteed to be sent to the same topic partition, as long as the number of partitions for the topic remains constant. If the number of partitions is altered, the key-ordering guarantee will be broken since the algorithm for assigning a partition is a function of the number of existing partitions. Kafka brokers are identified by an ID and each broker contain certain topic partitions. The topic partitions are automatically replicated across a number of brokers when a topic is created, but the brokers do not necessarily contain all partitions for every topic. The number of brokers that hold a partition replica depends on the replication factor property of the topic. The replication factor has to be less than or equal to the number of brokers. With a replication factor of N, clients can tolerate N-1 brokers being down either for maintenance or due to failure. Having a high replication factor increases the fault tolerance and resilience of the system, but will require more permanent storage and might result in higher latency. At any time only one broker can be the leader for a partition, and the other brokers containing the partition will be synchronized replicas of the leader. It is the leader that receive and serve data, and the replicas are ready to serve the data should the leader go down. The reason for dividing topics into partitions is that each partition can handle a certain throughput. Having more partitions will provide better parallelism and the ability to have one consumer for each topic partition running in what is called a consumer group. Another benefit is that more brokers in the cluster will be leveraged since different partitions might have different leader brokers.

Producers and Consumers

Producers publish data to the topics in the Kafka cluster. The producers are automatically assigned a partition when they connect to a topic. They automatically identify the broker that is the leader for that specific partition and sends the data. In case of broker failures, the producers will recover by itself and start producing to the broker that has become the new leader for the partition it was assigned. The system load can be balanced between the brokers if the partitions have different leaders. Another property of Kafka producers is that they can be configured to receive acknowledgments from the leader, the leader and the replicas, or none when sending messages.

Consumers subscribe to a topic and poll data from it. Just as for the producers, the consumers are automatically assigned one or more partitions and identifies the leader which will serve the data. They also recover if a broker failure occurs. When reading data from a topic, data from all partitions are served to the consumer unless the consumer belongs to a consumer group with more than one consumer running. There is no ordering guarantee when the number of consumers is less than the number of partitions, but the data is read in order within a partition. By using consumer groups and running one consumer per partition, each consumer will receive the data from its partition in order. This means that data from a topic might be delivered out of order, but data from a particular partition are delivered in order. Consumers within a consumer group read from exclusive partitions and if there are more consumers than partitions, some consumers will be inactive. Kafka stores the offsets of the messages that the consumer/consumer group has consumed. The offsets are committed to a topic named `__consumer_offsets`. When a consumer has received and processed data from a Kafka topic, it should commit the offsets so that it can continue reading from where it left off in case of failure.

When it comes to delivery semantics, there are three options. The first semantic is "at most once". This semantic can be used when it is acceptable to loose messages to avoid duplicated messages. If the messages are not sent again when an acknowledgment times out, the message might not be written to the topic depending on if the broker failed before or after the message was persisted. The consumer commits the offsets as soon as the message is received and if the processing of the message fails, the message will be lost. The second delivery semantic is "at least once". Producers receive acknowledgments from the Kafka broker when a message has been written to the topic. If the acknowledgment fails or times out, the producer might assume that the message was not written to the topic and retry sending it. If the broker failed after writing the message to the topic, the message will be duplicated on the topic. The message will be given different offsets and delivered to the consumer more than once. The offsets are committed after the message is processed. This semantic may result in duplicate processing of messages so it is important to make use of idempotent consumers to filter out duplicate messages. The last and the hardest to achieve delivery semantic is "exactly once". It requires cooperation between consumers, producers, and the messaging system. The send operation of the producers is idempotent. The messages are still sent when an acknowledgment times out, but it will not be written to the Kafka topic more than once. This is achieved by assigning a sequence number to the messages/batches of messages which will be used by the broker to eliminate duplicate messages. The sequence number is stored in a log so that if the leader goes down, any replica can look it up to find out if it is a duplicate [28].

Apache Zookeeper

Apache Zookeeper is a coordination and configuration server that Kafka requires in order to run. Like the Kafka brokers, the Zookeeper servers are distributed across multiple nodes to provide high availability and consistency. Zookeeper also has a leader that handles all writes, and a set of followers that handle reads. If the leader node fails, a new one is selected. Having Zookeeper handle coordination and configuration data, the developer can focus on the application logic. One of the tasks that Zookeeper performs is leader election for the Kafka cluster. It is responsible for maintaining the leader-follower relationship across all topic partitions and that there is only one leader for each partition. If the leader fails, it is the responsibility of the Zookeeper server to elect a new leader and notify the selected broker. Zookeeper contains a list of all the brokers that belong to the cluster and it sends notifications to Kafka if a broker goes down or comes back online. Additionally, Zookeeper contains the configuration of the Kafka topics. It maintains a list of all existing topics, the number of partitions for each topic, information about where the replicas are located, and it knows which broker is the leader for each partition. It is also the Zookeeper server that holds the access control lists for all topics which have information about authorizations for different users/services [49].

Kafka Guarantees and Configuration

As we have seen, Kafka, for instance, guarantees automatic per-partition ordering, replication, and persistence. Kafka also promises guaranteed delivery and high throughput provided that the system is configured properly. To meet the promised delivery guarantee, the producer can be configured to retry sending messages. The default option for the retries property is 0, meaning it will not retry sending at all and messages could, therefore, be lost. Setting the property to a large number will make the producer retry until the message is successfully received by the broker. The retry property guarantees delivery, but might lead to messages being sent out of order. This becomes an issue if the system relies on key-based message ordering. To deal with the ordering problem, the consumer has a property to control how many produce requests that can be made in parallel. This property is restricted to ≤ 5 for idempotent producers, which means that ≤ 5 produce requests can be unacknowledged at the same time. The value of this property depends on whether the system requires message ordering or not. To ensure message ordering this property could be set to

1, but this might lead to lower throughput. As explained, idempotent producers are used for preventing duplicate messages. Idempotent producers should have the retries property set to a large number to ensure delivery. They also require the acknowledgment property to be set to 'all' which means that the producer needs to receive an acknowledgment from the leader broker and all its synchronized replicas. This will make the delivery more reliable since it is guaranteed that all the running replicas will be synchronized with the leader at all times.

Some systems might also achieve higher throughput with proper configuration. Compression can be applied at the producer level to get a much smaller request size. It will also result in faster data transfer and better disk utilization in the Kafka brokers. Which compression algorithm to use depends on the system and the data that is being transmitted. Both producers and consumers will have to commit some CPU cycles for compression and decompression, but this is a very minor cost. Another property that helps to achieve high throughput is batching. When a Kafka producer reaches the maximum number of parallel send request, it starts batching the requests that are waiting to be sent. Batching messages will increase throughput by increasing the number of messages sent, as batches have a higher compression rate which in turn leads to better efficiency [17]. Producers defaults to sending messages as soon as possible and does not batch unless sending is blocked. To force batching a small delay can be introduced to increase the chances of messages being sent in batches. This means that at the expense of a little higher latency, the system can achieve higher throughput, compression, and efficiency. If the batch is full before the delay period, the producer sends the batch to the broker. The properties configuration may vary from system to system, and finding the optimal configuration is a difficult task.

Kafka Security

Kafka does not have any default security mechanisms. Messages are transmitted as plain text and can be sent and read by any client that has access to the IP-port combination of any broker in the cluster and the port. This IP-port combination can also be used to configure the Kafka cluster by for instance deleting or edit topics. There is a need for security mechanisms to ensure the confidentiality and integrity of the system. The first step towards secure communication in Kafka is to encrypt the data being transmitted and stored. Kafka can be configured to utilize TLS when transmitting messages. The Kafka server holds a private server certificate that is signed by a Certificate Authority (CA), and the client holds a public CA certificate. The Kafka server presents its signed certificate to the client. The client then uses the stored public CA-signed certificate to verify the server certificate. Once the certificate is verified, a secure and encrypted channel is established. This process is called the TLS handshake [26]. The client uses the CA public key to encrypt messages and the server uses its private key to decrypt the messages. By encrypting the messages, confidentiality is ensured. The integrity is ensured by including a message integrity check using a message authentication code. The code is used to confirm the authenticity of the sender and to prevent undetected loss or alteration during message transmission.

The second step towards securing communication is to authenticate clients. Authentication ensures that only clients that can prove their identity can connect to the Kafka cluster. There are two options for authentication clients in Kafka. The first is TLS authentication where users present their TLS certificate and are granted access if the certificate verification is successful. The second option is Simple Authentication and Security Layer (SASL) TLS authentication which is the option we went for within our prototype. SASL in Kafka implements multiple protocols, but we chose to use GSSAPI which uses Kerberos authentication. Kerberos is a protocol for authentication over insecure networks. The data exchanged is encrypted and the authentication is based on tickets that are issued to principals. A principal is a unique name associated with a service instance. A trusted 3rd party Key Distribution Center (KDC) is used by Kerberos to maintain the principal database.

Alternative Message-Oriented Middleware Technologies

There are several good MOM alternatives to Kafka. In this section, we briefly discuss some of them and look at what functionality they provide and how they differ from Kafka. The first technology we discuss is RabbitMQ [38]. RabbitMQ is an open-source message-queueing server that allows users to share data or queue processing jobs. When producers produce faster than the consumers can process, RabbitMQ can buffer messages. The messages are stored in DRAM as long as there is free space. When the DRAM is full, messages will be written to a persistent store which decreases the performance of the system. RabbitMQ provides an API that lets the user determine the routing logic, in contrast to Kafka which supports topic-based routing. In Kafka, the user can control which partition the messages are sent to by providing a key. In RabbitMQ, the routing logic is more customizable and can be whatever the user needs. Another feature RabbitMQ provides is ordering guarantee. Messages are sorted when written to a queue which means that messages that have to be resent due to failure, is delivered in order. In the case where a load balancer is used between the producer(s) and the broker to reach the scalability that can be achieved with the use of partitions in Kafka, the ordering guarantee no longer applies since the messages leave the load-balancer via different channels. When the consumer(s) have acknowledged a message, the RabbitMQ brokers deletes the message while the Kafka brokers keep them in a persistent store for a configured period of time. As we have discussed, Kafka performs automatic replication if the replication factor is properly configured and there are available brokers. RabbitMQ does not perform automatic replication of queues, but replication can be configured during the creation of a queue [11].

We also considered Google Cloud Pub/Sub [18] which is a cloud-based MOM solution. Could Pub/Sub provides messaging capabilities such as security, durability, and scalability. In contrast to Kafka, Cloud Pub/Sub provides some default security mechanisms. Only clients that have access to the private key of the service can access the broker. It is similar to Kafka in the way that messages are acknowledged and temporarily stored to ensure message delivery. Cloud Pub/Sub offers both push and pull subscriptions. A push-based subscription receives the message as soon as possible, while pull-based subscriptions store and queue the messages for consumers to pull. In the case where there are multiple subscriptions on a topic, each subscription will receive a copy of the message. Subscriptions with more than one consumer will only send the message to one of them. Also, if data is sent to a topic with no subscribers, the data may be lost. Cloud Pub/Sub is horizontally scalable which means that it is possible to increase the number of server instances. The load balancing mechanisms of Cloud Pub/Sub routes published messages to the geographically closest data center which helps retain low latency for the clients. Published messages are also replicated across different regions so that clients can receive messages with low latency. Clients are assigned the data center that has the lowest latency on the connection. Any of the data centers that hold the data can serve clients and the load is distributed between a set of available forwarders. Kafka scales by adding more partitions to a topic, not by increasing the number of servers. The partitions of a topic is evenly distributed across the brokers that comprise the Kafka cluster. Each partition has a leader broker which handles all writes meaning that the load is distributed across the brokers. The load balancing is performed by the producers and is based either on message keys, or a round-robin algorithm. The data is replicated across brokers so that all brokers can serve the data should the leader fail, but the serving is not based on geographical location.

Chapter 3

Current Software Architecture and System Requirements

VegVokteren is a monitoring and control system that has been under continuous development for nearly 20 years. The system is used by the traffic control centers in the western and northern regions of Norway to monitor infrastructures such as tunnels, mountain passes, and bridges. In addition to the traffic control centers, the system is utilized by several other users that have an interest in the information the solution provides about facilities and alarms. These users might need access to the system to perform work such as maintenance or construction on the facilities. Examples of facilities are tunnels, bridges, mountain passes, or intersections, and examples of objects within the facilities are cameras and signal lights.

The incentive behind developing a new architecture is the concern that VegVokteren will not be able to fulfill the users' expectations in the future, even though the current solution is stable and has shown to have a high up-time. The number of facilities is also increasing, and they are getting more complex. Facilities also contain more equipment than earlier, and there is a growing need for more advanced functionality in the system.

3.1 Current Solution

The almost 20-year-old, first version of VegVokteren provides the foundation of the current solution. A large part of the system has been renewed in later versions, and it is continuously extended with new functionality. VegVokteren has a centralized database and is built by using what was, at the time of development, recommended solutions and technologies. The solution is centered around the database, and a lot of the processing is conducted inside the database as stored procedures which poses one of the challenges of the current system. Below we take a closer look at the elements that comprise the existing solution.

VegVokteren was well suited to solve the tasks it was built for when it was first put into use back in 2000. Today, a lot has changed when it comes to both the facilities and the functionality of the system. The facilities are getting larger, and more and more facilities are registered in the system which is in continuous development. The number of users is also increasing. Earlier the majority of the system's users were the operators at the traffic control centers in the northern and western regions of Norway. These operators now constitute the minority of the user group, while the majority consists of entrepreneurs performing maintenance and construction work on the facilities. The traffic control center operators need to have access to the system 24 hours a day to be able to react to occurring events, and the entrepreneurs access the system for instance when they are performing maintenance or construction work. The amount of information about the facilities has also increased significantly. These changes are so significant that changes to the current architecture are necessary for the system to meet the requirements of the users.

Technical Platform - Instances

The technical configurations and setup can differ from instance to instance, which makes it require a lot of effort to install and run a new instance of VegVokteren compared to if there was a standardized configuration applying to all instances. Figure 3.1 shows the main and backup instances of VegVokteren at Statens Vegvesen and how they are connected to the facilities. As mentioned in the introduction to this chapter, the system is used by the traffic control centers in the western and northern regions of Norway and each region has a total of four instances of VegVokteren. The main and backup instance is shown in figure 3.1. The main instance is the one that is used when everything works as intended. There is also a backup instance that at any point in time is synchronized with the main solution. The two instances are independent of each other and they send, receive and store the same data. The

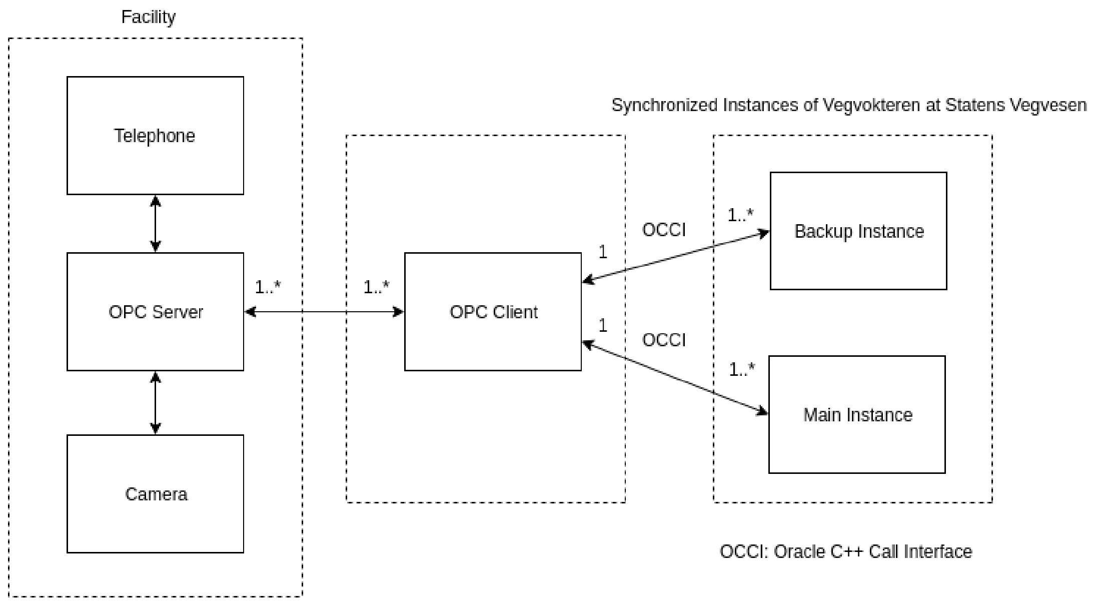


Figure 3.1: Communication between the instances of VegVokteren at Statens Vegvesen and the facilities

backup instance is used if the main instance crashes, is taken down for maintenance, or is in the process of being upgraded. Each region also has a test lab which serves as a test environment for Statens Vegvesen. In this environment, it is possible to perform configuration and testing of, for instance, new functionality and bug fixes. The last instance is a simulator that is used to train new users of the system. It gives users the ability to build different scenarios/sequences of commands, manipulate object state, and trigger alarms to simulate real events. The simulator is in production but still under ongoing development.

