Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A framework for dynamic configuration of IoT nodes based on events

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Manuel Filz, B.Sc.
Matrikelnummer 1228549

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Dr. Michael Vögler

Wien, 20. Februar 2017

Manuel Filz	Schahram Dustdar

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A framework for dynamic configuration of IoT nodes based on events

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Manuel Filz, B.Sc.
Registration Number 1228549

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Dr. Michael Vögler

Vienna, 20th February, 2017

Manuel Filz | Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Manuel Filz, B.Sc.
Meidlinger Hauptstraße 11, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. Februar 2017

Manuel Filz

Danksagung

Ein großer Dank gilt meinen Betreuern Schahram Dustdar und Michael Vögler für die umfassende Unterstützung während der Entstehung meiner Diplomarbeit. Besonders möchte ich mich bei Michael für seine großartige Betreuung, sein wertvolles Feedback sowie seine fortlaufenden Bemühungen auch während der Zeit, in der er nicht mehr an der Technischen Universität Wien tätig war, bedanken. Das flexible Umfeld und der drucklose Umgang ermöglichte es mir, das Entstehen der Diplomarbeit mühelos mit meinen beruflichen Tätigkeiten in Einklang zu bringen.
Ebenfalls möchte ich mich bei Franziska Kellner für die Unterstützung beim Korrekturlesen bedanken.
Ein großer Dank gebührt außerdem meiner gesamten Familie und meinen Freunden, die mich stets in meinen Vorhaben unterstützt und motiviert haben. Insbesondere möchte ich meine Schwester und meine Eltern hervorheben, die immer für mich da waren.
Zu guter Letzt möchte ich mich bei meiner Freundin Christine Kellner für das liebevolle Verständnis und die entgegengebrachte Geduld bedanken.

Kurzfassung

Das Internet der Dinge (IoT) ist ein Konzept, das derzeit zunehmend an Bedeutung gewinnt, da es sowohl von technischer, sozialer, als auch ökonomischer Relevanz ist. Im Wesentlichen bezeichnet es die Vernetzung einer Vielzahl an Geräten, die mit Hilfe ihrer eingebauten Sensoren in der Lage sind, ihr Umfeld zu erfassen und diese Daten über das Internet auszutauschen. Die enormen Datenmengen, die dadurch erzeugt werden, bieten aus technischer Sicht sowohl Chancen als auch Herausforderungen. Während es innovativen Applikationen die Möglichkeit eröffnet, Zusammenhänge aus unserem täglichen Leben für neue Anwendungen zu nützen, bedarf es für die richtige Verarbeitung passende Werkzeuge. Complex Event Processing (CEP) ist ein solches Werkzeug, das im Stande ist, relevante Strukturen durch die Analyse von Datenströme in Echtzeit zu erkennen. Der Einsatz im IoT Umfeld bietet Applikationen ein adäquates Mittel, ihre Konfigurationen an wechselnde Umgebungsbedingungen wie Standort, Wetter, und andere Faktoren anzupassen.

Im herkömmlichen Sinne wird CEP in einer zentralen Architektur eingesetzt, bei der eine einzige Verarbeitungseinheit aus unterschiedlichen Datenquellen gespeist wird. Um jedoch den hohen Anforderungen an Performanz und Skalierbarkeit im IoT Umfeld gerecht zu werden, benötigt es hier den Einsatz von verteilten Verarbeitungsmodellen, die Komponenten je nach Bedarf hinzufügen können. Unglücklicherweise bedeutet die Implementierung von verteilten Systemen mehr Aufwand und höhere Komplexität.

Das Ziel dieser Arbeit besteht daher darin, ein Framework umzusetzen, das die Entwicklung von CEP-basierten IoT Applikationen unterstützt. Durch ein modulares Design der Zielanwendung sowie klar definierten Kommunikationsstrukturen werden die genannten Anforderungen erfüllt. Zusätzlich übernimmt das Framework Basisaufgaben wie die Koordination von Komponenten, die Verteilung von Abfrageregeln, und die Bereitstellung von Fehlertoleranz. Ein weiterer Bestandteil der Arbeit ist eine eigene Abfragesprache. Die Sprache ist für den Gebrauch im IoT Umfeld optimiert und ermöglicht es den Benutzern, Abfragen und Regeln möglichst prägnant und intuitiv zu formulieren.

Ebenfalls Teil dieser Arbeit ist eine prototypische Implementierung, die als Grundlage für Evaluierungen dient. Neben funktionalen Anwendungsfällen im Bereich von Smart Buildings werden ebenfalls quantitative Messungen durchgeführt, um die Nützlichkeit und die Anwendbarkeit des Frameworks zu demonstrieren.

Abstract

The Internet of Things (IoT) represents an evolving field of technical, social, and economic significance. At its core, it connects a plethora of devices able to collect and sense their environment by using embedded sensors and exchange gathered information via the Internet. The enormous amount of data generated by these devices produces both new possibilities and new challenges. Opening up an endless array of new opportunities for innovative applications, the high volume of data demands the development of new tools and technologies. One such technology is Complex Event Processing (CEP), designed to detect patterns by analyzing streams of data in real-time. Applied to the IoT, it allows applications to make decisions tailored to their specific environment. As a result, applications can change their configuration depending on different requirements such as location, weather, and many other factors.

However, CEP in the traditional sense of a central processing unit does hardly scale across thousands of nodes. This limitation can be overcome by applying a distributed processing model, allowing to add components as required. Yet while distribution allows such applications to meet their requirements, it comes at a cost: the effort necessary for implementation of distributed systems are very high due to their increased complexity.

Therefore, this thesis proposes a framework for supporting and facilitating the construction of CEP based IoT applications. By design, the framework leverages a modular design of the concrete application and employs well-defined communication structures to satisfy aforementioned requirements. It also cares about low-level tasks of distributed systems such as the coordination of different components, the distribution of rules, and measures for improving fault tolerance. Moreover, this thesis proposes a custom language for defining queries and rules. The proposed language is optimized for the IoT context and enables users to express patterns in an intuitive and succinct way.

Additionally, this thesis presents a prototype implementation of the framework. This prototype implementation is consequently used to carry out evaluations. Besides functional use cases in the field of smart buildings, the evaluation focuses on quantitative measurements. By discussing and analyzing multiple test cases, the thesis discusses the feasibility and applicability of the proposed framework.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

The world has entered a new era of computing, a digital transformation, which is commonly linked with the term Internet of Things (IoT). It describes how devices embedded with sensors and network connectivity, are collecting and exchanging data with the existing Internet infrastructure. This transformation had started with the advent of smartphones more than a decade ago, and someday will have us surrounded by smart, connected devices that anticipate our every need and desire. Possible areas of applications range from an automobile that uses its embedded sensors to alert the driver when tire pressure is critically low or techniques to increase food production by optimizing water and fertilizer consumption, based on recorded data like temperature, soil moisture, and content of nutrients.
Experts forecast that by the year 2020, IoT will include more than 26 billion units and will generate incremental revenue exceeding $300 billion, mostly in maintenance activities and services. Another impressive figure predicts that more than 5 billion people will be connected by 2020. To sum this up in one sentence, the development of IoT is looking into a promising future[1].
The IoT has also given rise to new technologies that alter established concepts and approaches. In the last decade, devices have been connected to back-end systems through various networks, but they have often operated in isolation from one another. As more complex use cases evolve in the future, new connection models with greater cooperative interaction between devices are expected to emerge.
In order to address this issue companies started creating initiatives to come up with standardizations. Such developments have the big advantage that whole communities can collaboratively work on new products, because ideas are better exchanged. MQTT [MQT16] is one example for a lightweight messaging protocol that has emerged

[1] http://www.gartner.com/newsroom/id/3165317

for the specific requirements of the IoT world. Similar to standardizations at the protocol layer, frameworks do the job of promoting and simplifying new development processes at the software layer. Using them, developers can devote more time on the requirements, instead of developing lower level details that have been solved already. Many applications of the same type have great similarities, and frameworks provide a common code basis and reduce the effort to write the code all over again.
A further challenge are the avalanches of data that are flowing from every device in the Internet of Things. While on the one hand this wealth of data provides new opportunities to derive insights that can help to better understand customers, it makes it also harder to interpret data properly. If use cases additionally require the data to be processed in real time, the challenges become even more difficult.
To address the aforementioned problems Complex Event Processing (CEP) can be used. CEP has become one of the most important fields in data processing. The core idea is that huge amounts of data are correlated and analyzed in real-time or near real-time, for example, to detect fraud patterns or monitor stock prices. Key considerations for these types of applications are latency, throughput, and the capability of detecting complex patterns in a high volume of events. CEP works a bit like a database turned upside-down. Instead of storing data and executing queries against stored data, the CEP engine holds queries and runs the data against these queries [Inc16a].
Generally, applications in the IoT environment should guarantee good scalability, to continue to function well when the number of nodes is changed in size or volume. However, CEP in the traditional sense of a single central processing unit could be a bottleneck. To ensure good performance and scalability characteristics even if the number of nodes is high and the environment is considerable dynamic, a distributed processing model could be applied, which distributes computation tasks to multiple CEP nodes.
Bass et al. [BCK12a] recommend the tactic “Manage Sample Rate“ as another approach to satisfy performance and scalability attributes. The idea is that by reducing the sample rates of data streams, the overall volume of data could be deliberately minimized. In terms of nodes in the IoT, this means that they must be dynamically configurable, to minimize transmission data based on the current utilization of an application.
The aim of this work is to provide a framework for applications in the IoT world, especially enriched by CEP capabilities. Applications implemented on top of the framework can jump right into the use cases of their own project. All the groundwork and special challenges discussed before are shielded by the framework. To be more precise, the framework aims to build a distributed network of CEP nodes, taking care of equally distributing queries to ensure scalability in the context of IoT.

1.2 Aim of the Work

Applying CEP to the IoT domain enables new possibilities to build sophisticated applications, which help to better understand customers. Consequently, a framework that facilitates the implementation of these applications, would be a convenient tool

for developers. Hence, the expected result of this thesis is a framework, which will support the development of distributed CEP solutions, designed for horizontal scalability, reliability, and easy management of IoT applications. Thereby, the framework will use the power of CEP to detect patterns among the vast amounts of data generated by IoT sensors. To ensure scalability, the framework will leverage a modular design of the implemented applications, by subdividing the scope of functions into independent components such as sensor nodes, CEP nodes, and configuration nodes. By adding and removing these components dynamically, applications build on top of the framework will have the possibility to adapt the number of components to their current needs.
Furthermore, users will be able to express use cases via a sophisticated and user-friendly language that is largely simplified and optimized for the proposed framework. The language will consist of two parts: queries and rules. While queries will describe patterns, which should be detected by the system, rules will define how to respond. Consequently, rules will demonstrate high potential towards self-reconfigurable IoT.
Queries registered at the framework, will be spread automatically among all CEP nodes. The distribution itself will be based on criteria such as CPU utilization and number of running queries. If the framework detects that a node is not working correctly, an intelligent redistribution will be carried out. All affected queries will be redistributed to other, healthy nodes, ensuring fault-tolerance.
Another design goal is that the framework should be extendable and not limited to any specific CEP technology. Expandability will be achieved by a clean and structured communication model between all components.
To show that the proposed mechanism of the framework works in practice, a proof-of-concept implementation will be developed and tested. For evaluation purposes, a use case scenario from the IoT domain will be defined and used.

1.3 Methodological Approach

The methodological approach consists of the following steps:

(i) Literature review.
In the first step, relevant literature is gathered and reviewed. The results serve as theoretical basis of the thesis.

(ii) Design of the framework.
In the second step, the design and architecture of the framework is created. The main goals of the framework are defined in Section 1.2.

(iii) Design of the query/rule language.
Next, the syntax of the query/rule language is designed. The goal of this language is to provide an interface that is on the one hand largely simplified and optimized for the needs of the framework, and on the other hand independent of any technology.

(iv) Proof-of-concept implementation
To prove the feasibility of the proposed architecture a prototype implementation will be developed. Technologies which might be used for implementation are part of the Java ecosystem.

(v) Evaluation
Finally, the implemented framework will be evaluated by using IoT use cases. It must be shown that the framework can support CEP applications in the IoT environment.

1.4 Structure

The remainder of this thesis is structured as follows. Chapter 2 discusses the two concepts of IoT and CEP. Moreover, it outlines relevant challenges, issues, and common patterns, followed by Chapter 3, which provides an overview on related work. For example, it references and explains various IoT applications that utilize CEP in order to build advanced IoT applications.
Chapter 4 then builds upon these theoretical considerations and elaborates on a use case definition that reflects the framework's function range and serves as evaluation foundation. From the broad area of possible IoT applications, a scenario in the context of smart buildings is chosen.
In Chapter 5 the framework is designed. Design decisions are discussed in great detail, especially ones that influence features of the framework. The definition of the custom language is done in an iterative approach in realizing the total scope of the language step by step. Expressiveness is added to the language with every additional requirement.
In Chapter 6 the prototype based on Java and Spring is implemented. In particular, it focuses on an effective implementation using design patterns proved over the last years. Every design pattern in use is shortly introduced to promote its benefits.
The prototype is then used in Chapter 7 to evaluate its feasibility. The evaluation is separated into two major parts. The first one concentrates on use cases defined in Chapter 4, and addresses the question, whether the requirements can be realized with the help of the framework. The second part measures properties that describe the distributed nature of the application. By conducting a quantitative evaluation, the performance of the framework is demonstrated.
Finally, Chapter 8 concludes this thesis, summarizes the key aspects of the design as well as results found during the evaluation of the prototype. At the end details on remaining open questions and possible extensions are discussed, which are issues for future work.

CHAPTER 2

Background

In this chapter, background information necessary for reading this thesis are discussed. Due to the fact that the thesis is placed around the two concepts IoT and CEP, these concepts are investigated in greater detail.

2.1 Internet of Things

The term Internet of Things was first used in the late 1990s by entrepreneur Kevin Ashton who was one the founders of the Auto-ID department at the Massachusetts Institute of Technology (MIT) [Ash09]. The department coined the term to illustrate the power of Radio Frequency IDentification (RFID) tags used in the context of supply chain management. In their research work, the group connected goods to the Internet in order to track them without the need for human intervention. Nowadays, the IoT has become a popular keyword for covering various aspects related to the extension of the Web and the Internet into the physical domain.
While the term IoT is relatively new, the vision of combining physical devices with networks or computers is not new at all. By the late 1970s, electrical grids were remotely monitored via telephone lines and with the advance of wireless technology in the 1990s, machine-to-machine (M2M) solutions for monitoring were used at large scale. Many of these M2M products, however, were built on closed networks and proprietary standards, rather than on widely adopted Internet standards as we know it today.
The first Internet compatible device, a toaster that could be turned on and off over the Internet, was presented at the Interop conference in 1990. The toaster was connected to the Internet with the TCP/IP protocol, and controlled with a Simple Networking Management Protocol Management Information Base (SNMP MIB) [Ste16]. Over the next several years, universities and companies launched multiple IP enabled products, including a refrigerator invented by LG Electronics [Wik16] and a soft drink machine featured by Carnegie Mellon University [Sci16]. While these products did not become

a commercial success, a broad field of research started and created the foundation for today's Internet of Things.
Atzori et. al. [AIM10] state that the name "Internet of Things" is ambiguous itself. It is syntactically composed of two terms: While the first one emphasizes an Internet-oriented perspective, the second one prefers a Things-oriented view. Additionally, the authors identify a third perspective, the Semantic-oriented one. Figure 2.1 shows the claim of the paper that the IoT paradigm is the result of the convergence of these visions.

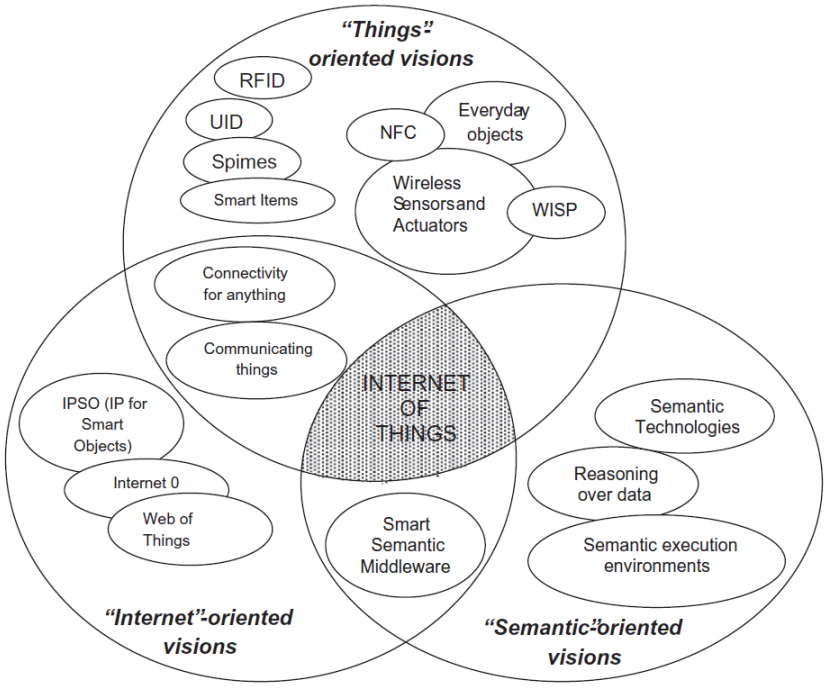


Figure 2.1: Convergence of different visions [AIM10]

The "Things"-oriented view focuses on the physical items and their identification. Simple items for that purpose are RFID tags, but they comprise just a small set in the overall vision. Further technologies that will connect the real and digital world are Near-Field Communication (NFC) and Wireless Sensor & Actuator Networks (WSAN) combined with so called smart devices. These are sensors not only equipped with usual wireless communication and memory, but also with new capabilities. Proactive behavior, context awareness, and collaborative communications are just a few required capacities.
The "Internet"-oriented vision addresses how the things can be integrated into today's and future Internet. It promotes the Internet Protocol (IP) as the "middleware" and the central network technology for connecting smart objects around the world. IP over Small Objects (IPSO), Constrained Application Protocol (CoAP), and IPv6 over Low Power Wireless Personal Areas Network (6LoWPAN) are technologies relevant to this vision.
Finally, the "Semantic"-oriented vision focuses on technologies for accessing and reasoning about the data generated by the IoT. It is supposed that the number of items involved

in the future Internet is extremely high. Therefore, subjects related to how to present, store, search, and organize information will become a key success factor. In this context, the authors expect semantics technologies are playing an important role.
An alternative way of outlining the IoT paradigm, but identifying the same requirements is proposed by [MSPC12]. Four pillars serve as basis for their arguments. While the first three perspectives -- conceptual, system-level, and service-level -- leverage the same challenges as identified in [AIM10], the last perspective involves the user itself as an important success factor. From a user perspective, the IoT will enable a large amount of new services, which shall support them in everyday activities. When a user has specific needs, he will make a request and an *ad hoc* application will satisfy them. The application itself will be automatically composed and deployed at run-time and tailored to the specific context of the user.
Enabling properties for IoT on system level are identified as following: [AIM10] [MSPC12]

- Device heterogeneity. IoT is characterized by a vast number of different devices, each equipped with different capabilities from the computational and communication standpoints. A high level of heterogeneity is tremendously important on protocol and architectural level. IP has become the dominant global standard for networking, and a broad incorporation into a range of devices is required.
- Scalability. The capability to handle a growing number of devices, affects IoT on different levels, including: naming and addressing, data communication and networking, knowledge management, and service provisioning.
- Ubiquitous data exchange. Wireless communication technologies play a prominent role to make almost everything “connectable“. The ubiquitous use of the wireless medium for exchanging data will encourage its further development in terms of spectrum availability.
- Self-organization capabilities. Following the user perspective, devices in IoT organize themselves autonomously in order to satisfy requested tasks. This includes strategies to perform device and service discovery without the need of an additional input.
- Semantic interoperability. Exchanging and analyzing massive amounts of data enable new opportunities for extracting information and knowledge. As a prerequisite, data with adequate standards, models and semantic description of their content, and well-defined formats must be provided.
- Security and privacy-preserving mechanisms. The IoT is immensely vulnerable to attacks for several reasons. First, devices are located in the public realm and therefore it requires little effort to physically attack them. Second, secretly listening to the conversation is due the wireless communication possible. Finally, most of the IoT devices are characterized by low energy and computing resources and thus, they are limited in supporting complex security mechanisms and algorithms. As a consequence, security and privacy should be an important part of the design and architecture of IoT solutions.

It cannot be overlooked that the main theoretical premises behind the above listed enabling properties are based on the traditional research line of distributed systems, where a distributed system is defined as a system driven by separate components, which are executed on different, interconnected, nodes. The design of architectures and protocols for distributed systems is therefore a key issue for the IoT.
While a commonly accepted full layered architecture for the IoT is missing due the vast number of different fields of application, Atzori et al. [AIM10] propose a SOA-based architecture for the IoT middleware as depicted in Figure 2.2. Thereby, the middleware is understood as a layer, connecting different, often complex and already existing applications that were not originally designed to be connected. Its feature of hiding the details of different technologies supports tremendously the work of programmers. Instead of spending a significant amount of time with recurring tasks not directly pertinent to their focus, they are able to concentrate entirely on the specific requirements of their business when developing individual solutions for the IoT.

