Distributed Sensor Management for an Industrial DevOps Monitoring Platform

Bachelor's Thesis

Simon Bela Nicolay Fritz Alexander Ehrenstein

September 26, 2019

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Sören Henning, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 26. September 2019

Abstract

The organization of sensors is fundamental for monitoring systems since sensor-associated data is substantial for generating information about the target infrastructure. In this work, we propose how a sensor management can be integrated into a microservice based architecture, for the monitoring of the electrical power consumption in industrial environments, in order to apply Industrial DevOps. Aiming at improving the performance of the system, we introduce the sensor management as a microservice that allows identifying sensors internally with numerical identifiers that are decoupled from the external representation of sensors. Moreover, we allow a flexible hierarchial organization of sensors which provides the basis for other microservices to aggregate monitoring data with respect to the most recent hierarchial organization. We provide a prototype implementation of our approach which we evaluate in various scenarios that cover different functional requirements of our approach. Although, we are not able to show that our internal format for the representation of sensors yields to an improved performance for realistic scenarios, we provide the base for a dedicated performance analysis in a realistic production environment, where it can be analyzed whether and how the performance of the system can be improved by representing sensors internally with a numerical format.

Contents

Chapter 1

Introduction

1.1 Motivation

In the vision of Industry 4.0 [Lasi et al. 2014], industrial facilities are highly integrated with internet technologies. Implementing IT[1] solutions in order to cope with the 4th industrial revolution can be a challenging task for small and medium sized enterprises, because of the lack of experience with their integration as well as the high financial and organizational challenges that come with the implementation and maintenance of these systems [Hasselbring et al. 2019]. As a solution to this problem, Hasselbring et al. [2019] propose Industrial DevOps as an approach for applying the culture and principles of DevOps in an industrial context. The goal of the Titan Project [2018] is to lower the organizational effort and the costs for enterprises for applying Industrial DevOps by creating an open-source platform, which is easy to use for employees that have no expertise in programming, and to make those companies cope with the challenges of the digitalization and the realization of Industry 4.0.

In industrial environments, one important cost factor is the electrical power consumption. Therefore, for industrial enterprises, it is useful to monitor the electrical power consumption of the industrial infrastructure for applying Industrial DevOps. Accordingly, Henning et al. [2019] propose the *Titan Control Center* as a scalable architecture for monitoring the electrical power consumption as part of the Titan platform. The Titan Control Center monitors the electrical power consumption of the target infrastructure based on existing sensors that send their measurements to the system. Hence, it is the key component that is accountable for continuously providing decision makers with information about the preceding power consumption, which is essential for the application of the DevOps cycle. As sensor-associated data is substantial for the functionality of a monitoring system, like the Titan Control Center, it is necessary to pay special attention to how this data is managed. This includes how sensors are represented within the system as well as how measurements of specific sensors are processed. Another goal of the processing of records is to generate as much meaningful information from the monitored data as possible which can be used for improving business processes. As a consequence, we suggest a monitoring system, such as the Titan Control Center, to include a dedicated component that is responsible for the management of sensors.

[1] Information Technology

1.2 Goals

The superordinate goal of this thesis is to develop an approach for the management of sensors in a distributed monitoring system, like the Titan Control Center. Hence, we define the following subordinate goals.

1.2.1 G1: Decoupling of the Sensor Representation from the Environment

In a monitoring system, such as the Titan Control Center, each monitoring record refers to a single sensor. This implies that each record needs to contain an identifier which allows to determine the sensor to which a record refers. On the one hand, industrial operators are motivated to allow these sensor identifiers to be as general as possible, e.g., to be able to use identifiers to which a human can easily relate. This can be accomplished by using strings for the representation of sensors. On the other hand, we assume using string-based identifiers to be inefficient. As a consequence, it is desirable to have a representation for handling the sensors internally which is more efficient than using string-based identifiers. Therefore, we aim to develop a suitable approach of decoupling the internal representation of sensors from the environment which allows us to use an internal format for monitoring records. This way, we aim to improve the performance of the system while still being able to access the system with the string-based sensor identifiers.

1.2.2 G2: Providing a Flexible Organization of Sensors

In the target industrial infrastructure of the Titan Control Center, individual physical sensors emit measurements, which provide direct information about the power consumption. Moreover, it is useful to organize these sensors according to certain aspects. This means that multiple sensors are grouped to allow the calculation of aggregated measurements, for example, the sum of all measurement values for all sensors within a specific group. Moreover, it should be possible to organize sensors with regard to multiple aspects simultaneously, e.g., on the one hand, according to their logical function within the target infrastructure and their location on the other. Thus, we aim to provide an easy-to-use interface through which a user can define these organizations of sensors. Additionally, it should be ensured that other components of the system have knowledge about the current organization of sensors at any time. Therefore the other components should be notified if the organization of sensors changes.

1.2.3 G3: Preserving Scalability and Fault Tolerance of the System

As the function of the sensor management is crucial for the function of the whole system, it should be designed fault-tolerant such that it compensates partial failures. In general, the sensor management should fit into the architectural design of the existing system. As

the architecture of the Titan Control Center is aimed to be scalable, we also aim to design of the sensor management as a scalable architecture.

1.2.4 G4: Implementation of a Software Prototype

With respect to the evolved architecture, we aim to create a prototype implementation of a sensor management component. This, involves the realization of backend logic of the sensor management, together with an extension of the graphical user interface of the system that allows the user to manage how sensors are organized. Additionally, the other components of the system need to be adapted so that the prototype can be integrated into the existing system.

1.2.5 G5: Evaluation of Feasibility and Performance

We aim to evaluate the implementation of our architecture. On the one hand, we want to evaluate the feasibility of our approach in order to demonstrate that our implementation fulfills the functional requirements. On the other hand, we want to identify deficiencies of the performance of the current version of the system with regard to the representation of sensors and compare the performance to our approach.

1.3 Document Structure

The remaining part of this work is structured as follows: In Chapter 2 we present the foundations and technologies that are the base for our approach which we discuss in Chapter 3. On the one hand, our approach includes taking the current version of the Titan Control Center into consideration and on the other hand, it includes deriving requirements that lead us to a suitable architecture. Consequently, we present a prototype implementation of the sensor management in Chapter 4. Finally, in Chapter 5, we evaluate our implementation. At the end of this work, in Chapter 6, we discuss related work, before we come to the conclusions of our thesis in Chapter 7.

Chapter 2

Foundations and Technologies

In this chapter, the foundations and technologies, which are relevant for our approach (Chapter 3) and implementation (Chapter 4), are introduced.

2.1 Distributed Systems

A distributed system is a system, where the individual components, which can be, e.g., running processes or databases, reside on different hardware units that are interconnected by a network [Tanenbaum and Van Steen 2007]. We often refer to these hardware units as nodes. The communication between the nodes is commonly realized directly with network protocols, such as the *Hypertext Transfer Protocol* (HTTP) or by message oriented middlewares.

2.2 Vertical Scalability

Today, in times of large bandwidth for internet services, the limiting factor for data transmission often is not the network but the computational resources of nodes within distributed applications. When a node's computational capacity is exhausted, the result is an increasing latency for serving requests from clients. Vertical scaling describes the enhancement of a system by adding resources to individual nodes [Barzu et al. 2017b]. This usually includes increasing the number of CPUs, adding memory or increasing the capacity of the node. The result of vertical scaling is that individual nodes, and as a consequence the system, are optimized for efficiently handling a higher load of the system.

2.3 Horizontal Scalability

Although vertical scaling can be used to increase the load a system can handle, it involves high costs due to the hardware required. Additionally to vertical scaling, we can decrease the load of individual nodes by adding machines to the system in order to decrease the latency for clients requesting the system which is referred to as horizontal scaling [Barzu et al. 2017a]. Additionally, scaling a system horizontally implies the existence of redundant

nodes. This also increases the systems abilities to cope with failures of single nodes which is prerequisite for providing fault-tolerance as described by Isermann [2006].

2.4 Microservices Based System Architectures

In the traditional architecture for web applications, the system consists of a single unit. This means, an application can only be deployed as a whole. To this approach of designing an application we refer as a monolithic architecture. In contrast, in a microservices architecture [Newman 2015] [Hasselbring and Steinacker 2017] the application consists of multiple small, independently deployable, autonomous units that work together and that can be scaled individually. Further, designing services in a stateless manner allows individual requests to a service to be understood to be isolated from each other [Fielding and Reschke 2014]. By designing microservices in a stateless way, we achieve horizontal scalability of instances as it does not matter for a client which instance of a respective microservice handles its requests. Choosing a microservices architecture implies that the system is a distributed system which entails implementing mechanisms that allow the microservices to communicate with each other. The communication between the services is often realized with Representational State Transfer (REST) [Fielding 2000] in combination with HTTP, which allows stateless data transmission, or with a messaging middleware. Even if introducing a microservice architecture often results in an higher initial effort than choosing a monolithic approach since we have to implement the communication between the services as well as setting up the environment, which usually consists of more machines than for a monolithic application, there are some advantages to consider: First, the individual microservices are usually designed to fulfill certain domain-specific tasks which enables them to be loosely coupled. As a consequence, extending or changing a component is less complicated. Moreover, designing small services results in less effort for the deployment [Hasselbring and Steinacker 2017]. Second, it is possible to use different technologies, such as programming languages or databases, for each microservice. Due to that, teams of experts for the respective domain of a microservice are able to explicitly address the specific needs of a particular microservice. This allows to address potential bottlenecks of our application more precisely.

2.5 Atomicity of Operations in Database Systems

Atomicity is one of the properties that comes with *ACID*[1] systems. An atomic operation follows the all-or-nothing principle, which means that either the whole operation succeeds, or the operation fails and no changes are applied. Often, the perception of atomicity is used in the context of database transactions that enable multiple operations to be treated as one.

[1] Atomicity, Consistency, Isolation, and Durability [Haerder and Reuter 1983]

2.6 Consistency Models in Distributed Systems

According to the CAP-Theorem [Gilbert and Lynch 2002], in the presence of partition tolerance, we have to choose between consistency and availability. Given a distributed system, this means we have to trade off consistency against availability according to the specific requirements. As a consequence, we have to choose an appropriate consistency model for our application.

The strictest consistency model implies that every read operation in a system returns the most recent value with respect to time. This is only possible if all preceding write operations are made persistent instantaneously, which is impossible to achieve in a distributed system with today's technology since the speed of the transfer of information is physically bounded. However, it is often sufficient to implement a relaxed consistency model to fulfill the requirements of the application. One relaxed consistency model is the sequential consistency model [Lamport 1979]. In this model, the result of an execution is the same as if all operations from the processes were executed in some sequential order, while the order of the operations from each individual process is preserved.

2.7 The Pipe and Filter Framework TeeTime

TeeTime [Wulf et al. 2017] is a Pipe-and-Filter (P&F) framework for Java. It allows to create and execute arbitrary P&F architectures. Two important abstractions in TeeTime are *stages*, and *ports*. Stages correspond to filters in a P&F architecture. Accordingly, they are used to transform incoming data before it gets passed to the next stage via an output port. In general, stages can have different amounts of input and output ports that serve for receiving and passing data from and to other stages within the P&F architecture. Additionally, TeeTime provides a set of predefined stages that we can use to implement our own P&F architectures.

2.8 The Apache Kafka Messaging Platform

Apache Kafka [Kreps et al. 2011], simply referred to as *Kafka*, is a messaging platform that follows a publish-and-subscribe pattern. Kafka allows clients to publish streams of data entries to a distributed store and to asynchronously consume data from the store. In Kafka, so-called topics act as a named channel for a specific type of data. Each topic can be considered to be a stream of data entries, where each entry is a key-value pair. The platform provides different *Application Programming Interfaces* (APIs). With the Producer API, data can be published into a topic. For this, the data must be serialized since the data within the topics is stored in form of bytes. With the Consumer API, the previously published data can be read from a topic and therefore has to be deserialized in order to work with it on application level. There also exist the Connect API that allows to connect

Kafka to third-party systems, such as databases, and the Streams API which can be used to perform enhanced stream processing in combination with Kafka. The data within the topics is decomposed into partitions, whereas each of these partitions can be replicated over multiple Kafka brokers. Both, the number of partitions and the replication factor can be individually defined by the user for each topic. As a consequence, Kafka is able to provide high scalability and fault-tolerance.

2.9 The Titan Control Center Prototype

The *Titan Control Center* [Henning et al. 2019] is responsible for the monitoring of the power consumption in the Titan platform. The Titan Control Center is a microservice-based system which consists of four microservices (RecordBridge, History, Configuration, Stats) that communicate with each other with REST via HTTP and with messaging via Kafka. The relationship between the microservices of the Titan Control Center is visualized in Figure 2.1. The different microservices of the Titan Control Center serve specific tasks. Each Record Bridge acts as an entry point for sensor measurements. There can exist multiple types of Record Bridges at once, each for a different type of sensor data or for a different protocol used for transmitting sensor measurements to the system. Further, at the Record Bridges, the data relevant for the monitoring is extracted from incoming data packages and it is published to the other components of the system via Kafka. Record Bridges can be implemented with the help of the Record Bridge framework. This framework comes with the Titan Control Center, and provides a set of standard definitions for integrating the Titan Control Center with the target infrastructure. The Configuration microservice allows to organize the physical sensors that send their measurements to the Record Bridges in a hierarchial manner. For this, it provides a REST API that can be accessed by the visualization frontend which allows the organization of sensors via a web-based interface. The History microservice has the task of persisting the obtained sensor measurements and to aggregate the measurements according the defined organization of sensors. Based on the aggregation performed by the History microservice, the Stats microservice calculates statistics concerning the power consumption of sensors.