In addition to the development environment, TrafSys hosts three instances. They have a server that at any time contains the latest version of the source code of VegVokteren, a configuration and upgrade server dedicated to configuration and upgrading of instances, and a test environment where new functionality and bug fixes are tested before being installed in the test environment of Statens Vegvesen.

Architecture - Modules and Relationships

VegVokteren is a real-time monitoring and control system built for high availability and needs to be constantly in operation. The facilities being controlled ranges from small with few objects (e.g. cameras and signal lights) and users, to large and complex facilities with a large number of objects and users. The system fetches status and alarms from these facility objects, streams video, and receives voice calls. The system also contains functionality that enables users to make decisions in reaction to events that occur, as well as functionality for generating reports that are stored and used for analysis. Examples of events are when a smoke detector is triggered, or when the door to a fire cabinet is opened.

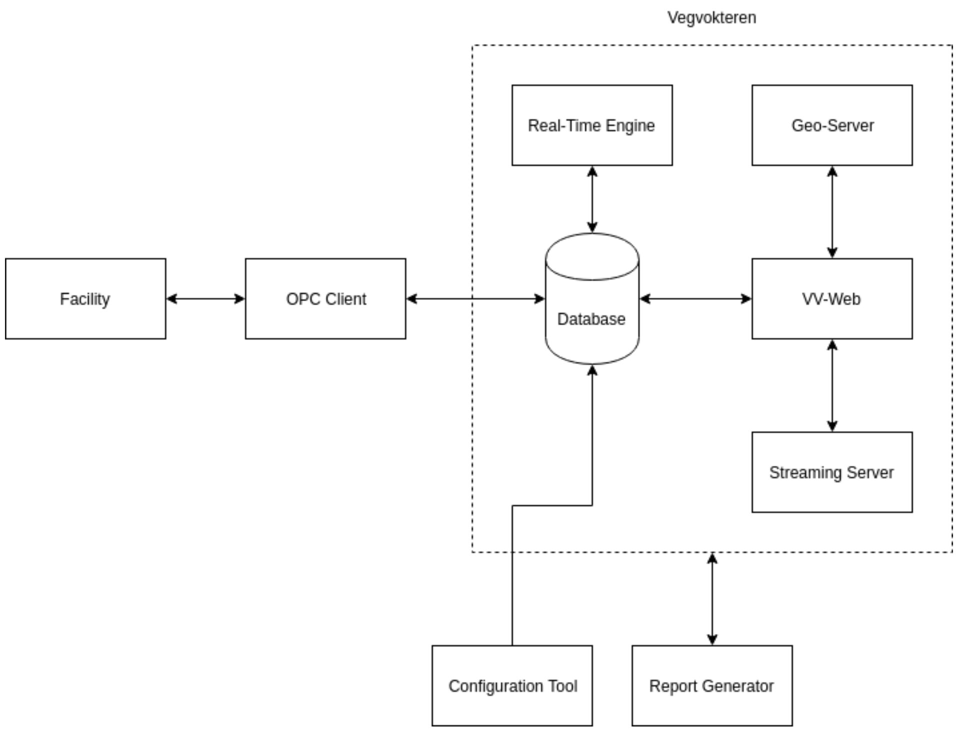


Figure 3.2: Architectural overview of VegVokteren

Figure 3.2 gives an overview of the main components of the existing system and the relationships between them. VegVokteren consists of five main components that enable monitoring, controlling and analysis of events that occur. After an event is generated, it is sent to VegVokteren which is the central monitoring and controlling component and the software the users interact with. This component contains a real-time engine which polls a specific table (event store) in the database containing bit values that represent object statuses, interprets and translates the bit-level values to a readable format, and writes it to another table in the database. VegVokteren also contains a database which is connected to a user interface called VegVokteren-Web (VV-Web). Both the real-time engine and the database are logging events, and alarms are stored in the database and handled by an alarm handler in the real-time engine. The definition of how the real-time engine is supposed to handle events and updates is defined in the configuration file of the facility. All the facility configuration files are loaded when the real-time engine is started. VV-Web is connected to a geo-server providing geographical data and functionality, and a streaming server controlling all video streams and snapshots from connected cameras. The system also contains a configuration tool component that is used to define a representation of objects in the facilities and generate configuration files. The configuration files are used for graphical and functional setup of the facilities. VegVokteren is also connected to a component that generates reports based on incoming events and other information. These reports are, as mentioned earlier, stored and used for analysis. The users of the system have different roles which define what a user is authorized to do within the system.

The communication between VegVokteren and the facilities is based on what was previously known as OLE for process control. OLE is a technology developed by Microsoft to integrate various types of data, and the abbreviation stands for Object Linking and Embedding. The standard was renamed to Open Platform Communications (OPC) in 2011 and is an industry-standard created by Microsoft and several other world-leading software and hardware companies to exchange real-time data [20]. OPC clients are used as middleware to connect VegVokteren to the facilities. An OPC client is connected to the database in VegVokteren and communicates with OPC servers that are connected to the hardware in the facilities, as illustrated in figure 3.3. An OPC server is a program that converts messages from the communication protocol of the hardware to one of several OPC protocols. An OPC client is a software component that receives data from the hardware or sends commands to the hardware through an OPC server. Both the main and the backup instance are connected to the facilities through OPC clients and servers, and both instances can access the same data at all times. In addition to having redundant instances of VegVokteren, some facilities are set up with redundant OPC servers to help meet the requirement of high availability. The equipment in the facilities is periodically polled by the OPC server(s) and the values of the equipment are updated through programmable logic controllers (PLC) that are connected to the equipment. A PLC is a device that receives information from an input device, processes the information and triggers an output such as an alarm. An Oracle C++ Call Interface is utilized by the OPC clients to communicate with the database, and the response is sent through a database management system (DBMS). There is a one-to-many relationship between the OPC clients and the OPC servers meaning that multiple clients can be connected to the OPC server(s) in the facilities. As can be seen in figure 3.4, the users

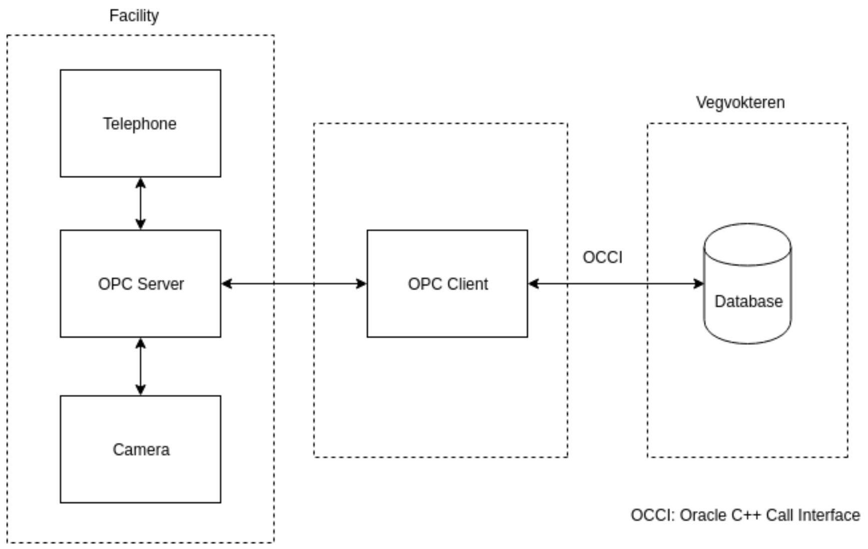


Figure 3.3: Communication between database and facility

interact with the VV-Web component which communicates with the database. VV-Web is a graphical interface and contains an access control system that manages access to different parts of VV-Web. The access control system controls access to the following components in VV-Web:

- An SVG image generator which is used to generate representations of the facilities based on XML data received from the database through the Oracle DBMS pipe.
- A module for asynchronous updating. This module receives asynchronous updates from the database and uses this information to update the alarm list when the status of an object is changed by an event in a facility. The alarm list contains a list of all active alarms.
- A Geo-Server providing information of where facilities are located and their associated objects, and functionality that enables users to search for facilities either by navigating a map or by text search.

Through an interface to VegVokteren, users can stream video from specific cameras at the facilities. The video streams are handled by a separate video streaming server which streams

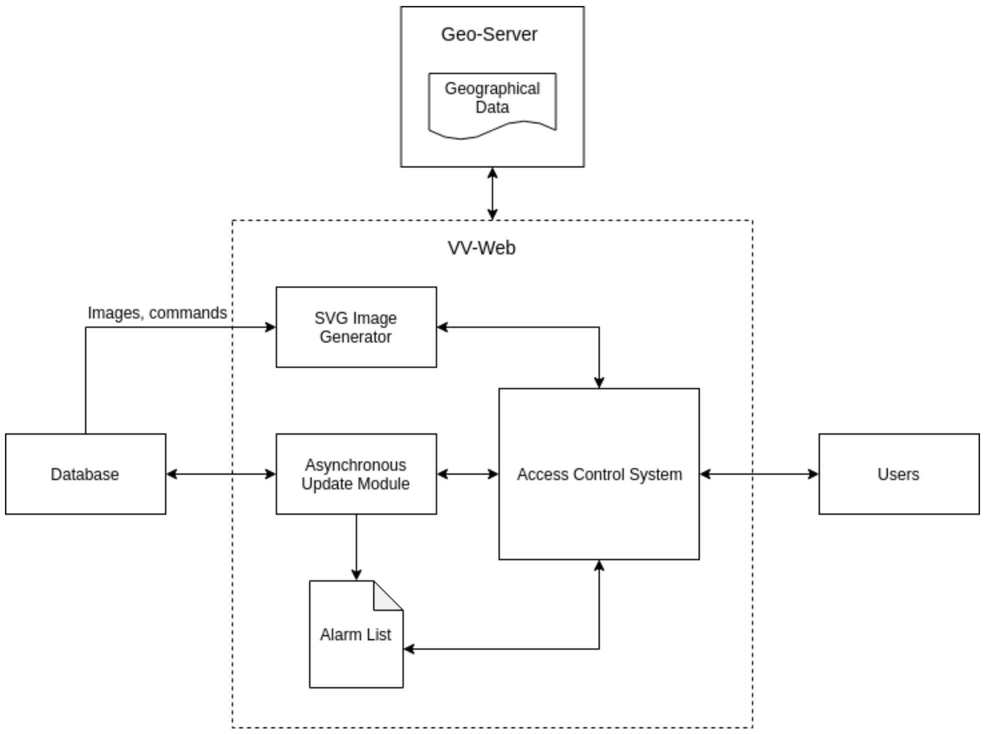


Figure 3.4: Overview - VV-Web

video from the selected camera. Users are also able to display snapshots from cameras directly in VegVokteren.

To build a representation of a facility with associated objects, a configuration tool is used. The tool is also used to specify the possible states an object can have. Figure 3.2 already showed how the configuration tool is connected to VegVokteren. The configuration tool consists of three tools which are used to set up a new facility. These can be seen in figure 3.5. The model editor is used to define facility objects, associated features, parameters, variables, and analogous values for the objects. It is also used to define the dependencies between the objects, and how they communicate. The SVG editor is used to generate backgrounds and graphical representations of facility objects. The output from the SVG editor is used by the screen editor to create several different displays consisting of background, objects, and graphical views. VegVokteren is updated at startup by XML files generated by the model editor and the screen editor. In the current solution, it is also possible to make updates directly on the database from the configuration tool.

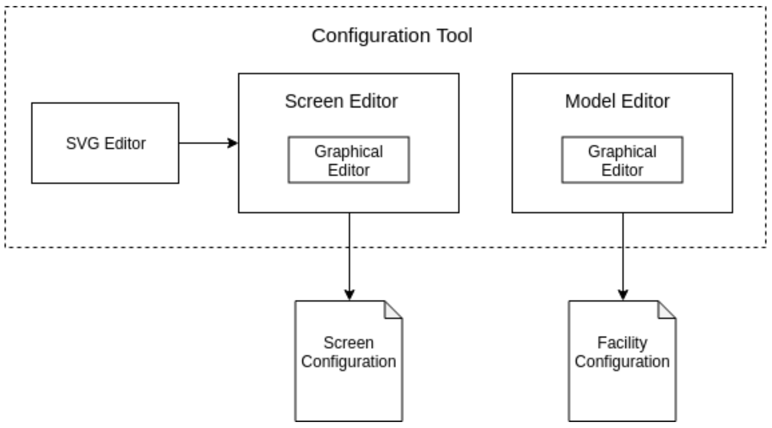


Figure 3.5: Overview of the configuration tool

3.2 Capacity and Performance of the Current Solution

TrafSys has conducted a few capacity tests of the current solution to evaluate how the system handles varying transaction volumes. The tests were carried out in two different environments. One environment was set up in a way that makes it possible to reconstruct it to later test a new architecture and determine if a new system meets the system requirements. The second environment is the backup instance in the western region of Norway. This environment is in production and could provide a good indication of when, or if, the current architecture encounters problems. The time consumption on reading and processing of messages between the database, the real-time engine and the web server for different transaction volumes and number of facility objects were measured in the tests. Two tests were carried out. The first test was an update test where different sized message loads were simulated and sent over a 30 second period. The messages updated status on the object in a facility. The second test was an OPC failure/Reset test. The OPC servers in the facilities are what connects facilities to VegVokteren, and all communication with the facility is lost if the OPC servers crash. If communication is lost, the facility objects' values are updated to indicate a failure. In the OPC failure tests, this was simulated by running SQL updates that changed the objects' values in the database. Transaction volumes of 100, 1000, 5000, and 10000 messages and 1, 3, and 5 objects were used for testing in the virtual environment. The tests were conducted on one facility. In the virtual environment, the tests indicate that the performance is relatively constant for all test scenarios, at 10 messages per second, up until 5000 messages. At 5000 messages the tests indicate that the performance decreases, but the results are hard to validate without further testing. The same tests were conducted on the backup instance in the western region of Norway, but with message loads of 1000, 5000 and 10000. Logs from the main solution in Bergen were also analyzed to determine what the average load is during a certain period and what the maximum number of messages processed per second is. As mentioned, the tests indicate that the performance is relatively constant at 10 messages per second in the virtual environment, meaning that it will take 100 seconds (1 minute and 40 seconds) to process 1000 messages. For the tests conducted on the live backup instance, the number of messages per second was 42. The performance starts decreasing at around 5000 messages per 30 seconds in both environments. To confirm this further testing needs to be carried out. It seems like VegVokteren might have limited capacity at higher workloads, but this does not normally cause any problems in production because the system does not reach the limitations very often. A problem they have experienced is when central communication objects (OPC server and client) fails. Errors in these components generate a large number of messages, and this has led to situations where alarms have been delivered with several minutes delay.

Main Challenges of the Current Architecture

The capacity tests described above show that the performance decreases at around 5000 messages per 30 seconds. The system needs to be able to handle an ever-growing amount of data and it has to have high uptime and availability. VegVokteren consists of multiple components which can run on one or more different servers. Earlier versions of the system were typically divided in two and run on two different computers. The reason for this was that one computer did not have sufficient resources. One machine hosted the database and the OPC, while the other hosted the real-time engine, filtering, and the webserver. Later, TrafSys started hosting all the components on the same physical hardware, because there was no real reason for hosting different parts on different machines. There is a tight coupling between the components in the system and if one of the central components crashes, the system becomes unavailable. This means that the system has a single point of failure independently of whether the system runs on one machine or are divided and run on multiple. The system is currently scaled by copying the entire monolith to another physical machine which provides an easy way of dealing with downtime and failovers. If software or hardware fails, the operators can use the backup while restoring the operation of the main system.

VegVokteren is in continuous development which means that it is occasionally extended with new functionality, tested, and configured with new facilities. Testing of new configurations and new features is conducted in Statens Vegvesen's test lab as explained in section 3.1. When new functionality is ready for production, the system has to be taken offline to be upgraded. This means that the system is unavailable for the time it takes to perform the upgrades. As larger and more complex facilities were added to the system, the stability was varying a lot. After an upgrade of the hardware and the database version were conducted, it has become a lot better. There are still some issues concerning the stability of the system. The current challenges concerning stability and availability are that the system is not being monitored and there are only a few preventive measures that are rarely carried out. This means that it is hard to detect problems and areas where improved performance is needed. Besides, a new instance has to be started physically upon crash. There is no easy way to synchronize the different instances and the system has to be taken offline to be upgraded or to fix errors.