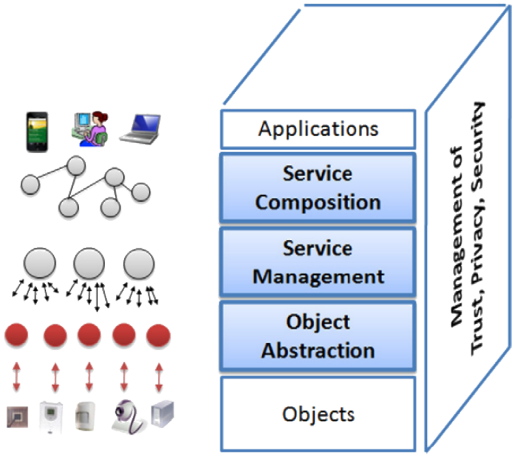


Figure 2.2: Architecture for the IoT middleware [AIM10]

A Service Oriented Architecture (SOA) is essentially a collection of services, decomposing complex and monolithic systems into well-defined components, which communicate with each other via standard protocols. Currently, many SOA deployments exploit Web-based protocols for enabling interoperability. In the context of IoT, SOA may be too heavyweight for being deployed on resources-constrained devices. Nonetheless, the idea itself in terms of abstracting functionality from the specific software implementation as well as for guaranteeing integration and compatibility of IoT technologies into the future Internet is very promising. Further key concepts of SOA, such as late binding and dynamic service composition, are expected to be inherited in IoT [MSPC12].
Trust, privacy, and security are considered as cross-cutting topics, which affect all layers:

- Objects/Devices. Devices occur in various types and at a minimum they are equipped with communication capabilities that either indirectly or directly con-

nected to the Internet. An example of direct connection is a RaspberryPi connected via Ethernet or Wi-Fi.

- Object abstraction. This layer enables the connectivity of the devices. Besides the well-known HTTP protocol, lighter-weight protocols on top of the IP/TCP stack play a prominent role. The two best known protocols in this area are MQTT and CoAP, which support resource constraints by the mean of small message sizes, message management, and lightweight message overhead.

- Service Management. This layer provides all functions necessary for the management of devices in the IoT scenario. A basic set encompasses: dynamic discovering, status monitoring, and service configuration.

- Service Composition. This is a common layer on top of a SOA-based middleware. Multiple services offered by devices are summed up to build one specific application. On this layer there is no notion about objects, the only rated assets are services.

- Applications. Applications are on the top of the architecture, supplying all the system's functionalities to the user.

2.2 Event Processing

A closer look at the literature reveals that there exist multiple different understandings about the concept of event processing. The term itself is used to describe several things, depending on the purpose and the structure of the actual use case. That is the reason why this thesis briefly outlines the evolution of event processing in order to create a common understanding.
The first concept of event processing started with discrete event simulation in the 1950s. The basic idea was to simulate the behavior of systems like factory production lines or control systems. Based on input data, the program generated events to mimic the interactions of the components of the system. A clock was used to simulate the real time by increasing its value by discrete "ticks". Such models were called discrete event simulations [Luc07]. The events had the forms of messages, which signals significant changes in the state of the application at a given point in time. This definition of an event is still valid today.
While in the following decades, similar systems were given different names, a more focused work started at the end of the 1990s. During this period the term Event Driven Architecture (EDA) was coined to define a design paradigm that is built around the concept of events. A steady development began, in which existing systems were adopted and new systems applied this paradigm. Subsequently, three fundamental types of event processing were born under the names simple, stream, and complex [Mic06].
In Stream Event Processing (SEP), notable events are considered as an important change in state. They initiate downstream actions and are commonly used to control real-time flow of business aiming to take out lag time. A well-known specification following the

principle of SEP is the Java Messaging Service (JMS) [SLW10].
The second type, Event Stream Processing (ESP), involves processing streams of notable and ordinary events. Typical ESP systems receive a large number of events, and use filters and other processing mechanism to decide, which ones are considered remarkable. Compared to SEP, individual events are less important, and instead the focus is on the variety of events. ESP is typically used to control the real-time flow of information, enabling real-time decision making around enterprise applications.
The third type of event processing is Complex Event Processing (CEP), which stresses the main focus on the correlation and composition of ordinary events into complex events. The event correlation may be casual or temporal. A common use case is to detect and respond to business anomalies, threats, and opportunities. Over the years, all three types converged to a common understanding and the distinction between them have been weakened, so that, today a lot of these systems are referred to as CEP systems [jbo16].

2.2.1 Complex Event Processing

CEP is a technique in which incoming low-level events are analyzed more or less as they arrive to predict higher-level, more valuable, summary information (complex events) that would go otherwise unnoticed, so the definition from the web portal [SL16]. The portal is supervised by Dr. David Luckham who originally conceived the paradigm at Stanford University. His book "The Power of Events" was an important milestone and describes CEP software as any computer program that allows users to perform calculations on complex events.
Figure 2.3 shows the schematic representation of a CEP system. In general, it consists of event observers, the CEP engine and event consumers. Event observers are the source of events and have only limited knowledge about the whole system. They only know that events occur and broadcast a signal to indicate that information. The CEP engine fulfills a range of tasks like filtering, transforming, and routing. Its functionality is normally controlled and defined by rules. The final receivers of events are event consumers, which provide a self-contained reaction consisting typically of numerous downstream activities.
As mentioned above, an event itself is an arbitrary activity in the system. A complex

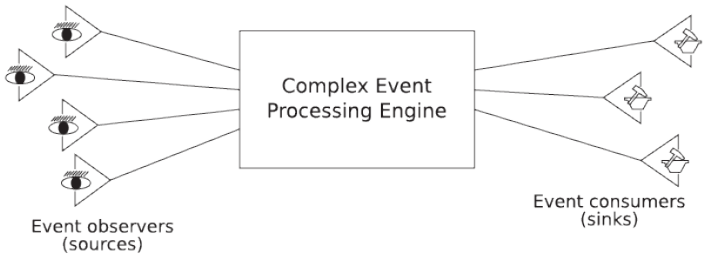


Figure 2.3: High-level view [CM12]

event further is an abstraction of one or more base events, related to each other by the

properties time, causality, and aggregation [Luc01].

- Time. The property time describes the order of events based on timestamps. An event can have one, two, or several timestamps according to the application. A typical timestamp is the start or stop time of the event.
- Causality. The property causality expresses the dependencies between specific events. The notation of A $\rightarrow$ B means that A must occur before B, or B depends on A. The Cause-Time Axiom stated by Luckham implies that if event A caused event B in system S, then no clock in S gives B an earlier timestamp than it gives A. In contrast, if A and B are causally independent, their arrival must not fit in any order.
- Aggregation. The property aggregation describes the *one-to-many* relationship. Rather to be the result of a single event, a complex event is the result of a series or set of events.

All three properties have some simple mathematical characteristics:

- Transitive. Whenever an event A is related to an event B, and B is in turn related to an event C, then A is also related to C. That means: A before B before C implies that A before C.
- Asymmetric. For all events applies, if A is related to B, then B is not related to A. That means: A caused B implies that B did not cause A.
- Partially ordered. Not the exact order of every event is specified. Only certain events depend on each other. The same property is prominent in the literature on distributed system [Lam78]. That means: If there are three events (A, B, C), then they are totally ordered, if and only if A $\rightarrow$ B $\rightarrow$ C happen in that order. However, if the requirement is that only A happens before C, then they are partially ordered. Under this circumstances multiple sequences satisfy the partial ordering: A $\rightarrow$ B $\rightarrow$ C; A $\rightarrow$ C $\rightarrow$ B; B $\rightarrow$ A $\rightarrow$ C.

Rules

Rules define how incoming events have to be processed within the CEP engine. There are different approaches existing to represent rules, which mainly depend on the adopted rule language. A widely applied approach is the declarative way via declarative languages. Instead of specifying the desired execution flow, the expected results are expressed. The syntax is usually derived from relational languages, in particular, Structured Query Language (SQL) and relational algebra, extended by additional *ad hoc* operators to better support timing requirements.

Delimitation

In contrast to traditional Database Management Systems (DSBMSs), CEP systems process information as a flow according to a set of preinstalled rules. As a consequence, CEP systems store only data that are relevant for the detection of the rules installed. Whereas, traditional DBMSs require data to be stored before it could be processed. Furthermore, queries are only executed when explicitly asked, quasi, asynchronously to its arrival. Both aspects are inconsistent with the requirements of CEP applications [CM12].

2.2.2 Design Patterns

There are many different design patterns that arise in CEP solutions [Cor16a]. The following presented design patterns include the basic functionalities of every non-trivial CEP.

Filtering

The most basic design pattern for event processing is the principle of event filtering as shown in Figure 2.4. In this case, a specified logical condition is evaluated based on the attributes of one or more events. If the evaluation returns true, the event is published to the destination stream. The filtering is not limited to a procedure, where an event after the other is evaluated and only the attributes of one event is involved. It is also possible to construct more complex filters that compare events to other events or compare events against computed metrics.

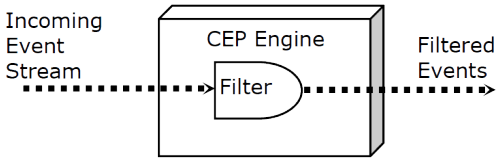


Figure 2.4: Filtering [Cor16a]

Aggregation over Windows

This design pattern combines several events to a single composite event. It is used in different flavors as depicted in Figure 2.5, differing along the following categories:

- Type of aggregation function.
  The typical set of aggregation functions encompasses sums, counts, minimum, maximum, standard deviation, and averages. Additionally, numerous CEP engines allow to implement custom aggregators.
- Type of window function: Time-based or count-based.
  Recent events are typically cached in so-called windows. A window is something

similar to an in-memory database, keeping and evicting certain events according to its strategy. In the case of a time-based window, events that arrived within a certain amount of time are kept, meanwhile a count-based window maintains a fixed number of events without any time restriction.

- Output frequency: Continuous or periodic.
  In the case of continuous output each incoming event updates the calculated expression, and an output event is created. In contrast, the periodic output publishes only periodically, for example, every minute, while the calculated expression is also updated continuously.

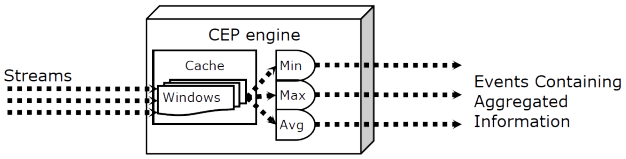


Figure 2.5: Aggregation [Cor16a]

Correlation (Joins)

More sophisticated CEP applications require that events are correlated across multiple streams. A join in a CEP application is similar with a join in SQL. As depicted in Figure 2.6, a join involves always one or more windows.
Joins arise in different flavors:

- Stream to window join
  Events arriving in a stream are joined with events stored in a window.
- Window to window join
  Events from multiple windows are retained and joined incrementally.
- Outer join
  Similar to SQL a distinction is made between left outer joins, right outer joins, and full outer joins. In a full outer join, each time an event arrives to one of the event streams, an output is produced.

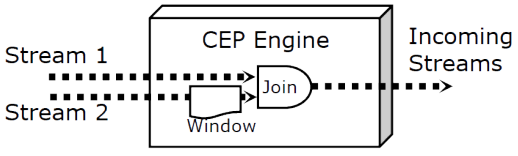


Figure 2.6: Correlation [Cor16a]

In general, joins are very CPU-intensive operations. Thus, CEP engines employ extensive indexing for optimizing.

Event Pattern Matching

While joins are pretty powerful regarding their functionality, the use of multiple joins can become relatively cumbersome. It is often desirable to have an intuitive syntax for expressing time-based relationships. For example, Figure 2.7 shows an example that should detect a pattern, in which four events in three streams occur in a specific sequence.

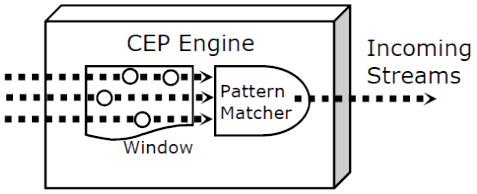


Figure 2.7: Event Pattern Matching [Cor16a]

The most common event patterns are the following:

- Event A followed by event B.
  Event B happens after event A.
- Event A and/or B.
  Both respective one event happens.
- Not event A.
  Event A does not happen.

2.2.3 Esper

All the above mentioned requirements have led to the development of specific systems designed to process complex events. One of the leading open-source CEP provider is Esper [Inc16a]. It uses a rich declarative language for rule specification, called Event Processing Language (EPL). The syntax is inspired by SQL to express querying, filtering, aggregation, and joins, from one or more streams of events. The main difference is that EPL replaces tables as the source of data by event streams and a concept called views, which are made of events instead of rows. In its use of the “select“, “from“, and “where“ clause the similarities are most striking: “select“ clauses specify event properties to retrieve, “from“ clauses define the view to use, and “where“ clauses express constraints:

```
1 select symbol, avg(price) as averagePrice
2    from StockTickEvent.win:length(100)
3    group by symbol
```

Listing 2.1: A sample EPL that returns the average price per symbol for the last 100 stock ticks [Inc16b].

Furthermore, the concepts of filtering, aggregation through grouping, and correlation through joins can be effectively leveraged. As long as no special time requirements are requested, equivalence between EPL and SQL is given.
The improved expressiveness becomes visible if the concept of time is required. As mentioned above the views define the data available for querying and filtering, but that is not all. Another fundamental task of views is to represent windows over streams. It enables the limitation of events considered by a rule and is commonly defined either by a specific time interval or by a specific number of elements into the past.
EPL embeds two different syntaxes for the definition of patterns. The common way is via so-called EPL patterns, which are defined as nested constraints, including logical operations, set operations, and iterations (Listing 2.2).

```
1 every StockTickEvent(symbol= "IBM", price > 80) where timer:within(60 seconds)
```

Listing 2.2: A sample pattern that alerts on each IBM stock tick with a price greater then 80 and within the next 60 seconds [Inc16b].

The alternative way exploits the well-known syntax of regular expressions for pattern detection (Listing 2.3). The expressiveness of both syntaxes reveal no differences.

```
1  select * from TemperatureSensorEvent
2
3  match_recognize (
4     partition by device
5
6     measures A.id as a_id, B.id as b_id, A.temp as a_temp, B.temp as b_temp
7
8     pattern (A B)
9
10    define
11
12    B as Math.abs(B.temp - A.temp) >= 10
13 )
```

Listing 2.3: A sample pattern that looks for two TemperatureSensorEvent events from the same device directly following each other [Inc16b].

Multiple event streams can be merged by using the insert clause. As a result, views get staggered onto each other in order to build a chain of views. For reasons of efficiency,

the Esper engine makes sure that views are reused among EPL statements.
As it is stated in the documentation, the main purpose of Esper is the facilitation of the development process of applications that compute large volumes of incoming messages or events. Therefore, the Esper CEP processing engine is available as component for the Java and .Net (Nesper) programming language. The data model of the implementation supports several ways for representing events. The most convenient way is via any standard Java/.Net object as depicted in Listing 2.4. The only requirement is that all event properties are accessible through getter and setter methods.

```
1  package at.model;
2
3  public class Event implements Serializable {
4
5     private String name;
6     private String value;
7
8     public Even(String name, int value) {
9        this.name = name;
10       this.value = value;
11    }
12
13    public String getName() {
14       return name;
15    }
16
17    public void setName() {
18       this.name = name;
19    }
20
21    public String getValue() {
22       return value;
23    }
24
25    public void setValue() {
26       this.value = value;
27    }
28 }
```

Listing 2.4: POJO as Event. All properties are acessible through getter and setter methods.

Additionally, the documentation recommends to use immutable objects since events represent states that occurred in the past and should therefore not be changed. XML Nodes are another way to represent events with the advantage that event properties can be evaluated by XPath expressions.
Applications making use of the Esper CEP engine, register rules via a set of methods at the engine. Typically, the output is received through listeners that are bound to exactly

one rule and implements a special interface. That interface gets invoked in the case if the associated rule is triggered. Listing 2.5 shows the wrapper construction for implementing a custom listener.

```
1 public static class CEPListener implements UpdateListener {
2  public void update(EventBean[] newData, EventBean[] oldData) {
3          // INSERT CUSTOM CODE HERE
4     }
5 }
```

Listing 2.5: Implementation of *UpdateListener*. The method *update* is invoked in the case if the associated rule is triggered.

CHAPTER 3

State of Art & Related Work

Today, there are many full CEP systems available and most of these systems use some variant of a Nondeterministic Finite Automaton (NFA). NFA is considered as a special case of a finite state machine without the limitation that its transitions are uniquely determined by its input symbols and source states. In the context of CEP, rules are converted into NFA models, whereby vertices define different states and edges are predicates of the corresponding rule. A rule is said to be matched when the NFA reaches the final state. Cugola et al. [CM12] provide a detailed discussion and a comprehensive overview of a substantial number of event processing implementations. The authors put forward the claim that traditional DBMSs can hardly fulfill the requirements of timeliness, which is precisely why during the last fifteen years, different research communities developed a number of new tools to support applications with strict timing requirements. They are called collectively Information Flow Processing (IFP) systems and their survey performs an exhaustive investigation of these tools regarding several aspects like system architecture, data model, rule model, and rule language. In their work, they outline the various stages of developments so far, which include active databases, Data Stream Management System (DSMS), and CEP systems. Prominent examples for DSMSs are STREAM [ABB+03] and TelegraphCQ [CCD+03].
Apart from Esper, multiple event engines have been proposed over the last years. Siddhi [SGLN+11], Cayuga [BDG+07], and SASE [WDR06] are research prototype systems equipped with special features in order to achieve high performance and scalability. For example, Cayuga employs several index structures and memory management techniques enriched with garbage collectors to improve the performance and to guarantee a small memory footprint. SASE is specialized in the processing of large sliding windows and reducing intermediate result sizes. Amongst others features, this is achieved through an auxiliary data structure, the so called Active Instance Stack (AIS). Siddhi tries to bring in stream processing aspects like multi-threading into the architecture of CEP engines. Furthermore, Siddhi proposes a pipelining architecture to handle temporal conditions

and keeps the runtime state of windows within event streams. As a result, many queries can use the same window, which leads to an improvement of the overall performance.
In addition to these research engines, there are also several commercial CEP engines available in the market, such as Oracle CEP 10g [Ora16a] and TIBCO BusinessEvents [TIB16]. Oracle CEP 10g is a fully Java-based CEP tool, which provides many built-in applications for supporting event processing within enterprises. The rule language illustrates similarities to the EPL from Esper, whereas TIBCO BusinessEvents uses an UML-based modeling approach for describing rules.
All the above mentioned CEP engine implementations, provide a custom grammar for specifying rules, however, literature contains few examples of generic rule languages as basis for integrating multiple engines. The paper [MCT14] proposes an approach to generate rules automatically. The authors put forward the claim that the complexity of rules is a limiting factor for the diffusion of CEP. Their framework learns from hidden causalities between the received events and the situations to detect, and as a result recommend CEP rules to the user.
There is a rapidly growing amount of literature on CEP utilized in the IoT environment. One such example is found in [Lee14], where CEP is used for traffic monitoring. The paper aims to detect complex events such as congestion or accidents. For that purpose, the authors implemented a prototype application and verified the usefulness of the architecture on the basis of a real-world dataset. They collected data from over four thousand vehicles, which transmitted up to fifteen thousand events at peak times. The evaluation of this use case scenario showed that the centralized CEP engine performed without loss in performance. Bottlenecks appeared as the number of vehicles was increased to a number of over ten thousand. Another example is given in [YCL11] where the authors propose a RFID-enabled framework for managing hospital data from different sources. They apply CEP to detect medically significant events to improve patient safety while reducing operational costs. Within the paper, a prototype system was developed to verify the feasibility of their approach. The results show that the processing delay and detection accuracy is acceptable, if only a small number of rules are installed. However, the authors state that the number of event rules have significant impact on the processing time, especially if they involve complex temporal and casual reasoning. In the particularly study, a doubling of rules led to a hundredfold increased processing time.
The last two examples demonstrate that the requirements of IoT can hardly be satisfied by applications based on standard architectures running on commodity hardware. In the first case the rate of incoming events was eventually too high for one CEP node, and in the second case the number of rules was beyond the processing capacity of a single instance. As a result, much of the current debate in the scientific literature revolves around the need for distributed solutions. For example, Govindarajan et al. [GSJM14] build a hierarchy of CEP nodes by implementing two kinds of CEP systems: A lightweight version primarily for the Android smartphone, complemented by a full-featured CEP engine installed on network nodes. The processing capabilities of the lightweight version are limited regarding filtering, sequencing, windows functionality, and simple aggregation. In the use case scenario presented by the authors, Android smartphones are the only

event generators. The events are categorized based on their sensor types and consequently put into distinct publish-subscribe queues. The lightweight version acts as subscriber and gets asynchronously notified as soon as an event arrives. If the lightweight engine produces an output, the engine forwards the result to the next CEP node, which can be either a lightweight or a full-featured CEP node. The next node is defined by a graph, which is in turn the result of a submitted rule. This graph is created at the moment the user submits a rule. The advantage of this approach is that not all events are processed centralized, but rather the computation load is distributed over a network of CEP nodes with distinct capabilities. Chen at al. [CFS+14] follow a similar approach by employing a distributed CEP architecture. The paper separates the detecting procedure into two parts: preprocessing and reasoning. The aim is to extract and filter useful information locally, and to transmit only useful information. As an example the authors mention the detection of an abnormal condition like: "The air condition system still turned on during the off-work time." [CFS+14]. In this case, the transmission of the exact power consumption by the air conditioner is not relevant. In fact, it is sufficient to send a warning message that the device is turned on. Following the approach outlined by the paper, this extraction is performed by the so called client-side CEP node. In contrast, the server-side CEP node compares the received warning message to data received by other nodes to make sure that the warning message is not just a false positive. Saleh et al. [SS13] avoid the transmission of unnecessary low-level data between nodes by applying a distributed solution on rule level. The main idea of this paper is that a large NFA expression resulting from a CEP rule can be split into smaller ones according to some criterion. After splitting, each subexpression contains a part of the CEP statement, that can be executed independently on distributed nodes. In case of dependencies between subexpressions, the possibility exists that affected nodes exchange intermediate events. The approach is primarily intended for sensor networks and the authors state that much research is still needed for a system that fully applies this approach.
What is evident, however, is that CEP in a single architecture is well established among all kinds of IoT domains. CEP in a distributed system for the IoT, remains still on a research level. Although examples of such approaches are evident in the literature they are not fully implemented or accepted.
Gaunitz et al. [GRF15] propose an idea for processing and analyzing enormous amounts of data. They use Apache Storm[1] in combination with CEP to provide a scalable event-driven architecture. Apache Storm was originally developed, to analyze the click behavior of users in real time and became soon a top level project of Apache. Apache Storm is based on the Map-Reduce algorithm to distribute work between a cluster of nodes. The authors put forward the view that the combination of Apache Storm and CEP bypasses the disadvantage of Apache Storm to change the topology dynamically. Meanwhile, it enables the exploitation of horizontal scaling features. Finally, the authors state that this proposal needs further research, because each CEP node works in its own context and is not visible to other nodes. As a result, only rules with limited expressiveness could be used in this configuration.