2.10 The Key Value Store etcd

etcd [Cloud Native Computing Foundation 2018] is a distributed key-value store with the focus on reliability. Its data model consists of a list of key-value pairs that are associated with a revision number. This revision number is incremented, each time an entry of the store is modified. Moreover, it is guaranteed that the revision number is unique for each data-manipulating operation. etcd allows to be operated as a clustered database, where the data can be replicated over multiple nodes. Although there exists a special node in every cluster, the so-called leader, which is responsible for executing each operation that

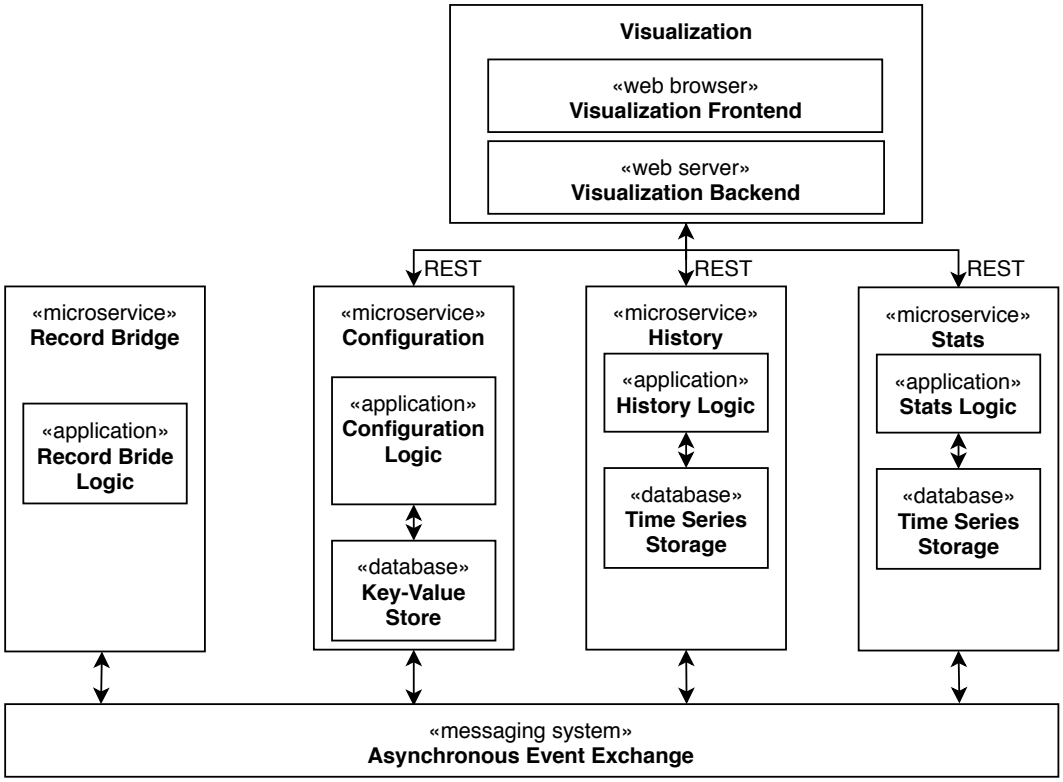

Figure 2.1. Microservice Architecture of the Titan Control Center [Henning et al. 2019].

requires consensus, each node provides an identical interface for the client. This is possible because if an operation requires consensus, the current leader controls the execution of a consensus algorithm to make all nodes agree to the modifications of the store. In case of a leader failure, a new leader gets elected automatically. For achieving consensus, the RAFT consensus algorithm [Ongaro and Ousterhout 2014] is applied.

etcd can be accessed via its *gRPC* [Cloud Native Computing Foundation 2015] API which supports various operations including the basic operations *put*, *get*, and *delete*. Furthermore, etcd supports the concept of ranges. Ranges are sets of keys that correspond to an alphabetical interval. This means, e.g., all keys with the same prefix can be retrieved or deleted in one operation.

Additionally, etcd guarantees sequential consistency which means that for all nodes in a cluster that have performed the same number of operations, the order of the performed operations will be identical. Moreover, etcd supports so-called mini-transactions. These transactions are atomic *if-then-else* constructs which allow to treat multiple read and write operations as a single operation. This behaviour can be used to implement higher order

atomic instructions, such as *test-and-set*[2], or distributed locking.

2.11 The Single Page Web Framework Vue.js

Vue.js [You 2018], simply referred to as *Vue*, is a JavaScript Framework for creating single-page web applications. In contrast to traditional web applications, where each page of a website is retrieved individually from the server and as a consequence has its own Document Object Model (DOM), a single-page web-application consists of one DOM which is manipulated in order to navigate between pages. Meanwhile, the application can communicate with a server in order to display information or to perform certain actions such as logging in a user.

The central organizational units in Vue.js are components. Components are reusable entities that are responsible for a specific part of the application, e.g., for a side-menu or another arbitrary view within the application. Components can contain other components leading to a hierarchial structure with a root component which contains all other components. Each component consists of the respective layout definitions that are based on the *Hypertext Markup Language* (HTML) with some Vue-specific extensions, scripts that contain the business logic, and styling definitions for the layout. Additionally to the component system, Vue provides extra functionality, e.g., a router which can be used to navigate between views and an event lifecycle which allows to perform tasks at certain events, such as fetching data from a server before rendering a component. Another important concept of Vue are directives. Directives are custom HTML attributes which enable to bind logic to layout elements. This allows, for example, to bind the state of a layout element to some variable in the background or to dynamically render a list of layout elements from a list that is defined in a script of the web application.

[2] An atomic statement that performs an action if a certain condition is true.

Chapter 3

Approach

In this chapter, we present our approach for the sensor management. Thus, we need to consider the current state of the existing system (Section 3.1), in our case the Titan Control Center. After that, we derive requirements for the sensor management (Section 3.2) and we present a suitable architecture (Section 3.3).

3.1 The Current Version of the Titan Control Center

As our approach builds on the current version of the Titan Control Center, we have to take the current architecture into consideration.

3.1.1 Flow of Sensor Data in the Titan Control Center

In order to discuss the management of sensors within the Titan Control Center, we have to analyze the flow of sensor data at the current state of the system. Figure 3.1 visualizes how sensor measurements are processed in the Titan Control Center. Within the system, monitoring records are represented by subtypes of the `IMonitoringRecord` interface from the monitoring framework Kieker [van Hoorn et al. 2012] which can be generated with the help of the Kieker Instrumentation Record Language (IRL) [Henning 2018a]. At the current version, the Titan Control Center is only able to monitor the active power consumption of the target system. The structure of the two central records, the `ActivePowerRecord` and the `AggregatedActivePowerRecord`, is shown in Figure 3.2. However, there also are more types of monitoring records used for representing statistical data within the Stats microservice. These records are similar to the records in Figure 3.2 and can be treated analogous with respect to the approach of this work.

When an instance of a Record Bridge receives a data package for a sensor, the relevant fields from the data are extracted. These are the *identifier* of the sensor, which is allowed to be an arbitrary string that must be globally unique, the *value in watts* that corresponds to the measured power consumption of the sensor, and the *timestamp* of the measurement which indicates when the measurement has been taken. This leads us to the creation of an `ActivePowerRecord` as described in Figure 3.2. Afterwards, the `ActivePowerRecord` is serialized and published to the Kafka topic *input*. The History microservice is able retrieve the monitoring records from the topic, aggregate them, persist them in its database,

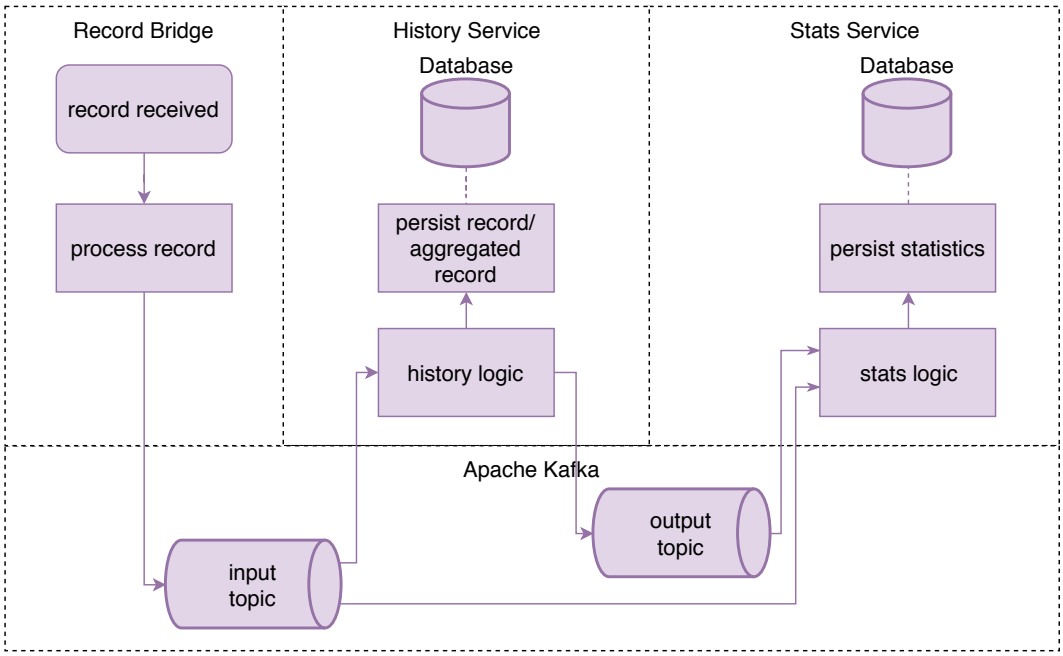

Figure 3.1. Data-Flow within the Titan Control Center (based on Henning et al. [2019]).

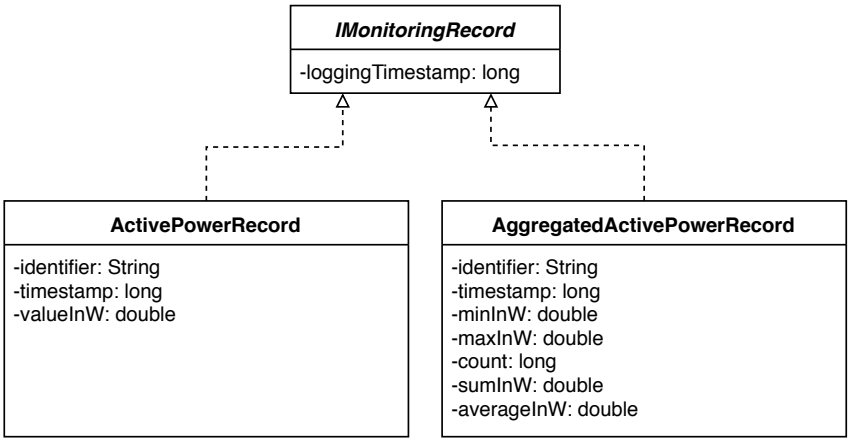

Figure 3.2. Monitoring-Records within the Titan Control Center (based on Henning et al. [2019]).

and publish the aggregated records to the *output* topic. As described in Figure 3.2, the aggregated records also have the fields *identifier* and *timestamp*, along with more fields that contain the aggregated measurements. Subsequently, all records can be processed by the

Stats microservice, which calculates and persists the respective statistics. Finally, when all records are persisted by the History microservice and the respective statistics by the Stats microservice, the processing of the data package from the sensor can be considered to be completed. After this point, the monitoring data is available to be requested by the visualization frontend of the Titan Control Center.

3.1.2 Organization of Sensors

The Titan Control Center receives data concerning the power consumption from physical sensors that themselves take measurements and send them to the monitoring system. Additionally, the user can group multiple sensors to abstract units, so-called aggregated sensors, which are used to compute aggregated measurements for all contained sensors. Moreover, aggregated sensors can be grouped by other aggregated sensors leading to a hierarchy of sensors with one root aggregated sensor, which we refer to as a sensor hierarchy. According to these sensor hierarchies, statistics and aggregated measurements are determined. At the current point there can only be one global hierarchy at a time, which the user can manage through the visualization frontend of the of the Titan Control Center. Moreover, when the hierarchy is updated, it is published to the other microservices via Kafka in form of a *JavaScript Object Notation* (JSON) string, which enables the history microservice to adjust the aggregation logic to the most recent hierarchy.

3.2 Requirements

Introduction of internal records

While processing monitoring records, as described in Figure 3.1, each record is associated with its sensor. This is achieved with the *identifier* field which contains the sensors original string-based identifier. Using string-based identifiers has the advantage that the operator of the system can freely choose the format of the identifiers which sensors are identified with. This, e.g., allows to use identifiers, a human can relate easily to, or that have been preset by the manufacturer of the sensors. Even if using ASCII[1]-based identifiers, each character needs to be represented by one byte. As at least for the `ActivePowerRecords` the remaining data only takes up 16 bytes in total, consisting of 8 bytes each for the fields *valueInW* and *timestamp*, the usage of string-based identifiers leads to a significant overhead with regard to memory, as an identifier does not contain any additional information which could be relevant for future computations. Moreover, we assume that handling these string-based identifiers makes the (de)serialization of monitoring records disproportionately complex. As a consequence, we assume the latency of the processing of records to be larger than necessary.

[1] American Standard Code for Information Interchange

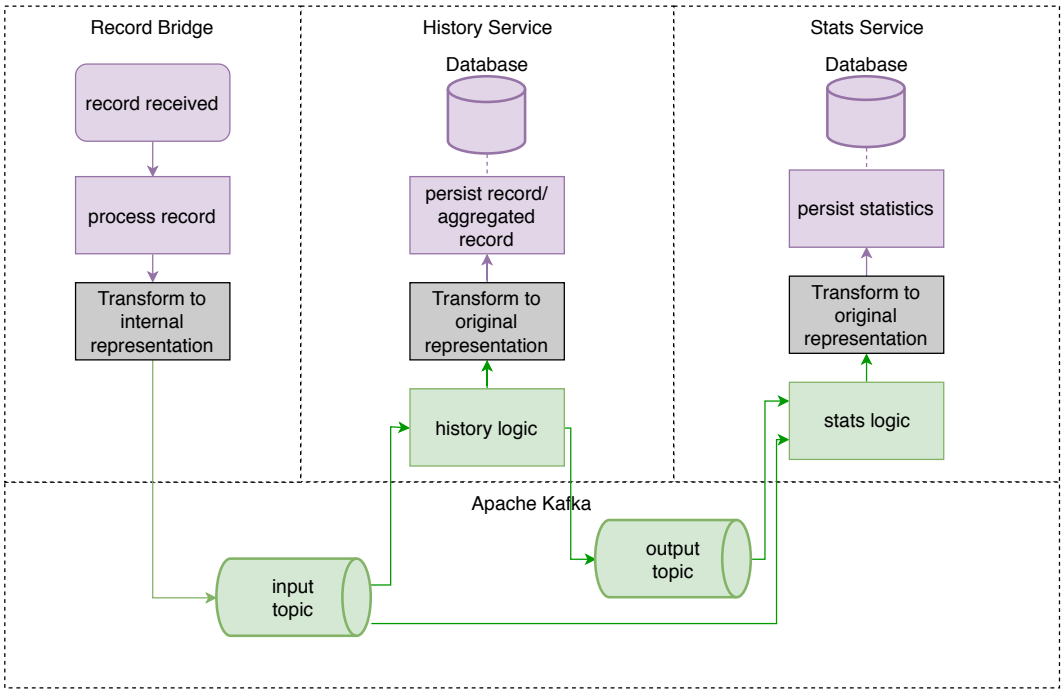

Figure 3.3. Data-Flow with internal records within the Titan Control Center (based on Henning et al. [2019]).