One of the challenges concerning scalability is decreased capacity in the interaction between the database, real-time engine, and the webserver when the message load is high. As discussed in the previous section, testing indicates at the capacity decreases at around 5000 messages per 30 seconds. Another problem with the current solution is that a new instance of VegVokteren has to be physically started to scale the system. There are also challenges when it comes to configuring the system. Configuring the system is time-consuming because the configuration has to be duplicated for all instances. Another potential problem is that the software is platform-dependent which requires all users to run the required platform. Upgrading software to newer versions requires a lot of effort because of the tight coupling between the components. This means that a minor change in a single module might require updates throughout the entire monolith which is time-consuming and harder than what is necessary.

The incentives behind a new software architecture are to ensure the stability, availability, and scalability of the system. The new architecture should facilitate quick and easy updates of the system whether it is being extended with new functionality, upgraded, or bugs are fixed. Currently, the architecture does not support these capabilities because of the tight coupling between the components and every change to the system makes the coupling even tighter. Every component of the system is dependent on the other components except the OPC client. Disabling an OPC client will not affect the rest of the system. Tight coupling prevents individual scaling of components, and also the availability of the system since it has to be offline when performing an upgrade. There is also a lot of logic in the database in the form of stored procedures. Every component has access to the database which makes it hard to follow the flow of data through the system. The database also has the responsibility of evaluating and updating all graphical objects in VV-Web, and it is hard to follow these updates and to troubleshoot.

As mentioned, it is not possible to scale individual components in the current solution. Being able to start new instances of the components that become bottlenecks would increase the performance of the system significantly, and the extra instances can be stopped when the system load decreases. This will save cost and resources compared to the entire system being scaled like it is done today. Having loosely coupled components that can be scaled independently would also increase the availability of the system and lead to a more stable system. A running instance of a component could be replaced with an updated version after the upgrade is complete without having to take the entire system offline. The component that is upgraded can be disabled and replaced by the updated version.

3.3 System Functionality and Requirements

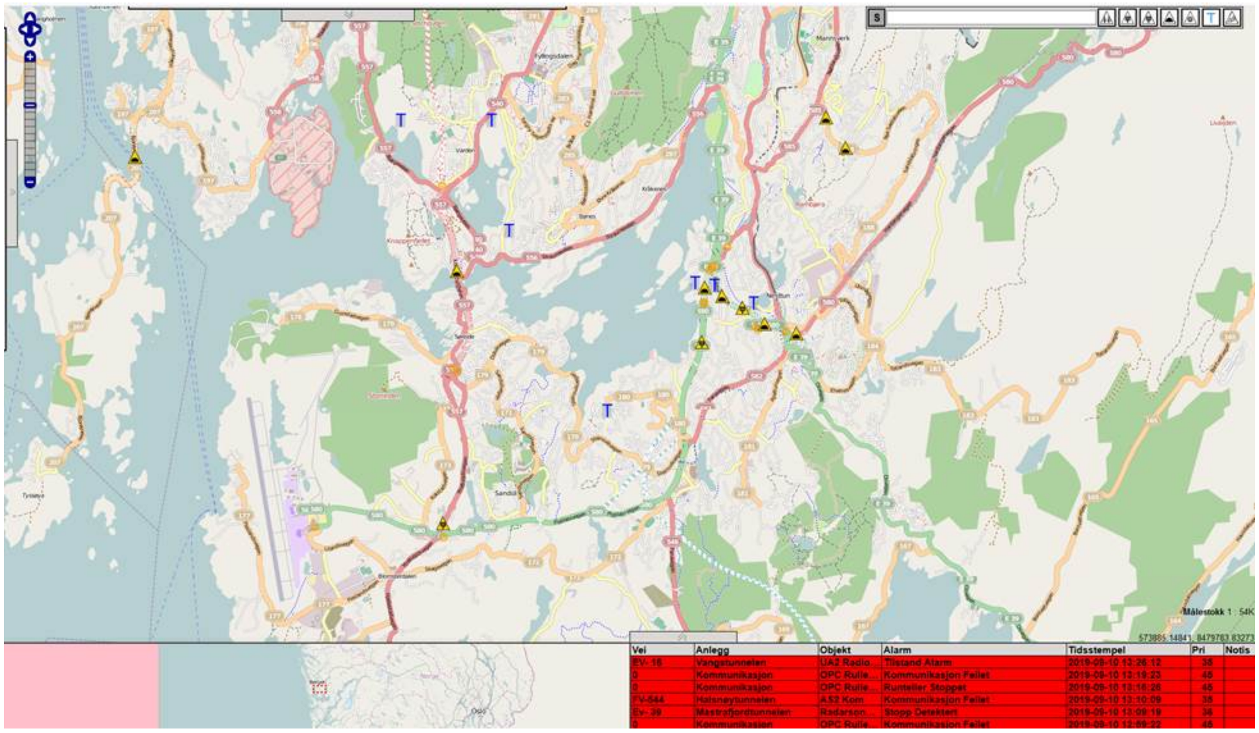
Even though the system architecture is getting updated to an event-driven microservice architecture, the system functionality and its requirements will remain the same until new equipment is added or further requirements are requested. It will still be used as a surveillance and control system, and give users the ability to control traffic flow and safety measures at different facilities.

3.3.1 Overall System Functionality

The main application window contains a world map with all the facilities dotted across, an alarm list and different menus as seen in Figure 3.6. Facilities are represented on the map as a triangle, which has a symbol in it depending on what type of facility it is. The triangles are also able to change color based on errors in the equipment or alarms within the facility.

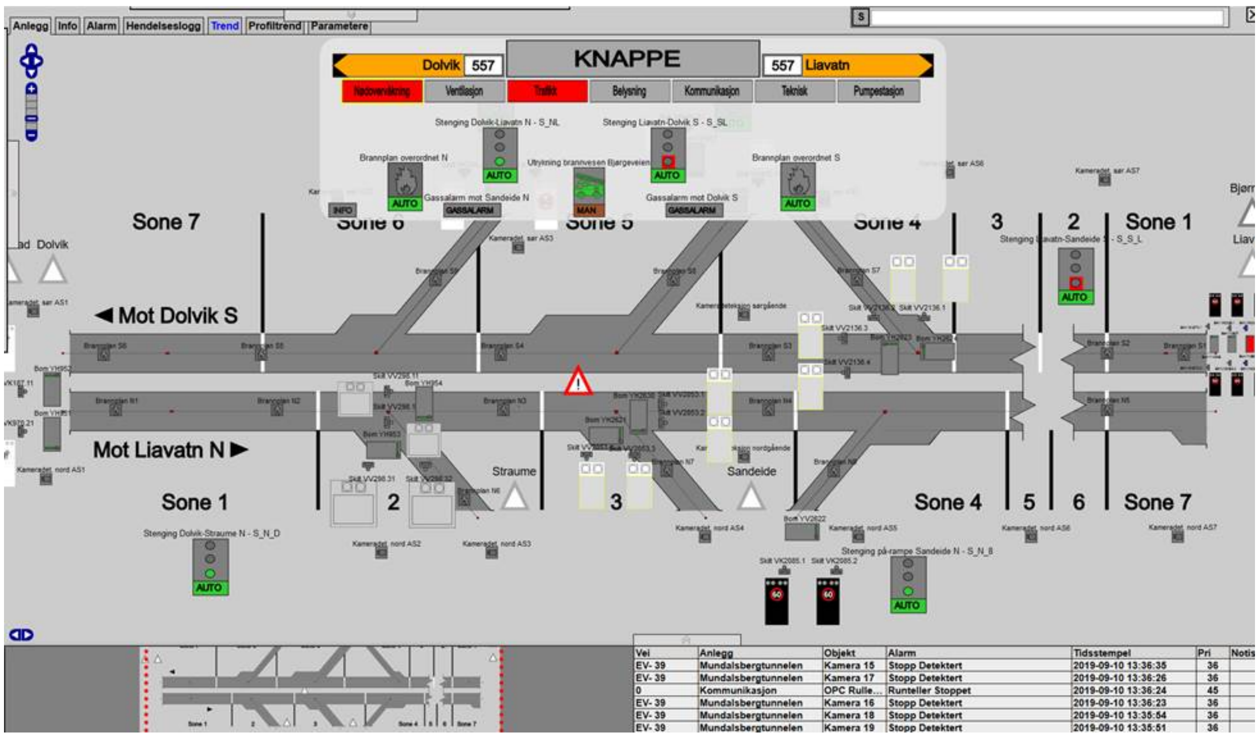
Each triangle is clickable, and takes the user to the facility map.

The facility map is a vector drawn representation of the facility and contains all the sensors and actuators within the facility, an example of this can be seen in Figure 3.7. All these objects have a visual indicator describing what kind of equipment it is. This map is used for controlling individual facilities and its equipment. One example can be that a user can manually close or open a barrier in front of or within a tunnel. Users can also initialize pre-defined procedures such as closing a facility for normal traffic and still be able to let emergency vehicles through.



Vei	Anlegg	Objekt	Alarm	Tidsstempel	Pri	Notis
EV- 16	Vangstunnelen	UA2 Radio...	Tilstand Alarm	2019-09-10 13:26:12	38	
0	Kommunikasjon	OPC Rulle...	Kommunikasjon Feilet	2019-09-10 13:19:23	48	
0	Kommunikasjon	OPC Rulle...	Runteller Stoppet	2019-09-10 13:16:26	48	
FV-544	Halsnøytunnelen	A52 Kom	Kommunikasjon Feilet	2019-09-10 13:10:09	38	
Ev- 39	Mastrafjordtunnelen	Radarson...	Stopp Detektert	2019-09-10 13:09:19	38	
0	Kommunikasjon	OPC Rulle...	Kommunikasjon Feilet	2019-09-10 12:59:22	48	

Figure 3.6: Map view in VegVokteren



Vei	Anlegg	Objekt	Alarm	Tidsstempel	Pri	Notis
EV- 39	Mundalsbergtunnelen	Kamera 15	Stopp Detektert	2019-09-10 13:36:35	36	
EV- 39	Mundalsbergtunnelen	Kamera 17	Stopp Detektert	2019-09-10 13:36:26	36	
0	Kommunikasjon	OPC Rulle...	Runteller Stoppet	2019-09-10 13:36:24	45	
EV- 39	Mundalsbergtunnelen	Kamera 16	Stopp Detektert	2019-09-10 13:36:23	36	
EV- 39	Mundalsbergtunnelen	Kamera 18	Stopp Detektert	2019-09-10 13:35:54	36	
EV- 39	Mundalsbergtunnelen	Kamera 19	Stopp Detektert	2019-09-10 13:35:51	36	

Figure 3.7: Facility overview in VegVokteren

The facility map is split up into different layers, where each layer is responsible for one type of equipment. The main layers are:

- **Emergency Surveillance:** Holds critical equipment such as barriers, signs, and cameras. This layer also contains shortcuts to the pre-defined routines discussed earlier.
- **Ventilation:** Is an overview and control window for the ventilation system present in most modern tunnels. This is mainly controlled automatically based on thresholds set for the CO and NOx sensors that the user can access in this layer.
- **Traffic:** A more detailed overview of the traffic flow and road network in a facility. The facility is divided up into zones and lanes and updates independently from each other based on camera input and speed sensors.
- **Communication:** An overview of the connection status for all access points in a facility. The access points act as hubs between the equipment and the OPC-server.
- **Technical:** Have a lot of same functionality as the communication layer but focuses on power management in a facility.

For as long as every piece of equipment has power, and a connection to the system, an operator is able to control the traffic flow through a facility and its emergency equipment. This functionality combined with the alarm list, which is a module that is present in every window in the application, ensures that operators can respond quickly if something happens at a facility.

In Figure 3.8 we can see a simplified flow sequence of a sensor update that triggers an alarm in the system. After VegVokteren gets notified of the sensor update, the event is stored and evaluated against pre-defined routines and thresholds within the system. If the new value passes a threshold that has a routine linked to it, the routine starts and triggers all attached actuators. In this case, operators are also notified about what has happened, and what procedures that are started. If the system triggers an alarm that does not have a routine linked to it, the operators are notified about the event. Then operators can decide what they have to do at the facility to ensure continued safe traffic flow.

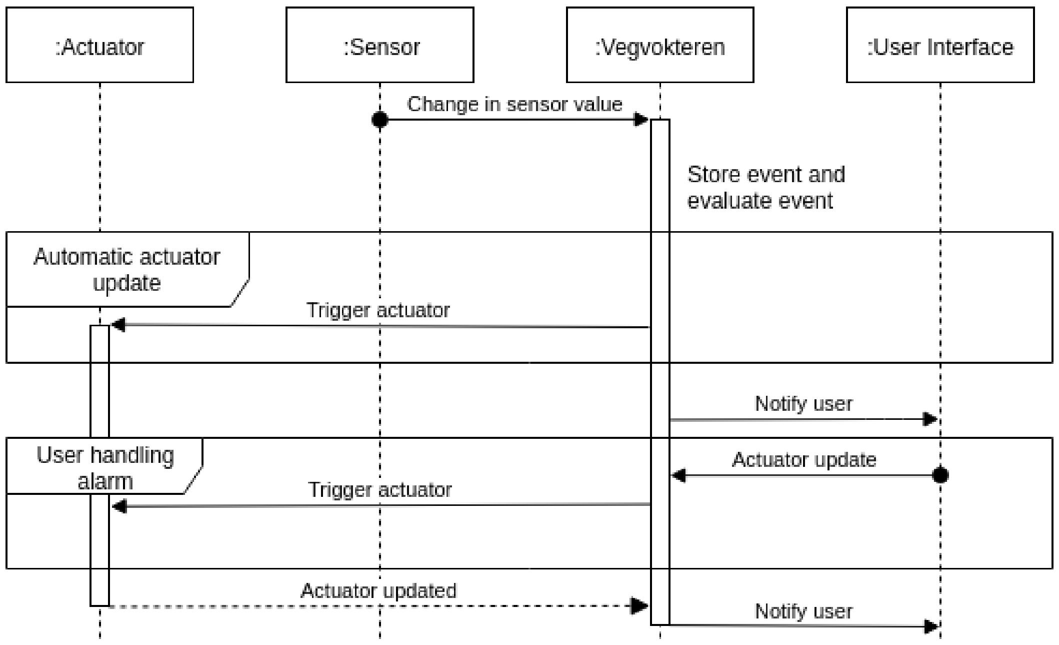


Figure 3.8: Sensor change results in an alarm

3.3.2 Non-Functional Requirements

The main goals for the new architecture are to ensure stability, availability and security [5]. A stable system is always running when it is expected to be running, and handle heavy load in a way that the system stays responsive even in periods with high system load. A system with high availability has a high degree of uptime, can be upgraded while running, and has to be running with limited functionality if parts of the systems goes down.

Scalability

VegVokteren is a system where the system configuration and functionality are constantly growing. It is therefore important that the system architecture is designed to scale. When a component is overloaded, there are two main ways of scaling that component:

- **Horizontal scaling:** Start an identical component in a new instance. This works well when the workload can be preserved in parallel.
- **Vertical scaling:** Allocate more hardware resources to the component. If the workload cannot be run in parallel, this is the only way of scaling up the component.

If the system is running on in-house servers, it can be difficult to scale vertically. New hardware resources have to be bought and the software has to be moved to the new hardware. In the current solution, it is an implicit prerequisite in the architecture that it have to use one database, one real-time engine, and one webserver. This means that you cannot move these without taking the system down, this will cause downtime in the period it takes to move the system to the new hardware.

Horizontal scaling makes it possible to scale to existing hardware, and it can facilitate the addition of new hardware resources without taking already running components down. This is advantageous in a cloud-based system where new instances can be started on-demand, but can also be useful on in-house servers. In the current solution, the only component of the system that can scale horizontally is the OPC-client.

Stability

To achieve a stable system it has to be able to work through a workload of any size without delays or losing data in the process. An unstable system can loose events or process them wrong, or crash entirely if the workload becomes too large. It is more common that large workloads cause delays in a system, which can result in important information gets delivered to late to the operators.

The system stability is supported by the system architecture by separation of functionality into components which has specific tasks. The architectural choices made for the new system have to facilitate that components can be isolated from the system in case of errors which may cascade the errors into other components. The components should also be made to be able to run in parallel over multiple instances, both to support scaling, but also if there is a hardware configuration that makes the component unstable, then it can be moved to a new server without taking the complete system down.

Security

Security is important in any software system. VegVokteren handles sensitive data and it is, therefore, important to control access to the system components and to communicate over secure channels. Since the data being transmitted is sensitive, cryptography should be used to ensure the confidentiality and integrity of the data and to authenticate the data source. It is important to ensure confidentiality and integrity to prevent unauthorized parties to gain access to the data and ensure that the data has not been tampered with during transmission.