[1] http://storm.apache.org/

Akbar et al. [ACMZ15] choose a different approach and demonstrate that machine learning methods can be used in conjunction with CEP in order to provide a proactive solution for IoT applications. Data from multiple sources is gathered and processed with an adaptive prediction algorithm called Adaptive Moving Window Regression (AMWR). The moving window refers to the way in which the prediction model is trained. Instead of training the prediction model once using large historical data, a moving window of data is utilized for training the model. Every time new data arrives, the algorithm calculates an error value and retrains the model accordingly. Thereby, the degradation of the prediction model should be simulated. Afterwards, predicted data is published on an event bus where the downstream CEP system can access the data. According to the authors, the proposed method is adaptive in nature and therefore can cope with dynamic environments. In recent years, several other research efforts were conducted with the objective of combining predictive analytics methods like machine learning with CEP. A recent example is given by the paper [DZJ02], which addresses applications for predicting traffic flows.

IoT in general is a broad concept for which no uniform architecture exists. Abdmeziem et al. [ATR16] present a summary of different IoT architectures proposed in the literature. These are either the result of academic research or business projects. Examples are the EU project SENSEI [PBEV09] and the Electronic Product Code (EPC) based IoT Architecture [HM11]. Additionally, Abdmeziem et al. summarize a commonly accepted high-level architecture that comprises three layers: Perception Layer, Network Layer, and Application Layer. As the name suggests, the focus of the Perception Layer lies on the perception of things around us. The second part of the layered architecture, the Network Layer, acts as mediator between the outer layers. In this context, it is in charge of processing the received data from the Perception Layer and transmitting data to the Application Layer through various network technologies. The Application Layer constitutes the front end of the IoT and exploits the possibilities offered by the IoT. Ilapakurti et al. [IV15] and Qin et al. [QSF+14] are two examples, which also survey the main techniques and specific requirements for applications in the environment of IoT with particular focus on CEP.

CHAPTER 4

Use Case Definition

Thousand of years ago, the first buildings ever constructed were primitive shelters made from stones, sticks, and other natural materials. While the discipline of house construction passed many worthy stages to reach the current level of modernization, the basic idea has remained the same - to provide a comfortable space for the people inside.
Today, there are several emerging trends, which will have significant impact on how buildings will be constructed in the future. One influencing factor is the ongoing urbanization. Studies show that today about 54% of the world's population lives in urban areas, a rate that is expected to grow by more than ten per cent till 2050[1]. Combined with the overall growth of the world's population, cities will in the future be confronted with a set of new challenges emerging from these developments. Amongst other things, satisfying the increased demand for energy in urban regions will be a main challenge for city planners. They will have to define strategies regarding the use of new technologies, establish optimized energy consumption mechanisms and realize energy saving initiatives to meet this ambitious challenge. One concrete strategy in terms of IoT will be to make buildings "intelligent". Those so called smart buildings have the ability to adjust and adapt facilities to the needs of its occupants. As a positive side effect, smart buildings can reduce unnecessary energy consumption by operating devices in the most efficient way. In the best case energy waste will vanish completely in the future.
When various tasks in a building are controlled based on real time data, the result is an intelligent building that is not only functionally efficient, but simultaneously also increases the comfort level. Occupants can benefit in several respects: Doors open and close automatically, waiting time for elevators is reduced or window blinds are controlled automatically. There are countless possibilities that are increasing the comfort level and will become standard in the future.
Another key factor for future-oriented buildings is the compliance with safety and security requirements. A building that monitors the arrival or departure of people, using bio-

[1] http://esa.un.org/unpd/wup/highlights/wup2014-highlights.pdf

metric identifications, making it possible to know who was in the building and at what time will become the norm. This in combination with smoke detectors and fire alarms placed strategically, alert occupants and inform the fire department personnel about the location of the fire, will increase safety standards.
Three key factor influencing how smart buildings will be constructed in the future are: Saving energy, increasing comfort, enlarging safety and security requirements. Before concrete use cases are worked out, which are satisfying these key factors by means of the proposed framework, this work briefly outlines a huge office building as a reference model. The assumption is that our reference model has several floors or type of rooms, which all serve a certain purpose that is common to most public buildings (hospitals, universities, etc.). According to their utilization, different properties are identified:

(i) Technical floor/room
is a space dedicated to mechanical and electronic equipment. Typically, server rooms are part of such floors, which require special treatment. For example, server rooms have to be air conditioned below 20 degree Celsius and need special regulations concerning fire protections.

(ii) Office floor/room
is a space equipped with offices and work spaces. These are typically used for conventional office activities like reading, writing, and computer work. Occupants should feel comfortable regarding temperature and lighting. In case of emergency, occupants should be informed immediately and the nearest safe emergency exit should be indicated.

(iii) Storage floor/room
is a space used for storing all kinds of items. Here less attention has to be paid to room temperature control while instead particularity attention must be given to the access control.

(iv) Entrance floor/room
is usually a large, specially designed hall or space. The very first access control is normally done in this area. The number of people entering or leaving the building through these areas are the basis of all further activities.

A picture of the reference building is depicted in Figure 4.1. In the following the key factors discussed are applied to our reference model in order to present in detail possible use cases.

4.1 Saving Energy

Building operators have different choices to reduce energy consumption of their facilities. Selecting walls, roofs and other assemblies based on long-term insulation, air-barrier performance and durability requirements, is only one opportunity. An alternative approach

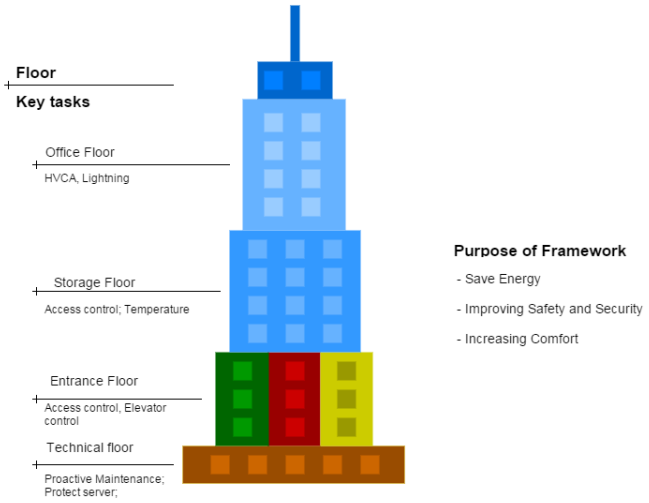


Figure 4.1: Reference model

is to avoid the waste of energy. Without appropriate measures, energy consumption units in buildings are often needlessly turned on. These so called "energy guzzlers" are often in a direct relation to the size of a building. Huge facilities such as the reference model offer multiple approaches for improvement. Some of them are exemplified and discussed in detail in the following:

(i) **Optimized heating, ventilation and air conditioning (HVAC)**
HVAC is responsible for a large proportion of the total energy consumption in buildings. Analyzing the building usage, allows to make reasonable decisions for the optimized usage of HVAC. Reasoning about room occupation is thereby a challenge. HVAC may for example only be activated for a specific room, if room usage exceeds thirty minutes. The task is to identify a causal connection between multiple single events.

(ii) **Proactive maintenance of equipment**
Proactive maintenance is a preventive strategy to maintain reliability of machines and equipment. Abnormal and inefficient operational modes should be detected as early as possible. For example, when an air conditioning equipment permanently blasts at full speed for a defined time period, maintenance personnel should be notified to inspect as early identification of minor issues may prevent serious damage.

(iii) **Optimized lightning**
Countless studies indicate that office staff does not care if light is turned off at the time they leave office[2]. As a result lights continue to shine during the whole

[2] http://aceee.org/files/proceedings/1996/data/papers/SS96_Panel8_Paper18.pdf

night or throughout the weekend. The most simple solution for this problem are of course motion sensors. Yet such a solution fails to react to more complex situations or even a whole sequence of events.
The described scenario is not only applicable for lighting. All kind of devices, which are controlled depending on their environment can be treated equally. As an example window blinds can be mentioned. Depending if the sun is shining or not, what the current season is and which workspaces are occupied, they can be opened or closed automatically.

4.2 Increasing comfort

A common way of achieving comfort is to use labor-saving devices. Their purpose is to make a task easier to perform than a traditional method. This effect can be intensified by feeding devices with real-time data. The aim is to anticipate the needs of occupants and provide a pleasant environment for everyone in the facility. Examples are given in two separate scenario:

(i) **Elevator**
This scenario outlines a predictive elevator system for the reference office building. As a precondition all employees are equipped with RFID tags on which their primary floor is stored. Once an employee enters the building, a reader activates the RFID tag. It gets supplied with power and transmit its data. This interaction happens without the individual concerned being aware of it. While the employee is approaching further to the elevator, the system prepares itself to the expected ride. If multiple passengers are arriving, it can sort them into groups of similar destinations and assigns specific elevators to each passenger.
The ability to change settings for specific passengers is a further feature. For example, passengers in wheelchairs can benefit from elevators that keep their doors open longer.

(ii) **Flow meters**
In this scenario flow meters gather data in real-time and provide not just water data, but information on outages, malfunction and quality criteria. The benefits of monitoring and managing water systems in buildings are for example: Water leaks can be quickly detected, maintenance costs are reduced and water is saved. From the comfort perspective, a rapid identification of leaks minimizes damages, while constant water quality monitoring is essential for specialized industries such as biotechnology companies.

4.3 Improving safety and security

Providing safety and security to occupants is a key requirement of every building. Failure to provide sufficient safety and security is usually not accepted.

(i) **Fire protection**
The key requirement for a fire protection system is obvious: Fire alerts have to be processed as fast as possible. A special characteristic is the real-time processing of the measured data. There is no need to store data permanently.
The evaluation itself could combine multiple sensor data in order to minimize measurement errors. The use of temperature and smoke sensors has often been proven to be useful for detecting fire in buildings [LST02]. The most important task after detecting is to provide information and warning without delay. Alerting and updating operational dashboards helps to evacuate people.

(ii) **Video surveillance systems**
A customary video surveillance system can be increased in value by integrating it into the framework of the reference building. Video recordings can be triggered based on detected motion, which on one hand simplifies the storage process and on the other makes it easier to browse through recordings. Additionally, video cameras can be connected with one another and automatically hand off recording to adjacent cameras. In case of emergency, the system can provide real-time feed to local law enforcement.

(iii) **Advanced intrusion detection systems**
There is a far-reaching consensus in the literature that security will shape the future developments of IoT in general, and in particular for smart buildings [XWP14]. The transition from closed networks to the public Internet at an alarming pace, raises all kind of security issues. If attackers for example could gain control of devices that regulate technical floors, they could turn off fans and cause overheating of servers. Much of the current debate revolves around the question of how do we protect billions of devices from intrusions and interference that could compromise personal privacy or threaten public safety.

With the development of smart building, services from the physical space are blurred with technologies from the information technology. Vast numbers of situation-aware sensors and devices are embedded in the reference building, which produce huge amounts of data.
Using the example of a standard office building, different use cases in order to achieve each key factor are presented. A closer look at the use cases shows that temporal constraints are required in nearly every scenario.

CHAPTER 5

Design

The term “Design“ refers to the description of high level structures and main conceptual elements of a software system and how they relate to each other. Thereby the structures and elements can be described in several different views to capture specific properties of interest.
For describing the design of the proposed framework a schematic view, a component & connector (C&C) view, and various sequence diagrams are used. The schematic view introduces the different components of the framework, including a description of the core tasks. The C&C places emphasis on the pathways of interaction, such as communication links and information flows of the system. Choosing the appropriate forms of interaction between computational elements is considered as an essential part of the design. These interactions represent complex forms of communication that must take place in a defined sequence, and therefore require nontrivial implementation mechanisms. For describing communication scenarios, sequence diagrams are used.

5.1 Conceptual Overview

The application is composed of three main components: *Device Node (DN)*, *Event Processing Node (EPN)*, and *Configuration Management Unit (CMU)* (depicted in Figure 5.1). Modularity is the key feature that has been kept in mind while developing this structure. Each component represents an autonomous unit, which can be deployed independently.

- The *Device Node (DN)* acts as data supplier and monitors the environment around it. DNs can be of various types with the minimum requirement that the interface for communication is implemented. The exact specification of the API is discussed in Chapter 6. In general, these devices are distributed at will and can connect dynamically to the system over the Internet. After a successful handshake with the

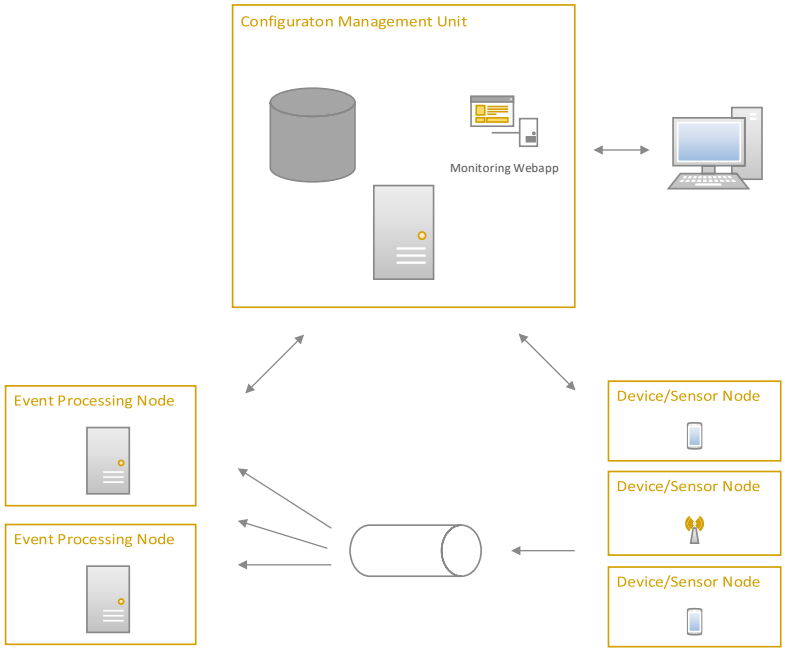


Figure 5.1: Schematic view

CMU, they are part of the system and fulfill specific observation tasks according their possibilities.

- The *Event Processing Node (EPN)* incorporates the CEP in a way that it allows a loosely coupled implementation of the concrete CEP engine in use. It follows the adapter approach to integrate existing implementations without modifying their source code. Similar to DNs, EPNs can be spread over the Internet randomly and join the system as required. The underlying assumption is that the system contains a vast number of DNs, which deliver data in an even larger amount. In order to guarantee software qualities like performance and scalability, multiple EPNs are deployed. The concept of horizontal scalability is applied here [VBD01]. More resources are added to the system in the form of logical units (EPNs) to handle a growing amount of work. In contrast, vertical scalability refers to adding more resources to a physical unit, for example by adding more memory to a single EPN.

- The *Configuration Management Unit (CMU)* acts as central registry and agent between the two other components. It holds crucial information about registered components and is responsible for distributing *rules*, *supply requests*, and *notification changes*. As previously mentioned, rules define actions for given conditions. Supply

requests denote special messages sent between CMU and DNs. Their purpose is to instruct EPNs either to start or to stop supplying data. During the registration process, DNs inform the CMU about the kind of data each of them is able to supply. For example, a temperature sensor located in the first floor of a building would disclose that it is able to supply the measured value “temperature“ in the domain “first floor“. Another type of messages exchanged between CMU and DNs are *notification changes*. As soon as a rule is triggered, the corresponding action can lead to a change in the configuration. For example, if the temperature exceeds a certain limit, the temperature sensor could be instructed to increase the resolution and to send measured values in shorter intervals.
The *Monitoring Webapp* serves as management tool for end users. On the one hand it offers an overview of all involved components, while at the same time it enables the user to install and uninstall rules. At the moment the user commits a new rule to the application, a basic rule validator examines the syntax of the rule. If the rule does not follow the specified syntax, the *Monitoring Webapp* informs the user by displaying an error message. This validation ensures that committed rules can be converted to workable CEP statements.

5.2 Pathways of interactions

The C&C view depicted in Figure 5.2 provides a picture of potential interactions of the framework. Additionally, a summary of all used connectors is given in Table 5.1. The rule for naming the interfaces is that the prefix is derived from the providing component. Two kinds of communication types are identified: the client-server pattern and the publish-subscribe pattern. The client-server pattern is build around the components *client* and *server*, and uses the request/reply connector type. Clients initiate interactions with servers by requesting services and wait for the results of those requests. The client must know the identity of the server, in order to be able to invoke services. In contrast, servers do not know the identity of clients in advance, they simple respond to the initiated client requests.

Based upon the client-server pattern, a predominant part of the communication flow of the framework complies to the Representational State Transfer (REST) architectural style. Additionally to the client-server constraint, it is build around non-blocking HTTP requests. These requests rely on the four basic HTTP verbs (POST, GET, PUT, DELETE) to tell the service to create, retrieve, update, or delete a resource. Thereby, any information is represented as a resource, which is accessible through a single addressing scheme based on Uniform Resource Identifier (URI). Another formal constraint of REST is statelessness. The communication between client and server must be stateless between requests. This means that each request from a client must contain all necessary information for the server that is needed for understanding the content of the request [BCK12b].
In comparison with the other off-the-shelf technology for web-based applications today, WS* and Simple Object Access Protocol (SOAP), REST has somewhat fewer characters than a message exchanged in SOAP. Therefore, REST has to be favoured in systems

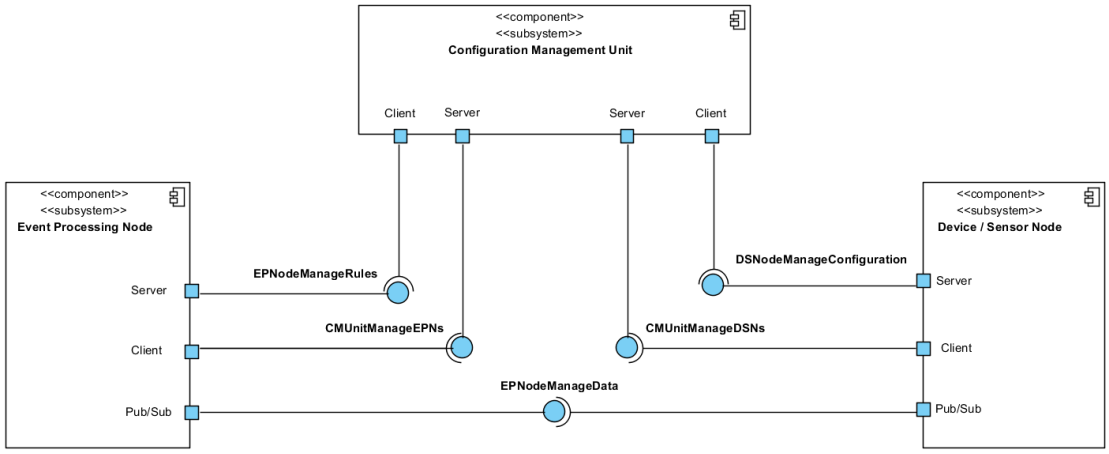


Figure 5.2: Pathways of interactions

exchanging a large number of messages. Another reason for choosing this style is interoperability, which was identified as an enabling factor in the IoT. REST has the advantage that only a HTTP stack is needed for message exchange. An alternative way

Component (Provided Port - Source)	**Component** (Required Port - Target)	**Connector Type(s)**	**Interface**	**Purpose**
DN	CMU	Client-server, REST	DSNodeManageConfiguration	Source receives *supply requests*, and *notification changes*.
EPN	DN	Publish-subscribe	EPNodeManageData	Source receives *sensor data*.
EPN	CMU	Client-server, REST	EPNodeManageRules	Source receives *rules*.
CMU	DN	Client-server, REST	CMUnitManageDSNs	Source receives *registrations*.
CMU	EPN	Client-server, REST	CMUnitManageEPNs	Source receives *registrations*.