With respect to **G1** from Section 1.2, the decoupling of the internal representation of sensors from the environment, we encounter these problems by introducing an internal format for the monitoring records which uses integer-based identifiers to identify the respective sensors for the records. For our case, this is sufficient since we assume the domain of a 4 byte integer to be big enough to represent all sensors of the target infrastructure.

Assuming that there exists a bijective mapping between the internal integer-based and the original string-based identifiers of sensors, we can convert the original format of records to the internal format before performing computationally extensive operations, and vice versa, when we need to obtain the original format of the record. With these integer-based identifiers, the size of the identifier field would be fixed and bounded to 4 bytes which would decrease the overhead with regard to memory. As a result, we expect the latency to be less when internally using numeric identifiers than when identifying sensors with string-based identifiers.

Figure 3.3 shows the flow of sensor data considering the usage of the internal record-format. The components that are colored in violet indicate that the representation of records relies on string-based identifiers. We also denote these identifiers as original identifiers. Respectively, the green components indicate the computationally extensive parts of the

system, where the representation of records is based on the numerical identifiers which we will also call internal identifiers. Each grey box represents a transformation procedure from the representation based on string-based identifiers to numeric identifiers or vice versa.

Flexible Organization of Sensors

Due to the fact that there can only exist one global sensor hierarchy at a time, the abilities for the analysis of the monitoring data are rather limited. According to **G2** from Section 1.2, we enhance these abilities by allowing to create multiple sensor hierarchies. This enables the user to organize the sensors according to different aspects at the same time.

Moreover, when the global sensor hierarchy changes at the current state of the system, the whole hierarchy is published to the other microservices via Kafka in form of a JSON string. Consequently, other microservices need to parse the whole hierarchy in order to adapt their model to the changed hierarchy. With our approach we provide more granular information regarding the changes of the sensor hierarchies. For this, we take into consideration that the values of an aggregated sensor are the result of performing an aggregation over all measurement values of the contained physical sensors. However, already computed aggregations from contained aggregated sensors are not reused during the aggregation. Hence, we aim to provide the basis for improving the aggregation by reusing already computed values. Accordingly, we introduce two APIs, where the first provides the relationship between physical sensors and all their parent sensors, and the second provides the relationship between sensors and their direct parents. This way, we achieve a higher granularity and provide the History microservice with the necessary information for reusing already computed aggregation results.

3.3 Architectural Design

Considering the current state of the system (Section 3.1), we come to an architectural design that is aimed to fulfill the requirements derived in Section 3.2.

3.3.1 Overview

With regard to the basic principles of the microservices system architecture described in Section 2.4, we design the sensor management as a new microservice which replaces the existing Configuration microservice. Ensuring that the Sensor Management microservice is stateless, we address **G3** from Section 1.2, since statelessness allows to scale the service horizontally in order to reduce the utilization of individual instances and to increase the fault-tolerance due to redundancy.

Figure 3.4 visualizes the structure of our approach. The Sensor Management microservice maintains the functionality of the Configuration microservice and it contains additional

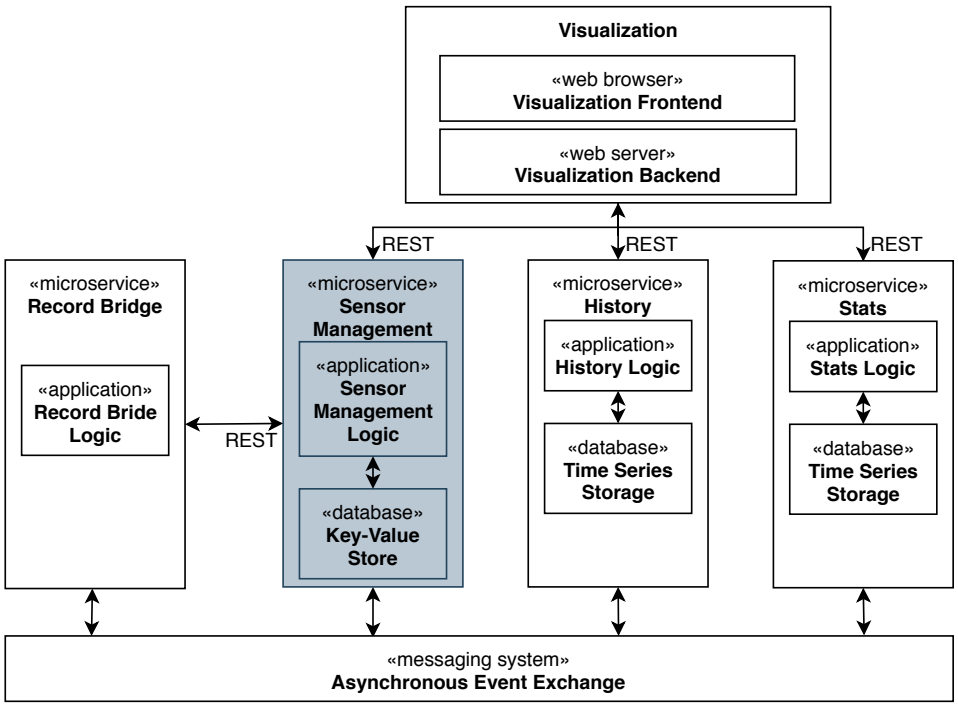

Figure 3.4. Our modified architecture of the Titan Control Center (based on Henning et al. [2019].). The introduction of the Sensor Management microservice is highlighted in blue.

business logic that is outlined in Section 3.3.2. As a consequence the structure of the remaining components can mostly be maintained and gradually adapted in the future to support the additional functionality of the Sensor Management microservice.

3.3.2 Sensor Management Microservice

The Sensor Management microservice has two central tasks: On the one hand, it is responsible for generating internal identifiers for sensors, maintain the mappings between original and internal sensor identifiers, and to allow other microservices to retrieve original and internal identifiers. On the other hand, it has the task of providing an interface for managing multiple sensor hierarchies at a time and of informing other components of the system about modifications of these sensor hierarchies. This leads us to the architecture described in Figure 3.5.

The microservice consists of three components. First, the *Sensor Identifier Registry* which is responsible for the generation and the retrieval of internal sensor identifiers, and second the *Sensor Hierarchy Repository* which is responsible for the management of the hierarchies of sensors. For both components, we use a key-value store database to persist data. Third,

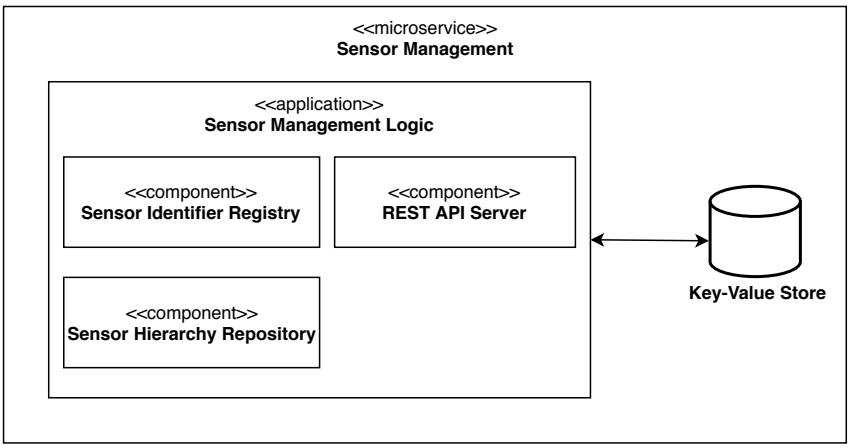

Figure 3.5. The architecture of the Sensor Management microservice.

the *REST API Server* component provides a REST API to access the Sensor Management microservice from other other components of the system via HTTP.

3.3.3 Adaptions to Other Components

The introduction of the Sensor Management microservice makes it necessary to adapt the remaining components of the system at certain points. That means, we adapt the microservices and also the visualization frontend of the system.

Microservices As indicated by Figure 3.3 the different microservices of the Titan Control Center need to transform between original and internal records. Because of that, the respective microservices need to perform HTTP requests to the Sensor Management API to retrieve the original or internal identifiers for sensors. As each mapping between an original and the corresponding internal sensor identifier does not change once created, we implement an in-memory cache for each component that stores mappings that are already retrieved. As a consequence, future records from sensors, which identifiers are already cached, can be transformed without performing additional HTTP calls.

Frontend As we allow to create multiple sensor hierarchies at a time, we need to adapt the visualization frontend in order to support the functionality of the REST API of the Sensor Management microservice. This means that we need to add a Vue component that allows the user to manage the list of sensor hierarchies. This includes creating, updating, and deleting sensor hierarchies. Additionally, we integrate the component to the visualization frontend presented by Henning [2018b, pages 49-56].

Chapter 4

Implementation

As described in Section 1.2, **G4** is to create a prototype implementation of the sensor management. In this chapter, we therefore present a realization of our architecture from Section 3.3 in Java. This includes the introduction of an internal format for monitoring records which allows us to decouple the internal representation of monitoring data from the environment (Section 4.1). After that, we discuss how the core of our implementation, the Sensor Management microservice, is realized (Section 4.2). Finally, we adapt the other microservices and the visualization frontend in order to be compatible with the Sensor Management microservice (Section 4.3).

4.1 Introduction of Internal Monitoring Records

We introduce the internal format by defining the classes `InternalActivePowerRecord` and `InternalAggregatedActivePowerRecord` for the internal representation of active power records and aggregated active power records. As indicated in Figure 4.1, these classes also implement the `IMonitoringRecord` interface from the Kieker framework. Both classes are equivalent to the existing classes `ActivePowerRecord` and `AggregatedActivePowerRecord` from Figure 3.2, except for the identifier field which has the type `int` instead of `String`. As mentioned earlier, there are several classes for representing statistic-related data within the Stats microservice. However, we do not explain the structure of these records and how we replace them with internal records, as the procedure is analogous to the active power records and the aggregated active power records.

As indicated in Figure 3.3, we replace the original records in the computationally extensive parts of the system with the internal records. These include the aggregation logic within the History microservice, the data within the Kafka topics, and the main logic of the Stats microservice which is responsible for calculating the different statistics.

4.2 Sensor Management Microservice

According to our architecture, the Sensor Management microservice consists of the three components REST API Server, Sensor Identifier Registry and Sensor Hierarchy Repository. The REST API Server allows to access the business functions of the service via a REST API. When a HTTP request is received, it delegates the request data to the corresponding

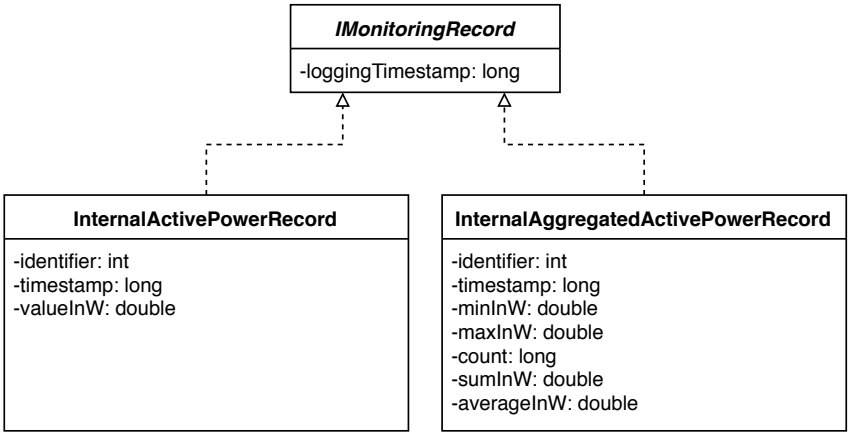

Figure 4.1. Structure of the internal records.

component, the Sensor Identifier Registry or the Sensor Hierarchy Repository which processes the request and returns the result. For both components the reliable key-value store etcd ist used as a database. For accessing etcd we use the official database client for Java *jetcd*[1] which encapsulates the respective gRPC calls.

4.2.1 REST API Server

To implement the REST API Server, we use the Java framework *Spark*[2]. It allows us to instantiate a web server object by specifying the host and the port. After that we can define routes on the server object for the different HTTP methods, whereby we can handle each request in a dedicated method. We have to consider routes for the two main functionalities of the Sensor Management microservice: First, routes for the retrieval of original and internal identifiers and second, routes for the management of the sensor hierarchies. Accordingly, the routes of the API of the Sensor Management microservice are defined in Table 4.1. When requesting one of the routes, the REST API Server invokes the respective component, the Sensor Identifier Registry or the Sensor Hierarchy Repository, which then handles the request. Hereby, the routes for managing the sensor hierarchies mainly correspond to the general *Create Read Update Delete* (CRUD) Operations. Additionally, we enable clients to request the list of identifiers and names of the existing sensor hierarchies. For each of the routes, the REST API Server invokes the respective method of Sensor Hierarchy Repository, which then requests the database in order to write or retrieve information, and returns the result.

[1] https://github.com/etcd-io/jetcd
[2] http://sparkjava.com/

Table 4.1. Routes for the retrieval of original and internal identifiers and the management of sensor hierarchies.

Route	Operation
`GET /sensor-id/:originalId/internal`	Retrieve internal ID for original ID
`GET /sensor-id/:internalId/original`	Retrieve original ID for internal ID
`POST /sensor-hierarchy`	Create a new sensor hierarchy
`PUT /sensor-hierarchy/:id`	Update a sensor hierarchy
`GET /sensor-hierarchy/:id`	Retrieve a sensor hierarchy
`GET /sensor-hierarchy/`	Retrieve all sensor hierarchies (IDs and names only)
`DELETE /sensor-hierarchy/:id`	Delete a sensor hierarchy

4.2.2 Sensor Identifier Registry

The main task of the Sensor Identifier Registry is to manage the bijective mappings between original and internal sensor identifiers and to allow to retrieve them. Therefore, the registry has to provide functionality for retrieving an internal identifier for an original identifier and vice versa. As our implementation should be fault-tolerant according to **G3** from Section 1.2, the data storage also should be designed fault-tolerant. As a consequence, it is essential to note that our application should work correctly even if we operate etcd as a cluster of database nodes. This makes it necessary to consider the CAP Theorem for our implementation. As in a cluster of database nodes, partition tolerance is inevitable, we have to trade off consistency against availability. For this, it is important to recall the guarantee of sequential consistency of etcd which we already described in Section 2.10.