It is important to control access to software systems. To achieve access control, authentication and authorization mechanisms have to be implemented. Authentication of users can be done by validating the credentials of a user or system component. Such credentials could be as passwords, personal identification numbers or other authentication factors. This is important to prevent unwanted access to a system. Authorization mechanisms should also be implemented to describe what actions an authenticated entity is allowed to perform in the system. Some users might have access to the entire system, while others are restricted to a subset of the system functionality and components. It is important to control the actions individual entities are allowed to perform within a system to prevent access to parts of the system that an entity does not need or should not be allowed to access.

Chapter 4

Generic Architecture

To test our hypothesis that an event-driven microservice architecture will be more scalable, more efficient and easier to maintain, we are going to implement a prototype that resembles VegVokteren (VV). The system is going to be a simpler and more generic version of VV, but it will exhibit the same data flow. This means that we will, to a degree, be simulating the facilities and their behavior. The data flow from the facilities will be piped into a message-oriented middleware, from which different services can asyncronously pull data. In this chapter, we propose a new system architecture for VegVokteren and explain our design choices.

4.1 Prototype Description

Our prototype consists of four independent components, all of which are explained in detail throughout the chapter. A simple visualization of this can be seen in Figure 4.1. It is created to mimic the current implementation of VegVokteren, but with one major difference: the message-oriented middleware handles all communication between the components.

The prototype is able to parse and process data coming from facilities and notify users if something notable happens at the facilities. These notable events can be that a sensor value has exceeded a threshold, or a manual actuator has been triggered within the facility.

The users are also able to start pre-defined procedures, to close part of or the entire facility. Users are also able to change the status of actuators manually through the command line interface front-end.

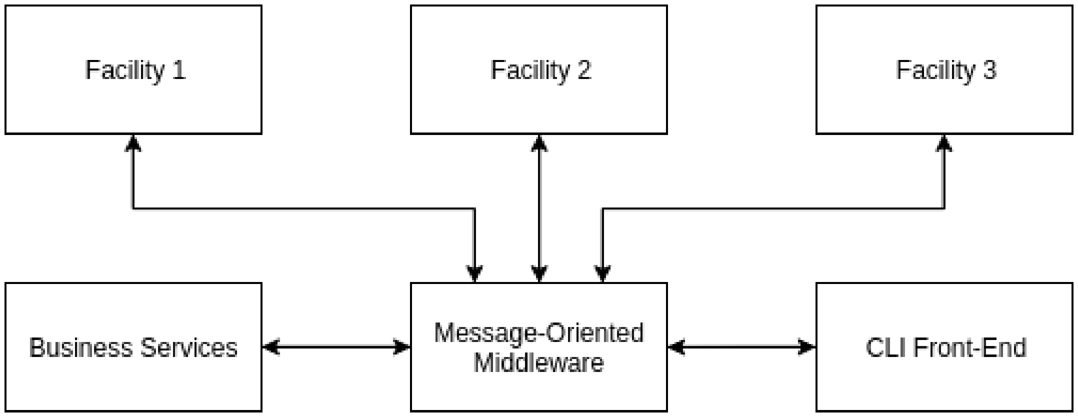


Figure 4.1: Crude architecture overview

As we have discussed in Chapter 3.1, much of the actual processing in VegVokteren happens inside the database. In our prototype, we have moved the processing to the business services component. This will be explained in further detail in Section 4.3, but in short, this is a component that consists of single-responsibility microservices that pulls and pushes data from and to the MOM.

4.2 Facility

We have chosen to simulate facilities instead of connecting our system to the real data stream from VV. This is to ensure that we can test, and reproduce results, with different parameters such as system load and the number of messages the MOM can handle. This also enables us to run a collection of facilities on a computer or server as independent and asynchronous processes, to come as close to VV's facilities as possible. This means that we can control the entire ecosystem, and set up different testing scenarios to test scalability, responsiveness, and performance for the architecture.

A facility is modeled as seen in Figure 4.2, as a rough version of VV's facilities. It has some metadata such as identifier (id), name, type and location, a status which can tell users if there is something that is needed to be done at the facility, and a boolean value which states if the facility is open or closed to traffic. In addition to this, it contains a list of actuators and sensors which can be defined when generating a facility. This enables us to regulate the amount of load we can put on the business services when testing. As an example, we can increase the number of sensors in one or all facilities if we want to investigate what happens to the system when we want to increase the load.

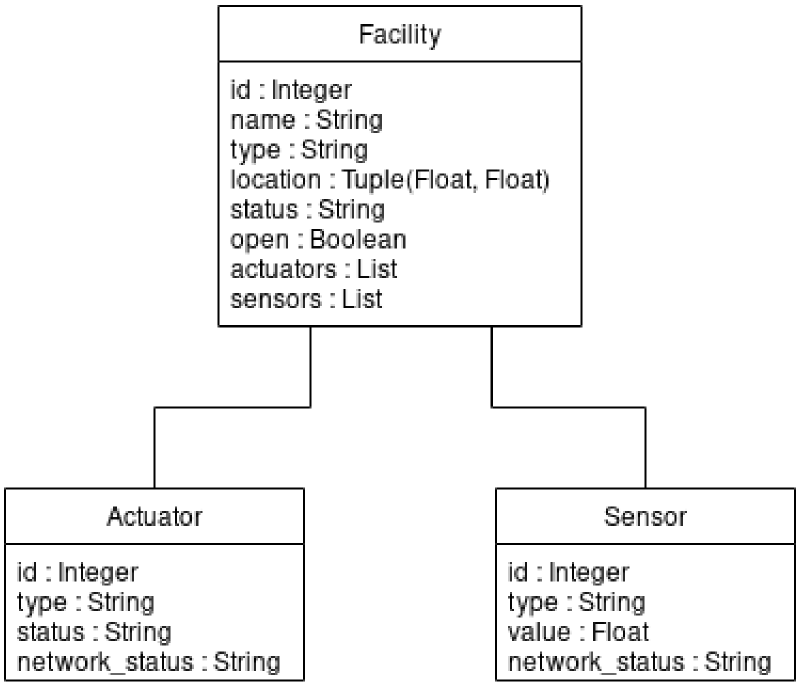


Figure 4.2: Model of a facility for the generic prototype

To run simulations we have made a tool to generate test scenarios, where we can define the number of actuator changes we want to occur within the facility each cycle. A cycle is simply a unit of time that is defined within the facility. When the facility has processed and published data for each of the messages in the simulation cycle, it sleeps for the defined time. We can also define how long a cycle is with this tool, this gives us even more control over our test scenarios. Each facility has its own individual scenario file, where the test scenario is stored. When we want to start a simulation, all the facilities will read their file and change the state of sensors, actuators, and the facility according to the commands it reads. The changes will then be pushed to the MOM, ready to be pulled by the different business services.

4.3 Business Services

The business services component is a collection of single-responsibility microservices that handles processing of the data coming from the facilities. Each service pulls and pushes data as needed from and to the MOM. This also includes communication between different business services. This is to ensure that services can run asynchronously and independently to achieve maximum efficiency.

Each object, i.e, facility, sensor or actuator have its own service that stores historical data and the current state of the object as this is requestable from the front-end CLI. These services provides data retrieval for different requests by the front-end through the MOM.

The other type of services handles the processing and evaluation of incoming data from facilities and commands from the users. If a command to update an actuator status is issued from a user, the system proceeds as shown in Figure 4.3. The command is pushed into a topic in the MOM, which is a service responsible for updating actuators is subscribed to. This service then parses the message into a command that a facility can understand, and passes it back to the MOM. The facility then pulls the message and changes the state of the actuator in question, and sends a message back to the MOM with the new state of it. This is then stored in both the actuator database and in the update service database.

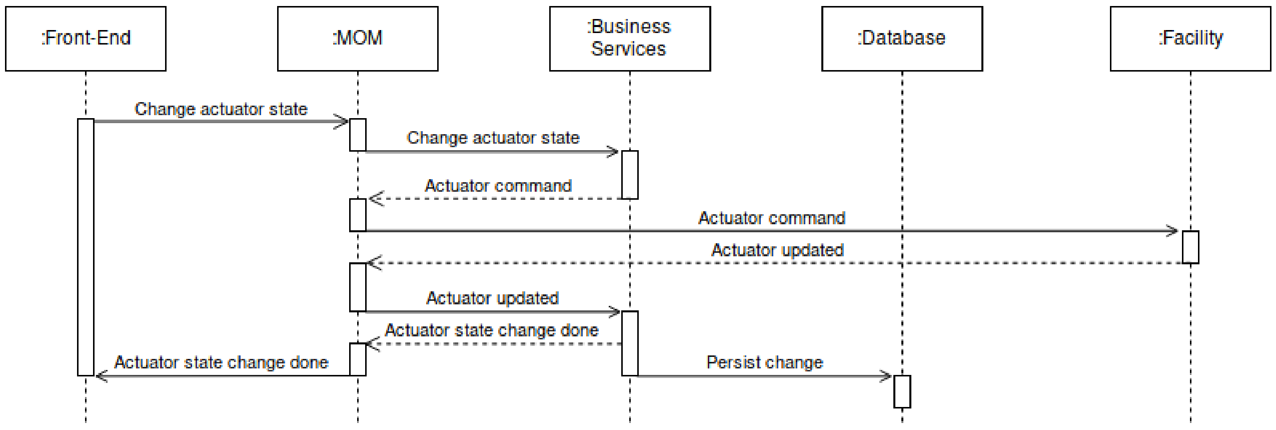


Figure 4.3: Manual change to an actuator system flow

4.4 Message-Oriented Middleware

VegVokteren revolves around interpreting and responding to event messages that are being generated in the facilities. In the current, database-centric solution, OPC clients communicate directly with the database which is tightly connected to the other system components. We propose that a message-oriented middleware (MOM) component is the central component responsible for connecting the different parts of the system. The concept of MOM involves using communication channels and a topology (discussed in section 2.2) to send data between applications and services. The message-oriented middleware should be a fault-tolerant and highly scalable component responsible for receiving, delivering and storing messages. By using MOM for inter-service communication, we obtain an architecture consisting of decoupled services that do not directly interact with each other. As Figure 4.1 illustrates, the MOM has replaced the database as the central component of the system. The event processing flow in our prototype is relatively simple and therefore we chose the broker topology and Apache Kafka as middleware for the implementation. Apache Kafka is a distributed, publish/subscribe, real-time event streaming platform that is highly scalable and fault-tolerant. Kafka is a good fit for our prototype because it provides good default behavior such as automatic load balancing, delivery guarantees and messages are automatically stored in a persistent store in the broker. Using Kafka as a middleware for communications, it is easy to add new microservices to the system. New microservices are connected to the system by either publishing or consuming an existing topic, or by publishing data to a new topic for use in other services. By sending all data through Kafka, the prototype will consist of services complying to the decoupling principle. By utilizing Kafka for all message transmission, services does not communicate directly. The services comprising the system does not know or care about which service is in the other end of the message transmission. They perform their tasks without knowing what the result will be used for or by whom.

4.5 Front-End CLI

In our prototype, we decided to make a command-line interface (CLI) instead of a web front-end. Unlike a graphical user interface (GUI), a CLI contains no graphical options and is simply a text-based interface allowing users to operate a system by typing in commands and responding to prompts. Developing a CLI takes only a fraction of the effort it takes to build a GUI, but it requires the user to have good knowledge of the commands to be able to work effectively. What we refer to as commands in our CLI is, in reality, functions in Python 3 scripts. The CLI enables developers to work quickly and utilize the powers of CLI applications. Using a CLI allows the user to perform repeatable tasks using loops and to use commands in scripts. CLI applications are based on standard I/O operations which make it possible to chain commands together in scripts to perform more complex tasks. Since there is no need to load graphical components, and commands can be processed in batches, CLI applications consume fewer resources than a GUI [14]. Also, a CLI is adequate for what we intend to test in this project.

CLI Style

Since the CLI is based on typing commands, naming the commands is a big part of the usability of the CLI. We chose to use a CLI style inspired by the CLI style guide developed by the Akamai team [45]. They recommend following the principle of least astonishment when naming the commands. The principle reads as follows: ”If a necessary feature has a high astonishment factor, it may be necessary to redesign this feature.” This means that commands should behave as the users expect and they should not be surprised by the system’s behavior. It is important to match the expectations of those who will use the system to decrease the chance of misunderstandings leading to misuse [42][22]. Our CLI style uses command names that describe the action it performs. An example of a command would be:

```
$ python3 alarm-list.py
```

The name `alarm-list` implies what the script returns; a list of all active alarms. Another important aspect to think about when designing a CLI is documentation. It is difficult to use a CLI for new users, especially if the documentation is of bad quality or lacking. The CLI will provide documentation for the alarm listing script by running the following command:

```
$ python3 alarm-list.py --help
```

We decided to use flags over arguments as the input of our script. Flags are optional, unordered and suitable for our system since there are no requirements concerning input parameters to our commands. Our commands have good defaults if flags are not provided which makes the CLI more user-friendly. A comparison between using arguments and flags when requesting an alarm list filtered by facility ID and sensor ID can be seen in listing 4.1. Using arguments could be confusing to the user if it is unclear what role the arguments play. Flags can be specified in any order, makes the command easy to understand, and ensures the user that the command is being run correctly. It is important to describe all flags so that the users receive the help they need when providing the `--help` flag/option.

Listing 4.1: Comparison between Flags and Arguments

```
# using arguments
$ python3 alarm-list.py 1 2
# using flags
$ python3 alarm-list.py --facility-id 1 --sensor-id 2
```

The example using flags in listing 4.1 shows how flags are used to provide facility identifier and sensor identifier to our commands. The command can be run without both flags in which case the command will return all alarms. If the command is run without one of the flags, for instance, sensor-id, the command will return all active alarms associated with the provided facility identifier.

Chapter 5

Implementation

In this chapter, we present the implementation of the prototype and explain how Apache Kafka was set up and utilized by the system components. We go into detail on the microservices that constitute the prototype and the methods and technologies used to implement them. The prototype is based on the architecture proposed in chapter 4, and in chapter 6 we show how the methods and technologies of choice make it possible to meet the requirements of the system. Figure 5.1 gives an overview of the data flow in the system. The flow starts in the facilities where sensors generate data which is sent to the Kafka topic `sensor_data`. The alarm interpreter service polls the sensor data from the topic, interprets it to detect alarms and stores alarms in a database before the data is published back to Kafka on the `alarms` topic. The CLI continuously polls the `alarms` to be able to list active alarms to the users.

5.1 Apache Kafka

The prototype that we developed revolves around reliable messaging through Kafka. If properly configured, Kafka is a fault-tolerant and secure messaging system that provides high throughput and guaranteed delivery of system events. In this section, we explain how the Kafka cluster was set up and configured, and how message confidentiality and integrity was assured.

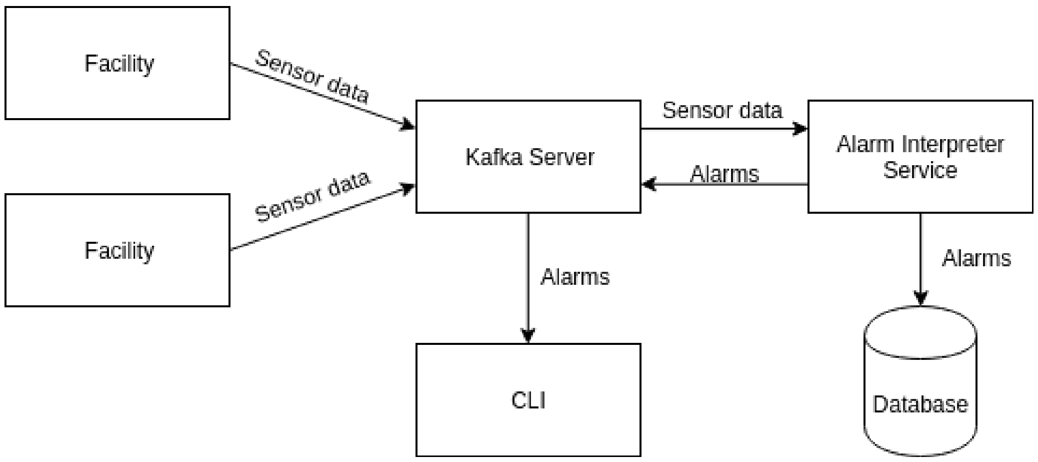


Figure 5.1: Overview of the data flow

Kafka Setup and Configuration

We decided to run our Kafka cluster on the Google Cloud Platform (GCP) because we needed access to it from multiple devices and to avoid running the Kafka and Zookeeper servers locally on our own limited hardware. The Kafka brokers were set up by launching a Kafka virtual machine (VM) image in the Google Compute Engine. When starting the compute engine instance, the Zookeeper and Kafka servers start automatically and is ready to be accessed via ssh. To be able to connect to the brokers remotely, a couple of configurations are necessary. Initially, the VM has one broker running in addition to the Zookeeper server. Port 9092 and 2181 was opened in the Google Compute Engine instance because they are the default ports that the Kafka and Zookeeper servers listen to, respectively. Next, we had to configure the broker to listen to its public network interface and specify the hostname and port it should advertise to producers and consumers. We assigned an external IP address to the virtual machine (VM) instance and set the `advertised.listeners` property of the Kafka server properties file to the chosen hostname-port combination. This leads to the Kafka broker being publicly available, listening on port 9092. The combination of hostname and port (hostname) is used by Kafka clients when connecting to the cluster.