Table 5.1: List of connectors

to exchange messages is the publish-subscribe pattern, in which clients and servers are decoupled. Instead of directly exchanging messages, a client (called publisher) sends a message to a third component, called message broker. The broker filters all incoming messages and distributes them accordingly to one or more clients (called subscriber), who

are receiving the messages. As a result, the publisher and the subscriber do not know about the existence of one another. The central meeting point is the message broker, which is known by both the publisher and the subscriber.
The decoupling of components can be differentiated in the dimensions space, time, and synchronization [EFGK03]. While space decoupling describes the condition in which publishers and subscribers do not need to know each other physically (either by a network address or a domain name), time decoupling states that the interacting parties do not need to run at the same time. The third dimension, synchronization decoupling, stresses the fact that publishers are not blocked while producing events and subscribers get asynchronously notified about events.
The publish-subscribe pattern provides better scalability compared to the traditional client-server approach [FJL+01]. One of the reasons for this is that the message broker can be highly optimized and parallelized for processing events. Caching and intelligent routing are further mechanisms, which improve scalability. However, to deal with millions of devices, clustered broker nodes with load balancers are needed.

5.2.1 Communication Scenarios

In the following three main use cases are discussed and how they influence the communication flow.

Registration of nodes

As already mentioned, nodes are in line with the vitality of the IoT nature and therefore join and leave the system as currently needed. In order to become part of the system, nodes perform a three-way registration handshake as depicted in Figure 5.3. The handshake starts with a request by the client sending a message with its own URL to the CMU. In response, the server replies an ID to the client. If the URL is not already known by the server, a new ID is generated, otherwise the corresponding ID is replied. The last part of the handshake differs between the types of nodes. While EPNs simply acknowledge the reception of the ID, DNs use the acknowledgment message to disclose information about the kind of events they are able to provide. After the three-way handshake, both the client and receiver have obtained an acknowledgment of the connection.
So far, it has been assumed that the CMU is up and running and the handshake works successfully the first time. Should this not be the case, the client initiates the registration process after a configurable delay again. The time interval is consequently increased with every unsuccessfully attempt. This approach guarantees that a first failed attempt will be tried again in a reasonable short time, while a fundamental problem burdens the system minimally.
For terminating the connection no explicit handshake is used. As long as heartbeat messages are send, nodes are considered as part of the system. System properties are an integral part of the heartbeat messages. The current CPU and memory utilization are examples for system properties, making it possible for the CMU to reason about current capacities.

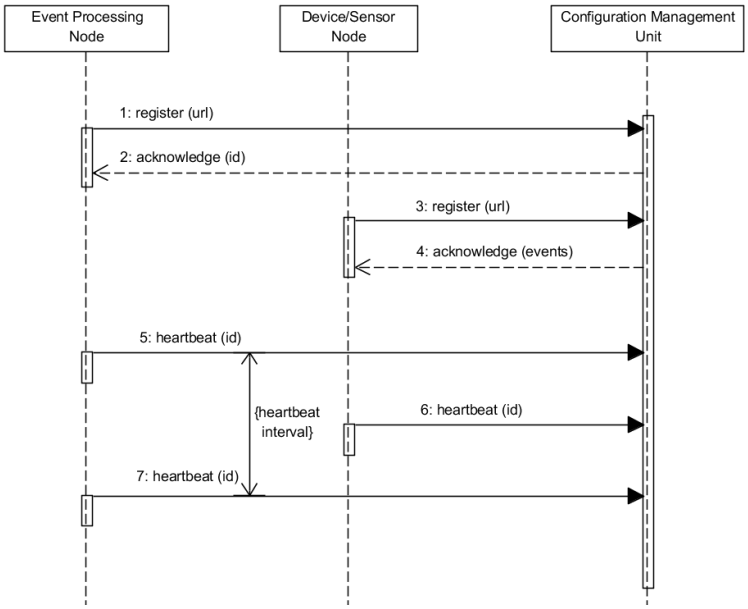


Figure 5.3: Registration of nodes

Rule submission

Figure 5.4 shows the communication flow within the framework triggered through a rule submission. After the user has entered a rule, the CMU determines which EPN should take over the processing. For the purpose of selection, *distribution policies* are used. They define the distribution behavior based on monitoring metrics. Policies can range from very simple and domain-independent (e.g., select the EPN with the lowest average CPU utilization) to application-specific solutions, such as policies that incorporate priorities of certain events. By default, the framework offers three domain-independent policies. All three follow the basic principle of selecting the lowest average value. The evaluated metrics are average CPU utilization, average memory utilization, and number of running rules.
Distribution policies are a well-suited tool for system engineers to influence how the application should handle a growing number of rules. The engineer steers the scalability of the application and may positively influence infrastructure costs and utilization via sophisticated *distribution policies*.
Whenever a new rule is submitted, the CMU executes the defined *distribution policy* and assigns the rule to an EPN. The EPN receives the rule and triggers multiple preparation steps sequentially. The first is to validate the syntactic correctness and to parse the declarative defined rule. The outcome of the parsing process is further transformed into one or more equivalent CEP engine statements. After preprocessing successfully, the statements will be installed at the CEP engine.
The free choice of CEP engines was identified as one of the main objectives of the proposed framework, which is precisely one reason why all these preprocessing tasks are

not executed at the CMU centrally. If that were the case, the CMU would be required, to care about all the individual preprocessing tasks for all the different CEP engines. This would clearly limit the interoperability of the framework. A pleasant side effect is that the load of the CMU is reduced.
Another outcome of the rule transformation are so called *event data sources*. They are used for identifying events by two dimensions (type, domain). The concept of *event data sources* is discussed in more detail in Section 5.3. For describing the communication flow it is sufficient to know that the identified *event data sources* are the basic ingredients for the message broker. Hence, the next step of the communication flow is the subscription of the EPN at the message broker by the mean of *event data sources*.
In order to meet the more challenging requirements in respect of constraint devices and to avoid unnecessary data transmission, DNs should emit events only if explicitly requested. For that purpose the *event data sources* are also send to the CMU, which in turn informs all potential data suppliers. It searches the database for registered DNs that comply to the *event data sources* and sends them so called *supply requests*.
After completing all these steps, the rule is successfully installed, which means the rule is registered at the CEP engine and the DNs supply the system with data.

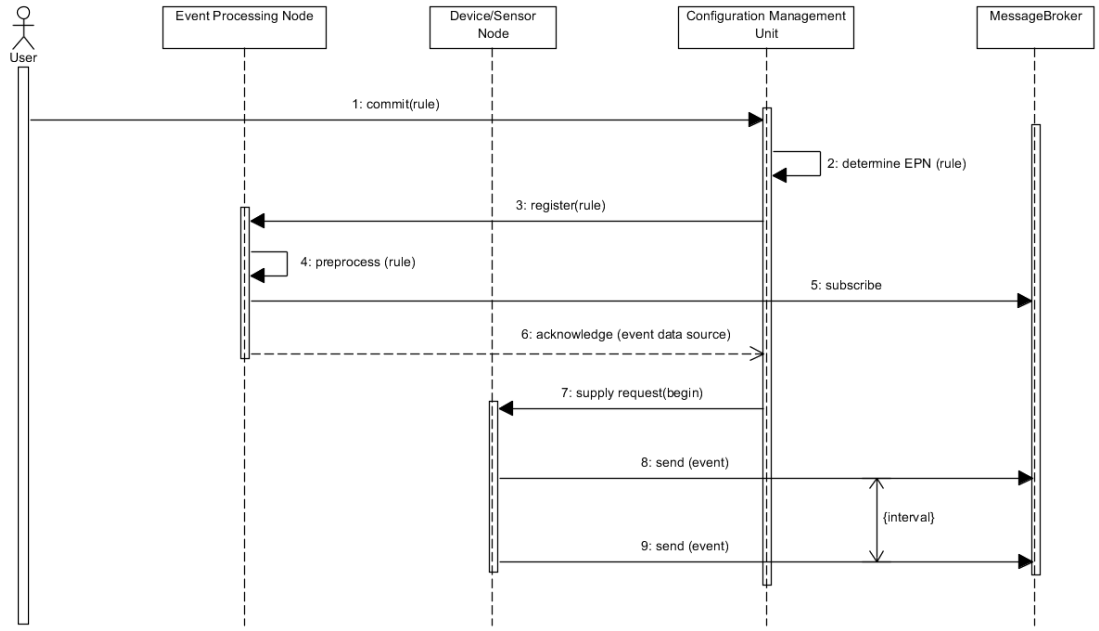


Figure 5.4: Rule submission

5.2.2 CEP triggers rule

Whenever a CEP statement is triggered, the EPN retrieves the associated rule from an in-memory repository and creates a so called *notification change*. This message is

transmitted to the CMU, in order to instruct affected nodes. *Notification changes* are intended to adopt the configuration of nodes. This principle is the foundation that enables developers to generate proactive applications. Figure 5.5 shows how the framework reacts to a triggered rule.

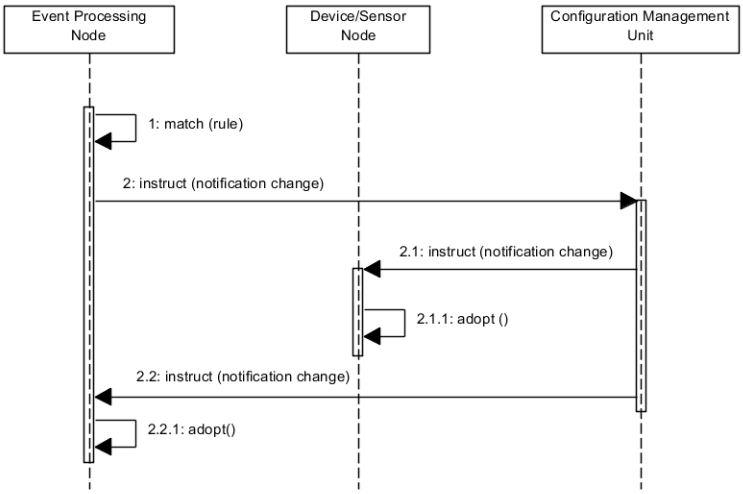


Figure 5.5: CEP triggers rule.

5.3 Data & Event Model

As outlined in Chapter 2 a primary goal of the IoT is to create situation awareness and enable applications, machines, and users to better understand their surrounding environments. The perceptiveness enables applications to take rational decisions and to respond to the dynamics of their environments. The basis for decisions is the correct interpretation of the observation and measurement data. That is precisely the point why the framework proposes a data & event model. Measured values like temperature, air pressure, light, and sound are just a few examples of different measured quantities the framework has to deal with. In addition to the diversity of measured values, mostly, their occurrences strongly vary in their quality, which makes the task of processing, integrating, and interpreting a non-trivial challenge.
To address this challenges, the framework proposes a data & event model that incorporates the basic characteristics of IoT data. The distinctive attributes are identified by Barnaghi et al. [BWDW13] and summarized in Table 5.2. As shown, the data model should for example take time and location information into account when reporting a measured value such as temperature thereby adding temporal and spatial information to simple values.
Another aspect that should be kept in mind during the process of defining a data & event model, is the dependency of the rule language. The rules must be able to evaluate

the same attributes, such as type (temperature, air pressure, etc.), location (location information), time (timestamps, freshness of data), and value (measurement value).

Attribute	**IoT Data**
Size	IoT data is often very small. Meta-data (e.g., measuring unit) can be significantly larger than the data itself.
Location dependency	IoT data is most of the time location dependent. Rather, location information improve the informative value of device and sensor data.
Time dependency	IoT data is time critical. Timestamps are needed to support requirements with time constraints.
Life span	IoT data is usually short lived or transient.
Number	The quantity of data is often very large.

Table 5.2: Characteristics of IoT data [BWDW13]

Based on these considerations, the data model is build upon three core data types: *EventType*, *Domain*, and *ModificationAdvice*.
By design these data types incorporate the main attributes for events, namely type, value, location, and time, which are identified as main properties for CEP in the context of IoT. Figure 5.6 depicts the hierarchy of the data model and how the core data types influence the model. The core data types are denoted as abstract classes, which points to the fact that they are not used directly, but serve as basis for further implementations. Developers and system architects can extend these types, and consequently customize their applications.

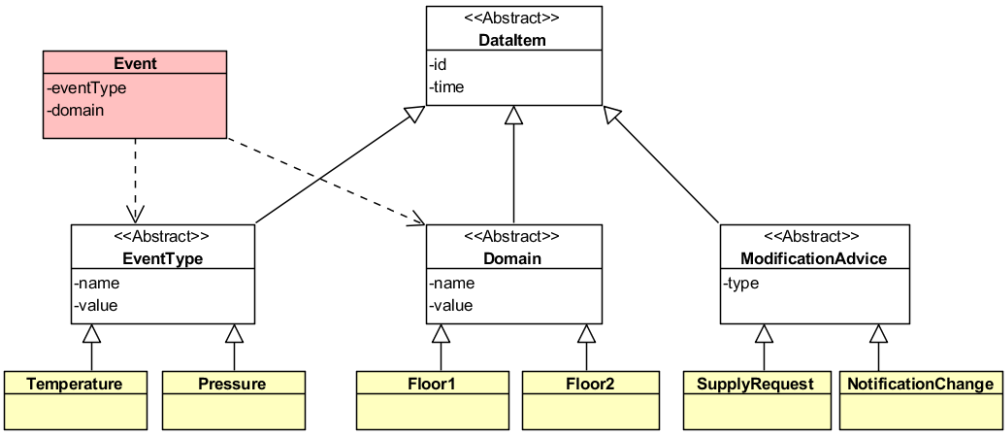


Figure 5.6: Data & Event Model containing the three core data types: *EventType*, *Domain*, and *ModificationAdvice*

The core data types are discussed in more detail in the enumeration below. Special attention is paid to their intended usage.

- *EventType* contains values, either measured, calculated or detected by the component DN. Different instances are used to deal with the multi-modal characteristic of sensors, whereby each instance defines exactly one measured type. For example, the physical units temperature and air pressure would represent two distinct instances. Using the framework, applications have the free choice of defining instances of *EventType* and extend them with custom meta-information. Sometimes it may be necessary to extend them with meta-information like measuring unit. For example, regarding temperature it makes a difference whether the measurement unit is degree Celsius or degree Fahrenheit.
  Within an application, developers should ensure that all nodes share a common understanding about the *EventType* in use.

- *Domain* contains origination information to precisely determine the provenance of items. One possible use case is the classification of event types according their geographical location. Conceivably they are only of interest if they belong to a specific domain.
  Similar to *EventType*, applications can define arbitrarily instances of *Domain* in order to customize the system. For example, if an application is written to observe a building, individual floors could be designed as single instances.

- *ModificationAdvice* contains instructions to change the configuration and behavior of the system. The delivery advice for *DNs* send by the *CMU* is an example of an built-in *ModificationAdvice*. It instructs sensors to start delivering the requested data.

- *Event* is composed of the two data types *EventType* and *Domain*, hereby serves as a wrapper. The significant attributes (value, type, location, time) are included in this envelope. While value and type are provided by *EventType*, *Domain* caries the location information. The attribute time is included in both of them.

5.4 Rule Language

The object of this section is to define a rule language, which is largely simplified and optimized for the needs of the proposed framework in particular and for the requirements of IoT in general. An important aspect is to reduce the complexity and administrative effort of writing rules. End users should benefit from a clear and intuitive grammar in terms of punctuation. It supports natural and domain specific notions that allow the language to morph to a concrete application domain. Another objective is to provide interoperability between various event processing engines. It is not surprising that in return the rule language will cover only a subset of all possibilities provided by other event processing engines discussed in Chapter 2. Nevertheless, the framework is designed

to accept “native“ grammars, in the sense that rules can also be written in the definite grammar of the event processing engine in use.
The definition of the rule language is divided into two parts. The first part concerns the ability to describe a pattern of events that is of interest. It should describe precisely not only the events, but also their causal and timing dependencies. This part is denoted as “query“ part. The second part, the “rule“ part, considers possibilities to specify actions that have to be taken whenever one or multiple queries are matched. This separation increases the reusability of queries by using them in multiple rules. This will be discussed in detail in Section 5.4.2.
Altogether, the Backus Naur Form (BNF) like notation is used to provide an approximation of the grammar. John Backus and Peter Naur introduced a formal notation to describe the syntax of a given language in the 1960s. It is used to formally define the grammar of languages, so that there is no disagreement or ambiguity as to what is allowed and what is not. The fundamental principle is that so called “productions“ transform non-terminal sequences into a sequence of terminals. As the name suggests, terminals are valid words of the language, whereas non-terminals are part of the speech and only allowed on intermediate stages. In this work, BNF is not only important to describe the syntax theoretically, but it is also used in combination with libraries to construct a parser for the prototype implementation mechanically.
The definition of both parts is done in an iterative approach in realizing the total scope of the language step by step. It starts with a simple requirement and increase expressiveness with every further requirement. The final aim is to obtain a grammar, in which both the syntax and the semantic is well defined and all assumed requirements can be expressed.

5.4.1 Query Part

Queries are templates that match the occurrence of certain events. From the grammar perspective, they are subdivided into several elements, including the *Condition*, the *Domain*, and the *Window* clause. The *Condition* clause is a portion of a query that restricts the events matched by specific conditions. The *Domain* clause indicates the origin of events, and the *Window* clause steers the time-oriented framing conditions. The basic structure of a query in BNF notation reads as follows (The question mark (?) denotes *zero or one occurrence*):

```
<query>      ::== 'CONDITION' <condition> <domain>? <window>?
```

The keyword “CONDITION“ is used to advise about the beginning of the *CONDITION* clause.

Requirement Q1

The simplest requirement covers the matching of single events. It also includes the task of evaluating values sent from *DNs*. An example is the following textual description:

- All events of type “temperature“ should be detected. As additional constraint, only events should be taken into account where the temperature value is greater than 30 degrees Celsius.

As a precondition, the grammar must allow the usage of event types in order to write expressions. For that reason, the data type *EventType* was introduced in the previous Section 5.3. On the one hand, the data type is needed by sensors and devices to distribute their measured values, on the other hand it establishes the connection to the syntax of the grammar. The name of the corresponding *EventType* is used, to specify placeholders in expressions for values sent from *DNs*.
For evaluation, the language supports basic arithmetic and comparison operators. Obviously, the operators =, <, >, <=, >= are used to reason about “is equal“, “is less than“, “is bigger than“, “is less or equal than“ respectively “is bigger or equal than“. These operators are defined between literal values and variables. Literal values represent constants which may be integers or strings. The preliminary syntax is depicted in Table 5.3.

```
<query>      ::== 'CONDITION' <condition>
<condition>  ::== <expression>
<expression>::== <property> <operator> <property>
<property>   ::== (<variable> | <string> | <integer>)
<operator>   ::== ('=' | '<' | '>' | '<=' | '>=')
<variable>   ::== ('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' | '0'..'9' | '-' | '_' )+
<string>     ::== ('A'..'Z' | 'a'..'z' | '0'..'9')+
<integer>    ::== ('0'..'9')+
```

Table 5.3: Query grammar covering requirement Q1.

Requirement Q2

Next, the *Condition* clause is extended by logic operators. Logic operators are used to connect two or more *Condition* clauses in a way, such that the result forms a valid query. A textual description of an example could read as follows:

- All events of type “temperature“ should be detected for which the temperature value is greater than 30 degrees Celsius or less than 10 degrees Celsius.

The grammar supports three kinds of operations:

- **NOT** A negation of an item is satisfied when it is not detected.
- **AND** A conjunction of items is satisfied when all the items have been detected.
- **OR** A disjunction of items is satisfied when at least one item has been detected.

Thereby a specific ordering relation between clauses is not important. In a more formal shape, this means that the used logic operators are associative. The extended grammar is outlined in Table 5.4. What appears is the distinction between unary (NOT) and binary (AND, OR) operators, and the productions regarding composite and single conditions. Single conditions are directly translated into expressions as discussed in the previous requirement, whereas composite conditions are build upon logical operators and eventually are also dissolved by expressions.

<query>	::==	'CONDITION' <condition>
<condition>	::==	(<singleCondition> | <compositeCondition>)
<singleCondition>	::==	<expression>
<compositeCondition>	::==	(<compositeOpUnary> | <compositeOpBinary> | <singleCondition>)
<compositeOpUnary>	::==	<compositeFuncUnary> <compositeCondition>
<compositeOpBinary>	::==	<singleCondition> <compositeFuncBinary> <compositeCondition>
<compositeFuncUnary>	::==	('NOT')
<compositeFuncBinary>	::==	('AND' | 'OR')
<expression>	::==	<property> <operator> <property>
<property>	::==	(<variable> | <string> | <integer>)
<operator>	::==	('=' | '<' | '>' | '<=' | '>=')
<variable>	::==	('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' | '0'..'9' | '-' | '_')+
<string>	::==	('A'..'Z' | 'a'..'z' | '0'..'9')+
<integer>	::==	('0'..'9')+

Table 5.4: Query grammar covering requirement Q1 and Q2.