Representation of Mappings between Original and Internal Identifiers

Since in etcd, we can only retrieve database entries by their key, we represent a mapping between an original identifier id_o and an internal identifier id_i in form of two key-value pairs that we insert into etcd. To avoid collisions of the inserted key-value pairs, when there accidentally exists an original identifier that is equal to an internal identifier, we assign a constant prefix to the keys for both key-value pairs, before we insert them into the store. Let $prefix_o$ be the prefix for the original identifiers, and $prefix_i$ be the prefix for the internal identifiers. A mapping (id_o, id_i) can be persisted by putting the key-value pairs $(prefix_o id_o, id_i)$ and $(prefix_i id_i, id_o)$ into etcd.

Generation of Mappings between Original and Internal Identifiers

We decide to generate the internal identifier for a sensor when it is requested for the first time at the Record Bridges. Consequently, once a sensor has received an internal

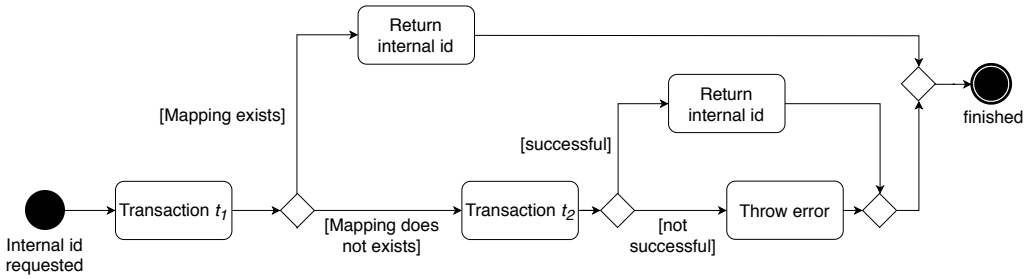

Figure 4.2. Process of retrieving an internal identifier.

identifier, it can be returned for all subsequent requests for that sensor. As etcd does not come with any auto-incrementing key functionality like in, for example, relational database systems, or any comparable counter, we have to implement the generation of internal identifiers by ourselves. We therefore implement a distributed counter which can be used to generate unique identifiers for the internal sensor representation. For this, we use etcd mini-transactions. As these are atomically executed, no concurrent operation can interfere while the transactions are being processed.

We have to reason about the three possible relationships between a sensors original- and internal identifier that could theoretically result from a naive generation of internal identifiers:

1. 1:1 relationship: One original identifier maps to exactly one internal identifier and vice versa.

2. 1:n relationship: One original identifier maps to multiple internal identifiers.

3. n:1 relationship: One internal identifier maps to multiple original identifiers.

The first scenario corresponds to the bijective mapping. This is what we aim to realize, so we do not have to consider it as problematic. The other two scenarios are indeed problematic as the mapping in both cases is not bijective. Therefore we have to ensure that the 1:n and n:1 relationship can not result from the generation of the internal identifiers.

Let *id_original* be the original identifier of a sensor S, let *prefix_original* be the key prefix for the original identifiers and let $lastIdKey$ be the key of the store entry that stores the distributed counter. The procedure *concat* represents the string concatenation. We can realize the generation of internal identifiers by using two consecutive mini-transactions t_1 and t_2, as visualized in Figure 4.2. The first transaction $t_1$is defined in Listing 4.1

Once an internal id is requested, this transaction performs a check whether there already exists an internal identifier for the sensor S. This is achieved by checking if there exists an entry in the store with the original identifier *id_original* of the sensor as the key, and which has been modified at a revision greater that 0. If this is the case, there is no need to generate an internal identifier because the mapping already exists and the corresponding internal

```
1  IF    getModRevision(concat(prefix_original, id_original)) > 0
2  THEN  get(concat(prefix_original, id_original))
3  ELSE  get(lastIdKey)
```

Listing 4.1. Pseudocode of the transaction t_1 which retrieves the last used value of the distributed counter.

```
1  IF    getModRevision(lastIdKey) == lastModRevision
2  THEN  put(concat(prefix_original, id_original), lastId+1)
3        put(concat(prefix_internal, lastId+1), id_original)
4        put(lastIdKey, lastId+1)
5  ELSE  nothing
```

Listing 4.2. Pseudocode of the transaction t_2 which increments the distributed counter and inserts a new mapping.

identifier is retrieved. Otherwise, the last internal identifier is retrieved which is then used for incrementing the distributed counter in the transaction t_2 defined in Listing 4.2.

Here, *lastId* is last internal identifier retrieved from t_1, *prefix_internal* is the key prefix for the internal identifiers, and *lastModRevision* is the revision of the distributed counter from t_1. This transaction performs a *test-and-set* operation that inserts the entries for a sensor mapping and increments the last inserted identifier. This is only done if the revision of the distributed counter has not changed since it was checked in t_1 that the sensor has not received an internal identifier before. Finally, the internal identifier is returned and the retrieval can be considered to be completed.

Since etcd provides sequential consistency, all nodes of the database cluster receive the operations in the same order. Furthermore, the key-value pairs for the mapping are only inserted once since in t_1 it is checked whether the mapping already exists. If this is the case, t_2 is not executed. Additionally, t_2 is only successful, if the last inserted identifier was not incremented by another process in the meantime. Consequently, every generated internal identifier is exactly assigned once. Therefore, with this model, we can ensure that 1:n and n:1 relations between original and internal identifiers can not occur and the mappings are always bijective.

However, there exists an edge case, where we have two or more processes and an interleaving execution of the transactions. Consider the execution visualized in Table 4.2.

The first column of **??** shows the time in abstract units, whereas the other two columns show the execution of the transactions t_1 and t_2 for two processes. Since t_1 is executed first for process 1 and directly afterwards for process 2, both processes receive the same value for the last generated identifier. Process 2 then succeeds with t_2, but afterwards t_2 fails for process 1, since the distributed identifier counter has been incremented by process 2 in the

Table 4.2. Interleaving execution of the transactions for two processes.

Time Unit	Process 1	Process 2
1	t_1	-
2	-	t_1
3	-	t_2
4	t_2	-

meantime.

When this case occurs, we throw an error which needs to be handled by the client of the Sensor Management microservice in a way that the retrieval of the internal identifier is retried. Nonetheless, it is guaranteed that there occurs no livelock, as among all competing processes, the one which executes t_2 first always succeeds. According to Tanenbaum [2009], this means that our implementation is starvation-free, since every process will retry and eventually successfully retrieve an internal identifier.

4.2.3 Sensor Hierarchy Repository

The Sensor Hierarchy Repository is responsible for managing the list of sensor hierarchies. Therefore it provides functionality to create, retrieve, update and delete hierarchies. As we allow to create multiple sensor hierarchies instead of just one, we need to identify each hierarchy. We achieve this, by identifying each hierarchy its root aggregated sensor. Additionally we allow to retrieve a list of all identifiers and names of all sensor hierarchies.

Each sensor hierarchy is stored as a key-value pair, where the key is the identifier of the hierarchy, and the value is the hierarchy itself in form of a JSON string, where each sensor in a hierarchy is represented by a JSON object with properties for the identifier of the sensor, a name, and optionally an array of child sensors with the same structure, if the sensor is an aggregated sensor. As we identify hierarchies with their root sensor, its identifier and name are considered to be the identifier and the name of the respective hierarchy. To avoid collisions with other keys used for the management of the mappings between original and internal sensor identifiers and to allow to retrieve all sensor hierarchies in one operation from the key-value store with an etcd range request, we define a constant prefix for all sensor hierarchy keys.

Creating Sensor Hierarchies When a POST request is sent to the route `/sensor-hierarchy`, we parse the body of the request which contains the sensor hierarchy that should be created. After that, we invoke the Sensor Hierarchy Repository which executes an etcd mini-transaction that performs a check if the registry already exists. If this is not the case and the registry has a valid JSON structure, the new registry is created. Otherwise, we

respond with a 409 (Conflict) HTTP status code.

Updating Sensor Hierarchies Updating a sensor hierarchy is similar to creating one. After having received a PUT request at the route `/sensor-hierarchy/:id`, where `id` is the identifier of the sensor hierarchy, we also parse the request body and invoke the Sensor Hierarchy Repository. In the following, we also perform an etcd mini-transaction which checks, whether the sensor hierarchy that we want to update exists and whether the new sensor hierarchy has a valid JSON structure. If this check succeeds, we overwrite the old sensor hierarchy with the updated one. Else, we respond with a 404 (Not Found) HTTP status code.

Retrieving Sensor Hierarchies For the retrieval of sensor hierarchies there are two cases. First, it is possible to retrieve a single sensor hierarchy by performing a GET request to `/sensor-hierarchy/:id`, where `id` is the identifier of the sensor hierarchy we want to retrieve. After that the Sensor Hierarchy Repository performs a *get* operation on etcd which returns the sensor hierarchy as a JSON encoded string. Second, we allow to retrieve the identifiers and names of all the existing sensor hierarchies. When a GET request is sent to the route `/sensor-hierarchy/`, we perform a range request over the prefix of all sensor hierarchy keys which returns the list of all sensor hierarchies. From this list, we filter the identifier and the name of each sensor hierarchy, which leads us to a list that is sent as a response.

Deleting Sensor-Hierarchies Deleting a sensor hierarchy can be done by sending a DELETE request to the route `/sensor-hierarchy/:id`, where `id` is the identifier of the sensor hierarchy that should be deleted. The request is delegated to the Sensor Hierarchy Repository which performs a *delete* operation on etcd.

4.2.4 Publishing of Sensor Hierarchy Related Events

According to **G2** from Section 1.2 we want to inform other components when the list of sensor hierarchies changes. We realize this by publishing changes of sensor hierarchies to the other microservices via Kafka, where the respective events are triggered by respective calls to the Sensor Hierarchy Repository. There are three main types of events that may occur: The first event occurs, when a new sensor hierarchy is added. The second event occurs when an existing sensor hierarchy is updated and the third event occurs when a sensor hierarchy is deleted.

When a new sensor hierarchy is created, it initially consists only of the root aggregated sensor. When a sensor hierarchy is deleted, we consider all contained physical and aggregated sensors as being deleted from that sensor hierarchy. When an existing sensor hierarchy is updated, we must compare the old- and the new version of the updated sensor hierarchy and publish information concerning all sensors which have been modified. We

provide two APIs: The first can be utilized for receiving information for the relationship of physical sensors with all of their parent sensors. The second can be utilized for receiving information for the relationship of both types of sensors, physical and aggregated sensors, with their direct parent sensors. For the first API, a physical sensor is considered as being modified within a sensor hierarchy if the set of its parent sensors is changed and for the second API, we consider a physical or aggregated sensor to be modified within a sensor hierarchy if its direct parent sensor is changed.

Format of Event Related Data

Since the aggregation of monitoring records in the Titan Control Center depends on the sensor hierarchies, we want to publish information about the relationship between sensors and their parent sensors with respect to the individual sensor hierarchies. We represent the event-related data for both APIs as pairs $(key, value)$, where the *key* itself in both cases is a pair of the internal identifier of the modified sensor and the identifier of the sensor hierarchy that contains the modified sensor. For the representation of the key-pair, we use the `Pair` type from the `org.apache.commons.lang3.tuple` package. The *value* of the representation depends on the type of information represented by it:

For the first API, the value consists of a set of internal integer-based identifiers from all the parent sensors, whereby we use the `Set` type from `java.util` package. Additionally, we wrap the set into an `Optional` which also is from `java.util`. As a consequence, we can encode the deletion of a physical sensor from a sensor hierarchy with an empty `Optional`.

For the second API, the value consists of an `Optional` containing the internal integer-based identifier of that direct parent sensor or an empty `Optional` if the sensor is the root sensor of the hierarchy, as this sensor does not have a direct parent. This `Optional` of the type `Integer` is wrapped by another `Optional` to encode the deletion of a sensor from the sensor hierarchy.

Publishing via Kafka

Based on the two formats of the event-related data for the two APIs, we create two Kafka topics that we use to publish event-related information every time changes to a sensor hierarchy are detected. For the first API, we use the topic *sensor-parents* to publish the relationship between physical sensors and their set of parent sensors. For the second API, we use the topic *sensor-parent* to publish the relationship between a physical or aggregated sensor and its direct parent sensor. Therefore, other microservices can subscribe to the changes of the list of sensor hierarchies in order to adapt to the current list of sensor hierarchies.

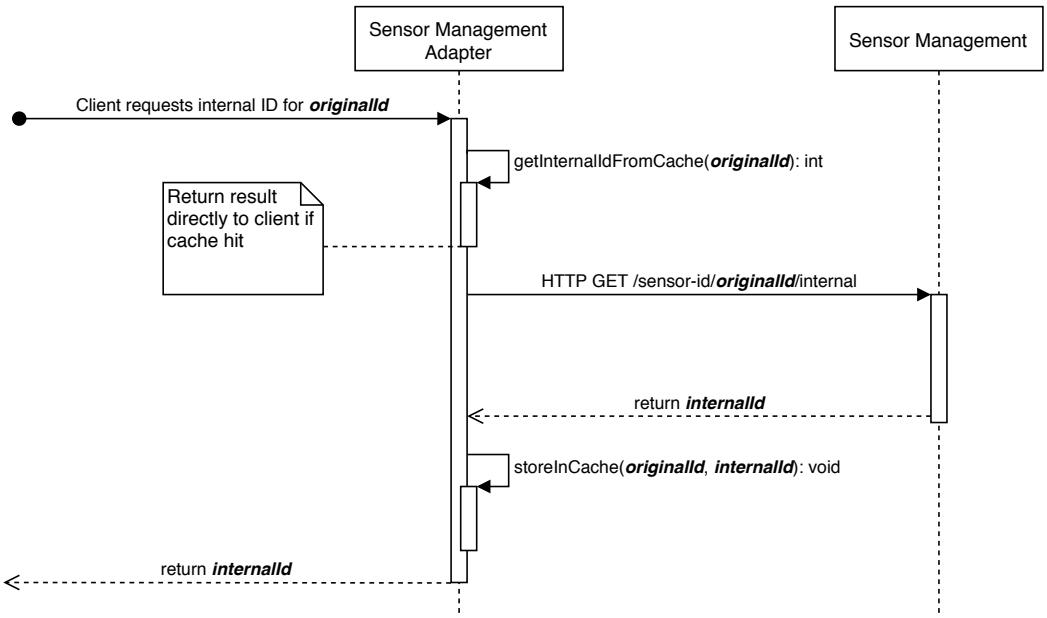

Figure 4.3. Interaction of an arbitrary client and the Sensor Management adapter.