The publicly available Kafka broker constitutes a cluster of size one. In our prototype, we set up a cluster comprised of three Kafka brokers. Since the prototype is running in a development and testing environment, we decided to run all our brokers on the same VM instance. In a production environment, the brokers should be run on different machines or VMs in case the machine goes down. Setting up two additional brokers in the same instance is simple and requires only a few steps. The first step is to copy the Kafka broker properties file as shown in listing 5.1.

Listing 5.1: Setting up configuration files for two additional brokers

```
$ cp /opt/kafka/config/server.properties /opt/kafka/config/server-1.properties
$ cp /opt/kafka/config/server.properties /opt/kafka/config/server-2.properties
```

The next step is to configure some of the properties in the files to separate them from each other. The broker ID is a unique identifier and is set to 0 in the original broker property file. Broker 1 and 2 are given IDs 1 and 2. They are configured to listen to ports 9093 and 9094, respectively. The brokers are also assigned separate files for storing their logs. Having multiple brokers configured and running enables us to create topics and replicate topic partitions across some of, or all of the brokers. The replication factor specifies how many replicas there are of each topic partition in the cluster. We configured our topics to have a replication factor of three and have three partitions. This way, all three topic partitions are replicated across the three brokers in our cluster, allowing two brokers to be down at the same time. Having three partitions also means that three consumer instances from the same consumer group can process messages in parallel. If there is a need for more consumers to consume a topic, more partitions can be added.

Producers and Consumers Configuration

The producers in our prototype system are all idempotent to ensure that messages are always delivered without duplicates. This means that the producers wait for acknowledgments from all brokers, retry sending messages until they are successfully received by the broker, and have a maximum of 5 unacknowledged requests. This ensures that messages are delivered exactly once. As discussed in section 2.3, batching messages can improve performance and increase throughput. In our prototype, we set the batch size of the producers to 32kB and we configured them to wait for 5 ms before sending a batch (if not full). This results in fewer requests sent and processed, and the introduced delay adds up to less than the sum of the transmission time of each request that would have been if the messages were sent individually. We also enabled compression to get smaller request sizes for faster data transfer and to save external storage. Combined with batching, compression increases the efficiency of message transmission and increases throughput.

The consumers were configured to mark the last consumed offset. In this way, the consumer can continue reading from the latest checkpoint to get data that were missed due to for instance a failure. This way the consumers do not have to read the data prior to the checkpoint, which will make the system more efficient. Setting checkpoints for every event ensures that no messages are lost, but it will impact the performance. The consumers of our system was configured with a checkpoint for every 200 messages. This results in lower performance impact as well as ensuring that messages will not be lost.

5.1.1 Kafka Security

The setup explained above has a few problems when it comes to security. By default, the data being sent between a client and the Kafka cluster is fully visible on the network as it is sent as plain text. Also, any client can access the cluster to produce and consume data on any topic. In our prototype, we implemented security mechanisms for encrypting messages during transmission, authentication, and authorization. Implementing these security mechanisms ensures secure communication between the microservices in the system. Kafka security is fairly new and the setup is complicated. The built-in security features came with version 0.9 (November 23, 2015) and have been continuously improved since [23]. We now take a look at how we implemented the security mechanisms in our prototype.

Encryption

The first step towards encryption is the server-side setup. An overview of the encryption setup can be seen in figure 5.2. Since our project is private, we chose to create our own Certificate Authority (CA) and use self-signed certificates instead of using a third party CA. The CA is located on the Kafka server in a sub-directory to separate it from the other configuration. We used OpenSSL which is a toolkit for the TLS and SSL protocols to create our CA [47]. To create the CA, we used the OpenSSL command to request a new private RSA encrypted key of length 4096 bits and a public key as shown in line 1 of listing 5.2.

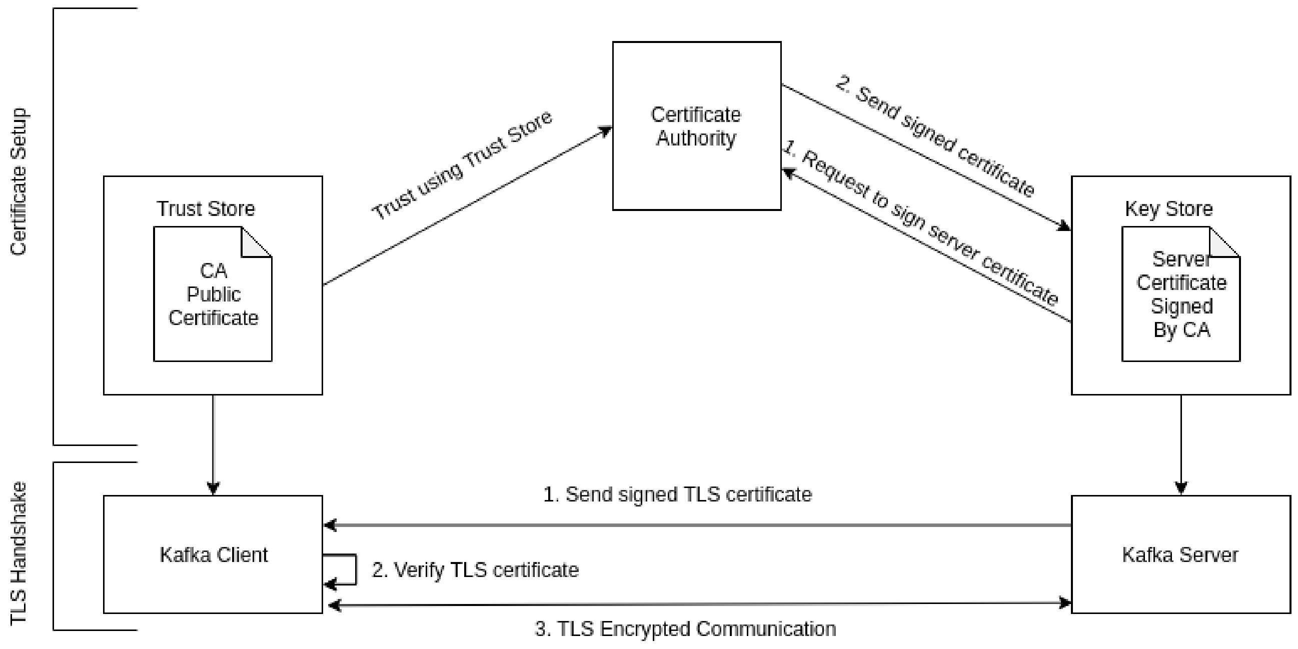

Certificate Setup
TLS Handshake
Certificate Authority
Trust using Trust Store
2. Send signed certificate
1. Request to sign server certificate
Trust Store
CA Public Certificate
Key Store
Server Certificate Signed By CA
Kafka Client
Kafka Server
1. Send signed TLS certificate
2. Verify TLS certificate
3. TLS Encrypted Communication

Figure 5.2: Encryption overview

Listing 5.2: The commands used to enable encryption

```
$ openssl req -new -newkey rsa:4096 -days 365 -x509 -subj
  ↪ "/CN=Kafka-Security-CA" -keyout ca-key -out ca-cert -nodes

$ keytool -genkey -keystore kafka.server.keystore.jks -validity 365
  ↪ -storepass $BROKERPASS -keypass $KEYPASS -dname "CN=<Hostname of
  ↪ the Google Compute Engine instance>" -storetype pkcs12

$ keytool keystore kafka.server.keystore.jks -certreq -file cert-file
  ↪ -storepass $BROKERPASS -keypass $KEYPASS

$ openssl x509 -req -CA ca-cert -CAKey ca-key -in cert-file -out
  ↪ cert-signed -days 365 -CAcreateserial -passin pass:$BROKERPASS

$ keytool -keystore kafka.server.truststore.jks -alias CARoot -import
  ↪ -file ca-cert -storepass $BROKERPASS -keypass $KEYPASS -noprompt

$ keytool -keystore kafka.server.keystore.jks -alias -CARoot -import
  ↪ -file ca-cert -storepass $BROKERPASS -keypass $KEYPASS -noprompt
$ keytool -keystore kafka.server.keystore.jks -import -file cert-signed
  ↪ -storepass $BROKERPASS -keypass $KEYPASS -noprompt

$ scp -i ~/kafka-security.pem <username>@<hostname>:<path to certificate>
  ↪ .

$ keytool -keystore kafka.client.truststore.jks -alias CARoot -import
  ↪ -file ca-cert -storepass $CLIPASS -keypass $KEYPASS -noprompt
```

The `ca-key` is the private key of our CA while `ca-cert` is the public key. The keys have a validity of one year. Next, we used the keytool command to generate the Kafka broker certificate and store it in a password protected server key store as seen in figure 5.2. Line 3 in listing 5.2 show the command used to achieve this. The command creates the certificate and stores it in the password-protected java keystore `kafka.server.keystore.jks`. The certificate has a validity of one year.

What we did next was to get a signed version of the generated certificate so that clients can verify the validity of the certificate. Getting the signed version consists of two steps. First, we used the `keytool` command to extract the certificate that we wanted to sign from the key store and used it to create a signing request named `cert-file` (line 5 in listing 5.2). The next step was to get the CA-signed certificate from the signing request which is shown in line 7 of listing 5.2. The signing request was used to get a signed certificate (`cert-signed`) from the CA as illustrated in figure 5.2. The `keytool` command was then used to create a trust store on the Kafka server and to import the CA public certificate into it to make our brokers trust the certificates signed by our CA. Line 9 of listing 5.2 show how the `keytool` command was used to create the trust store and import the public CA certificate. Next, the commands shown in lines 11 and 12 of listing 5.2 was used to import the public CA certificate and the signed server certificate into our Kafka server key store. We added the location of the key store and the trust store, with corresponding passwords, to the properties files of each of the Kafka brokers. It was also necessary to modify the listeners and advertised listeners properties to enable SASL (Simple Authentication and Security Layer) TLS. This was the last step in the setup of the server-side trust store and key store, so now we could continue with the configuration of the Kafka broker.

Now that the Kafka server was configured, we could continue with the client-side TLS setup. As we did on the server, we created a directory for TLS configuration to separate it from the other configuration. The clients also needed a trust store so that our clients can verify the TLS certificate from the Kafka server endpoint. We use a "chain of trust" version of setting up the trust store for our clients. This involves copying the public CA certificate into our clients' trust stores so that our clients can trust all certificates signed by the CA. The secure copy command was used to copy the CA certificate from the CA (located in the Google Compute Engine VM) as shown in line 14 of listing 5.2. The `keytool` command in line 16 of listing 5.2 was then used to create a password protected Kafka client trust store and to import the CA certificate. Next, we created a properties file which will have to be provided as an additional parameter by the Kafka clients when connecting to the Kafka cluster. The contents of the file specify the security protocol (TLS), the location of the client trust store, and the password to the trust store. This concludes our setup TLS. A TLS handshake, as seen in figure 5.2, is performed between a Kafka client and the Kafka server before data is transmitted. Encryption is enabled and messages can be sent over a secure channel (see section 2.3).

Authentication

The authentication workflow starts with the client presenting its credentials to authenticate to the 3rd party Key Distribution Center and request a ticket, illustrated in step one and two in figure 5.3. The KDC issues a ticket if the authentication is successful. If a ticket was issued, the client can connect to the Kafka server as shown in step three of figure 5.3. The key characteristics of the authentication workflow are that all communication is encrypted, no passwords will be transmitted and that tickets will be automatically renewed by the clients if their credentials are valid. Once the clients are authenticated in a Kafka cluster, only one

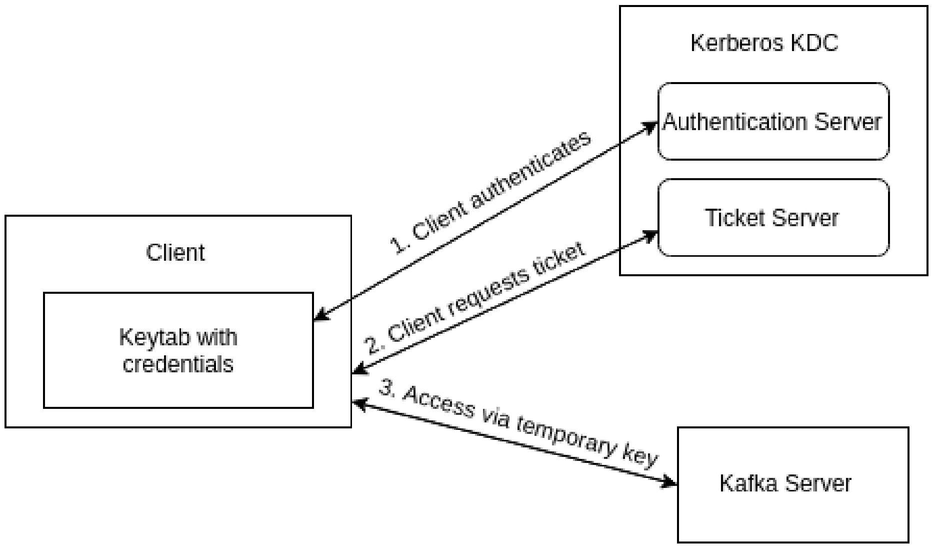


Figure 5.3: Kerberos Authentication workflow

step remains before secure communication is achieved. Authentication needs to be combined with authorization so that Kafka knows what actions different clients are allowed to perform on the cluster. To authorize clients, Kafka uses access control lists (ACL) which is a list of rights associated with the clients [52].

For authenticating clients in the Kafka cluster, we used the Simple Authentication and Security Layer (SASL) and the GSSAPI protocol as mentioned in section 2.3. We launched a new virtual machine in the Google Compute Engine to host our Kerberos server. Port 88 (default for Kerberos KDC) was opened for incoming connections from the clients' IP addresses and the Kafka server VM instance's internal IP address. On the VM instance, we installed the Kerberos server. The next step was to edit three configuration files. First, we specified a default realm as shown in listing 5.3, lines 1 to 7, in the `kdc.conf file`. The name of the realm on line 5 was changed from `EXAMPLE.COM` to `KAFKA.SECURE`. The default properties of the realm (would be specified between lines 5 and 7) are not modified.

Listing 5.3: Content added to the kcd.conf file

```
 1 [kdcdefaults]
 2   default_realm=KAFKA.SECURE
 3
 4 [realms]
 5   KAFKA.SECURE = {
 6     ...
 7     }
 8
 9 [libdefaults]
10   default_realm=KAFKA.SECURE
11   kdc_timesync = 1
12   ticket_lifetime = 24h
13
14 [realms]
15   KAFKA.SECURE = {
16   kdc = <hostname of the Kerberos server>
17   admin_server = <hostname of the Kerberos server>
18   }
19
20 $ sudo kdb5_util create -s -r KAFKA.SECURE -P <password>
21
22 $ sudo kadmin.local -q "add_principal -pw <password> admin/admin"
23 $ sudo kadmin.local -q "add_principal -randkey r@KAFKA.SECURE"
24 $ sudo kadmin.local -q "add_principal -randkey w@KAFKA.SECURE"
25 $ sudo kadmin.local -q "add_principal -randkey rw@KAFKA.SECURE"
26
27 $ sudo kadmin.local -q "add_principal -randkey
     ↪ <username>/<hostname>@KAFKA.SECURE"
28
29 $ sudo kadmin.local -q "xst -kt <path>/admin.user.keytab
     ↪ admin@KAFKA.SECURE"
30
31 $ kinit -kt <path>/admin.user.keytab admin
32
33 $ kafka-console-producer.sh --broker-list <hostname>:<port> --topic
     ↪ kafka-security-topic --producer.config <path to Kerberos properties
     ↪ file>
34 $ kafka-console-consumer.sh --bootstrap-server <hostname>:<port> --topic
     ↪ kafka-security-topic --consumer.config <path to Kerberos properties
     ↪ file>
```

The next configuration file we had to edit was the `kadm.acl` file. This file contains only one line by default which specifies a principal naming pattern for our admin principals. We configured it so that any principal matching the pattern "*<username>/admin@<realm>" is treated as an admin in the system. The last file we edited was the krb5.conf file. In this file we replaced the `[libdefaults]` and `[realms]` properties with the cofiguration shown in listing 5.3 on lines 9 to 18. The `kdc_timesync` property maintained its default value. The file, in addition to what is shown in the listing, contains information specifying where the different logs are stored. Next, we used the tool `kdb5_util` to create our Kerberos database for our realm. The command is seen on line 20 in listing 5.3. Now we used the client tool, called `kadmin.local` to create admin principal matching the naming pattern specified in the `kadm.acl` file. We also created user principals for other clients of the system. The creation of the password-protected admin principal `admin/admin@KAFKA.SECURE` is shown in listing 5.3 on lines 22 to 25 as well as how to create some example principals for normal clients that are not password protected. We also needed to create a principal for our Kafka service matching the name pattern `<username><hostname>@<realm>` where hostname is the hostname of our Kerberos server. To create a service-based principal, we used the command on line 27. The next step was to export each of our principals into keytab files of their own by using the `xst` command in our query. Line 29 in listing 5.3 demonstrates how this was done using the admin principal as an example. The command creates a keytab file in the specified location and exports the principal `admin@KAFKA.SECURE` into it. What we did next was to download the four user keytab files to our local clients and the Kafka service keytab file to the Kafka server. Each user will have a keytab file, which should be accessible only for the owner because it contains a user's identity.