Requirement Q3

In the next step, the grammar of the query language is extended by the ability to evaluate from which spatial scope an event originates. In view of this requirement in Section 5.3 the data type *Domain* was introduced. To recap, the data type is an autonomous entity due to the importance of spatial information in the IoT context. By way of illustration, the following requirement could be requested:

- All events in which the value of the type "temperature" exceeds 25 degrees Celsius in the premises of a building should be detected. Particular attention is given

to office spaces, so the air conditioning system can provide cooling exactly where needed.

The temporary result of the extended grammar by domain information is depicted in Table 5.5. The keyword “FROM“ is used to advise about the beginning of the clause *Domain*. Similar to event types, the name of the corresponding *Domain* is used to specify placeholders for requested domains.

<query>	::==	'CONDITION' <condition> <domain>?
<domain>	::==	'FROM' (<variable> | <variable> ',' <variable>)
<condition>	::==	(<singleCondition> | <compositeCondition>)
<singleCondition>	::==	<expression>
<compositeCondition>	::==	(<compositeOpUnary> | <compositeOpBinary> | <singleCondition>)
<compositeOpUnary>	::==	<compositeFuncUnary> <compositeCondition>
<compositeOpBinary>	::==	<singleCondition> <compositeFuncBinary> <compositeCondition>
<compositeFuncUnary>	::==	('NOT')
<compositeFuncBinary>	::==	('AND' | 'OR')
<expression>	::==	<property> <operator> <property>
<property>	::==	(<variable> | <string> | <integer>)
<operator>	::==	('=' | '<' | '>' | '<=' | '>=')
<variable>	::==	('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' | '0'..'9' | '-' | '_')+
<string>	::==	('A'..'Z' | 'a'..'z' | '0'..'9')+
<integer>	::==	('0'..'9')+

Table 5.5: Query grammar covering requirement Q1, Q2, and Q3.

Requirement Q4

Another common use case is aggregate functions, where multiple events are grouped together as input on certain criteria to form a single value of more significant meaning. The definition of time windows is closely connected, in order to provide a time frame for the aggregate function. For instance, a request including an aggregate function over a time window could be:

- All events of type “temperature“ should be detected, where the average value exceeds 50 degree Celsius in a time window over the last 10 minutes.

The grammar supports following aggregate functions:

- **SUM** The *summation* is the addition of a sequence of values.

- **AVG** The *average* is the sum of a list of values divided by the number of values.
- **COUNT** The *counting* finds the number of events.
- **MAX** The *maxima* finds the event with the highest value.
- **MIN** The *minima* finds the event with the lowest value.

The final grammar of the query part is depicted in Table 5.6. It is noteworthy that both time-based and count-based window types are supported. Aggregate operators are integrated as extra option for dissolving a single condition.

<query>	::==	'CONDITION' <condition> <domain>? <window>?
<domain>	::==	'FROM' (<variable> | <variable> ',' <variable>)
<window>	::==	<windowType> '(' <integer> ')'
<windowType>	::==	('WIN:TIME' | 'WIN:LENGTH')
<condition>	::==	(<singleCondition> | <compositeCondition>)
<singleCondition>	::==	(<expression> | <aggregateCondition>)
<compositeCondition>	::==	(<compositeOpUnary> | <compositeOpBinary> | <singleCondition>)
<compositeOpUnary>	::==	<compositeFuncUnary> <compositeCondition>
<compositeOpBinary>	::==	<singleCondition> <compositeFuncBinary> <compositeCondition>
<compositeFuncUnary>	::==	('NOT')
<compositeFuncBinary>	::==	('AND' | 'OR')
<aggregateCondition>	::==	<aggregateFunc> '(' variable ')' <operator> <integer>
<aggregateFunc>	::==	('SUM' | 'AVG' | 'COUNT' | 'MAX' | 'MIN')
<expression>	::==	<property> <operator> <property>
<property>	::==	(<variable> | <string> | <integer>)
<operator>	::==	('=' | '<' | '>' | '<=' | '>=')
<variable>	::==	('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' | '0'..'9' | '-' | '_')+
<string>	::==	('A'..'Z' | 'a'..'z' | '0'..'9')+
<integer>	::==	('0'..'9')+

Table 5.6: Query grammar covering requirement Q1, Q2, Q3, and Q4.

5.4.2 Rule Part

By the means of queries, it is possible to express patterns of interest. Towards a reactive system, the gap between pattern matching and taking actions is closed. Therefore, rules are introduced, which imply a causal relationship between the queries that trigger an action and the task to be done. Generally, rules allow the system to respond in the case a query is striking. The principal skeleton of a rule in BNF notation is depicted below:

Rules are subdivided into several language elements, including *Query*, *Window* and *Reaction* clauses. Their names are largely self-explanatory and refer to the queries that

```
<rule>  ::== <queries> <window>? ('TRIGGERS') <reaction>
```

should be matched according to the window that should be considered. Furthermore, the keyword “TRIGGERS“ serves as introduction of the *Reaction* element, which in turn servers as placeholder for the definition of reaction advices.

Requirement R1

A basic use case is the modification of configuration settings. For instance, if the temperature in a room exceeds a certain limit, all temperature sensors could be instructed to increase the resolution and send current values in shorter intervals. Therefore, reactions should express which modification should be applied, not forgetting, in which domains and to which devices. Logically, reactions are triples consisting of *EventType* (which type of device), *Domain* (which spatial area), and *ModificationAdvice* (what kind of change) as shown in Table 5.7. It should be noted that queries are determined via additional

```
<rule>              ::== <queries> ('TRIGGERS') <reaction>
<queries>           :==  <variable>
<reaction>          ::== <eventType> ',' <domainInfo> ',' <modificationAdvice>
<eventType>         ::== <variable>
<domain>            ::== <variable>
<modificationAdvice> ::== <variable>
<variable>          ::== ('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' | '0'..'9' | '-' | '_' )+
<string>            ::== ('A'..'Z' | 'a'..'z' | '0'..'9')+
<integer>           ::== ('0'..'9')+
```

Table 5.7: Rule grammar covering requirement R1.

names, which are issued during registration time. These names serve as unique identifiers in the rule grammar. As a positive consequence of this concept, queries could be reused in multiple rules.

Requirement R2

In addition to the current expressiveness, the rule grammar is further extended to handle timing requirements between queries. Of interest here is how to express “followed by“ relationships.
In order to implement this requirement, the production “query“ is extended by the symbol "->". It says that the query noted on the left-hand side must occur temporal before the query stated on the right-hand side. If this is the case the extended query is matched and triggers the reaction of the rule. Our target is therefore achieved by specifying an

ordering of the queries. The time frame is once again defined by the window concept as it has been already discussed before. The final rule grammar is shown in Table 5.8.

<rule>	::==	<query> <window>? ('TRIGGERS') <reaction>
<query>	::==	(<variable> '->' <query>) | <variable>
<reactions>	::==	(<reaction> ';' <reactions> | <reaction>)
<reaction>	::==	<eventType> ',' <domainInfo> ',' <modificationAdvice>
<eventType>	::==	<variable>
<domain>	::==	<variable>
<modificationAdvice>	::==	<variable>
<window>	::==	<windowType> '(' <integer> ')'
<windowType>	::==	('WIN:TIME' | 'WIN:LENGTH')
<variable>	::==	('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' | '0'..'9' | '-' | '_')+
<string>	::==	('A'..'Z' | 'a'..'z' | '0'..'9')+
<integer>	::==	('0'..'9')+

Table 5.8: Rule grammar covering requirement R1 and R2.

CHAPTER 6

Implementation

This chapter discusses the prototypical implementation of the design described in Chapter 5. First, a closer look at the development environment is taken, including frameworks and common design patterns. Second, low-level implementation details are discussed in order to gain deeper insights into various aspects of the prototype and its realization.

6.1 Frameworks and implementation patterns

The technological foundation of the prototype is the programming language Java that looks back on a history of more than twenty years. The long history and wide adoption have created a comprehensive ecosystem of frameworks, libraries, and documentations, for supporting and accelerating the development process. Tools of this ecosystem are Spring and Vaadin, which are intensively used for the prototype implementation.

6.1.1 Spring framework

The Spring framework is an application framework and Inversion Of Control (IoC) container for the Java platform. It was initially created as an alternative to Enterprise JavaBeans (EJB), which is a server-side component framework taking care of common challenges, developers encounter when implementing an enterprise application. According to official sources from Oracle[1], EJB enables rapid and simplified development of distributed, transactional, secure, and portable applications based on Java technology [Cor16b]. In general, Spring pursues the same intention.
The second basic concept of Spring, the IoC mechanism, is a design principle in which classes of a program receive dependencies (other classes) from a generic framework. This process is essentially the opposite of traditionally procedural approaches, where classes and methods are explicitly responsible for their dependencies. Fowler [Fow16]

[1] `https://www.oracle.com/index.html`

suggests renaming the principle to Dependency Injection (DI) due to the fact that the term IoC is not sufficiently specific for object-oriented programming. Hence, DI is a form of IoC, where object B is passed into object A either through constructors, setters, or service look-ups, which the object A will "depend on" in order to behave correctly. The advantage is that it helps to decouple components and to foster the effective reuse of software items (classes, modules, etc.). During development, an interface, which is implemented by object B, ensures that functionality in form of methods is available. The definition of which object (B, C, D, ...) is injected, is done externally. For example, if it is needed to change the properties of object B or the object B itself, no changes on the code are required, all adjustments are done via an external configuration file.
The Spring framework comes in a modular approach, which allows developers to choose parts they are interested in. Presently, the Spring framework consists of features that are split into about twenty modules satisfying all kinds of requirements. The downside is that these modules make the process of finding the appropriate configuration quite time consuming for unexperienced developers. In order to address this issue, the Spring team provides a solution under the name Spring Boot. It is an add-on to Spring framework and facilitates the development process of stand-alone, production-grade applications. Thereby, Spring Boot bundles modules, known as starters, which possess well-known interoperability.
Another advantage of Spring is that it does not need an application server. Instead, the application is deployed in a web container such as Tomcat or Jetty.
As a consequence, Spring Boot is preferred over EJB for the prototype implementation, complemented by Maven[2] as build and dependency management tool. Working with Spring Boot makes the set up of the project rather simple and clear. A handful of starters is added to the project descriptor and all parent dependencies with configured versions are resolved automatically. Since Maven is used, the file *pom.xml* represents the project descriptor. It acts as recipe to build the project [Sof16a].
In the following common Spring Boot issues are presented and discussed, which occur in slightly different forms more than once. Instead of repeating the same concept multiple times, a general description is presented, which serves as reference during the detailed discussion of the prototype.

6.1.2 Entry-Point

Out of the box, Spring Boot uses a *public static void main* entry-point that launches an embedded web server. By using the annotation *@SpringBootApplication*, Spring Boot attempts to configure the application based on information largely derived in a convention-over-configuration manner. The framework scans the classpath and draws the appropriate conclusion regarding the configuration. For example, if the framework detects the implementation of an in-memory database on the classpath, and no database connection has been configured manually, then it auto-configures an in-memory database. Listing 6.1 depicts the entry-point in a Spring Boot application.

[2] `https://maven.apache.org/`

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 @SpringBootApplication
4 public class Application {
5
6 public static void main(String[] args) {
7    SpringApplication.run(Application.class, args);
8    }
9 }
```

Listing 6.1: Implementation of a Spring Boot entry-point to launch an embedded web server.

Persisting Data

For data management, Spring Boot provides an extensive support for working with SQL databases. The functional range stretches from direct Java Database Connectivity (JDBC) access to complete object-relational mapping (Object-relational mapping (ORM)) technologies. Boot is leveraging the module Spring Data for ORM and offers the convenient concept of *Repository*, which generates database queries directly from the method names. Convention-over-configuration is once again the key factor. The query builder mechanism parses method names and searches for specific phrases like *find...By*, *read...By*, and so on. Thereby, the first *By* acts as delimiter and indicates the criteria. For example, the declaration of the method *findByLastname(@Param("lastname")String lastname)* in the entity *Person* generates the implementation of a query to retrieve all persons with an specific name. The entire implementation is automatically provided by the Spring framework.
Subsequently, it is sufficient to extend the entity class by *CrudRepository* instead of *Repository*, to become a full-fledged Data Access Object (DAO). It provides sophisticated CRUD (create, read, update, delete) functionality for the class that is being managed, all possible without writing a single line of code. Listing 6.2 shows a class in its specialized role as DAO. The interface is extended by the *@CRUDRepository* interface to instruct the framework about its role.

```
1  import org.springframework.data.jpa.repository.Query;
2  import org.springframework.data.repository.CrudRepository;
3  import org.springframework.data.repository.query.Param;
4  import configuration.management.model.DeviceDLO;
5
6  public interface DeviceNodeRepository extends CrudRepository<DeviceDLO, Long> {
7
8     public DeviceDLO findByName(@Param("name") String name);
9
10    public DeviceDLO findByAuthority(@Param("authority") String authority);
11 }
```

Listing 6.2: Example of Spring Boot repository interface

Implementing the API

The majority of the prototype's communication follows the REST architectural style. It is a set of constraints, such as the utilization of a client server architecture, a uniform interface structure, and the principle of being stateless. REST does not make the use of HTTP compulsory, but it is most commonly and it is also the choice for the prototype. All three types of components (DNs, EPNs, CMUs) expose their functionality via an API. The implementation of the APIs is done by means of controllers, which Spring maps to HTTP resources. The mapping including data serialization and binding is done by Spring as soon as it detects the annotation *@RestController*. Many annotations which help developers to refine the HTTP resource exist. For example, *@RequestBody* annotated parameters access the HTTP request body with the convenience that Spring creates the object automatically. Another essential setting is stated by the annotation *@RequestMapping*. It defines the HTTP verb and the HTTP path to access the resource properly.
Regarding serialization, Spring offers built-in data converters to serialize objects to JavaScript Object Notation (JSON) for consumer of the API. Listing 6.3 depicts all the functionalities described in the lines above.

```
1  @RestController
2  public class CMUManageDeviceImpl implements CMUManageDevice {
3
4     @Autowired
5     private DeviceNodeTransformer transformer;
6
7     @Autowired
8     private DeviceNodeRepository repository;
9
10    @Override
11    @RequestMapping(value = "/registrations/devices", method = RequestMethod.GET)
12    public @ResponseBody ResponseEntity<List<Address>> getAll() {
13
14       List<Connection> dns = transformer.toRemote(Transformer.makeCollection(
      repository.findAll()));
15
16       return new ResponseEntity<List<Connection>>(devices, HttpStatus.OK);
17    }
18 }
```

Listing 6.3: Implementation of an API using various Spring Boot annotations for refinement.

The business logic behind the methods of the APIs is structured in so called *Action* tasks, which exhaust the Chain of Responsibility (CoR) design pattern. The CoR belongs to the group of behavioral patterns. Gamma et al. [GHJV95] describe these patterns as complex communication patterns, which are difficult to follow at run-time. CoR processes a request through a series of activity objects to decouple sender and receiver of a request.

The activities participating in the chain decide if they serve the request, otherwise they forward the request to the next in the chain. The most valuable asset for our scenario is the capability to construct the set of activities dynamically. By reusing activities in different chains, reusability, a great mantra of software engineering, is achieved. Figure 6.1 depicts the core structure of the CoR pattern as it is used in the prototype. The abstract class *Activity* acts as chain link between concrete activities.

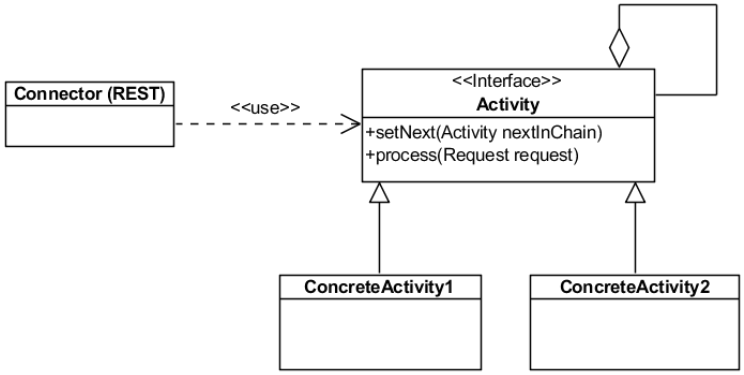


Figure 6.1: Structure of the design pattern Chain Of Responsibility. The class *Activity* acts as chain link between concrete implementations.

A list of *Activity* tasks used throughout the prototype including a description is depicted in Table 6.1.

Packaging

Spring Boot pursues a build mechanism that creates executable jars with an embedded container (Tomcat or Jetty) during build time. It aims to package a project into a lightweight, runnable artifact that is ready for distribution, straightaway. As already mentioned, Maven is used as build tool for the prototype. By executing the command *mvn clean install*, Spring Boot intercepts the jar build task and includes all the dependencies and objects like the web server into the resulting archive. This one archive file runs the entire Spring application with no fuss: no build tool required, no setup, and no web-server configuration [Lon16].

6.1.3 Vaadin

Vaadin enables developers to build user interfaces in a way that facilitates a component based approach. It provides a library of ready-to-use components, supplemented with a framework to create own components. Vaadin runs a servlet in a Java web server, serving HTTP requests. Additionally, Google Web Toolkit (GWT) is used for low-level rendering tasks at the browser. GWT is largely invisible for applications that do not require any custom GWT components.
While conventional HTML pages receive content with page updates, Vaadin employs

Label	Activity	Task description
A1	ValidateAddress	Activity validates addresses. It ensures that IP addresses (x.x.x.x, whereby x is between 0 and 255) and port number (x, whereby x is between 1024 and 65535) is in a valid range.
A2	ValidateDatasource	Activity validates event data sources regarding syntax and existence.
A3	StartDelivery	Activity starts delivery procedure. First, it monitors if the maximum number of delivery tasks is not exceeded. Second, it determines, if a new delivery task has to be started and stores relevant information.
A4	StopDelivery	Activity stops delivery procedure. It stops the thread of the delivery task and removes relevant information from the database.
A5	SetConfiguration	Activity sets configuration attributes, whereby existing attributes are overwritten.
A6	StatementValidation	Activity validates syntax of statement (query or rule) against our defined grammar.
A7	StatementTransformation	Activity transforms statement (query or rule) into an EPL representation.
A8	StatementPersistence	Activity stores statement (query or rule) into the database.
A9	StatementDeletion	Activity deletes statement (query or rule) from a database. Deletion is only possible, if statement is not active.
A10	StatementActivation	Activity activates statement (query or rule). EPLs are registered on the event processing engine.
A11	StatementDeactivation	Activity deactivates statement (query or rule). EPLs are unregistered from the event processing engine.
A12	SubscribeTopic	Activity subscribes to topic hosted by message broker.
A13	UnsubscribeTopic	Activity unsubscribes from topic hosted by message broker.
A14	NodePersistence	Activity persistence component (EPN or DN) into the database.
A15	DataSourcePersistence	Activity persistence data source of EPN into the database.
A16	Heartbeat	Activity processes heartbeat sent by a node. Message contains workload information regarding the node.

Table 6.1: List of concrete *Activity* tasks

extensively AJAX. AJAX stands for Asynchronous JavaScript and XML and enables pages to send requests to the server using an asynchronous mechanism. This way, only small parts of the page can be updated as needed [Grö11].
The Model-View-Presenter (MVP) is a pattern, which is inextricable linked to the process of developing applications with Vaadin. In contrast to the similar Model-View-Controller

(MVC) pattern, the view does not interact directly with the model. All invocations from the view are delegated to the presenter. The inverse communication takes place through an interface, which helps to simplify the mocking of the view for testing purposes. In general, the MVP isolates the view implementation better than in MVC, which is beneficial in unit testing of both presenter and model [Grö16]. Figure 6.2 illustrates the basic structure of the MVP pattern.

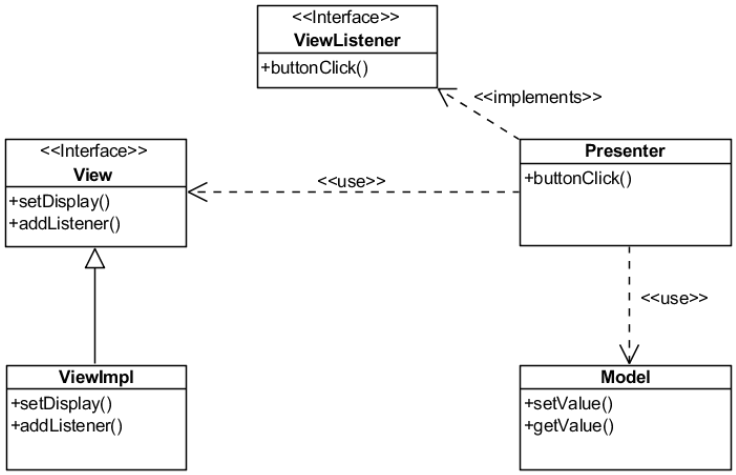


Figure 6.2: MVP structure (adapted from [Grö16]). View and model do not interact with each other directly.

6.1.4 JMS

Java Messaging Service (JMS) is a Java-based interface that was developed to provide a means for Java programs to access messaging systems. Messaging allows to integrate applications in a loosely couped and scalable manner. A component sends a message to a destination, and the recipient can receive the message from the destination without knowing anything about the receiver.
Messaging systems are classified into different models that specify which client receives a message. The publish-subscribe messaging model is used for the exchange of events (messages) between the components EPNs and DNs. This model is preferred, because it is very likely that multiple EPNs have to receive the same event (message). In other words, a message has multiple consumers. The central point in the publish-subscribe model is the topic, which operates somewhat like a bulletin board [Ora16b].
JMS applications are portable across different JMS providers, because the JMS architecture abstracts provider-specific information. The core elements of the JMS architecture are depicted in Figure 6.3. The *Connection Factory* is the object a client uses to create a connection to a provider. It encapsulates a set of connection configurations. *Destinations* represent the abstract object a client uses to specify the target of messages it produces and the source of messages it consumes. The topic in the publish-subscribe messaging

model is an example for a *Destination*. While the *Connection* represents the physical connection between a client and a provider, the *Session* is a single-threaded context for producing and consuming messages [Ora16c].

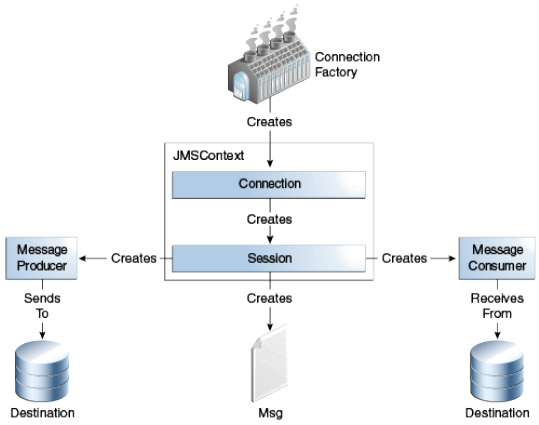


Figure 6.3: JMS API Architecture [Ora16c]

6.2 Prototype implementation

The project layout (Figure 6.4) of the prototype follows the component separation defined in Chapter 5, with the addition that a common module is applied in order to share code. For the sake of convenience, the prototype also provides a web-application mainly for management purposes. The classes for the web-application are outsourced in a separate module. The module functionality of Maven is applied to leverage project inheritance. In the following each module is discussed in detail.

6.2.1 Common Module

The common module encapsulates all functions that lay the foundation for the remaining components. In principle, all listed classes here provide some sort of utility functionality.