4.3 Adaptions to Other Components

Additionally to the replacement of the original record format with the internal record format as mentioned in Section 4.1, we need to adapt the implementation of the other microservices in order to function appropriately in combination with the Sensor Management microservice.

4.3.1 Sensor Management Adapter

We provide a reusable component, the Sensor Management adapter which encapsulates HTTP calls to the REST API of the Sensor Management microservice for retrieving original and internal identifiers. Moreover, the Sensor Management adapter internally caches the retrieved mappings in memory by using a hash-map, which avoids unnecessary HTTP traffic. Figure 4.3 visualizes the procedure of retrieving an internal identifier for an original identifier of a sensor with the Sensor Management adapter. However, the procedure is analogous for the retrieval of an original identifier for an internal identifier. First a client which is considered to be another microservice, for example, an instance of a Record Bridge or the History microservice invokes the Sensor Management adapter. After that, the Sensor Management adapter performs a lookup operation whether the requested mapping is cached. If the lookup succeeds, the requested identifier is returned immediately to the client

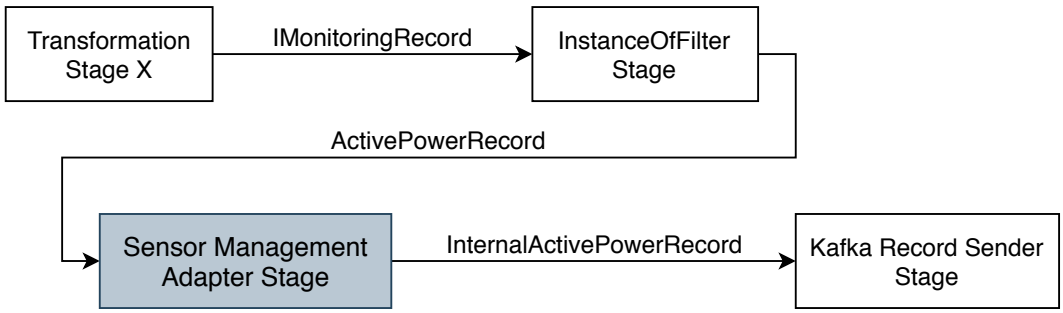

Figure 4.4. Pipe-and-Filter structure of the Record-Bridge Framework. The TeeTime stage added as part of our approach is highlighted in blue.

and the retrieval can be considered successful. Otherwise, the Sensor Management adapter performs a HTTP GET request to the REST API of the Sensor Management microservice requesting the internal identifier. Subsequently, after the response is received, the mapping is stored in the cache and the requested internal identifier is returned to the client.

Adaptions to the Record Bridge Framework

One central component of the Record Bridge framework is the `RecordBridge` class from the package `titan.ccp.kiekerbridge`. With the help of this class it is possible to create a Record Bridge from a stream of monitoring records or from a custom TeeTime configuration. However, the RecordBridge class is built on an internal TeeTime configuration. Two important TeeTime Stages in this configuration are the following:

1. *InstanceOfFilter Stage*: A stage that has one input port and filters objects that enter the Stage via the input port for objects that are instances of a specific type. After that, these objects are forwarded to a matching output port.
2. *Transformation Stage*: A stage that has one input port and one output port. The stage is used to transform instances of a certain type that enter the stage via the input port into instances of another type which are then sent to the output port.

To help developers with the implementation of custom Record Bridges, we integrate the Sensor Management adapter into the existing Record Bridge framework. This allows to use the framework without having to pay attention to the existence of the internal format for monitoring records. We achieve this by adding an additional TeeTime *Transformation Stage* to the underlying P&F architecture of the framework. Figure 4.4 illustrates the P&F structure of the Record Bridge framework with the additional *Sensor Management Adapter Stage*. Here, the *Transformation Stage X* represents an arbitrary TeeTime stage that can have an arbitrary number of preceding stages and needs to be created according to the needs of the respective Record Bridge. This stage outputs

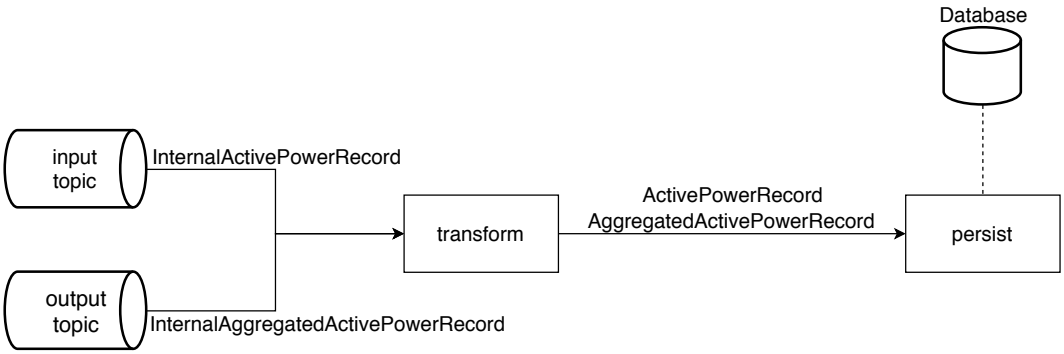

Figure 4.5. The data flow within the History microservice.

`IMonitoringRecords`. These `IMonitoringRecords` pass an *InstanceOfFilter Stage* which filters the input records for `ActivePowerRecords`. The *Sensor Management Adapter Stage* is a *TransformationStage* that retrieves the internal identifiers for the original identifiers of the incoming `ActivePowerRecords`, outputting `InternalActivePowerRecords` accordingly. Finally, these `InternalActivePowerRecords` are serialized and sent to Kafka in the *Kafka Record Sender Stage*.

Adaptions to the History Microservice

The History microservice is responsible for persisting both types of monitoring records, the `ActivePowerRecords` and also the `AggregatedActivePowerRecords`. As described in Figure 4.5, it consumes `InternalActivePowerRecords` and `InternalAggregatedActivePowerRecords` from the respective Kafka topic *input* which contains the `InternalActivePowerRecords` and the *output* topic which contains the `InternalAggregatedActivePowerRecords`. As the Kafka Streams API is used for the processing of the records from both topics, we can apply a for-each operation on the stream of records in which we transform the internal records to original records, before they are persisted in the database of the History microservice. This way, we can assure the consistency of the data even if the mappings between original and internal identifiers get lost. For the Stats microservice, the procedure is analogous. This means that we transform the internal statistic records to the respective original statistic records which are then persisted.

4.3.2 Visualization Frontend Integration

We add a new Vue component which is responsible for providing a graphical interface for managing the list of sensor hierarchies.

Figure 4.6 shows a screenshot of the rendered component. The component can be navigated to by clicking on the *Sensor Management* button (❶) in the left side-menu. Before

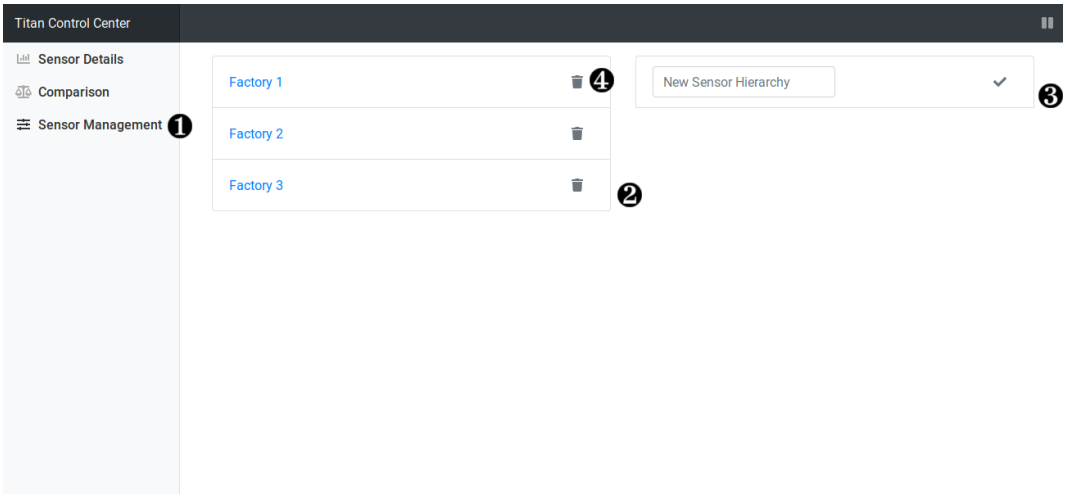

Figure 4.6. The rendered sensor hierarchy list component.

the component is loaded, we perform a HTTP GET request to the Sensor Management microservice to the route `/sensor-hierarchy/` in order to retrieve the identifiers and names of all existing sensor hierarchies. After that, we use the Vue directive *v-for* to render the list of sensor hierarchies (❷) dynamically from the fetched data. Moreover, on the right side, we add a menu (❸) which allows to enter a name for a new hierarchy. When submitting the form, we send the form data to the Sensor Management microservice via an HTTP POST request to the route `/sensor-hierarchy` and we add the new sensor hierarchy dynamically to the list. When clicking on an item of the list, the user is directed to a view based on Henning [2018b, page 55] where the respective sensor hierarchy can be updated. By clicking on the delete button (❹) of a list item, we can delete a sensor hierarchy from the list which is realized by performing an HTTP DELETE request to the route `/sensor-hierarchy/:id`, where `id` is the identifier of the sensor hierarchy and which also leads to the deletion of the respective item from the rendered list.

4.3.3 Consequences for Data Consistency

Taking into account that we store all monitoring data in their original form, meaning with the original sensor identifiers, we consolidate the consistency of all data that is stored in the databases of the History microservice and the Stats microservice. The reason for this is that the internal format for monitoring records is only used at runtime by the microservices and while the records are transmitted via Kafka. As a result, in case of a loss of data in the Sensor Management microservice that affects the mappings between sensor identifiers, only data within the Kafka topics is would be meaningless since it would not be possible to identify the sensors to which the individual internal monitoring records refer.

Chapter 5

Evaluation

Our final goal **G5** according to Section 1.2 is to evaluate the implementation of our approach. The evaluation is split into two main sections. In the first step, we evaluate the feasibility of our approach (Section 5.2) and in the second step we compare the performance of our approach with the performance of the current state of the Titan Control Center (Section 5.3).

5.1 Evaluation Environment

We deploy the different components of the Titan Control Center on an Ubuntu 18.04.2 machine with the following hardware:

CPU	Intel(R) Core(TM) i5-6200U
Clock Rate	2.30∼GHz
Cores	2 (4 virtual)
RAM	8 GB

We start the individual microservices from the Eclipse IDE[1] (version 2019-03(4.11.0)). While performing the evaluation, we do not run other processes than the ones necessary for the evaluation. For accessing the visualization frontend of the system, we use the Chromium web browser (version 76.0.3809.87).

5.2 Evaluation of Feasibility

In the following section, we evaluate whether our implementation fulfills the functional requirements. The general approach for the evaluation of the feasibility is to evaluate selected scenarios which each are comprised of manual integration tests.

5.2.1 Sensor Management Adapter Test Tool

We implement a Sensor Management adapter test tool in order to be able to verify that the generation and the retrieval of identifiers is working correctly for certain scenarios. This tool enables us to generate internal identifiers and retrieve internal and original identifiers

[1] Integrated development environment

for each other by invoking the Sensor Management adapter introduced in Section 4.3.1 and logging the results to the standard output.

5.2.2 Sensor Hierarchy Event Reader Tool

We create an event reader tool which utilizes the Kafka Consumer API in order to subscribe to the changes that have been performed to the individual sensor hierarchies. We process the incoming events by also logging the contained data to the standard output.

5.2.3 Methodology

Our approach for the evaluation of the feasibility consists of two main parts. The first is the evaluation of the retrieval of internal and original identifiers which we address in scenario 1 to 3. The second is the evaluation of the management of sensor hierarchies which we address in the remaining scenarios 4 to 6. For all six scenarios, we start the Titan Control Center in our environment described in Section 5.1 from our IDE. We decide to not horizontally scale any of our services since we are only interested in analyzing central aspects of our implementation. For generating synthetic sensor data, we use the Sensor Simulation Tool from Henning [2018b, page 61] in combination with the proper Record Bridge. In our implementation, the retrieval of internal and original identifiers is encapsulated by the Sensor Management adapter. As the different microservices only access the Sensor Management microservice by means of the adapter, we consider it as being sufficient to evaluate identifier retrieval by examining the Sensor Management adapter in scenario 1 to 3. For this, we use the test tool from Section 5.2.1.

For scenario 4 to 6, we configure the Sensor Simulation Tool to generate measurements for the existing physical sensors. Additionally, we set up the Sensor Management microservice so that a default sensor hierarchy is loaded initially. We trigger the creation, the update and deletion of a sensor hierarchy manually via the visualization frontend. After that, we try to validate the triggered operation depending on the scenario by either retrieving the list of sensor hierarchies or by retrieving the sensor hierarchy that has been created or updated. Moreover, we run the event reader tool from Section 5.2.2 during the evaluation in order to verify that the correct events are published to the respective Kafka topics.

Scenario 1: Generate an Internal Sensor Identifier

In the first scenario we validate the generation of internal identifiers for original identifiers. We start all components of the Titan Control Center without simulating any sensors. Moreover, we do not load any sensor hierarchy on startup of the Sensor Management microservice. Therefore, the system has no information about any sensors that might exist initially. In the next step, we invoke the Sensor Management adapter with the test tool in order to retrieve an internal identifier for the original identifier *new-original-id*.

Scenario 2: Retrieve an existing Internal Sensor Identifier

The second scenario constitutes the retrieval of internal identifiers for existing mappings. It builds on the first scenario where an internal identifier has been generated for the original identifier *new-original-id*. This means that we do not need to reset the databases of the different microservices after having evaluated scenario 1, but we need to restart the test tool to ensure that the retrieved internal identifier from scenario 1 is not cached in memory by the Sensor Management adapter. We restart the test tool and we let the Sensor Management adapter retrieve the existing internal identifier for the original identifier *new-original-id*.