To be able to request tickets, we installed the Kerberos client tools on all clients and edited the `krb5.conf` file the same way as on the Kerberos server (lines 9 to 18 in listing 5.3 lines). We were now ready to test if we were able to get a Kerberos ticket and confirm that a ticket was issued by calling the command `klist` (shown on line 31 of the listing) which will list all Kerberos tickets. We also installed the Kerberos client tools on the Kafka server and edited the `krb5.conf` file to like before. After that, we confirmed that we were able to request and receive a ticket like we did on the clients.

The brokers were already configured to utilize SASL TLS, but the GSSAPI needed to be enabled and we needed to specify the Kerberos service name to be the same as the Kafka service principal name. We applied these configurations by editing the server properties files on the Kafka server. We now created a file, called a `JAAS` file, that lets Kafka know how to perform SASL authentication by enabling the use of the Kafka service keytab file, the path to the keytab file and to use the Kafka service principal. The path to the security configuration (`JAAS file`) was set as an environment variable so that the JVM can access the configuration.

The clients also needed a `JAAS` file to be able to utilize SASL authentication. In the file, we specified that the Kafka clients require the Kerberos login module and that they should use the ticket cache to authenticate using tickets. Next, we created a Kerberos properties file for our clients where we specified all the necessary properties to use the Kerberos TLS enabled the setup. The file contains the security protocol (SASL TLS), the Kerberos service name, the location of our trust store, and the password to the store. We gave the JVM access to the security configuration by setting an environment variable as we did on the Kafka server. Clients that had been granted a ticket could now authenticate to the Kafka server and produce and consume any topic by providing the Kerberos properties file as shown in listing 5.3 on lines 33 and 34. If a consumer or producer is created without a valid ticket in the client's ticket cache, a Kafka `LoginException` will be thrown and the client will have to request a new ticket from the Kerberos server. This concludes the complicated Kerberos SASL authentication setup.

Authorization

The Kafka clients and Kafka server can now authenticate to each other, but we still need to define the permissions of our clients. For that, we used ACLs. ACLs can be applied on three levels. Topic-level ACLs restrict which users can read and write data to the topics of our Kafka cluster, ACLs for consumer groups restrict which client can use a specific consumer group, and cluster-level ACLs defines which clients can create, delete or edit topics. To be able to authorize clients, we need to configure our Kafka brokers to support ACLs. We added the following properties to the server properties files on the Kafka servers:

```
authorizer.class.name=kafka.security.auth.SimpleAuthorizer
super.users=User:admin;User:kafka
allow.everyone.if.no.acl.found=false
security.inter.broker.protocol=SASL_SSL
```

Superusers in Kafka are authorized to perform any operation on the Kafka broker without the need for an ACL. The `super.users` property was configured to treat the admin principal and the Kafka service principal as super users. The inter broker protocol specifies the protocol used for communication between brokers and is set to SASL_SSL. This property is not directly related to enabling authorization, but it needed to be set to `SASL_SSL` to make sure that the inter broker communication uses Kerberos for authentication. `SASL_SSL` uses TLS in practice, but Kafka uses SSL to refer to encryption between the clients and the servers. Having created our principals and topics, we could now create the ACLs. ACLs for granting reading permissions to the user principals `reader` and `writer` is created using the `kafka-acls.sh` script from the Kafka CLI as shown in listing 5.4.

Listing 5.4: Creating ACLs for reading access

```
$ kafka-acls.sh --authorizer-properties zookeeper.connect=<hostname>:2181
  ↪ --add --allow-principal User:reader --allow-principal User:writer
  ↪ --operation Read --group=* --topic <topic>
```

The authorizer property `zookeeper.connect` was set to a combination of the public DNS of our Zookeeper server, which in our case is the same as the Kafka server, and port 2181 (default for Zookeeper). This is because the ACLs are managed by Zookeeper. We granted user principals `reader` and `writer` reading access to the specified topic. The group property indicates which consumer group the client should commit offsets to. In our example, the consumer will not commit offsets to a consumer group since a group is not specified. Running the script in listing 5.4 creates ACLs for the topic and the consumer group. The same script can be used to grant writing access to the topic by replacing `-\--operation Read` with `-\--operation Write` and omitting the group property since the group property only applies to consumers. We created ACLs for all clients in the system. A ticket is issued to a specific principal stored in the local keytab file, and access is based on the ACL(s) that exists for that principal. If a client attempts to perform an operation it is not authorized for, it will result in a `TopicAuthorizationException` since we configured the Kafka broker to not grant access if no ACL is found. Superusers, though, only need a valid ticket to be able to perform operations on the Kafka cluster. This concludes the complicated process of setting up Kafka security. Authorized clients can now produce and consume data which is encrypted during transmission.

5.2 System Components

This section provides an overview of the different components that are implemented to represent a simpler and more generic architecture and system of VegVokteren (VV). As we discussed in the previous chapter these components are implemented as microservices, each with their well-defined responsibility. The different components are divided up into three groups, each of which are described in this section. The separation of the components have no functional implications on the system, but we found that it was easier to implement, modify and tests the system if the components are divided up into groups.

Kafka in Python

All of the components are implemented in Python 3. The reasoning behind this is to show that with a microservice architecture, how developers can build a system as a polyglot application. This also enables us to try out different frameworks for Kafka, where we chose Kafka-Python. Kafka-Python is a client for the Apache Kafka distributed stream processing system [39]. It is a high-level client that is designed to function similar to the official java client. In Listing 5.5 and 5.6 we can see that initialization of both the consumer and producer is fairly easy. The difference between the producer and the consumer is mainly that the producer does not need a topic to initialize. This is because the producer can be used for multiple topics, as the producer's *post*-function takes a topic name as one of its input arguments. The `value_deserializer` variable in both of the initializations is how the message is serialized, and this enables us to serialize objects as dictionaries in Python.

Listing 5.5: Consumer initialization

```
from kafka import KafkaConsumer
import json
consumer = KafkaConsumer(
    consumer_topic
    bootstrap_servers="0.0.0.0:9092"
    group_id="update_service"
    value_deserializer=lambda x: loads(x.decode('utf-8'))
)
```

Listing 5.6: Producer initialization

```
from kafka import KafkaProducer
import json
producer = KafkaProducer(
    bootstrap_servers="0.0.0.0:9092"
    value_deserializer=lambda x: dumps(x).encode('utf-8')
)
```

In Listing 5.7 we can see that the consumer polls new messages from the broker. This will acknowledge all the messages polled with the `group_id` defined in the consumer. This means that if there is another consumer with the same `group_id` it will not pull the same messages. The loops are there to ensure that we are working with a single message at the time.

Listing 5.7: Get messages from a topic

```
data = consumer.poll(timeout_ms=100)
for _, messages in data.items():
    for message in messages:
        try:
            #Process the message
        except ValueError as e:
            #Log error
```

Facility

A facility runs on a computer as a seperate process. This means that we can run multiple facilities asynchronous on a machine, and they will generate and send data to the Kafka broker. When a facility starts, it begins to run through a generated data file that simulates the sensors changing values. It then converts the change in sensor data to a JSON format and publishes this to the sensor_data topic in the Kafka broker. A facility is also able to receive commands through a consumer within the facility. The command is parsed and the appropriate actuator is updated. Once this is done the facility publishes a new message to the broker to notify other parts of the system that the actuator change has been completed.

The facilities run through the data file in cycles, where a cycle consists of a pre-defined number of sensor updates. The length of a cycle is defined in the implementation of the facility, and for testing, this is set to one second. This means that we can control how many messages each facility is sending to the Kafka broker each second, and as a result of this, we can control the load of the system.

Business Services

Business services refer to the processing and routing services for the system. The group consists of an update service, which handles commands from the front-end and passes it out to the facility in question, and an alarm interpreter service, which is explained in further detail in a later Section 5.3.

The update service is subscribed to a topic in the Kafka broker which the front-end produces messages to. These messages consist of which facility the command is going to, what actuator it is going to change, and what the new status of the actuator should be. It then parses the message to the message format the facility expects and sends the information to a topic the facilities are subscribed to. The service stores the message in its database, so commands can be retrieved at a later stage, and to see where changes to facilities have been made.

State and Storage Services

To retrieve information about the state of different facilities, actuators and facilities we have implemented simple services for each of these. Each of them keeps track of their data, and stores it independently. This enables a user to see historic data about a facility, or a range of sensors or actuators. Users are also able to retrieve the current state of one or all of the facilities when that is required.

5.3 Alarm Interpreter Service

The Alarm Interpreter Service is the service responsible for processing the data streams generated by the Facility objects. The service both consumes and produces data to the Kafka cluster and is developed with Java and Spring Boot. Spring Boot is an open-source framework which uses software project management tools, Maven in our case, for building projects and adding other frameworks and APIs as dependencies. Automatic configuration of dependencies does a lot of work for the developers. By for instance providing the dependency 'spring-boot-starter-web', Spring Boot starts and configures an embedded Tomcat web server and deploys the application to it every time the application is packaged and run. Developers do not have to go through the process of deciding on a web server, configuring it, and organize a deployment process. Spring Boot automatically configures the microservice-based on the dependencies of the project. In earlier versions of spring, the auto-configuration was enabled by using the class-level `@EnableAutoConfiguration` annotation in the class running the main method. Annotations in spring boot are used to enable interfaces and also to configure the application instead of using XML configuration files. An example on how to use annotations to configure a Spring Boot application can be seen in listing 5.8.

Listing 5.8: Example of how to use annotations in Spring Boot

```java
@SpringBootApplication
@EnableDiscoveryClient
public class AlarmServiceApplication{

  public static void main (String[] args){

    SpringApplication.run(AlarmServiceApplication.class, args);
  }
}
```

The 'spring-boot-starter-web' dependency adds the Tomcat and Spring MVC jars to the project, and Spring Boot configures the application as a web application based on an assumption made by looking at the dependency. The framework loads and configures the dependencies in a standard way so that developers do not have to spend time configuring the dependencies unless the standard configuration is not adequate. Spring Boot has a project initializer [40] which lets developers select the required dependencies and initialize a Spring Boot project.

The consumer class of this service, `SensorDataConsumer`, is annotated with the `@CommadLineRunner` annotations which enables CommandLineRunner interface. The interface is described in the documentation [1] as a: "Interface used to indicate that a bean should run when it is contained within a SpringApplication. Multiple CommandLineRunner beans can be defined within the same application context and can be ordered using the Ordered interface or `@Order` annotation.” The interface contains a run method which is executed after the application context is loaded. In this class, we initialize and configure our Kafka consumer. The consumer is subscribed to the `sensor_data` topic and continuously polls data from it. If there is any data available on the topic, the methods return immediately. We set the timeout of the `poll()` method to 50 ms which means that if there are no immediate data available, the consumer will wait for 50 ms and then eventually return an empty recordset. Each record of a consumed recordset is parsed to a JSON object before some simple processing rules are applied. If a record proves to be an alarm, the database is queried for an entry with the same sensor ID. If the query returns an entry it means that the sensor has sent an alarm previously. Every alarm stored in the database has a field saying if it is still active or has been dealt with. If the returned entry is not active, the status is set to active. Otherwise, nothing is done. If there is no entry in the database with the same sensor ID, the alarm is stored. Spring Boot provides JPA support by adding the ’spring-data-jpa’ dependency to our project. By creating a repository class extending `JpaRepository <Alarm, long>`, we can perform generic CRUD database operations on Alarm objects. Listing 5.9 shows the repository class.

Listing 5.9: The AlarmRepository class

```java
public interface AlarmRepository extends JpaRepository<Alarm, long> {

  Alarm findBySensorId(long sensorId);

  @Query("SELECT alarm FROM Alarm alarm WHERE alarm.active = :active AND
      ↪ alarm.published = :published")
  List<Alarm> findActiveNotPublished(@Param("active") boolean active,
                                     @Param("published") boolean published);
}
```

The repository class will be detected by the Spring Data JPA repository infrastructure which will then create a Spring bean for it [2][3]. We use dependency injection to inject a repository object into our `SensorDataConsumer` class. Injection in our case is the passing of the dependency (our repository class) to the `SensorDataConsumer` class which is done by the Spring Boot container (application context). The `SensorDataConsumer` define its dependencies the container deals with creating the bean for the class [9].

Listing 5.10: Injection of the Kafka producer class

```
@Autowired
AlarmProducerWithCallback producer;
```

We use a Kafka producer to publish all active alarms to a Kafka topic `alarms`. The producer is configured to be idempotent (see section 2.3), and apply compression and batching 2.3. The producer is injected into the `SensorDataConsumer` class by using the `@Autowired` annotation as demonstrated in listing 5.10. By using the annotation, the application context takes care of instantiating the object at runtime and maintains its state. The producer published all active alarms in the database the topic every time a new alarm is detected and persisted in the database.

5.4 Command-Line Interface

As explained in section 4.5, Python 3 scripts constitute our CLI. We chose to implement the CLI in Python because Python provides multiple clean and simple packages for developing CLIs. We chose the Click package [46] which enables developers to develop powerful CLIs quickly and with little code.

Alarm List Command

Listing 5.11 shows an example of a Click command. Arguments and flags are declared in decorators (as seen in listing 5.11) making them accessible as parameters for the CLI functions.

Listing 5.11: Command for listing alarms using Click

```
@click.command()
@click.option(
    '--facility-id',
    default=None,
    help='List alarms associated with the facility with id 'facility-id''
)
@click.option(
    '--sensor-id',
    default=None,
    help='List alarms associated with the sensor with id 'sensor-id''
)
def list_alarms(facility_id, sensor_id)
```

Click provides developers with the option of providing defaults for missing arguments. In our example, the default values are 'None'. When the defaults are passed as for both parameters, the function will return all active alarms. If one parameter is not specified while the other is, the function will return all alarms matching the specified parameter. This means that if only the parameter `facility-id` is specified, the returned list will contain all alarms associated with that specific facility with no regard to the sensor ID.

Kafka Client

As we saw in section 5.3, the alarm interpreter service publishes active alarms to the Kafka topic `alarms`. Our front-end service contains a Kafka client script that polls a list of Kafka consumer records. The consumer records contain a timestamp of when it was created which is used by the Kafka client to calculate the elapsed time of each alarm to see if it meets the requirements of the system. The elapsed time and the values of the consumer records are properties of `Alarm` objects that are created and persisted in the database. The database at all times contains all active alarms and it is used by the `list_alarms` function seen in listing 5.11 to display alarms. The Kafka client also parses the object's string representation to JSON and publishes it to another topic. This is done to make the elapsed time available to all clients in the system so that it can be used for evaluating the system's performance.

Chapter 6

Prototype Validation and Experimental Results

In this chapter, we explain how our prototype explores some of the limitations of the current software architecture of VegVokteren. We validate our Kafka cluster setup and configuration to investigate whether it is fault-tolerant, and we demonstrate how it can be scaled. We show how to scale the components of our prototype, using the alarm interpreter microservice as an example. The fault tolerance of the microservices will also be demonstrated. Additionally, we discuss how our architecture improves the extendability compared to the current solution. We also discuss the performance of the prototype based on tests we conducted,

6.1 Scalability Analysis

To be able to scale any component of the prototype, we need to configure the Kafka topics. The way scaling in Kafka works is by adding more partitions to a topic. Having a reasonable replication factor will also help because it will result in more brokers being leveraged since the partitions will have different brokers as leaders. As we have previously discussed, the number of active consumers from the same consumer group can not be larger than the number of partitions of the topic. This means that if there is a need to scale the system when the number of consumers is equal to the number of partitions, the topic being consumed will have to be extended with additional partitions.