```
Prototype
├── Common-module
├── DN-module
├── EPN-module
├── CMU-module
└── Monitoring-Webapp
```

Figure 6.4: Project structure. Each item represents a Maven module with *Prototype* as parent.

The main objective of this module is to become more agile in case of changes and extensions.
Apart from the data model (Data, EventType, Domain, and ModificationAdvice), the module consists a list of enumerations for different purposes. The most significant of these is the enumeration *RESOURCE_NAMING*, which serves as helper item to simplify the management of exposed methods via resource paths. The predefined constants combine HTTP path and HTTP verb in order to access the underlying method. An extract from the enumeration is depicted in Listing 6.4.

```
1    CM_REGISTER_DEVICE("/registrations/devices", RequestMethod.POST),
2
3    CM_HEART_BEAT_DEVICE("/registrations/devices/{id}", RequestMethod.PUT)
```

Listing 6.4: Extract from the enumeration *RESOURCE_NAMING*

Another noteworthy aspect of the common module is the abstract class *Transformer* that supports the efficient transformation of objects. Transformation is needed in different scenarios, such as, converting between communication and persistent objects. The corresponding abstract class is illustrated in Listing 6.5.

```
1  public abstract class Transformer<LOCAL, REMOTE> {
2     public abstract LOCAL toLocal(REMOTE remote);
3
4     public abstract REMOTE toRemote(LOCAL local);
5
6     public Collection<LOCAL> toLocal(List<REMOTE> remotes) {
7        return remotes.stream().map(r -> toLocal(r)).collect(Collectors.toList());
8     }
9     public List<REMOTE> toRemote(List<LOCAL> locals) {
10       return locals.stream().map(l -> toRemote(l)).collect(Collectors.toList());
11    }
12 }
```

Listing 6.5: Abstract class *Transfomer*

The module also contains the class *XMLParser* that implements methods for marshalling and unmarshalling of Extensible Markup Language (XML) files. Marshalling means that the state and the codebase of an object is recorded in such a way that when unmarshalling is applied, a copy of the object is obtained [RSL99]. The Java API for XML Processing (JAXP) is used for parsing. Thereby, the document is converted into a tree of nodes that represent the full content of the file. Once the tree of the document is created, a program can examine and manipulate the nodes at will [Fla05].
The parser is especially needed to read and write the bootstrapping file. As the name suggests, it plays an important role in the bootstrapping process by containing basic details. According to the design of the prototype, all nodes (DNs, EPNs) need to register at the CMU. The information required for registration, are obtained via the bootstrapping file, which is stored locally. The structure of the configuration file is

defined via a XML Schema Definition (XSD). Extracts from the file are depicted in Listings 6.6, 6.7, and 6.8. The root element *bootstrapping* contains two main ingredients *events* and *addresses*. Both elements are basically wrappers to surround a list of elements respectively (Listing 6.6).

```
<xs:element name="bootstrapping">
   <xs:complexType>
      <xs:all>
         <xs:element ref="events" />
         <xs:element ref="addresses" />
      </xs:all>
   </xs:complexType>
</xs:element>
```

Listing 6.6: Extract from bootstrapping file: Core element

The element *addresses* holds crucial connection information (host, port), for the communication setup to other components of the system. Especially, address information regarding the CMU and the location of the message broker are essential to perform basic tasks such as the registration handshake (Listing 6.7).

```
<xs:element name="addresses">
   <xs:complexType>
      <xs:sequence maxOccurs="unbounded" minOccurs="0">
         <xs:element ref="address" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="connection" >
   <xs:complexType>
      <xs:sequence>
         <xs:element type="xs:string" name="name"/>
         <xs:element type="xs:string" name="host"/>
         <xs:element type="xs:string" name="port"/>
      </xs:sequence>
      <xs:attribute type="Component" name="component" />
   </xs:complexType>
</xs:element>
```

Listing 6.7: Extract from bootstrapping file: Addresses

The structure of the second wrapper element (*events*) is largely self-explanatory as it represents the data model. The bootstrapping file contains all *events* that the respective node supports (Listing 6.8).

```
<xs:element name="events">
   <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="event" />
```

```
5        </xs:sequence>
6     </xs:complexType>
7  </xs:element>
8  <xs:element name="event">
9     <xs:complexType>
10       <xs:sequence>
11          <xs:element ref="deviceType" />
12          <xs:element ref="domain" />
13             </xs:sequence>
14    </xs:complexType>
15 </xs:element>
```

Listing 6.8: Extract from bootstrapping file: Events

6.2.2 DN Module

The class diagram (Figure 6.5) illustrates the structure of the module by showing the module's main classes, their attributes, operations, and the relationships among them. It is important to note that classes marked with yellow background, originate from the Spring framework, whereas transparent classes are part of the prototype. For the

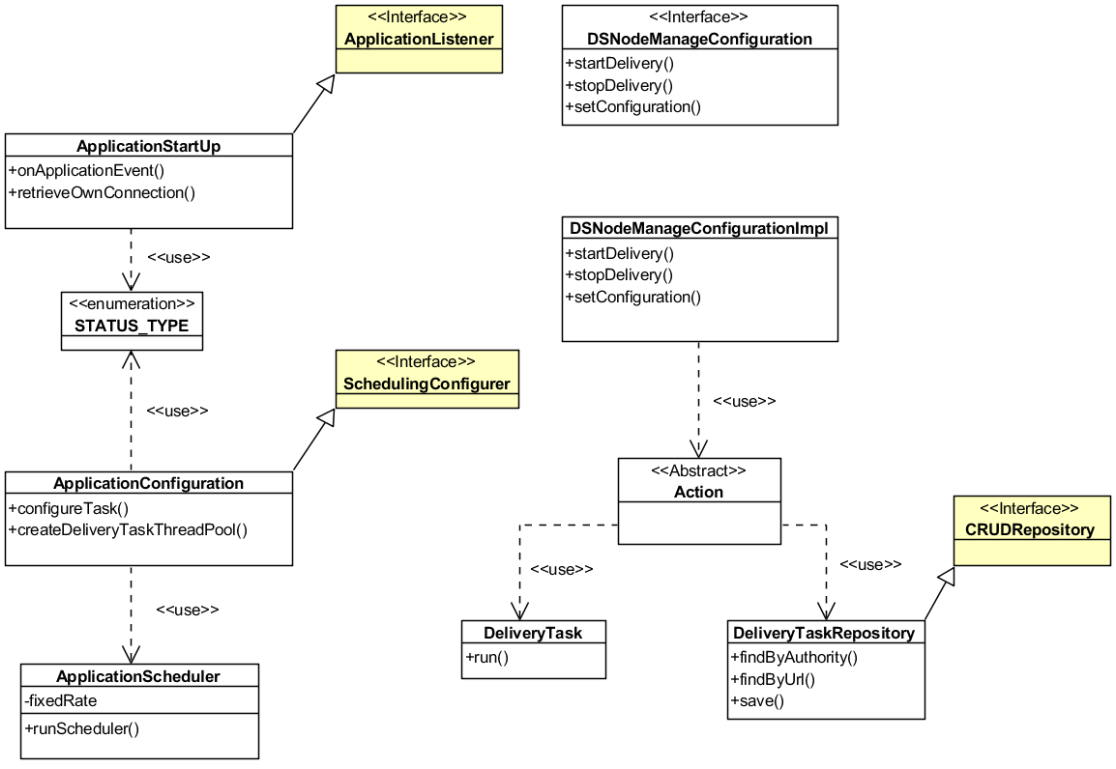


Figure 6.5: Class diagram of the DN module

registration process, the IP address and the port number of the corresponding node is needed. It makes sense for convenient reasons to determine this information on the fly in order to keep configuration effort as small as possible. To this end, the module polls its own connection information (IP address and port number) as soon as the application is started. The class *ApplicationStartUp* represents a listener, which is registered as an instance of the *ApplicationListener* in the Spring context. After Spring has started up and has finished its initializing, it signals that state to all instances of the *ApplicationListener*. In response, code is executed to retrieve IP address and port number of the running application.
For the remaining time-dependent activities, the class *ApplicationScheduler* is implemented. During start up, it schedules the registration procedure in all its variations from a seamless registration till a procedure at which the duration between two retries will be increased continuously. Afterwards, it ensures that heartbeat messages are sent to the CMU during normal operation. The interval at which the scheduler carries out the activities can be modified via the *fixedRate* attribute.
The annotation *@EnableScheduling* plays an important role, because it enables Spring's scheduled task execution capabilities. By default, all bean methods annotated with *@Scheduler* are delegated to a thread pool executor with pool size ten [Sof16b].
The interface *SchedulingConfigurer* offers an entry-point to obtain more control over specific scheduling settings. By using a custom implementation of the *Trigger* interface, the calculation of the next execution time is done on the fly, which enables variation. In order to keep an overview of the activities the enumeration *STATUS_TYPE* serves as helper. Listing 6.9 shows an excerpt of the class *ApplicationConfiguration*, showing the definition of a custom *Trigger*.

```
1  @Override
2  public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
3
4     taskRegistrar.setScheduler(applicationExcutor());
5     taskRegistrar.addTriggerTask(new Runnable() {
6        @Override
7        public void run() {
8           lastExecution.setTime(execution.getTime());
9           lastStatus.setCurrent(status.getCurrent());
10          applicationScheduler.carryOutActivity();
11          }
12       }, new Trigger() {
13          @Override
14          public Date nextExecutionTime(TriggerContext triggerContext) {
15          return calculateNextExecutionTime(lastExecution, lastStatus);
16       }
17    });
18 }
```

Listing 6.9: Extract from the class ApplicationConfiguration

Furthermore, class diagram 6.5 shows the provided interface *DSNodeManageConfiguration*. The module receives all kind of modification advices via this interface. The individual methods and their mapping to the associated HTTP verb and HTTP resource are summarized in Table 6.2. As stated at the beginning of this section, the business logic behind these methods are realized by a chain of *Activity* tasks. If the node gets instructed to supply an EPN instance with data (via modification advice), the device stores the instruction as *DeliveryTask*. Accessing the stored instructions is done via *DeliveryTaskRespository*. Concurrently, the delivery task is executed in an extra thread as long as the stop delivering advice is received.

Method	**HTTP path HTTP verb**	**Action(s)**	**Description**
startDelivery	/delivery/start POST	A1 A2 A3	Instruction to start delivery process. HTTP body contains the Modification-Advice.
stopDelivery	/delivery/stop POST	A1 A2 A4	Instruction to stop delivery process. HTTP body contains the *Modification-Advice*.
setConfiguration	/configurations POST	A1 A5	Instruction to set configuration. HTTP body contains *ModificationAdvice*.

Table 6.2: Overview of all supported HTTP paths and HTTP verbs of the DN module. Every combination of path and verb is associated with a respective Java method in the *DSNodeManageConfiguration*.

6.2.3 EPN Module

Figure 6.6 shows the class diagram of the EPN module, emphasizing the core elements of the implementation. Once again, classes with yellow background are associated with the Spring framework.

One design goal of the framework is the openness towards different event processing engines. The system should be independent on how its engines are initialized and employed. This specifically means, that the implementations of the event processing engines and all related classes are encapsulated, using the design pattern abstract factory. The pattern guarantees that only families of related objects work together [GHJV95]. Every participating event processing engine is plugged into the system by a concrete implementation of the interface *EngineFactory*. The prototype uses Esper as the event processing engine and the class *EsperEngineFactory* creates the family of Esper relevant classes. The family compromises the classes *EsperEngine*, *EsperEngineListener*, and *EsperLanguageTransformer*, which are concrete implementations of the corresponding

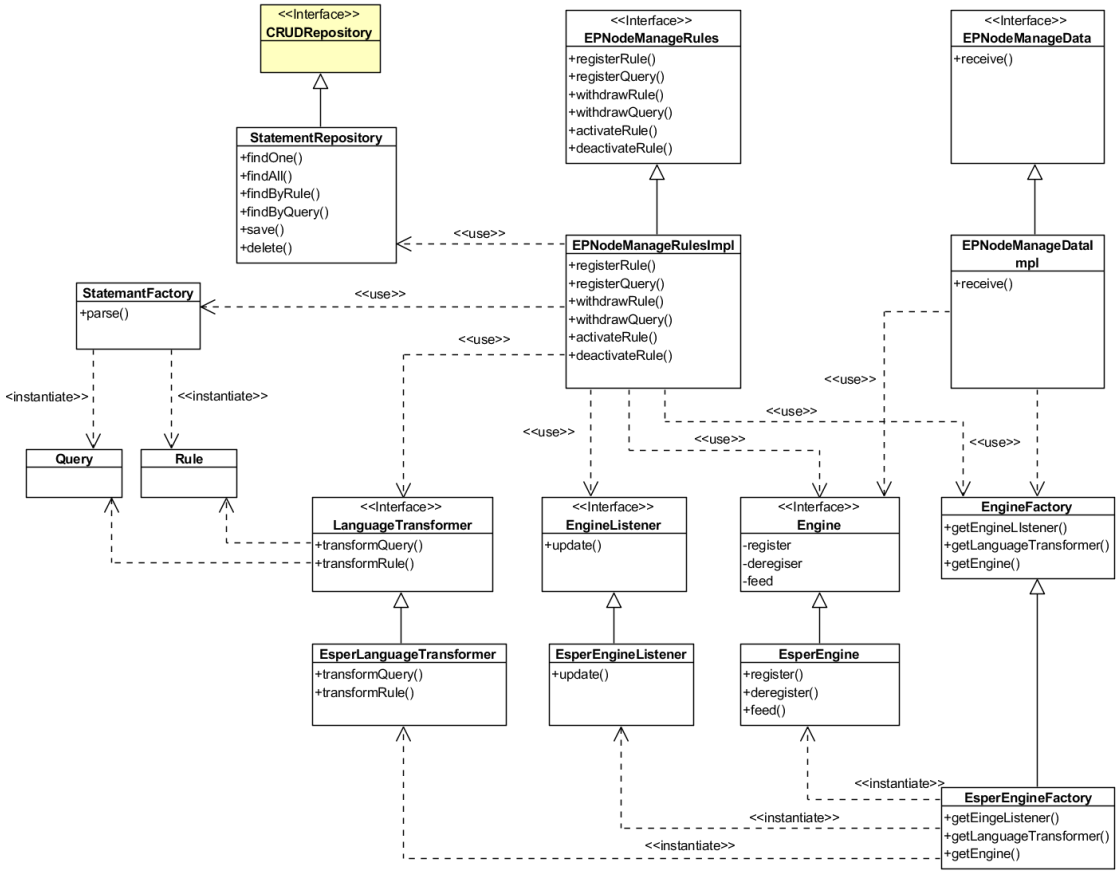


Figure 6.6: Class diagram of the EPN module

abstract classes *Engine*, *EngineListener*, and *LanguageTransformer*. The module deals only with the abstract classes and without any information on the implementation. The actual implementation of *EngineFactory* that the module applies is determined at runtime and can therefore vary from node to node.
The class *Engine* conducts the communication with the actual event processing engine. The communication content is primarily focused on statement registering and event passing.
In general, rules are passed to the system through the API *EPNodeManageRules*. Table 6.3 lists the methods of the API *EPNodeManageRules*, once again with the familiar pattern of representing the business logic as a chain of *Activity* tasks. Once a query or rule is received, the syntactic correctness is validated by the *LanguageFactory*. In case of success, it instantiates objects of the types *Query* and *Rule*, which encapsulate all parts of the fragmented statement. For this purpose the library ANother Tool for Language Recognition (ANTLR) [Par16] is engaged, which takes as input the grammar and generates a parser that can build and walk parse trees. The aforementioned objects *Query*

and *Rule* are the output of the tree traversal. Afterwards, the *LanguageTransformer* transforms the relevant statement parts of these objects into an interpretable statement of the concrete event engine in use. The next part (*Transformation process*) discusses in detail, how the *EsperLanguageTransformer* performs the translation to a EPL.
Rules instruct the event processing engine to find notable events. If such a condition is detected, the class *EngineListener* fires and initiates associated downstream activities. Depending on the action part of the rule, the next steps are started. Typically, the CMU is contacted in order to distribute modification advices. The event processing engine is

Method	**HTTP path** **HTTP verb**	**Action(s)**	**Description**
registerRule	/registrations/rule/{name} POST	A6 A7 A8	Registration of rule. HTTP body contains rule.
registerQuery	/registrations/query/{name} POST	A6 A7 A8	Registration of query. HTTP body contains query.
withdrawRule	/deregistrations/rule/{name} DELETE	A9	Deregistration of rule.
withdrawQuery	/deregistrations/query/{name} DELETE	A9	Deregistration of query.
activateRule	/activations/rule/{name} GET	A12 A10	Activation of rule.
deactivateRule	/activations/rule/{name} GET	A13 A11	Deactivation of rule.

Table 6.3: Overview of all supported HTTP paths and HTTP verbs of the EPN module. Every combination of path and verb is associated with a respective Java method in the *EPNodeManageRules*.

fed by the second API *ENodeManageData*. The message broker with its topics acts as data supplier.
For persisting rules and queries the repository *StatementRepository* is employed. The underlying database is realized as in-memory database (HyperSQL DataBase). The time critical parts share a lot of commonalities with the DN module discussed above. The *ApplicationScheduler* in combination with *STATUS_TYPE* are once again responsible for the start up and the heartbeat procedure (Figure 6.5).

Transformation process

The basis for the transformation process is the result produced by the ANTLR tree parser. The declarative string representation of rules are converted into structured objects. In the first place, these objects are not suitable for the Esper event processing engine. The remaining gap consists of transforming the query and rule objects back into a declarative representation, which complies to the syntax of EPL. An conceptual overview is depicted in Figure 6.7. The syntactical correctness of the transformation is verified by registering

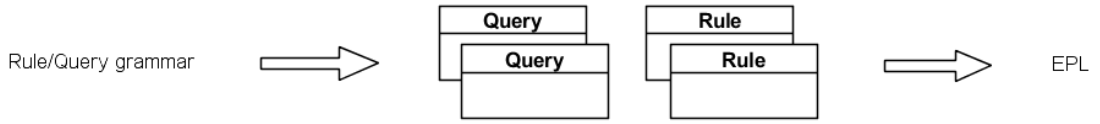


Figure 6.7: Conceptual overview of the transformation process between rule/query grammar and EPL

the output at the Esper engine. If the statement does not comply to the grammar, the statement is rejected immediately. However, the guarantee that the transformation preserves the semantically meaning of the expression is much more challenging. The formal proof that rules and queries of our rule language are semantically equivalent to the result of the transformation is not part of this thesis. Nonetheless, more details of the transformation process are presented. The pattern depicted in Listing 6.10 is used as a basic structure for the query transformation.