Scenario 3: Retrieve an existing Original Sensor Identifier

In the third scenario we validate the retrieval of original identifiers for existing mappings. This scenario also builds on scenario 1 as there must already exist a mapping. We start the test tool and call the Sensor Management adapter in order to retrieve the original identifier of the mapping generated and retrieved in scenario 1 and 2.

Scenario 4: Create a new Sensor Hierarchy

The 4th scenario involves verifying whether it is possible to create new sensor hierarchies. Therefore, we start all microservices of the Titan Control center, including the visualization frontend. Moreover, we configure the Sensor Simulation Tool to additionally simulate 4 physical sensors. We open the visualization frontend in the web browser and add a new sensor hierarchy with the name *New Factory* through the Vue component we introduced in Section 4.3.2. After that, we reload the website, which forces the web browser to reload the data from the server and we check whether the new sensor hierarchy is rendered as an item of the hierarchy list. Simultaneously, we run the event reader in order to check if the correct event-related data is published to the according Kafka topics.

Scenario 5: Update an existing Sensor Hierarchy

The 5th scenario builds on the 4th scenario, where we created a new sensor hierarchy *New Factory*. Moreover, the following parts of this scenario build on each other. First, we validate if we can add sensors to the hierarchy created in scenario 4. For this, we create two aggregated sensors via the visualization frontend and we assign the existing physical sensors (1-4) to the aggregated sensors. This leads us to an updated sensor hierarchy as described in Figure 5.1. After that, we save the updated sensor hierarchy and we reload the website in order to check if the changes have been persisted in by the Sensor Management microservice.

Second, we move the *Aggregated Sensor 1* to the *Aggregated Sensor 2*, which leads us to the updated hierarchy as described in Figure 5.2 We also save the updated hierarchy and reload the website in order to check if the changes have been persisted.

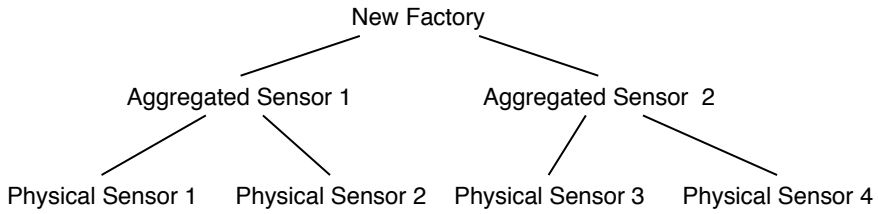

Figure 5.1. The sensor hierarchy after having added the aggregated and physical sensors.

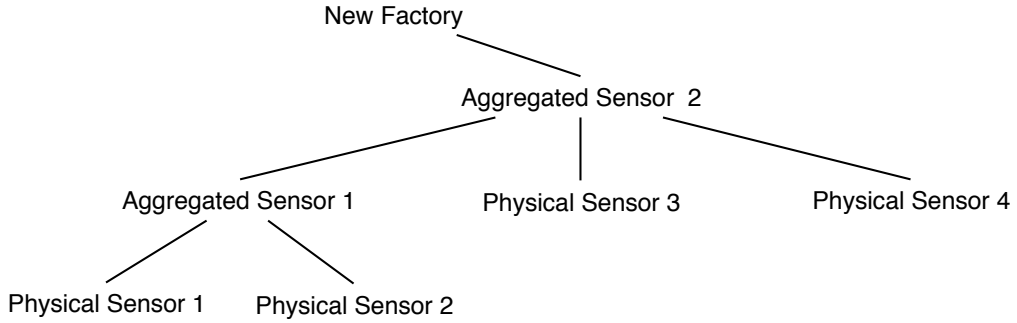

Figure 5.2. The sensor hierarchy after having moved *Aggregated Sensor 1* to *Aggregated Sensor 2*.

Third, we delete the *Aggregated Sensor 1* from the hierarchy. Accordingly, all contained physical sensors are also deleted from the hierarchy. The resulting hierarchy is visualized in Figure 5.3. Analogously to the previous operations, we reload the website in order to check if the changes have been persisted. As in scenario 4, we run the event reader tool while we carry out the operations of this scenario, in order to check if the correct event-related data is being published to the according Kafka topics.

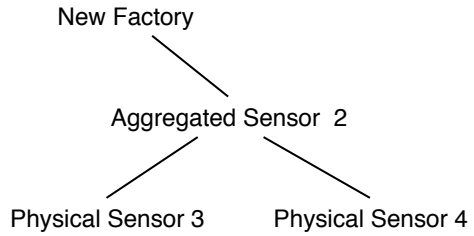

Figure 5.3. The sensor hierarchy after having deleted *Aggregated Sensor 1* and all its contained sensors.

Scenario 6: Delete an existing Sensor Hierarchy

The 6th scenario describes the deletion of a sensor hierarchy. We delete the hierarchy created in scenario 4. We trigger this manually by clicking the *delete* button for the corresponding item in the sensor hierarchy list described in Section 4.3.2. Afterwards, we reload the website in order to check if the deleted hierarchy has been removed from the list of hierarchies. Additionally, we verify that the correct information has been published to Kafka by checking the output of the sensor hierarchy event reader tool.

5.2.4 Results and Discussion

In the following section, we describe our observations for each scenario and we discuss whether the results match our expectations on the behaviour of our implementation.

Scenario 1: Generate an Internal Sensor Identifier

For scenario 1, the Sensor Management microservice returns the internal identifier *1* for the original identifier *new-original-id*. This is the expected result, as there where no internal identifiers generated previously. We conclude that generating internal identifiers is working as expected.

Scenario 2: Retrieve an existing Internal Sensor Identifier

For scenario 2, the Sensor Management microservice returns the internal identifier *1* for the original identifier *new-original-id*. This is expected since there already exists a mapping for the original identifier *new-original-id*. We therefore conclude that retrieving internal identifiers for existing mappings is working correctly.

Scenario 3: Retrieve an existing Original Sensor Identifier

For scenario 3, the Sensor Management microservice returns the original identifier *new-original-id* for the internal identifier *1*. This is the expected result as the original identifier *new-original-id* maps to the internal identifier *1* (see scenario 1). We conclude that retrieving original identifiers for existing mappings is working. Together with the results from scenario 1 and 2, this means that generating internal identifiers and retrieving both types of identifiers is working correctly.

Scenario 4: Create a new Sensor Hierarchy

After having created the new sensor hierarchy with the name *New Factory*, the corresponding item appears in the list of sensor hierarchies. Reloading the page yields to the same list, which means that the changes have been made durable successfully. The sensor hierarchy

Table 5.1. Events for the relationship between sensors and their direct parents for creating a hierarchy.

Sensor	Status
New Factory	*no parent*

event reader tool is able to detect an event for the new sensor hierarchy *New Factory*. Although the information is provided in form of internal sensor identifiers, we here visualize the results with the names of the respective sensors to increase the comprehensibility. To find out to which sensor the internal identifiers refer, we use the *etcdctl*[2] command line client for etcd that allows us to query the key-value store. For the relationship between a sensor and its direct parent we receive the information depicted in Table 5.1. The value *no parent* for the direct parent sensor indicates that the sensor is the root sensor of the hierarchy. The event signals that a new root aggregated sensor with the name *New Factory* is added to the new hierarchy. This is expected, as we identify sensor hierarchies with their root aggregated sensor. For the relationship between physical sensors and all their parents, we do not recognize any events since a created sensor hierarchy does not contain any physical sensors initially. This means both the topics *sensor-parents* and *sensor-parent* provide the correct information according to the creation of a sensor hierarchy.

Scenario 5: Update an existing Sensor Hierarchy

Having added the sensors, the updated hierarchy remains the same after reloading the website. This means, the changes have been persisted successfully. The event reader tool detects the event-related information shown in Table 5.2 for the relationship between physical sensors and all of their parent sensors. For the relationship between sensor and their direct parent sensors, the events are depicted in Table 5.3.

After having moved the *Aggregated Sensor 1* to the *Aggregated Sensor 2*, the hierarchy remains the same after reloading the website. The event reader tool detects the data visualized in Table 5.4 for the relationship between physical sensors and all their parents. For the relationship between sensors and their direct parents, the data shown in Table 5.5 is received.

After having deleted the *Aggregated Sensor 1*, we determine that the hierarchy has been persisted as the hierarchy remains the same after reloading the page. At the same time, the event reader tool receives the information depicted in Table 5.6 for the relationship between physical sensors and all of their parent sensors. For the relationship between sensors and their direct parents, we receive the data shown in Table 5.7.

We can verify that the emitted events correspond to the changes performed to the hierarchy for adding, moving, and deleting sensors. Therefore, we conclude that updating sensor hierarchies is working as expected for the selected scenarios.

[2]https://github.com/etcd-io/etcd/tree/master/etcdctl

Sensor	Parents
Physical Sensor 1	New Factory Aggregated Sensor 1
Physical Sensor 2	New Factory Aggregated Sensor 1
Physical Sensor 3	New Factory Aggregated Sensor 2
Physical Sensor 4	New Factory Aggregated Sensor 2

Table 5.2. Events for the relationship between physical sensors and all their parents for adding sensors.

Sensor	Direct parent
Aggregated Sensor 1	New Factory
Aggregated Sensor 2	New Factory
Physical Sensor 1	Aggregated Sensor 1
Physical Sensor 2	Aggregated Sensor 1
Physical Sensor 3	Aggregated Sensor 2
Physical Sensor 4	Aggregated Sensor 2

Table 5.3. Events for the relationship between sensors and their direct parents for adding sensors.

Sensor	Parents
Physical Sensor 1	New Factory Aggregated Sensor 2 Aggregated Sensor 1
Physical Sensor 2	New Factory Aggregated Sensor 2 Aggregated Sensor 1

Table 5.4. Events for the relationship between physical sensors and all their parents for moving sensors.

Sensor	Direct parent
Aggregated Sensor 1	Aggregated Sensor 2

Table 5.5. Events for the relationship between sensors and their direct parents for moving sensors.

Sensor	Status
Physical Sensor 1	*deleted*
Physical Sensor 2	*deleted*

Table 5.6. Events for the relationship between physical sensors and all their parents for deleting sensors.

Sensor	Status
Aggregated Sensor 1	*deleted*
Physical Sensor 1	*deleted*
Physical Sensor 2	*deleted*

Table 5.7. Events for the relationship between sensors and their direct parents for deleting sensors.

Sensor	Status
Physical Sensor 3	*deleted*
Physical Sensor 4	*deleted*

Table 5.8. Events for the relationship between physical sensors and all their parents for deleting a sensor hierarchy.

Sensor	Status
New Factory	*deleted*
Aggregated Sensor 2	*deleted*
Physical Sensor 3	*deleted*
Physical Sensor 4	*deleted*

Table 5.9. Events for the relationship between sensors and their direct parents for deleting a sensor hierarchy.

Scenario 6: Delete an existing Sensor Hierarchy

When a sensor hierarchy is deleted, it is removed from the list of hierarchies. Moreover, this remains the same after reloading the website. We conclude that the deletion of a sensor hierarchy is persistent. For the relationship between physical sensors and all their parents, the event reader tool receives the data visualized in Table 5.8. For the relationship between sensors and their direct parents, we detect the events shown in Table 5.9.

As the emitted events correspond to the deleted sensor hierarchy, we conclude that both APIs provide the correct information for deleting a sensor hierarchy. Hence, we conclude that deleting an existing sensor hierarchy is working as expected.

5.2.5 Threats to Validity

We only evaluated the feasibility of our approach in the environment described in Section 5.1. This includes the hardware environment on the one hand and on the other hand the web browser used for the evaluation of managing the sensor hierarchies. As a consequence, our results are only conditionally transferable to other environments since there may be differences in the behaviour in a real distributed environment with horizontally scaled microservices, or the compatibility of the visualization frontend may not be given for arbitrary web browsers. Additionally, the generation and the retrieval of sensor identifiers has only been evaluated by means of the Sensor Management adapter. Therefore, we can not conclude that the generation and retrieval is working correctly regardless of the systems state. In general, our evaluation only comprises central scenarios and does not cover all particular edge cases.

5.3 Evaluation of Performance

The evaluation of performance is separated into two main parts. In the first step, we analyze the performance of the current version of the system and in the second step we analyse the

performance of our approach. After that, we discuss the results of our analysis.

5.3.1 Methodology

For both parts, the experimental setup is the same, except for the respective version of the Titan Control Center that can either be the current version or the version of our approach. As a deployment environment we use the environment described in Section 5.1. Within the scope of this work, we choose to only evaluate the performance in form of the latency for the processing of records in the Record Bridges and the History microservice. We make some adaptions to the current version of the system as well as to our implementation: First, we create a Record Bridge that simulates one physical sensor by generating artificial active power records with a frequency of 5 records per second. Each time a record is generated, we set the *timestamp* of the record to the current unix epoch time in milliseconds. Therefore, each record contains the information when itself has been created. Moreover, we adapt the History microservice so that before the active power records and the aggregated active power records are persisted in the database, we again measure the current time in milliseconds. Together with the timestamp of the record itself we are able to calculate the latency for the processing of records from the moment they are received until they are persisted by the History microservice. We configure the Sensor Management microservice so that it loads an initial sensor hierarchy that consist of a root aggregated sensor which contains the sensor that is simulated by the Record Bridge. Additionally, we configure Kafka Streams so that the incoming active power record immediately triggers the creation of an aggregated active power record at the History microservice. As a consequence, we have the same amount of aggregated active power records as active power records.

We execute 14 scenarios, where in each we analyze the latency within the system for a certain length of original identifiers. We start with a length of 2^2 characters in the first scenario and we double the identifier length with every scenario until we have an id length of 2^{15} characters in the 14th scenario. All original identifiers consist of characters that can be encoded by two bytes each when UTF-8 encoded, except for the first character which only requires one byte. As a consequence, the amount of bytes required to represent an original identifier is bounded by 2 times the length of the original identifier. For each scenario, we emit 2.000 active power records, where we do not consider the first 1.000 records because of the warmup time of the Java Virtual Machine [Horký et al. 2015]. This way it also is ensured that we do not consider the first few calculated latencies since these are affected by the Sensor Management adapter performing HTTP requests in order to retrieve the sensor identifiers before they are eventually cached.