Listing 6.1: Adding partitions to a Kafka topic

```
$ kafka-topics.sh --alter --zookeeper <ip>:2181 --topic <topic> --partitions 4
```

The script in listing 6.1 is a part of the Kafka CLI and will add partitions to the specified topic. There are a few things to consider before adding partitions to a topic. If the data streams in question are relying on key-based ordering, adding more partitions will cause the ordering to be broken since the partition allocation algorithm is a function of the number of partitions (discussed in section 2.3). A topic can not be altered to have fewer partitions, it is only possible to increase the number of partitions.

To demonstrate that we are able to scale individual services in our prototype we ran three brokers, having topics with a replication factor of three, and using three partitions. We use the alarm interpreter service as an example to prove the scalability of individual microservices.

Table 6.1: Topic description

Topic:sensor_data	Partition: 0	Leader: 1	Replicas:1,2
Topic:sensor_data	Partition: 1	Leader: 2	Replicas:0,2
Topic:sensor_data	Partition: 2	Leader: 0	Replicas:0,1

As we can see in table 6.1, every partition has one leader and is replicated across all the brokers. Now that Kafka is set up for testing, we start an instance of the alarm interpreter service. The instance is automatically assigned all three partitions since it is the only running instance belonging to its consumer group. Next, we start a second instance of the alarm interpreter service running on a different port. Once the second instance is up, the following is logged in the first instance:

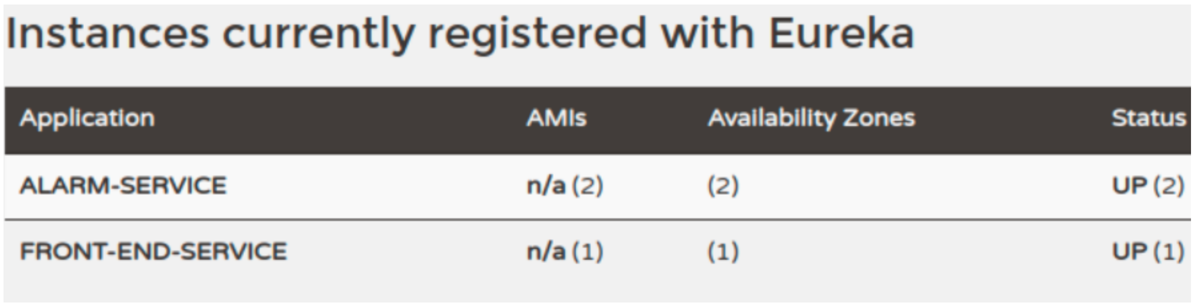
```
Attempt to heartbeat failed since group is rebalancing
Revoking previously assigned partitions [sensor_data-2, sensor_data-1,
    ↪ sensor_data-0]
(Re-)joining group
Successfully joined group with generation <current generation>
Setting newly assigned partitions: sensor_data-1, sensor_data-0
```

Kafka performs automatic rebalancing for the consumer group that now has two active consumers. Partition 2 is assigned to the second alarm interpreter service instance. We now have 3 partitions and 2 consumers which means that if there is a need to scale up even more, another consumer of the same consumer group can be started without having to alter the topic. For scaling beyond 3 consumers, we would have to use the previously mentioned script in listing 6.1 to add more partitions to the `sensor_data` topic.

The test described above shows how we can easily scale the services of our prototype individually. Scaling the other components can be conducted with the same approach.

6.2 Fault Tolerance Analysis

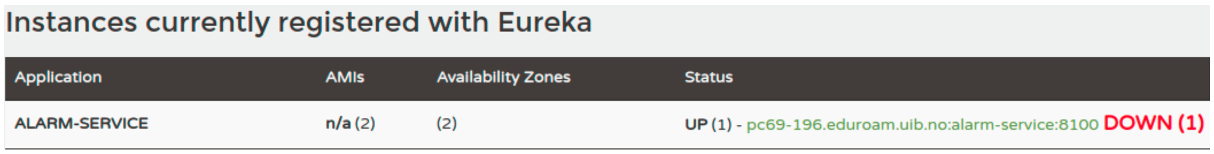
Our prototype needs to be able to cope with failures in service instances and Kafka. In theory, our multi-broker Kafka setup should be able to carry out its responsibility even when one or two brokers are down. We performed a simple test to confirm that our system works as intended when there is a failure in a broker or a broker is taken down. We also tested what would happen if one of our alarm interpreter service instances crashed. The tests conducted in this section starts where we left off in the scalability tests; with three brokers and two running instances of the alarm interpreter service component. We used a Eureka naming server [32] to check the status of our instances during testing. Eureka is a server for client-side service discovery and service registration developed by Netflix. A screenshot of service monitoring in Eureka can be seen in figure 6.1. Figure 6.1 shows that there are two


Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ALARM-SERVICE	n/a (2)	(2)	UP (2)
FRONT-END-SERVICE	n/a (1)	(1)	UP (1)

Figure 6.1: Running service instances displayed in Eureka

running instances of the `alarm-service`, while figure 6.2 shows the status of the `alarm-service` instances when one instance is taken down during the testing.



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ALARM-SERVICE	n/a (2)	(2)	UP (1) - pc69-196.eduroam.uib.no:alarm-service:8100 DOWN (1)

Figure 6.2: Status after shutting down one instance of alarm-service

Fault Tolerance in Kafka

Instance 1 of the alarm interpreter service is consuming partitions 0 and 1, while instance 2 is consuming partition 2. In this test, we observe what happens when broker 1 is shut down. As we can see in table 6.1, broker 1 is the leader for partition 0 which is consumed by instance 1. When broker 1 is shut down, the following Kafka warning is logged in instance 1:

```
Connection to node 1 (/<ip><port>) could not be established. Broker may
    ↪ be unavailable.
```

When we inspect the `sensor_data` topic, we obtain the results shown in table 6.2. As we can see, broker 2 has been elected new leader for partition 0. The shutdown has no effect on how the system runs since the leader election is conducted automatically by Kafka.

Table 6.2: Topic description after shutting down broker 1

Topic:sensor_data	Partition: 0	Leader: 2	Replicas:1,2
Topic:sensor_data	Partition: 1	Leader: 2	Replicas:0,2
Topic:sensor_data	Partition: 2	Leader: 0	Replicas:0,1

Fault Tolerance in the Alarm Interpreter Service

Table 6.3: Initial partition assignments

Instance: 1	Partition: 0
Instance: 1	Partition: 1
Instance: 2	Partition: 2

Next, we look at what happens when we shut down one of the alarm interpreter service instances. For this test, we shut down instance 2 which consumes partition 2. Initially the partitions were assigend as shown in table 6.3. When instance 2 is shut down, instance 1 has its previously assigned partitions revoked and is assigned all three partitions. Listing 6.2 shows the console log when pratitions is reassigned.

Listing 6.2: Console log when instance 2 is shut down

```
Attempt to heartbeat failed since group is rebalancing
Revoking previously assigned partitions [sensor_data-1, sensor_data-0]
(Re-)joining group
Successfully joined group with generation 3
Setting newly assigned partitions: sensor_data-2, sensor_data-1,
   ↪ sensor_data-0
```

Again, Kafka performs automatic rebalancing. Instance 1 performs the tasks of instance 2 until it restarted. In section 6.1, we described what happened when the second service is restarted. Kafka will rebalance and instance 2 will be assigned, in our example, 1-2 partitions. During the test, we also confirmed that no data was lost when a broker or a service was shut down. The rebalancing Kafka performes, in addition to the replication of messages, ensures that all data is delivered to the intended destination.

6.3 Extendability Discussion

One of the problems TrafSys are experiencing with the current solution is that is it hard to extend the system with new features and components and make changes in existing services. Minor changes to an existing component might lead to necessary updates throughout the system and might be a time-consuming process. One of our hypotheses was that a microservice architecture that utilizes Kafka helps solve this problem. When a producer publishes a message to a topic, it is available for the subscribers of that topic. If the data is relevant for more than one microservice, consumers in other services can subscribe to the topic and start receiving data from it. In our project, we utilized idempotent producers and the "exactly once" delivery semantic. This means that the Kafka brokers expect an acknowledgment from all consumer groups which are subscribed to a topic. Hence, new components, in our case, can be implemented, integrated into the system and start receiving data without having to do any changes in any of the other services. Adding new microservices to the prototype can be done without the risk that the new service will lead to errors in the other services. Microservices can also be removed from the system without problems except if the service is a producer and publishes data that one or more other microservices rely on. In this case, the microservice should not be taken down before a replacement service is in place or necessary changes have been completed in the services relying on it. Also, it is no problem to add more topics to the Kafka cluster for either existing or new services to use. The microservices that constitute our prototype are decoupled and mutually independent which only communicates with Kafka. No service knows which service will receive the data it produces or which service produces the data it consumes. The services are only responsible for producing, consuming or both producing and consuming.

To demonstrate this we started an instance of the alarm interpreter service and started sending data. After the services were running, we started a new Kafka consumer belonging to another consumer group through the Kafka CLI using the command in listing 6.3.

Listing 6.3: Starting a Kafka consumer with another consumer group

```
kafka-console-producer.sh --broker-list <ip>:<port> --topic sensor_data
    ↪ --group test_group
```

Both consumers consume the `sensor_data` topic. Once the consumer was up and running, it started to receive data. We made sure that they got the same data by looking at the offsets of the incoming messages. As we have discussed, an offset is a per topic, unique identifier. Since both consumers got messages with the same data, we know that they both got all messages that were produced to the topic. This is because Kafka expects an acknowledgment from every consumer group before the offset is committed.

6.4 Response Time

One of the requirements for the system is that if there is something that triggers an alarm in a facility, the operators have to know this within three seconds. In tests on the current version of VV, they found that the system handles the processing of ten messages/second(m/s) [5]. This means that under heavy workloads on the system, operators can get an alarm several minutes after it got triggered at the facility. We want to know if Kafka performs better as a message pipeline for this system, and have tested how long our prototype takes to send a message from a facility until it arrives at an operator. In Trafsys' tests, they ran 1000, 5000 and 10000 messages on one object, and measured the time it took the system to process all of them. Due to limited hardware resources to run our tests on, we are limited to a maximum of 2500 messages/second from facilities before we max out our hardware. There is also listed an example on 5000 messages/second to show what happens when the load exceeds the hardware it is running on.

Alarm Trip Time

This section will be dedicated to how long an alarm message uses from when it is triggered in a facility until it shows up in the CLI front-end. Our tests were performed with just one facility, with varying amount of messages produced per second. The messages will go through the Kafka broker and will get picked up by the Alarm Interpreter service (AIS), which decides if the sensor data is below a certain threshold causing an alarm. If it is an alarm, a new message goes out to the front-end, again through Kafka.

In these tests we wanted every data point from the facilities to be interpreted as alarms, causing a consistent and reliable load on the components involved. For every column in Figure 6.3, we performed five tests, discarded the first one due to high variance in connection time to the broker. The average, median and standard deviation are averages of the remaining four tests, while the maximum and minimum are across all four tests. In every test, facilities produce messages for 25 seconds and then shut off. This means that every test has 25 times more messages in total than the number of messages per second.

Our tests show consistent tests where the median is close to the average and have a low standard deviation. As the load goes up, so does the average time for messages to reach the CLI.

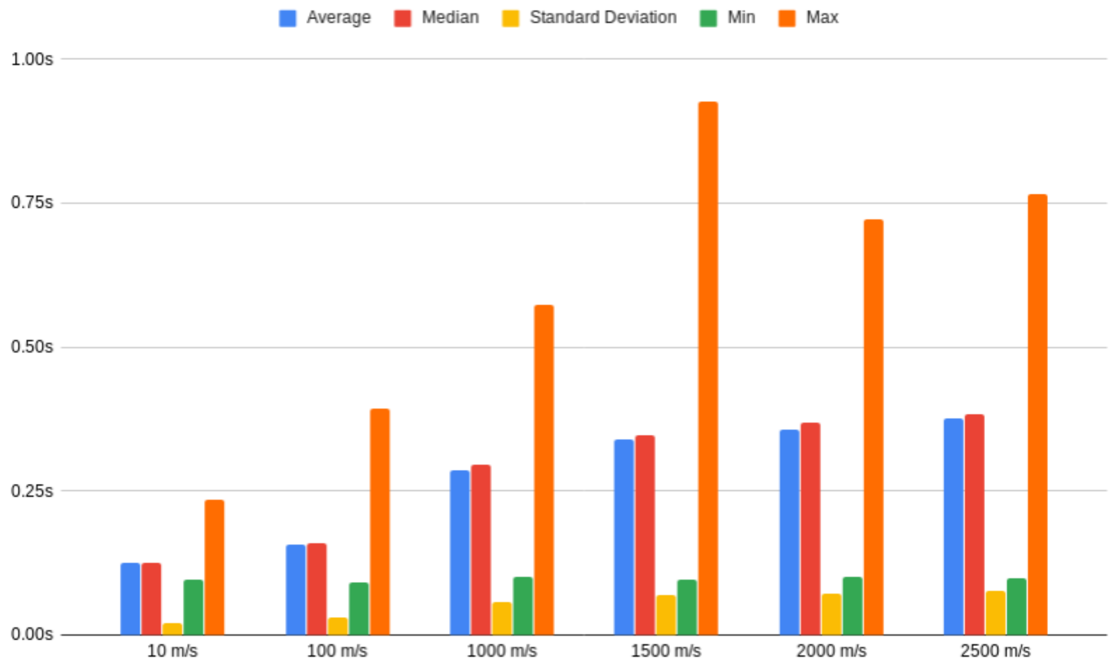


Figure 6.3: Alarm trip time with varying system load

We also performed a test at 5000 messages/second, as shown in Figure 6.4. To keep figure 6.3 readable, we moved this result out to a separate graph. Even though the values are higher with 5000 messages/second, we see much of the same patterns as we did in the tests with lower load, with a small difference in the median and average, and still a low standard deviation.

Our tests results have shown that on our hardware the performance takes a severe hit when the load increases. We find this troubling as Apache Kafka benchmarks [27] have reported throughput as high as 200 000 messages per second.

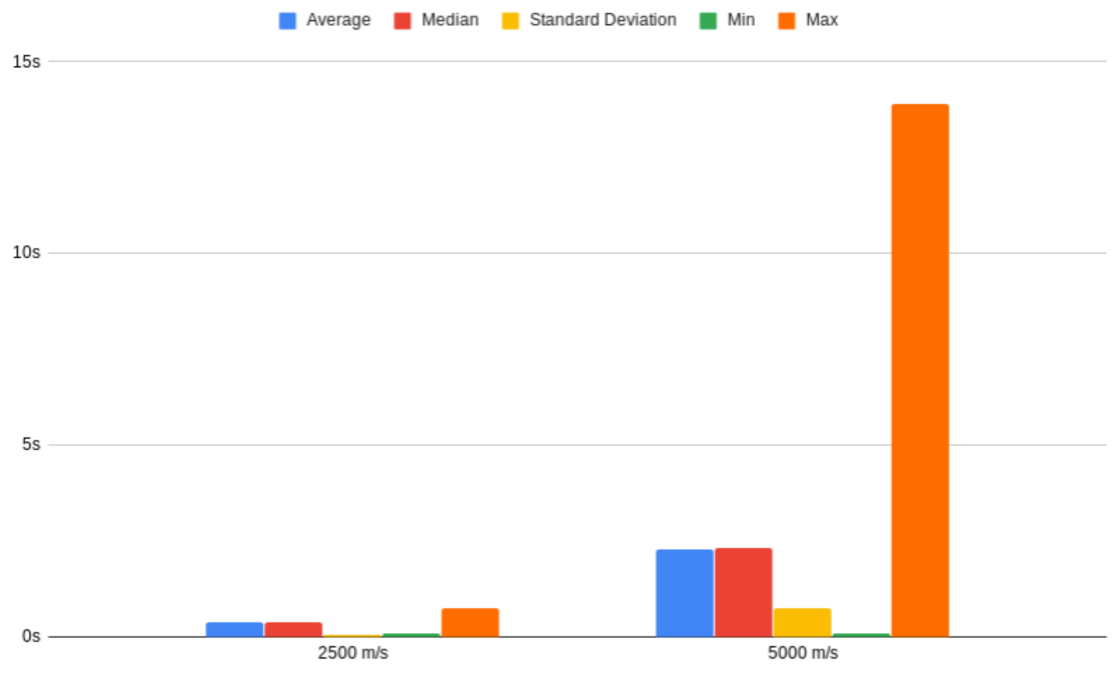


Figure 6.4: Alarm trip time with more load than our hardware could handle

6.5 Threaths of Validity

To end up at a conclusion we have built a representative prototype of VegVokteren, and the results we have shown are tests performed on that prototype. Our implementation builds upon some simplifications of VegVokteren and has been tested in a different environment than the servers Trafsys are going run their system on. Therefore, there are most likely going to be some discrepancies between Trafsys' and our test results. Our prototype contains just a few services, that do a very low amount of processing. In the real system, there will be more services, more processing and more communication to, and from databases. This will most likely increase the overall responsiveness of the system, and will lead to lower throughput.