```
1 insert into QueryEvent select '[query_name]' as name from Event [window] as e where
      [where_condition] [where_domain]
```

Listing 6.10: Basic structure of an EPL statement for transforming a basic query.

The basic structure contains an *insert into* clause, which specifies the result of the *select* clause as an event in a further stream. A new event is inserted into the stream called *QueryEvent* as soon as the *select* clause is matched. The insertion into a further stream is needed in order to reflect the statement hierarchy consisting of queries and rules. The placeholders used in the pattern and the special case, where two patterns (statements) for the transformation are used, are discussed in the following.
The pattern contains four placeholders, which are filled during the transformation process. The first one, *query_name*, is used as query identifier in the stream *QueryEvent*. While *window* is one-to-one taken from the input, *where_condition* represents restrictions regarding properties of events we are interested in and how these properties are logical combined. Restrictions regarding *Domain* are reflected by the placeholder *where_domain*. In general, they are AND-linked at the end of the EPL statement.
If the rule is equipped with an aggregate function, a single EPL is not enough for mapping the requirement. In such a case, two streams of events are used. The pattern structure for the two EPLs is depicted in Listing 6.11.

```
1 insert into AggregatedValue select [aggregated_value] as value from Event as e
      where [where_condition] [where_domain]
2
3 insert into QueryEvent select '[queryName]' as name from AggregatedValue [window]
      where [aggregated_operation]
```

Listing 6.11: Basic structure of an EPL statement for transforming an advanced query.

The first EPL inserts new events in the stream *AggregatedValue* as soon as the placeholders for *where_condition* and *where_domain* are fulfilled. The placeholder *aggregated_value* holds the values, which are the ingredients for the aggregation function.
The second EPL announces Esper to perform the desired aggregate function on a time window defined in the placeholder *window*, once again, one-to-one mapped from the query. The result is also inserted into the stream called *QueryEvent*.
So far, a way was found to transform queries of the grammar into EPL statements of the Esper event processing language. It serves as basis for the remaining step, the transformation of rules. To recap, the transformation of queries is done by creating new events and inserting them into different streams. In doing so, the *EventListener* is never informed about a triggered statement, a fact that corresponds fully with the expected behavior. Only if the superordinate rule is matched, the *EventListener* should receive a notification. For this purpose, an EPL pattern is used that looks disparate to the previous structure (Figure 6.12).

```
1 select * from pattern [every (a=QueryEvent(name = '[query_name1]') -> b=QueryEvent(
      name = '[query_name2]')) where [window]]
```

Listing 6.12: Basic structure of an EPL statement for transforming a rule.

The pattern uses the two important operators *followedBy* ($\rightarrow$) and *every* for pattern matching in Esper. The *followedBy* operator specifies that first the left hand expression must turn true and only then the right hand expression is evaluated, ensuring that event *a* (*query_name1*) "causes" event *b* (*query_name2*). In addition, the pattern evaluation should be restarted when it evaluates to true or false. To this effect, the *every* operator is used. In the syntax of EPL the parentheses around the *every* subexpression are crucial. It guarantees that the pattern matcher is restarted as expected, namely, at the time event *b* occurs and matches the pattern. Additionally, it fires the *EventListener* only once independently how often event *a* occurred before. In contrast, if the parenthesis are omitted, it would fire for every single event *a*.

6.2.4 CMU Module

The CMU glues the system together. For that reason, it primarily stores several relevant information, including:

- Registered DNs, which are managed by the *DeviceNodeRepository*:

  - Information regarding connection address: IP address and port number.
  - Information regarding last activity: Timestamp of last heartbeat message.
  - Information regarding event data sources able to provide: Combinations of *EventType* and *Domain* transmitted as part of the handshake procedure.

- Registered EPNs, which are maintained by the *CRUDRepository* called *EventProcessingRepository*:
  - Information regarding connection address: IP address and port number.
  - Information about last activity: Timestamp of last heartbeat message.
  - Information about system utilization received as payload of the heartbeat message.
  - Information regarding installed rules: CMU tracks the deployment of rules to know at any time, which rule runs on which node.

- Committed Queries: Information about committed queries by users (*StatementRepository*).

- Committed Rules: Information about committed rules by users (*StatementRepository*).

All these information are stored by the mean of *CRUDRepository* enhancements. Besides persisting all these information, the CMU is mainly engaged in the interaction between system and users.
If the user wants to detect a pattern, he expresses his needs via queries and rules, and passes them to the CMU via the APIs *CMUnitManageDNs* and *CMUnitManageEPs*. In order to finally activate a rule on an EPN, the activation commando must be executed. During the activation process, the CMU selects an EPN based on strategies defined by the user. The implementation of strategies follows the structure depicted in Listing 6.8, whereby the three built-in strategies named *MinCPUUtilization*, *MinRAMUtilization*, and *MinNumberOfActiveRules* implement the interface *Selector*. For harmonisation, the *SelectionFacade* wraps the implementation of the *Selectors* and makes the usage for developers more intuitive. Besides rule load balancing, the framework's capability to handle failovers is an important feature. If the framework detects a faulty node, it initiates the redistribution of the rules running on the faulty node, thereby considering only healthy nodes. A node is assumed to be erroneous, if the timestamp of the last received heartbeat message exceeds a certain threshold. Both, the threshold and the time interval nodes sending heartbeat messages are configurable.
Table 6.4 represents the APIs *CMUnitManageDSNs* and *CMUnitManageEPNs* in the way already seen for the nodes EPN and DN. Business logic is implemented by a chain of *Action* tasks.

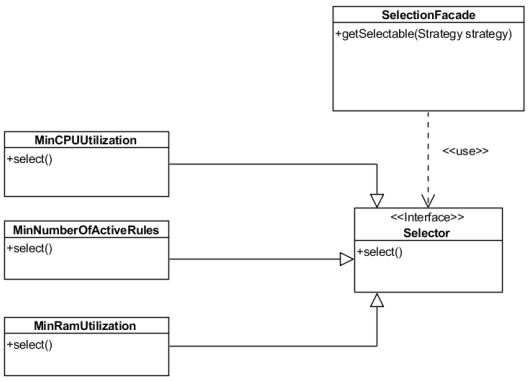


Figure 6.8: Selection pattern

Method	**HTTP path** **HTTP verb**	**Action(s)**	**Description**
register	/registrations/devices/ POST	A1 A14	Registration of DN. HTTP body contains address information.
registerDataSource	/registrations/devices/sources/ {id} POST	A1 A15	Registration of data sources (HTTP body) identified by id.
heartbeat	/registrations/devices/ {id} PUT	A1 A16	Heartbeat message identified by path variable id.
register	/registrations/eventprocessing/ POST	A1 A14	Registration of EPN HTTP body contains address information.
heartbeat	/registrations/eventprocessing/ {id}/{value1}/{value2} PUT	A1 A16	Heartbeat message identified by id inclusive measurement data (value1 and value2).

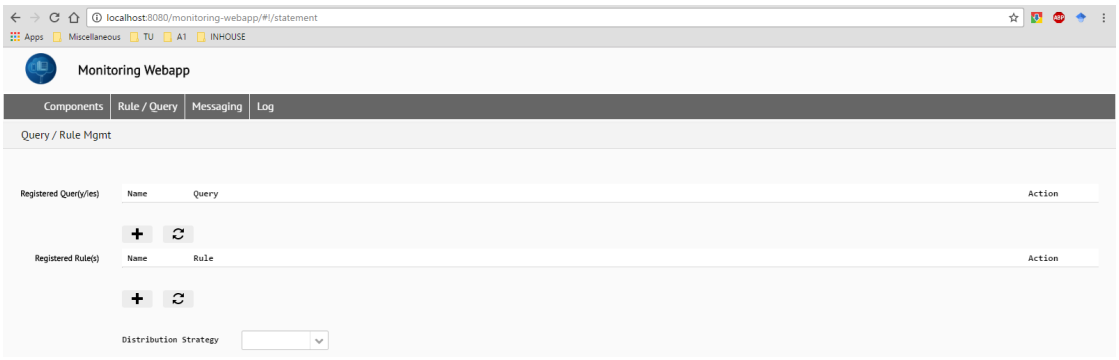
Table 6.4: Overview of all supported HTTP paths and HTTP verbs of the CMU module. Every combination of path and verb is associated with a respective Java method in the *CMUnitManageDSNs* and *CMUnitManageEPNs*.

6.2.5 Monitoring Webapp

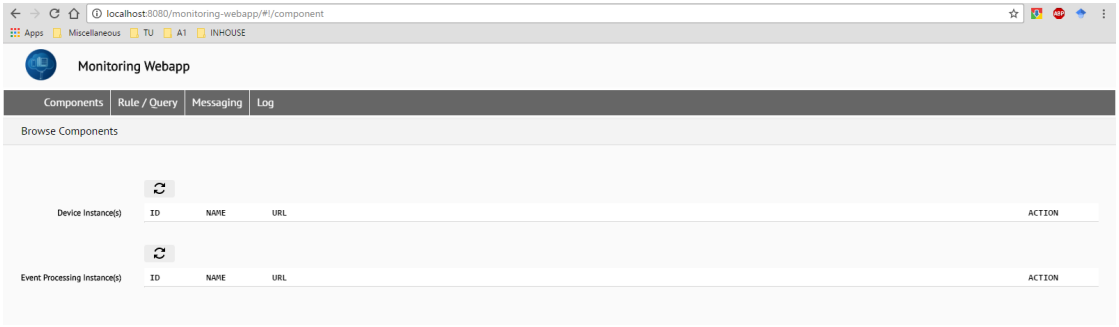
The *Monitoring Webapp* enables the user to manage the framework. It is a graphical user interface implemented with help of Vaadin. The *Monitoring Webapp* utilizes the APIs provided by the CMU for data exchange and supports the following operations:

- Installing and uninstalling of queries. During installation query syntax is checked. If it does not conform, user is immediately informed (Figure 6.9a).

- Installing and uninstalling of rules. During installation rule syntax is validated. If it does not conform, user is immediately informed (Figure 6.9a).
- Activation and deactivation of rules. The user can choose distribution strategies for selecting an EPN on which the rule is deployed (Figure 6.9a).
- Monitoring of registered DNs, including performance capacities, last heartbeat message, and event data sources (Figure 6.9b).
- Monitoring of registered EPNs, including performance capacities, last heartbeat message, registered queries and rules (Figure 6.9b).



(a) Monitoring Webapp: Page for query and rule management



(b) Monitoring Webapp: Page for browsing components

Figure 6.9: Pages of the Monitoring Webapp

6.2.6 Message broker

The message broker provides the JMS topics for exchanging messages (events) between instances of DNs and EPNs. A fully compliant JMS provider used for the prototype is Apache ActiveMQ [Fou16a], which offers many advanced features. The capability to deploy a broker within the Java Virtual Machine (JVM) is just one example leveraged in our implementation. The following Listing 6.13 starts the message broker within the

Monitoring Webapp [Fou16b]. The examples briefly outlines the interaction between framework and message broker.

```
1 BrokerService broker = new BrokerService();
2 broker.addConnector("tcp://localhost:61616");
3 broker.start();
```

Listing 6.13: Starting an embedded message broker.

Listing 6.14 shows how a connection with the JMS provider is established. It is noteworthy that besides of the class *AcitveMQConnectionFactory* nothing is revealed about the JMS provider. After receiving a *ConnectionFactory* all custom interaction is done on behalf of the JMS API.

```
1    @Override
2    public synchronized void start() {
3       ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://
      localhost:61616");
4
5       Connection connection = connectionFactory.createConnection();
6       connection.start();
7    }
```

Listing 6.14: Start method establishes a connection with the JMS provider.

Listing 6.15 reflects the structure of the JMS architecture. Starting with an existing *Connection*, the object *Session* is created. The *Session* in turn creates the *Topic*, which is identified by a name. With the help of the *MessageProducer* events are put into the *Topic*.
The *propertyValue* serves as selector for events and is determined by the names of a data source.

```
1    @Override
2    public synchronized void produce(Event event) {
3       Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
4
5       Topic topic = session.createTopic(MESSAGE_TOPIC);
6       MessageProducer producer = session.createProducer(topic);
7
8       String propertyValue = event.getEventType() + ":" + event.getDomain();
9       ObjectMessage message = session.createObjectMessage(event);
10      message.setStringProperty(PROPERTY, propertyValue);
11
12      producer.send(message);
13   }
```

Listing 6.15: Produce method writes an event into the topic.

Listing 6.16 depicts the way how listeners are registered in order to consume events from the JMS provider, once again reflecting all elements of the JMS architecture. The selector parameter acts as filter to specify events of interest.

```
1   @Override
2   public synchronized void consume(String selector, MessageListener
     messageListener) {
3
4      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
5
6      Topic topic = session.createTopic(MESSAGE_TOPIC);
7
8      MessageConsumer consumer = session.createConsumer(topic, selector);
9      consumer.setMessageListener(messageListener);
10  }
```

Listing 6.16: Consume methods installs a message listener.

CHAPTER 7

Evaluation

In this chapter the performance of the framework is evaluated using the prototype implementation presented in the previous chapter. The evaluation is divided into two major parts. The first concentrating on the use cases defined in Chapter 4, examining whether the requirements can be implemented on behalf of the prototype implementation. In addition to this feasibility study, the second part concentrates on metrics that describe the system. By conducting a quantitative evaluation, the performance and effectiveness of the framework should be demonstrated with the help of measured figures.

7.1 Use case evaluation

In Chapter 4 a series of use cases were discussed, outlining the functional scope of smart buildings. Three key factors were defined: saving energy, increasing comfort, and enlarging safety and security requirements. It is very likely that these three factors will shape the way buildings are constructed in the future. For the sake of evaluation, use cases referring to these factors are reconsidered.
The evaluation starts with use cases connected to the key factor saving energy (Section 4.1).

(i) Optimized heating, ventilation and air conditioning (HVAC)
Reasoning about room occupation were identified as significant important for optimizing HVAC. HVAC may only be activated if one or more persons spend some time in a room, while short stays for reasons of cleaning or picking up things should be omitted. The basic task is to prove a causal connection between door movement and movement in the room. One way of reasoning could read as follows: Rooms are assumed to be occupied, if after the door to the room was opened or closed, movements are recorded in the room for a certain amount of time. The delay is needed to prevent false positives. Listing 7.1 puts the described scenario

into practice with the help of the proposed framework and its language. The solution consists of two queries, which are the basic ingredients for the final rule that is connected through a followed-by relationship. In practice, the discrete events ("doorAction" and "movement") could be emitted by two simple sensors: one, a proximity sensor to observe the door, and secondly a motion sensor to record movements in the room. If the number of movement events exceeds a certain threshold it is assumed that a person occupies the room for a longer period of time.

```
1 /* Query1 */
2 CONDITION name="doorAction" FROM OfficeRoom
3
4 /* Query2 */
5 CONDITION COUNT(movement) >= 5 FROM OfficeRoom WIN:TIME(600)
6
7 /* Rule */
8 Query1 -> Query2 TRIGGERS Heating, OfficeRoom, ON_OFF = 1 WIN:TIME(600)
```

Listing 7.1: Implementation of use case optimized HVAC.

The result of the transformation into EPL statements is depicted in Listing 7.2. In total, four EPL statements are created and installed at the Esper CEP engine.

```
1  /* EPL1 */
2  insert into QueryEvent select 'Query1' as name from Event as e where e.
       eventType.name = 'doorAction' and e.domain.name = 'OfficeRoom'
3
4  /* EPL2 */
5  insert into AggregatedValue select count(e) as value from Event.win:time
       (600 seconds) as e where e.eventType.name = 'movement' and e.domain.
       name = 'OfficeRoom'
6
7  /* EPL3 */
8  insert into QueryEvent select 'Query2' as name from AggregatedValue
       where value >= 5
9
10 /* EPL4 */
11 select * from pattern [every (a=QueryEvent(name = 'Query1') -> b=
       QueryEvent(name = 'Query2')) where timer:within(600 seconds)]
```

Listing 7.2: EPL statements for use case optimized HVAC.

Figure 7.1 shows a graphical representation of two test sequences, which were conducted to determine if the functional requirements are met. Both sequences evaluate room occupation as expected. While the first sequence signals that the room is occupied, the events of the second sequence do not satisfy our definition for occupation, and thus do not signal room occupation.

(ii) **Proactive maintenance of equipment**
The aim here is to detect performance problems before they cause expensive outage. Listing 7.3 offers a generic solution, which is applicable to all kinds of machines in the reference building. The assumption is that equipments having an average load

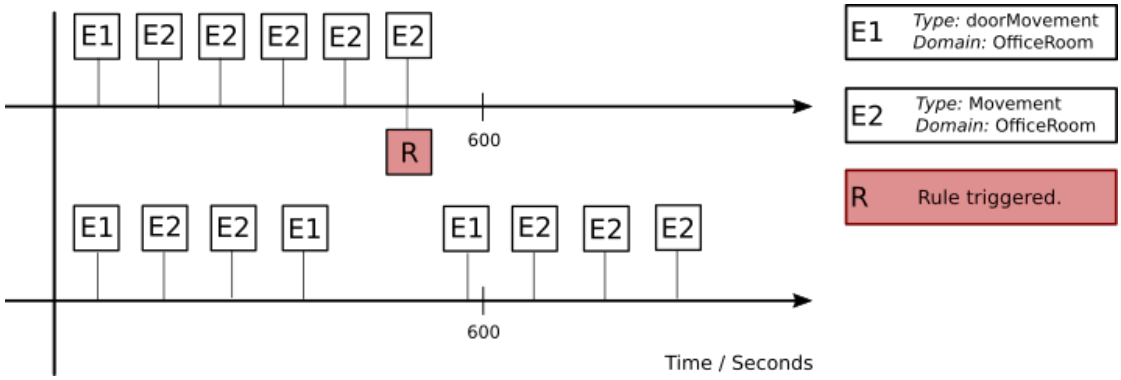


Figure 7.1: Two separate test sequences for evaluating use case optimized HVAC. Events E1 and E2 are fired as pictured on the timeline. The symbol R shows if and when the rule is triggered. The first sequence (upper timeline) signals room occupation, the second one (lower timeline) does not.

over 90 percent may trigger a proactive maintenance operation. Once again, the statements are converted into EPL statements. The result is depicted in Listing 7.4 and consists of two EPL statements.

```
1 /* Query */
2 CONDITION name="utilizationInPercent" AND AVG(value) >= 90 FROM
      TechnicalFloor WIN:TIME(600)
3
4 /* Rule */
5 Query TRIGGERS TechnicalFLoor, Warn_Message = "Keep an eye on your
      equipment"
```

Listing 7.3: Implementation of use case proactive maintenance of equipment.

```
1 /* EPL1 */
2 insert into AggregatedValue select avg(e.eventType.value) as value from
      Event.win:time(600) as e where e.eventType.name = '
      utilizationInPercent' AND e.domain.name = 'TechnicalFloor'
3
4 /* EPL2 */
5 select * from AggregatedValue where value >= 90
```

Listing 7.4: EPL statements for use case proactive maintenance of equipment.

Figure 7.2 shows the result of one test sequence, which was conducted for evaluation. During the sequence, multiple events are emitted to simulate changing workloads. As expected, every time the average value exceeds the threshold, a notification is triggered.

(iii) **Optimized lighting**
For the evaluation, an office room is considered, which contains two working places (desk A and desk B), each equipped with separated lighting. Desk A is located

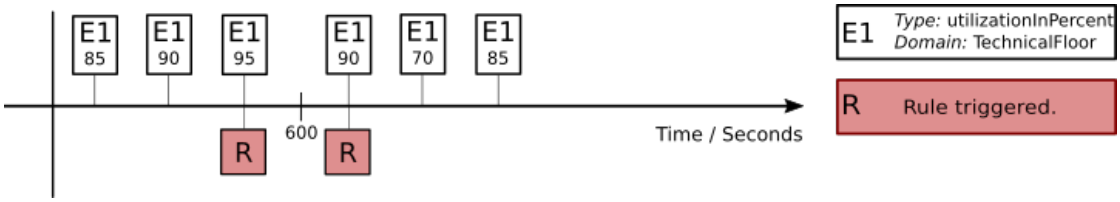


Figure 7.2: Test sequence for evaluating use case proactive maintenance of equipment. Event E1 is fired as pictured on the timeline. The symbol R shows if and when a rule is triggered.

next to the door. As a consequence every person with desk B as destination, has to pass desk A. The aim is to switch the lamp of each desk on or off corresponding if a person occupies or vacates one of these desks. Obviously, reasoning about occupation under these conditions is not a trivial task. Sequences of events have to be interpreted correctly, to avoid false positives. For example, both desks are occupied and a person leaves desk B. The task of the system is to detect that the person starts moving at desk B, passes desk A and closes the door. In a simple solution, every region (door, desk A, desk B) would judge their scope only based on single events. Thus, region A would produce a false positive due the movement in that region. Listing 7.5 offers an implementation for the framework to handle this situation without false positives.