5.3.2 Results

The results of the analysis for the latency of records of the physical sensor is shown in boxplot diagrams in Figure 5.4 and Figure 5.5. Each box represents the measured latencies that lie between the first and third quartile which means that each box represents 50% of

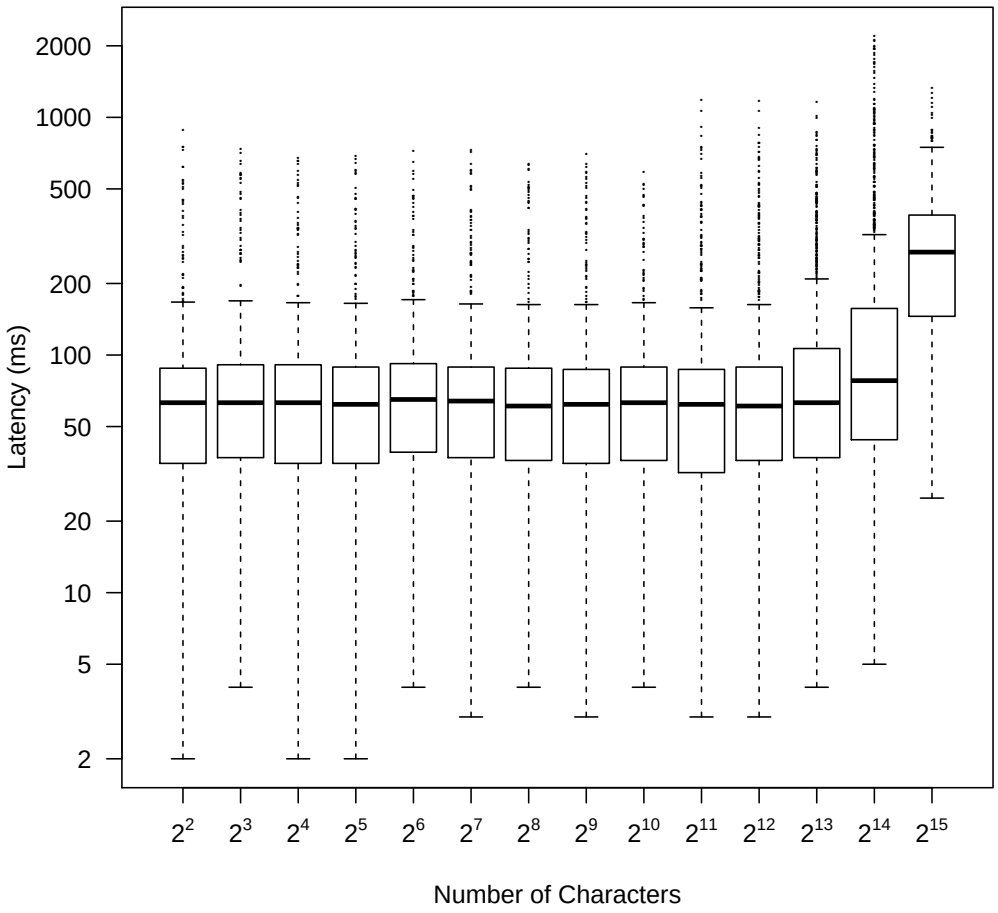

Figure 5.4. Results performance analysis of the current version of the Titan Control Center for the physical sensor.

the sample. Additionally, the vertical lines between the upper and lower horizontal marks show the total range of the sample. The upper bound of each line represents the maximum value and the lower bound of each line represents the minimum value of the sample accordingly. Statistical outliers are represented by the points above the upper horizontal mark. A value is considered a statistical outlier, if it is greater than the 3rd quartile plus 1.5 times the range between the 2nd and 3rd quartile.

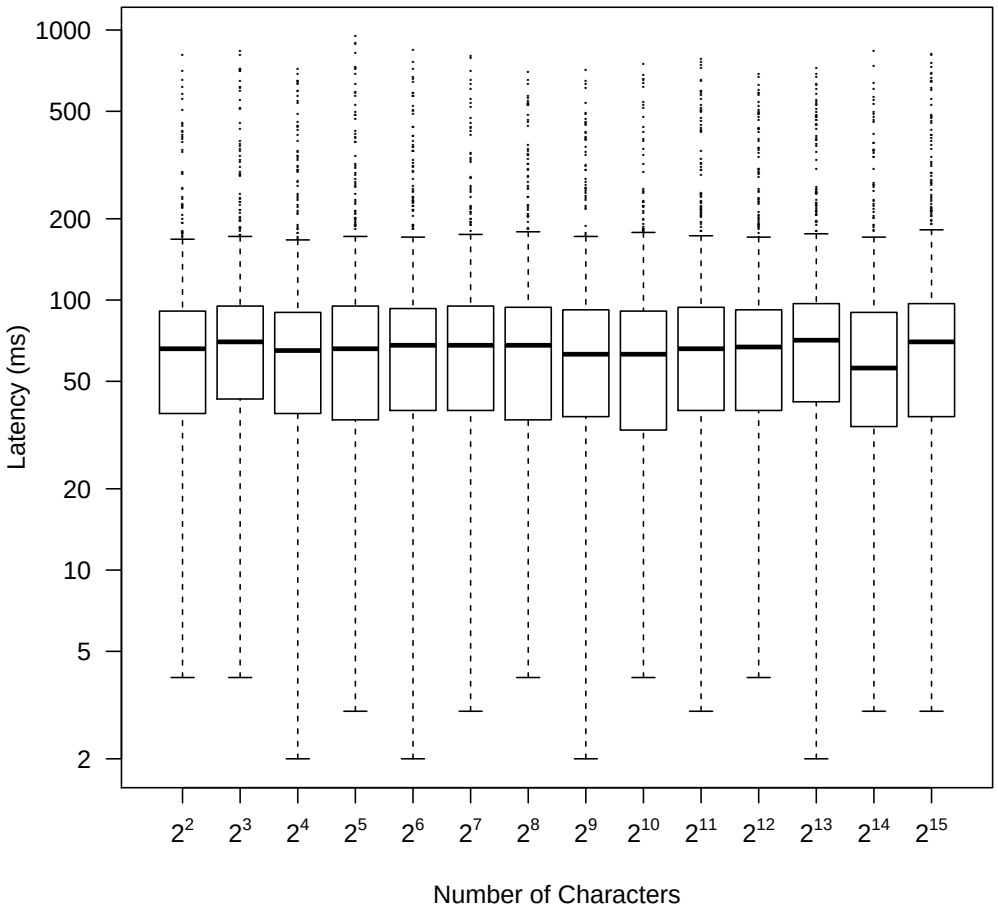

Figure 5.5. Results performance analysis of our version of the Titan Control Center for the physical sensor.

Figure 5.4 visualizes the results for the current version of the system. We see that the latency for the most records is between 40ms and 100ms up to the sample with an identifier length of 2^{13} characters. For larger identifiers, an increase of the latency is observable since for 2^{14} characters, 50% of the sample lie between approximately 50ms and 170ms and for 2^{15} characters, the 2nd quartile is at ca. 165ms and the 3rd quartile at ca. 400ms.

The results of the analysis of our approach are shown in Figure 5.5. Here, the majority

of the measured latencies is also between 40ms and 100ms and we can not determine an increase of the latencies for larger identifiers at all.

Figure 5.6 and Figure 5.7 show the results of the analysis of the latency for aggregated records. Figure 5.6 shows the results for the current version. Similarly to the results for the records from the physical sensor in the current version, we can only observe an upward trend for the latencies for identifiers larger than 2^{13}. In general, 50% of the latencies lie between approximately 250ms and 400ms up to a length of 2^{10} characters. After that the latencies drop before increasing massively for an identifier length of 2^{14} and 2^{15} up to a point where 50% of the sample lie between ca. 700ms and 1600ms for 2^{15} characters. In Figure 5.7 the results from our approach and the aggregated sensor are depicted. As at the results for our approach for the records from physical sensors, the latency is mostly constant for all lengths of identifiers examined at mostly 300ms to 400ms. This is, similarly to the results for the aggregated records in the current approach, much higher than for the normal monitoring records.

5.3.3 Discussion

The fact that the latency for the current version only increases significantly for identifiers with a length greater than or equal to 2^{14} is surprising, as we assumed that the latency would increase for much shorter identifiers. Moreover, for identifier lengths less than 2^{14}, there is no significant improvement observable for normal and aggregated monitoring records. Using the internal format for identifying sensors yields to approximately the same latencies for both approaches. However, for very large identifiers, our observations correspond to the expected results, meaning that the latency for the current approach increases while it remains mostly constant for our approach. We explain this with the fact that our introduced format uses fixed-sized identifiers and therefore, the length of the original identifiers is considered to have no practical influence on the latency.

Altogether we conclude that our results do not justify using an internal format for the monitoring records instead of using the original format as we were not able to show that our approach has performance advantages for realistic scenarios, where the identifier length is less than 2^{14}. Notwithstanding, we suggest to perform a more sophisticated analysis of the performance in a more dedicated environment, e.g., in a real distributed environment with multiple nodes. This way, we may achieve more meaningful results.

5.3.4 Threats to Validity

It is important to note that the environment we used is not specifically designed to run applications like the Titan Control Center and we also run all components of the Titan Control Center on one physical machine. Thus, these different components are additionally competing for computational resources. As a consequence, next to the desktop environment which needs to be run, there are many more processes that run in parallel to the Titan Control Center which would not exist in a production environment.

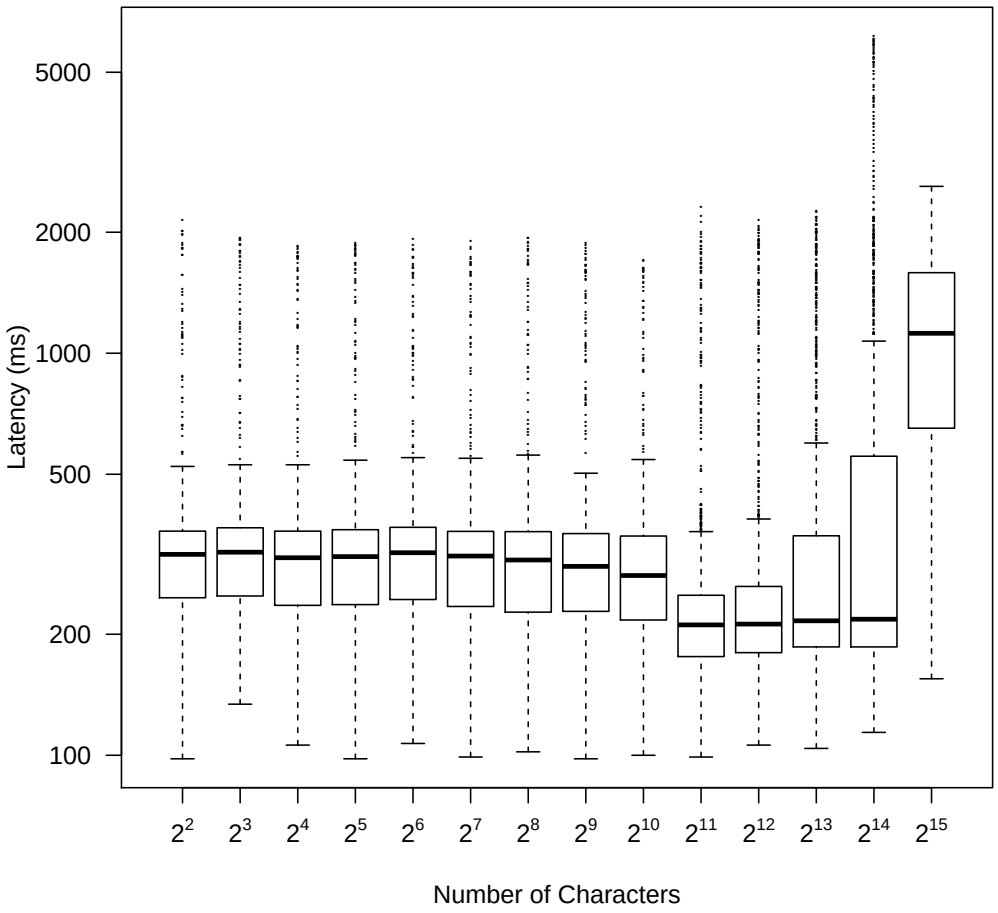

Figure 5.6. Results performance analysis of the current version of the Titan Control Center for the aggregated sensor.

We identify two possible reasons for the latencies being generally large in our samples: First, there may be an unidentified factor which has a higher influence on the latency than the size and type of the sensor identifiers. As a consequence, the difference between both approaches would not be measurable. Second, using numeric internal identifiers may not be appropriate for reducing the latency for the processing of monitoring records.

We explain the fluctuation in our samples with the lack of multitasking abilities of the

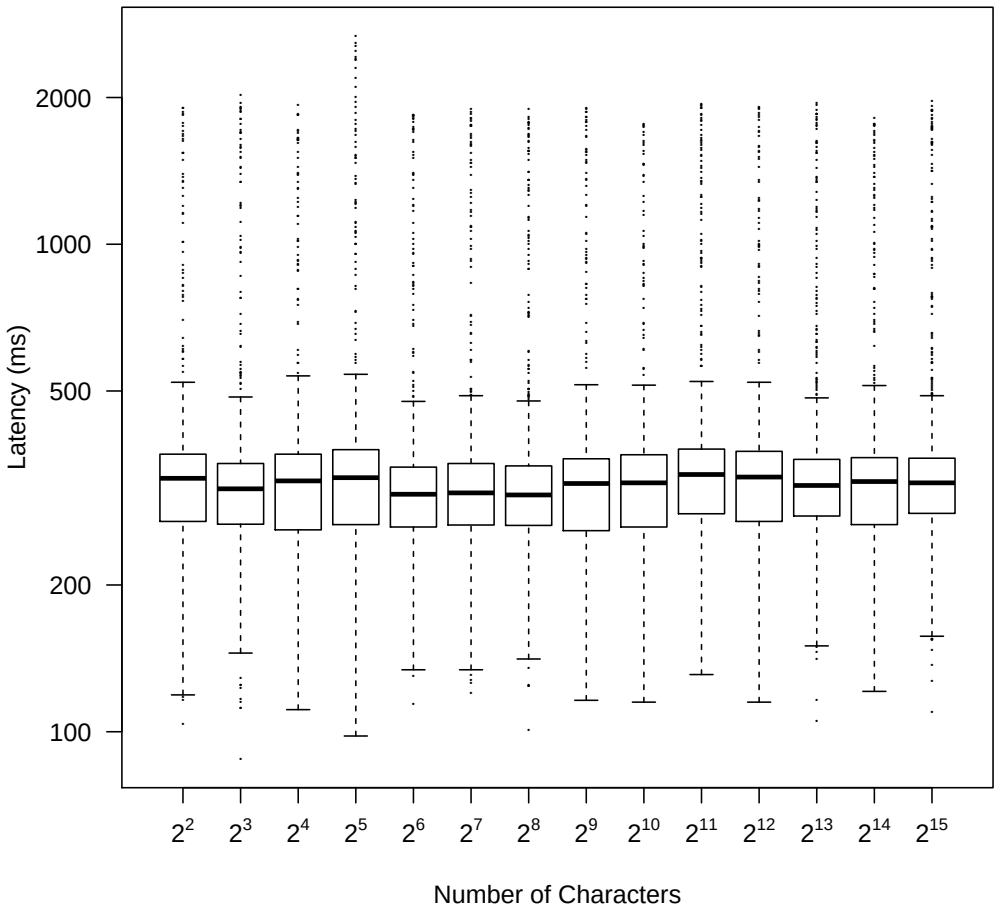

Figure 5.7. Results performance analysis of our version of the Titan Control Center for the aggregated sensor.

environment to cope with all the processes running at the same time. As a consequence, the results of our analysis only give a very limited impression how the system may perform in a production environment. Moreover, we only have evaluated the performance in by measuring the latency for the processing of monitoring records from the Record Bridges to the History microservice. Accordingly, we have do not know whether there may be any significant difference in the latencies for the processing of records within the Stats

microservice. Finally, we also have only evaluated the performance by means of the latency. Hence, we can not conclude what the impact or our approach has on the throughput for monitoring records.