Even though we have researched the security of both the Kafka framework to some extent, we have not implemented any security measures into our prototype. The reasoning behind this is, although necessary, the impact on performance, extendability and fault tolerance would probably be minor.

6.6 Test Environment

Our system primarily runs on laptops, but the Kafka broker runs in the Google Cloud Platform. When running tests, all services have been running on one machine. Our tests results when measuring perfomance of the system are likely to be impacted by this setup.
Specs for the laptops:

- **CPU:** Intel i5-7200U @ 2.5GHz
  - **Boost clock:** 3.1GHz
  - **Cores/threads:** 2/4
- **Memory:** 8GB (2x4GB)
  - **Clock:** 1867MHz

- **CPU:** Intel i5-6300HQ @ 2.30GHz

  - **Boost clock:** 3.20GHz
  - **Cores/threads:** 4/4

- **Memory:** 8GB

Chapter 7

Conclusion and Future Work

In this chapter, we summarize the work presented in this thesis and how the proposed event-driven microservice architecture address the problems stated in the research questions.

7.1 Summary

In this project, we developed an event-driven microservice prototype. The prototype utilizes a cluster of Apache Kafka message brokers to handle all communication between the services comprising the prototype and Apache Kafka. The implemented prototype is a more simplistic version of the current system based on the core functionality of VegVokteren. This thesis discussed the microservices that was implemented and the main components of our prototype:

1. Facility. This is a service that generates a data file that simulates sending data. The data is published to Kafka in a JSON format and contains facility and sensor identifiers, sensor value, and a timestamp.
2. State and Storage Service. This service retrieves information about the state of the facilities, actuators, and sensors. The reasons for this is to be able to look at historical data and get the current state. The current state can be used if a sensor fails to restart it in the state it was in before it failed. If a sensor fails due to the state it was in, it can be restarted from an earlier state.

3. Alarm Interpreter Service. The interpreter polls sensor data from Kafka and interprets it to detect alarms. Alarms that are detected are stored in a database and pushed to another Kafka topic.

4. Command-Line Interface. We implemented a CLI as our front-end for this prototype. The CLI continuously polls the topic with active alarms and stores them in the database. It is also the CLI that has the logic for calculating the time it takes from an alarm is published from the facilities until it reaches the front-end. The CLI has a command for displaying the alarms which can be filtered by the identifiers of the facilities and the sensors.

To summarize, the prototype generates messages in facility objects and sends them to Kafka. The alarm interpreter service polls the sensor_data topic and interprets the data to detect alarms. The alarms are published to an alarm topic which is polled by the front-end CLI and presented to the users. The Kafka cluster and clients were configured to encrypt messages during transmission, use SASL TLS authentication to let the clients and the cluster authenticate to each other, and use ACLs to control access to the topics in the cluster.

7.2 Research Questions and Results

The challenges that were stated in the research questions are discussed in chapter 3. The research questions for this thesis were:

1. Can an Event-Driven Microservice architecture make it easier to

   I Further extend the system with new functionality and services

   II Meet the requirements for scalability, fault-tolerance, and security

   III Meet the requirement for response time

We now discuss how the prototype has contributed to investigate these research questions. Research question I was answered in section 6.3 where we discuss how utilizing Kafka as our MOM decouples the microservices comprising the system. In chapter 2 we discussed how event-driven architecture and microservices decouples system services, and in section 4.4 we discussed how Apache Kafka also aids in decoupling services. The decoupled nature of the prototype makes the process of extending the system with more functionality simpler than what it is in the current solution of VegVokteren. Adding or removing a microservice require no, or minor, changes in other services of the system. This was demonstrated by conducting an experiment which showed that consumers belonging to different consumer groups will get the same messages.

To answer research question II, we showed in chapter 4 how event-driven microservices and Kafka enables scalability of the decoupled microservices of our prototype. Section 6.1 described the experiments that were conducted on the prototype to demonstrate that it is possible to scale services independently. Kafka performs automatic rebalancing when two instances of the same service are running and the messages are distributed among them if the partition count of the topic is greater than or equal to the number of running instances. Section 2 also discussed how replication of topic partitions makes the message transmission fault-tolerant. We conducted experiments that demonstrate how Kafka performs automatic rebalancing in our multi-broker cluster setup when an instance of a service or a Kafka broker goes down which are described in section 6.2. This makes the prototype fault-tolerant and reliable. The security mechanisms of Kafka were also presented in chapter 2 and a detailed discussion of our security setup was explained in section 5.1. To meet the security requirement, we enabled encryption, authentication, and authorization to prevent unauthorized access to our Kafka cluster.

To answer research question III we conducted performance tests with different sized workloads as described in section 6.4. With our own limited hardware, we were limited to sending 2500 messages per second before the performance decreases. The experiments show that up until 2500 messages per second, the average and maximum alarm response time is well within the requirement of three seconds. The experiments conducted with 5000 messages per second show that the average response time is within the response time requirement, while the maximum is reaching the front-end around 14 seconds after it is transmitted from the facility. This is 11 seconds more than what is stated in the requirement.

7.3 Future Work

Due to lack of time, some work has been left for future work. Perhaps the most interesting experiment would have been to perform testing with real data from one or more of the facilities that Trafsys is monitoring and controlling. This kind of testing would require integration with the equipment in the facilities, but would provide good indications as to whether or not an event-driven microservice architecture which utilizes Kafka for communications is a good technology choice for VegVokteren. Also, a more detailed analysis of the message flow and the microservices in the prototype itself could help uncover what works well and areas where improvements are necessary or different strategies must be implemented. Another interesting experiment would be to run our prototype on more powerful hardware and to run multiple instances of our services to test if the prototype is able to handle higher workloads and meet the response time requirement. Some further aspects that would be interesting to investigate as part of future work are:

1. Would hosting the Kafka cluster closer to the facilities decrease the time it takes for an alarm to reach the system operators? If there is a significant decrease in time, it would be beneficial to move the Kafka cluster to help meet the requirement of response time.

2. How will an increase in the number of facilities impact the response time? The number of facilities that are monitored and controlled by VegVoketeren is continuously increasing. It is hard to foresee what impact that will have on an architecture like the one we have discussed in this thesis.

3. Is the architecture presented in this thesis applicable to other, similar systems? If further testing with real data indicates that the architecture performs well for these kinds of systems, it might be applicable to systems in other business areas.

The system is lacking a lot of functionality compared to the current solution (discussed in section 3.1) which could be implemented in the future. This includes:

1. VV-Web which is the graphical interface which contains access control mechanisms and displays graphical objects.

2. Geo-server which provides geographical data and functionality to VV-Web such as information about where facilities are located and associated objects.

3. Streaming server which controls all video streams and snapshots from cameras in the facilities.

4. Configuration tool which is used to define representations of objects in the facilities and generate configuration files.

5. Report generator which generates reports based on incoming events and other essential data to be used for analysis.

Integrating all the above-mentioned components could prove to be difficult. Most of them would publish data relevant to other components to Kafka, and consume data from Kafka that is relevant for the component itself. A component might be divided into several microservices based on the number of tasks they perform. Video streams from the streaming server should not be sent through Kafka since Kafka is built for message streaming and not for video streaming. Research on how to best stream video would have to be conducted as part of future work.

VegVokteren is not only a monitoring system but also a control system. Our prototype is lacking the control mechanisms of the current solution and is serving only as a monitoring system. A natural expansion of the system would be to implement the control mechanisms to be able to manipulate the objects in the facilities. To be able to manipulate the facilities and sensors, we would have to expand the CLI with functionality that allows users to enter commands based on the action that should be performed. The user input would be published to Kafka and polled by the back-end services to perform updates.

In this thesis, we have described an event-driven microservice prototype system for monitoring. We have shown that the prototype aids in solving the challenges of the current solution of VegVokteren. The prototype scales well and is both extendable and fault-tolerant. The performance tests show that it performs well on reasonably high workloads and meets the requirement of response time.

List of Figures

List of Tables

Listings

Bibliography

[1] Interface CommandLineRunner.
URL: `https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html`.

[2] org.springframework.data.jpa.repository.
URL: `https://docs.spring.io/autorepo/docs/spring-data-jpa/1.7.x/api/org/springframework/data/jpa/repository/JpaRepository.html`.

[3] Spring Data.
URL: `https://spring.io/projects/spring-data`.

[4] A. Abeysinghe. Event-driven architecture: The path to increased agility and high expandability. September 2016.
URL: `http://wso2.com/whitepapers/event-driven-architecture-the-path-to-increased-agility-and-high-expandability/`.

[5] Trafsys AS. Forprosjektrapport: Ny Vegvokterarkitektur. 2017.

[6] C. Ashish, S. Pelluru, T. Takebayashi, S. Manheim, K. Whitlatch, and T. Nevil. Service Bus queues, topics, and subscriptions. 2018.
URL: `https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions`. 12/03/2019.

[7] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, May 2016. ISSN 0740-7459. doi: 10.1109/MS.2016.64.

[8] IBM Knowledge Center. Common message delivery patterns, 2010.
URL: `https://www.ibm.com/support/knowledgecenter/en/SS9H2Y_7.7.0/com.ibm.dp.doc/mq_commonmessagedeliverypatterns.html`. 12/03/2019.

[9] Shigeru Chiba and Rei Ishikawa. Aspect-Oriented Programming Beyond Dependency Injection. In Andrew P Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 121–143, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31725-8.

[10] Melvin E. Conway. How do committees invent?, April 1968.
URL: `http://www.melconway.com/Home/Committees_Paper.html`.

[11] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Industry Paper : Kafka versus RabbitMQ A comparative study of two industry reference publish / subscribe implementations. doi: 10.1145/3093742.3093908.

[12] M. Edwards, O. Etzion, M. Ibrahim, S. Iyer, H. Lalanne, M. Monze, C. Moxey, M. Peters, Y. Rabinovich, G. Sharon, and K. Stewart. A conceptual model for event processing systems. *Architecture*, pages 1–47, 2010.

[13] Y. Engel and O. Etzion. Towards proactive event-driven computing. (April):125, 2011. doi: 10.1145/2002259.2002279.

[14] Tom Fellmann and Manolya Kavakli. A command line interface versus a graphical user interface in coding VR systems. *Proceedings of the 2nd IASTED International Conference on Human-Computer Interaction, HCI 2007*, pages 142–147, 2007.

[15] Martin Fowler. Polyglot persistence, November 2011.
URL: `https://www.martinfowler.com/bliki/PolyglotPersistence.html`.

[16] Martin Fowler and James Lewis. Microservices, March 2014.
URL: `https://martinfowler.com/articles/microservices.html`.

[17] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building LinkedIn's Real-time Activity Data Pipeline. 35(2):21–45, 2012.

[18] Google. Cloud Pub/Sub.
URL: `https://cloud.google.com/pubsub/`.

[19] Jim Gray. Learning from the amazon technology platform, June 2006.
URL: `https://queue.acm.org/detail.cfm?id=1142065`.

[20] Schwarz M. H. and Boercsoek J. A Survey on OLE for Process Control (OPC) . 2007.

[21] Simon Halvorsen and Øystein Hegglid Follo. Implementation of the generic architecture.
URL: `https://github.com/Folb/masterTrafsys`.

[22] Beth Isaksen and Valeria Bertacco. Verification Through the Principle of Least Astonishment. pages 860–867.

[23] Ismael Juma. Apache Kafka Security 101, 2016.
URL: `https://www.confluent.io/blog/apache-kafka-security-authorization-authentication-encryption/`.

[24] M. Kim, J. Shin, S. T. Chanson, and S. Kang. An approach for testing asynchronous communicating systems. 1996. doi: 10.1007/978-0-387-35578-8_19.

[25] Holger Knoche and Wilhelm Hasselbring. Using microservices for legacy software modernization. *IEEE Software*, 35:44–49, 05 2018. doi: 10.1109/MS.2018.2141035.

[26] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. On the Security of the TLS Protocol: A Systematic Analysis. In Ran Canetti and Juan A Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 429–448, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40041-4.

[27] Jay Kreps. Benchmarking apache kafka: 2 million writes per second (on three cheap machines), April 2014.
URL: `https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines`.

[28] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka : A Distributed Messaging System for Log Processing. 2011.

[29] E. N. Pinson M. D. McIlroy and B. A Tague. Unix time-sharing system: Forward. 1978.

[30] B. M. Michelson. Event-Driven Architecture Overview: Event-Driven SOA Is Just Part of the EDA Story. *Patricia Seybold Group and Elemental Links*, 2006. doi: 10.1571/bda2-2-06cc.
URL: `http://www.omg.org/soa/Uploaded`.

[31] NICOLAS NANNONI. *Message-oriented Middleware for Scalable Data Analytics Architectures*. PhD thesis, 2015.
URL: `http://kth.diva-portal.org/smash/get/diva2:813137/FULLTEXT01.pdf`.

[32] Netflix. Eureka.
URL: `https://github.com/Netflix/eureka/blob/master/README.md`.

[33] O'Reilly Media, inc. Event-driven architecture broker topology, 2015.
URL: `https://learning.oreilly.com/library/view/software-architecture-patterns/9781491971437/assets/sapr_0203.png`. 13/03/2019.

[34] O'Reilly Media, inc. Event-driven architecture mediator topology, 2015.
URL: `https://learning.oreilly.com/library/view/software-architecture-patterns/9781491971437/assets/sapr_0201.png`. 13/03/2019.

[35] Nathan Peck. Microservice principles: Decentralized data management, September 2017.
URL: `https://medium.com/@nathankpeck/microservice-principles-decentralized-data-management-4adaceea173f`.

[36] Nathan Peck. Microservice principles: Decentralized governance, 2017.
URL: `https://medium.com/@nathankpeck/microservice-principles-decentralized-governance-4cdbde2ff6ca`.

[37] Nathan Peck. Microservice principles: Smart endpoints and dumb pipes, September 2017.
URL: `https://medium.com/@nathankpeck/microservice-principles-smart-endpoints-and-dumb-pipes-5691d410700f`.

[38] Pivotal Software. RabbitMQ.
URL: `https://www.rabbitmq.com/`.

[39] Dana Powers. kafka-pyhton 1.4.6, 2019.
URL: `https://pypi.org/project/kafka-python/`.

[40] Spring Boot Project. Spring Initializr.
URL: `https://start.spring.io/`.

[41] M. Richards. Event-driven architecture. In *Software Architecture Patterns*, chapter 2. O'Reilly Media, Inc., February 2015. ISBN 9781491924242.

[42] Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009. ISBN 0123749573, 9780123749574.

[43] W. Sthepen. What is stream processing? event stream processing explained, February 2018.
URL: `https://www.bmc.com/blogs/event-stream-processing/`. 14/03/2019.

[44] B. Stopford. *Designing Event-Driven Systems. Concepts and Patterns for Streaming Services with Apache Kafka.* 2018. ISBN 9781492038221.

[45] The Akamai Team. Style Guide, .
URL: `https://developer.akamai.com/cli/docs/style-guide`.

[46] The Pallets Team. Click. .
URL: `https://click.palletsprojects.com/en/7.x/`.

[47] John Viega, Matt Viega, and Pravir Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications.* 2002. ISBN 0-596-00270-X.

[48] C. L. Wang and P. K Ahmed. Dynamic capabilities: A review and research agenda. *The International Journal of Management Reviews*, 9:31–51, 2007.

[49] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a Replicated Logging System with Apache Kafka.

[50] Zhenghe Wang, Wei Dai, Feng Wang, Hui Deng, Shoulin Wei, Xiaoli Zhang, and Bo Liang. Kafka and its Using in High-throughput and Reliable Message Distribution. *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, 2015. doi: 10.1109/ICINIS.2015.53.

[51] L. Yan and W. Dong. Complex Event Processing . February 2010. doi: 10.1.1.457.4226.

[52] Aaram Yun and Yongdae Kim. On Protecting Integrity and Confidentiality of Cryptographic File System for Outsourced Storage. 2009.