Chapter 6

Related Work

This work builds on the Titan Control Center introduced by Henning [2018b] which is the component of the Titan platform that is responsible for the monitoring of the electrical power consumption. Our approach for a sensor management can be regarded as an extension of the existing system. The information, gathered by the monitoring with the Titan Control Center, is central for the Industrial DevOps approach proposed by Hasselbring et al. [2019]. The authors introduce a coherent cycle consisting of monitoring and analysis, identification of requirements, adapting or reconfiguration of the system, and automatically testing the adjustments before replacing the software in production where it is monitored again. Moreover, a organizational structure, which encourages people from different domains to work together in order to allow quick adjustments of the system, is presented.

6.1 Other Sensor Management Approaches

During this work, we found some IoT[1]-related publications that cover similar aspects as our approach: Pham et al. [2016] present a system architecture for the management of heterogeneous devices in a home network which is able to provide monitoring functions. For this, the authors make a distinction between direct and indirect device management. In the direct management approach, devices communicate directly with the management service. In contrast, the indirect approach involves a home gateway (HGW) that handles issues related to the protocols used within the home network. They combine both approaches depending on ability of the devices to be able to communicate with the management service by themselves. The authors propose a simple and an intelligent implementation of the HGW, whereas the simple implementation only forwards data to the management and the intelligent implementation performs a preprocessing of data before it is forwarded to the management. Therefore, the HGW can be considered as an example for edge computing as described by Giang et al. [2018]. In the Titan Control Center, the Record Bridges can function as edge components that operate similarly to the HGW, if they are deployed in the production environment along with the target infrastructure [Henning 2018b]. Similarly to the visualization of the Titan Control Center, the authors provide a web interface in

[1] Internet of Things

order to access the data gathered by the management, but they also allow or to control the devices in the home network.

Cai et al. [2014] describe a design for a management system for devices with low power Wi-Fi based sensors. These sensors are able to communicate over network on IP based protocols such as CoAP[2] which is a protocol for IoT based on REST. Similarly to our work, the authors introduce the sensor management as a component that is loosely coupled with an IoT application. Different to our design, the approach does not explicitly constitute a microservice architecture. Instead, the sensor management is realized with a dedicated Management Server (MS) that provides functionality for registering, configuring and monitoring sensors. The IoT application has an interface that allows users to observe and monitor the sensors indirectly over the MS and directly without the MS. In contrast to our approach, there is a built in authentication mechanism that ensures that only authorized sensors can communicate with the system. Once a sensor is authenticated, the MS observes and monitors the sensor. Thereby, the monitoring includes the health of the sensors which is determined by the sensors sending keepalive messages in predefined intervals to the MS that generates an alarm when a sensor times out. Additionally, the MS subscribes to the sensors which leads the sensors notify the MS of their status periodically.

6.2 Identification in Distributed Systems

Leach et al. [2015] specify a Uniform Resource Namespace for UUIDs (Universal Unique Identifiers), often referred to as GUIDs (Globally Unique Identifiers). UUIDs are 128 bit numbers that can be used for the identification of entities in distributed systems. Using UUIDs can be regarded as an alternative for the distributed counter for generating internal sensor identifiers presented in this work. There are different types of UUIDs: We can create time-based, name-based, and randomized UUIDs which can be used for different scenarios depending on the requirements. Time-based UUIDs are the only type that require a global lock and are therefore more complex to generate than the other types. The other two types of UUIDs have the advantage they can be generated without any type of higher order assignment mechanism, while still having a very high probability to be unique. For generating name-based UUIDs, a hash function is usually applied to a sequence of characters as part of the assignment algorithm. In order to generate randomized UUIDs, either truly random numbers or pseudorandom numbers can be used as part of the generation algorithm.

[2]https://coap.technology/

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we presented an approach for the management of sensors in a distributed monitoring system such as the Titan Control Center. For this, we first explained our motivation and the goals for this thesis. After that, we introduced the foundations and technologies that were important for the main part of the work. In our approach, we first analyzed the existing version of the Titan Control Center and together with the goals defined before, we derived requirements for our architecture we presented afterwards. Subsequently, we realized our architecture in form of a Java-based prototype implementation of the sensor management and evaluated the feasibility and the performance of our implementation.

We introduced an internal format for monitoring records which is decoupled from the external representation of sensors and we enabled other microservices of the system to transform between original and internal monitoring records by using the API of the Sensor Management microservice. Additionally, we enhanced the existing organization of sensors within the system in order to allow a more flexible modeling of the target infrastructure. For this, we reintroduced the existing hierarchial sensor organization by presenting a design which allows to create arbitrary many sensor hierarchies. This increases the analysis capabilities for gaining information in the context of Industrial DevOps.

We preserved the scalability and the fault-tolerance of the Titan Control Center by designing the sensor management as a stateless microservice. As a consequence, the system retains its horizontal scalability which is characterizing for the microservices architecture pattern. In addition, having the possibility to run multiple instances of a service provides fault-tolerance which also is reinforced by using etcd for storing the data for the Sensor Management microservice as data can be replicated over multiple database nodes.

We evaluated the feasibility and the performance of our implementation. With the feasibility evaluation we demonstrated that our implementation works correctly for common scenarios and that our approach fulfills the functional requirements. Within the bounds of our performance evaluation, we were not able to show that using our internal record format is more efficient than the original format in terms of the performance for reasonable sensor identifiers. Though, we have created the base for a more detailed analysis of the system in a realistic production environment.

We provide our implementation as a package [Ehrenstein 2019] that contains the Sensor

Management microservice and the remaining components of the Titan Control Center that have been adapted in this work in order to be able to integrate them with the Sensor Management microservice. Additionally, the package contains the sources and results from our evaluation.

7.2 Future Work

Our work imposes changes to most components of the Titan Control Center that we plan to work on in the future. We propose to integrate the part of the Sensor Management microservice into the architecture of the Titan Control Center which is responsible for providing the support for multiple sensor hierarchies simultaneously instead of only having one global hierarchy. As a consequence, it must be ensured that the History microservice aggregates monitoring data according to all existing hierarchies and that the Stats microservice computes statistics for all sensors of the hierarchies. To accomplish this, we can utilize the Kafka-based APIs of the Sensor Management microservice that allow the other microservices to subscribe to the changes of the sensor hierarchies. Moreover, the visualization frontend needs to be adapted so that the monitoring data from all hierarchies can be visualized properly.

Taking the results of our performance evaluation into consideration, we plan to reevaluate the performance of our approach in a more realistic production environment. This way, we aim to observe more meaningful results which may let us assess at certainty whether our approach improves the performance for reasonable original sensor identifiers, compared to the current version of the Titan Control Center. Depending on the results of this analysis, we may be able to decide whether using an internal format for the representation of monitoring records is justified and whether it should also be integrated into the existing architecture. Alternatively, we may decide whether integrating the decoupling of the internal representation from the environment in combination with an alternative internal format, for example, using name based UUIDs for representing sensors, is more appropriate.

Bibliography

[Barzu et al. 2017a] A. Barzu, M. Barbulescu, and M. Carabas. Horizontal scalability towards server performance improvement. In: *2017 16th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. Sept. 2017, pages 1–6. DOI: 10.1109/ROEDUNET.2017.8123729. (Cited on page 5)

[Barzu et al. 2017b] A. Barzu, M. Carabas, and N. Tapus. Scalability of a web server: how does vertical scalability improve the performance of a server? In: *2017 21st International Conference on Control Systems and Computer Science (CSCS)*. May 2017, pages 115–122. DOI: 10.1109/CSCS.2017.22. (Cited on page 5)

[Cai et al. 2014] X. Cai, Y. Wang, X. Zhang, and L. Luo. Design and implementation of a wifi sensor device management system. In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. Mar. 2014, pages 10–14. DOI: 10.1109/WF-IoT.2014.6803108. (Cited on page 48)

[Cloud Native Computing Foundation 2015] Cloud Native Computing Foundation. *A high performance, open-source universal RPC framework*. Accessed: 2019-09-20. 2015. URL: https://www.grpc.io. (Cited on page 9)

[Cloud Native Computing Foundation 2018] Cloud Native Computing Foundation. *A distributed, reliable key-value store for the most critical data of a distributed system*. Accessed: 2019-09-01. 2018. URL: https://etcd.io/. (Cited on page 8)

[Ehrenstein 2019] S. Ehrenstein. *Thesis Artifacts for: Distributed Sensor Management for an Industrial DevOps Monitoring Platform*. Sept. 2019. DOI: 10.5281/zenodo.3460683. URL: https://doi.org/10.5281/zenodo.3460683. (Cited on page 49)

[Fielding and Reschke 2014] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. Accessed: 2019-05-10. 2014. URL: https://tools.ietf.org/html/rfc7230. (Cited on page 6)

[Fielding 2000] R. T. Fielding. REST: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California* (2000). (Cited on page 6)

[Giang et al. 2018] N. K. Giang, R. Lea, M. Blackstock, and V. C. M. Leung. Fog at the edge: experiences building an edge computing platform. In: *2018 IEEE International Conference on Edge Computing (EDGE)*. July 2018, pages 9–16. DOI: 10.1109/EDGE.2018.00009. (Cited on page 47)

[Gilbert and Lynch 2002] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Sigact News* 33.2 (2002), pages 51–59. DOI: 10.1145/564585.564601. (Cited on page 7)

[Haerder and Reuter 1983] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys* 15.4 (Dec. 1983), pages 287–317. DOI: 10.1145/289.291. (Cited on page 6)

[Hasselbring et al. 2019] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, and M. Wojcieszak. Industrial devops. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Mar. 2019, pages 123–126. DOI: 10.1109/ICSA-C.2019.00029. (Cited on pages 1, 47)

[Hasselbring and Steinacker 2017] W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Apr. 2017, pages 243–246. DOI: 10.1109/ICSAW.2017.11. (Cited on page 6)

[Henning et al. 2019] S. Henning, W. Hasselbring, and A. Möbius. A scalable architecture for power consumption monitoring in industrial production environments. In: *2019 IEEE International Conference on Fog Computing (ICFC)*. June 2019, pages 124–133. DOI: 10.1109/ICFC.2019.00024. (Cited on pages 1, 8, 9, 12, 14, 16)

[Henning 2018a] S. Henning. Monitoring electrical power consumption with kieker. *Softwaretechnik-Trends* 38.3 (2018). (Cited on page 11)

[Henning 2018b] S. Henning. Prototype of a scalable monitoring infrastructure for Industrial DevOps. Master thesis. Kiel University, Department of Computer Science, Aug. 2018. (Cited on pages 17, 30, 32, 47)

[Horký et al. 2015] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. Dos and don'ts of conducting performance measurements in java. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: ACM, 2015, pages 337–340. DOI: 10.1145/2668930.2688820. (Cited on page 39)

[Isermann 2006] R. Isermann. *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media, 2006. DOI: 10.1007/3-540-30368-5. (Cited on page 6)

[Kreps et al. 2011] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: a distributed messaging system for log processing. In: *Proceedings of the NetDB*. 2011. (Cited on page 7)

[Lamport 1979] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* C-28.9 (Sept. 1979), pages 690–691. DOI: 10.1109/tc.1979.1675439. (Cited on page 7)

[Lasi et al. 2014] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & Information Systems Engineering* 6.4 (June 2014), pages 239–242. DOI: 10.1007/s12599-014-0334-4. (Cited on page 1)

[Leach et al. 2015] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. Accessed: 2019-09-21. 2015. URL: https://tools.ietf.org/html/rfc4122. (Cited on page 48)

[Newman 2015] S. Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015. (Cited on page 6)

[Ongaro and Ousterhout 2014] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pages 305–319. (Cited on page 9)

[Pham et al. 2016] C. Pham, Y. Lim, and Y. Tan. Management architecture for heterogeneous IoT devices in home network. In: *2016 IEEE 5th Global Conference on Consumer Electronics*. Oct. 2016, pages 1–5. DOI: 10.1109/GCCE.2016.7800448. (Cited on page 47)

[Tanenbaum 2009] A. S. Tanenbaum. Modern operating systems. In: Prentice Hall, 2009, page 459. (Cited on page 24)

[Tanenbaum and Van Steen 2007] A. S. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007. (Cited on page 5)

[Titan Project 2018] Titan Project. *The Industrial DevOps platform for agile process integration and automatision*. Accessed: 2019-09-09. 2018. URL: https://industrial-devops.org. (Cited on page 1)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pages 247–248. (Cited on page 11)

[Wulf et al. 2017] C. Wulf, W. Hasselbring, and J. Ohlemacher. Parallel and generic pipe-and-filter architectures with TeeTime. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Apr. 2017, pages 290–293. DOI: 10.1109/icsaw.2017.20. (Cited on page 7)

[You 2018] E. You. *Vue.js*. Accessed: 2019-08-15. 2018. URL: http://vuejs.org. (Cited on page 10)