```
1  /* Query1 */
2  CONDITION name="movement" FROM OfficeRoomAreaA
3
4  /* Query2 */
5  CONDITION name="movement" FROM OfficeRoomAreaB
6
7  /* Query3 */
8  CONDITION name="doorClosed" FROM OfficeRoom
9
10 /* Rule1 */
11 Query2 -> Query1 -> Query3 TRIGGERS Lightning, OfficeRoomAreaB, Off = 1
       WIN:TIME(10)
12
13 /* Rule2 */
14 Query1 -> Query3 TRIGGERS Lightning, OfficeRoomAreaA, Off = 1 WIN:TIME
       (10)
```

Listing 7.5: Implementation of use case optimized lighting.

The proposed solution contains two issues for discussion. Firstly, it is noteworthy that the same event for both movement queries (*Query1* and *Query2*) is used. The assignment and the differentiation are based entirely on the domain information, with the addition that a domain hierarchy (*OfficeRoomAreaA* and *OfficeRoomAreaB* are subsets of *OfficeRoom*) is applied. Secondly, the missing functionality to reason about the abstinence of events adds a minor flaw to the proposed solution. It is

evident that *Rule1* and *Rule2* have some relevant commonalities. To be precise, *Rule1* is a subset of *Rule2*. The workaround is to overrule *Rule1* if both events occur concurrently. The possibility to specify something like *NOT Query2 -> Query1 -> Query3 TRIGGERS ...* would be a superior solution, but logical operators are currently only supported on query level.
The statements are transformed into five EPL statements, which are depicted in Listing 7.6.

```
1  /* EPL1 */
2  insert into QueryEvent select 'Query1' as name from Event as e where e.
       eventType.name = 'movement' and e.domain.name = 'OfficeRoomAreaA'
3
4  /* EPL2 */
5  insert into QueryEvent select 'Query2' as name from Event as e where e.
       eventType.name = 'movement' and e.domain.name = 'OfficeRoomAreaB'
6
7  /* EPL3 */
8  insert into QueryEvent select 'Query3' as name from Event as e where e.
       eventType.name = 'doorClosed' and e.domain.name = 'OfficeRoom'
9
10 /* EPL4 */
11 select * from pattern [every (a=QueryEvent(name = 'Query2') -> b=
       QueryEvent(name = 'Query1') -> c=QueryEvent(name = 'Query3')) where
       timer:within(10 seconds)]
12
13 /* EPL5 */
14 select * from pattern [every (a=QueryEvent(name = 'Query1') -> b=
       QueryEvent(name = 'Query3')) where timer:within(10 seconds)]
```

Listing 7.6: EPL statements for use case optimized lighting.

The result of the practical evaluation is visualized in Figure 7.3. The rules are fired as expected, including the minor flaw.

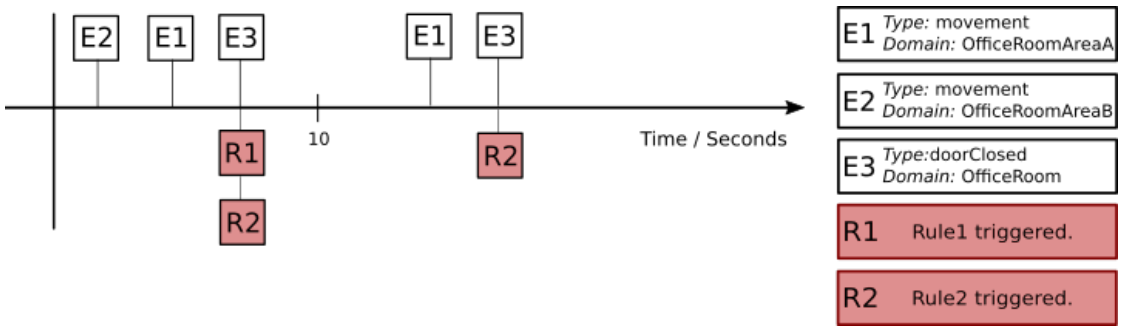


Figure 7.3: Test sequence for evaluating use case optimized lighting. Events E1, E2, and E3 are fired as pictured on the timeline. The symbols R1 and R2 show if and when a rule is triggered.

Increasing comfort was identified as second key factor (Section 4.2). Possible implementations of the concrete use cases are discussed in the following.

(i) **Elevator**
The aim is a smart elevator with the ability to anticipate near-future rides, by predicting passengers, their destinations, and special requirements of individual passengers. Adopting the behavior for passengers is depicted in Listing 7.7. The information *specialTreatment* is a property flag within the event to encounter persons with special needs.
The situation that a person needs special treatment is detected by the query. The query triggers a configuration change rule to set the opening time for elevators to 5 seconds. That rule affects all elevators, which are currently in the *EntranceFloor* domain.

```
1 /* Query */
2 CONDITION name="buildingEntered" AND specialTreatment = 1 FROM
      EntranceFloor
3
4 /* Rule */
5 Query TRIGGERS Elevator, EntranceFloor, KEEP_DOOR_OPEN_IN_MS = 5000
```

Listing 7.7: Implementation of use case elevator.

The corresponding EPL statements are illustrated in Listing 7.8.

```
1 /* EPL1 */
2 insert into QueryEvent select 'Query' as name from Event as e where e.
      eventType.name = 'buildingEntered' and e.eventType.specialtreatment
      = 1 AND e.domain.name = 'EntranceFloor'
```

Listing 7.8: EPL statements for use case elevator.

The graphical representation 7.4 illustrates the conducted test sequence. Worthwhile emphasizing is the demonstrated ability to extend events by custom properties.

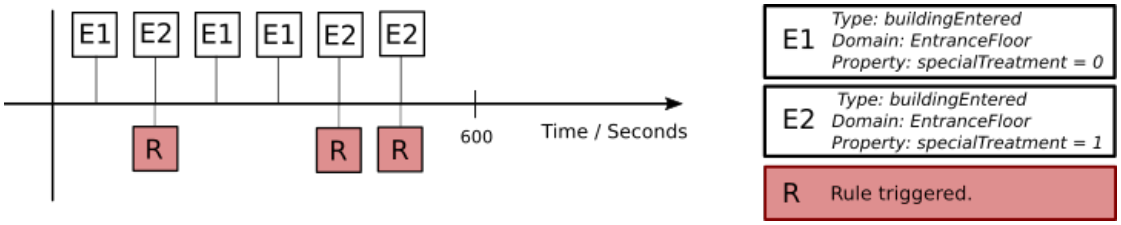


Figure 7.4: Test sequence for evaluating use case elevator. Events E1 and E2 are fired as pictured on the timeline. The symbol R shows if and when a rule is triggered.

(ii) **Flow meters**
The use case flow meters suggests to use the framework as remote control in order to determine the status values and working conditions of water supplying

equipment. Based on historical experience, maintenance personal should be able to give a rough estimation about the consumption of various consumers. This leads to an understanding of what is normal and what are indications of system failures. The basic statement structure of this use case is comparable to the use case proactive maintenance of equipment (Listing 7.9). Once again, an aggregation value is calculated over a specific event type with regard to a defined domain.

```
1 /* Query */
2 CONDITION AVG(waterUsageInLitre) > 100  FROM OfficeFloor WIN:TIME(3600)
3
4 /* Rule */
5 Query TRIGGERS Device, OfficeFloor, Warn_Message = "Keep an eye on your
      water consumption"
```

Listing 7.9: Implementation of use case flow meters.

Due to the similarity, the result of the EPL transformation and the evaluation of the use case can be looked up under the use case proactive maintenance of equipment, differing only in variables, constants, and values.
Finally, issues about the key factor improving safety and security are discussed (Section 4.3).

(i) **Fire protection**
Temperature and smoke sensors are applied to detect fires as fast as possible in the reference building (Listing 7.10). The query triggers fire alarm for a specific area if two conditions are satisfied within one minute: The measured temperature must be above a certain threshold and the smoke detector must trigger alarm. The temporal component is a one-minute long sliding window within both condition must occur.

```
1  /* Query1 */
2  CONDITION name = "Temperature" AND value >= 40 FROM OfficeFloor
3
4  /*Query2 */
5  CONDITION name = "SmokeDetector" FROM OfficeFloor
6
7  /* Rule1 */
8  Query1 -> Query2 TRIGGERS FireProtectionSensors, OfficeFloor, FireAlarm
       =1 WIN:TIME(60)
9
10 /* Rule2 */
11 Query2 -> Query1 TRIGGERS FireProtectionSensors, OfficeFloor, FireAlarm
       =1 WIN:TIME(60)
```

Listing 7.10: Implementation of use case fire protection.

The EPL transformation produces four statements depicted in Listing 7.11.

```
1  /* EPL1 */
2  insert into QueryEvent select 'Query1' as name from Event as e where e.
       eventType.name = 'Temperature' and e.eventType.value >= 40 and e.
       domain.name = 'OfficeFloor'
3
4  /* EPL2 */
5  insert into QueryEvent select 'Query2' as name from Event as e where e.
       eventType.name = 'SmokeDetector' and e.domain.name = 'OfficeFloor'
6
7  /* EPL3 */
8  select * from pattern [every (a=QueryEvent(name = 'Query1') -> b=
       QueryEvent(name = 'Query2')) where timer:within(60 seconds)]
9
10 /* EPL4 */
11 select * from pattern [every (a=QueryEvent(name = 'Query2') -> b=
       QueryEvent(name = 'Query1')) where timer:within(60 seconds)]
```

Listing 7.11: EPL statements for use case fire protection.

The interesting aspect of this use case is the demonstration that queries can be reused in multiple rules. The evidence is given in Figure 7.5, which documents the conducted test sequence.

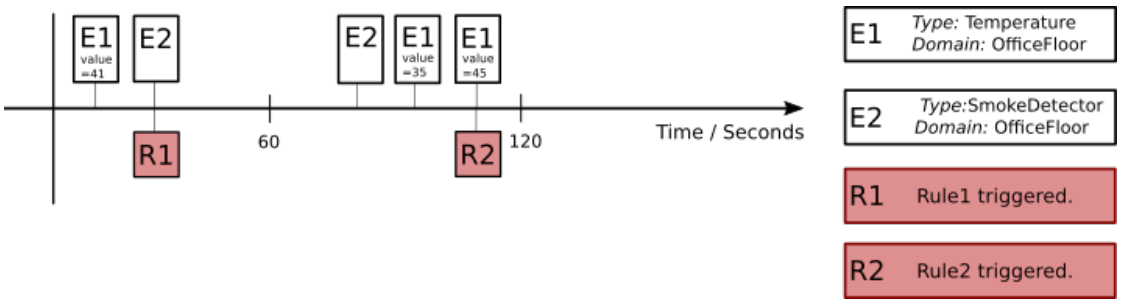


Figure 7.5: Test sequence for evaluating use case fire protection. Events E1 and E2 are fired as pictured on the timeline. The symbols R1 and R2 show if and when a rule is triggered.

(ii) **Advanced intrusion detection systems**
The proposed framework could help to identify complex attack patterns in real time. Denial of Service attacks could be prevented by defining threshold values for the number of permitted messages. If those values are exceeded, an alert can be triggered and the affected device can be temporarily deactivated. Listing 7.12 presents corresponding statements. From their construction and functional scope, they resemble use cases as already discussed. In particular, calculating an aggregated value and examining it, was a matter of discussion during use case proactive maintenance.

```
1 /* Query */
2 CONDITION count(Message) > threshold FROM OfficeFloor TIME:WIN 120
3
4 /* Rule */
5 Query TRIGGERS Message, OfficeFloor, Warning="DoS warning"
```

Listing 7.12: Implementation of use case advanced intrusion detection systems.

7.2 Quantitative evaluation

Before proceeding with the actual evaluation test cases, the next lines serve as short introduction. It presents the used evaluation setup and fosters a short debate about metrics.

7.2.1 Evaluation setup

The evaluation setup consists of four devices interconnected via Ethernet to form a local network. The devices are all different in their configuration and specification (hardware and software), which reflect the actual scenario assumable in the IoT world. Two of the four nodes are devices, which belong to the Raspberry Pi project [Fou16d], a series of credit card-sized single-board computers. In comparison with the other two components, their resources are fairly limited. The main objective of the Raspberry Pi project is to lower entrance hurdles for people who are interested in programming. All models of the project feature a Broadcom processor and are currently available in Model A or B, featuring small differences in RAM size and number of I/O ports. These machines share many restricted properties, as identified in the IoT context. For example, they are limited regarding storage and peripheral capabilities. The setup is complemented by two DELL devices, a desktop PC and a laptop. Table 7.1 summarizes all devices and

No	**OS**	**Typ**	**System Information**
R1	Raspbian (Debian)	Raspberry Pi 3 Model B	BCM 2837 64bit ARMv8 Cortex A53 QCore 1.2Ghz per core 1GB RAM
R2	Raspbian (Debian)	Raspberry Pi 2 Model B	BCM 2836 32bit ARMv7 Cortex A7 QCore 900Mhz per core 1GB RAM
L1	Ubuntu 14.04.5 LTS	Dell Laptop XPS L521X	Intel Core i7-3632QM 2.2Ghz per core 4GB RAM
PC1	Windows 10	Dell PC Optiplex 755	Intel Core Duo E4600 2.4Ghz per core 8 GB RAM

Table 7.1: Devices of evaluation setup

their characteristics. Particularly worth mentioning is the operating system Raspbian, because it is rather unknown compared to Windows and Ubuntu. Raspbian is Raspberry's official supported operating system. It is a Debian-based computer operating system with compilation settings adjusted to produce code that runs on the Raspberry Pi. The version pays special attention to the ARM architecture of the processor.
The topology of the network is depicted in Figure 7.6. All devices belong to the subnet 192.168.0.100/24. PC1 takes over the task of the CMU and simulates events emitted by DNs. L1, R1, and R2 are components, which host instances of EPNs. For the sake

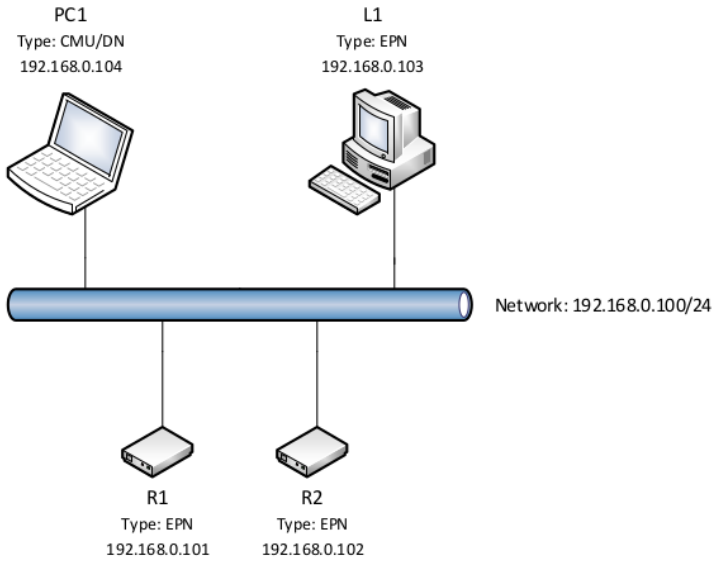


Figure 7.6: Network topology of evaluation setup

of automation and repeatability, Apache JMeter[1] is used to perform and simulate load behavior. As the name of the product indicates, the tool is entirely written in Java. It allows to prepare a sequence of HTTP requests and executes them multiple times with the possibility to steer dynamic properties and control timing. For HTTP processing, it provides a rich set of properties regarding header and cookie management. In contrast to a web browser, it does not render HTML pages or execute Javascript code found in HTML pages. Instead, all requests are recorded, including metrics like response time and size of the HTTP request [Fou16c].
For hardware monitoring the Java library called Operating System and Hardware Information (OSHI) is used. Compared to related libraries, OSHI does not require the installation of any additional library to retrieve system information, such as CPU load, memory usage, number of processors, and sensor data. It integrates native code through Java Native Access (JNA) and supports multiple platforms. It provides Java programs painless

[1] `http://jmeter.apache.org/`

access to native shared libraries without writing anything but Java code[2]. Listing 7.13 demonstrates the simple handling of OSHI for retrieving hardware information from native libraries.

```
1 SystemInfo systemInfo = new SystemInfo();
2 HardwareAbstractionLayer hadwareAbstractionLayer = systemInfo.getHardware();
3 CentralProcessor centralProcessor = hadwareAbstractionLayer.getProcessor();
4 GlobalMemory globalMemory = hadwareAbstractionLayer.getMemory();
5 double cpuLoad = centralProcessor.getSystemLoadAverage(3)[0] * 100);
6 double memUsage = globalMemory.getAvailable() * 100 / globalMemory.getTotal();
```

Listing 7.13: Retrieving hardware information by using OSHI.

7.2.2 Metrics

As with any other software, the framework performance depends strongly on the machine it is installed on. There are many different machine metrics, which are crucial for the software that runs on it. However, CPU load and memory usage are particular important as they give a reasonable impression of how well a machine is performing [Mol09].

System CPU Load

The CPU load of the entire system is reflected by the metric system CPU load, which is basically a snapshot value at a particular point in time. Short-term peaks can lead to erroneous conclusions, therefore an averaging over a period of time is preferable.
Listing 7.13 contains a method call named *getSystemLoadAverage* for retrieving the CPU load. Looking more closely reveals that the method returns load average numbers, as they are common in Unix-like systems. The values are measurements of the computational work the system is performing. In detail, the load average numbers are the sum of the number of runnable entities running on the available processors and the number of runnable entities queued to the available processors averaged over a period of time. The different periods are 1, 5, and 15 minutes, respectively. A completely idle machine has load average of zero. Each running processor either using or waiting for the CPU resource adds to the load average. The prototype uses the figure calculated over a period of 15 minutes to reason about CPU load.

System Memory Usage

Because all components of the prototype run inside the JVM, JVM memory and garbage collection are areas of special interest, when considering system memory usage. The heap space is of particularly importance for JVM applications as the heap space is the place, where Java objects reside in. Via the *new* operation heap space is consumed for newly created objects, while unused objects may be recovered via the process of garbage collection. The heap space is consumed at the time when the JVM starts up and may

[2] https://github.com/oshi/oshi

increase or decrease in size while the application runs. There are two parameters for setting the heap size: *-Xms:<size>* and *-Xmx:<size>*. The first, *-Xms:<size>*, sets the minimum heap size that is allocated during startup. The second, *-Xmx:<size>*, defines the maximum heap size that is available for the JVM. If heap usage exceeds this maximum, the JVM throws an out of memory exception and stops executing. The prototype uses following settings:

–Xmx: 512m
–Xms: 256m

As CPU load, heap memory describes a value at a specific point in time. For the purpose of evaluation, an adequate level of profiling is once again preferred. The JVM reserves memory as a single continuous block from the operating system. If different parameters for initial (-Xmx) and maximum (-Xms) heap space are defined, the allocated size of the JVM may change dynamically over time. So, monitoring memory on system level means that only expansions of the heap space are noticed. Minor changes induced by new objects are neglected, making the examination more independent of how JVM internals works (garbage collection). Finally, the values on system level for available memory and total memory are compared with following fraction:

$$memory\_usage = \frac{available\_memory}{used\_memory} \tag{7.1}$$

7.2.3 Rule Distribution

A core feature of the framework is the capability to distribute rules according specific strategies. The framework offers three built-in strategies based on CPU utilization (MinCPUUtilization), RAM utilization (MinRAMUtilization), and number of running rules (MinNumberOfActiveRules). Always the EPN having reached the minimum value should be selected to receive the next rule for processing. The test includes the evaluation of these individual strategies in two respects: Firstly, the evaluation concentrates solely on the distribution behavior without any interference with events, which may influence the performance of the EPN instances. This mode is referred to as the *distribution without load* mode. The second round is carried out under continuous flow of events and is termed *distribution with load* mode. Altogether, the test case described in the following is performed six times, once for each combination of built-in distribution strategy and mode.
The basic test structure consists of three REST calls, performing the activities query registration, rule registration, and rule activation (Listing 7.14). The used statements serve only testing purposes and are not meaningful in practice. The query detects events of type “Event“ with values over one hundred. Also for the rule a simple format was selected. As soon as the query is matched, the rule is triggered, whereby the trigger actions are of no importance for this particular test case, therefore the triple “Event, Domain, Setting=1“ does not trigger any particular downstream actions and acts merely as placeholder. The third statement activates the rule and induces the CMU to distribute

the rule according the strategy. The strategy is passed as parameter to the system. The values 1,2, and 3 are reserved for the strategies. The placeholder *counter_value* is filled by JMeter.

```
1  /* Query registration
2  *  CONDITION name='Event' AND value>100
3  */
4  POST /registrations/query/Query${counter_value}
5
6  /* Rule registration
7  *  Query${counter_value} TRIGGERS Event, Domain, Setting=2
8  */
9  POST /registrations/rule/Rule${counter_value}
10
11 /* Rule activation */
12 POST /activations/rule/Rule${counter_value}/1
```

Listing 7.14: Rest calls for test rule distribution

A test run is comprised of 90 cycles, each time carrying out the basic test structure in roughly 100 seconds. This means that approximately every second a new rule is registered at the system and should be distributed over all available EPNs. At the beginning of each test run, all involved components are already started. All participating EPNs are registered at the CMU and ready for receiving rules. The components are in a stabilized state so that the average measurements are not affected by previous computation loads. For load simulation, JMeter is used to generate events. Listing 7.15 depicts the structure of the test event with which the system is constantly fed. Its value is over 100, so that every single event fires the test query.

```
1  <common.data.model.DeviceData>
2     <id>-1</id>
3     <name>default</name>
4     <sensorData>
5        <rawValue class="int">123</rawValue>
6     </sensorData>
7     <device>
8        <id>-1</id>
9        <name>event</name>
10    </device>
11 </common.data.model.DeviceData>
```

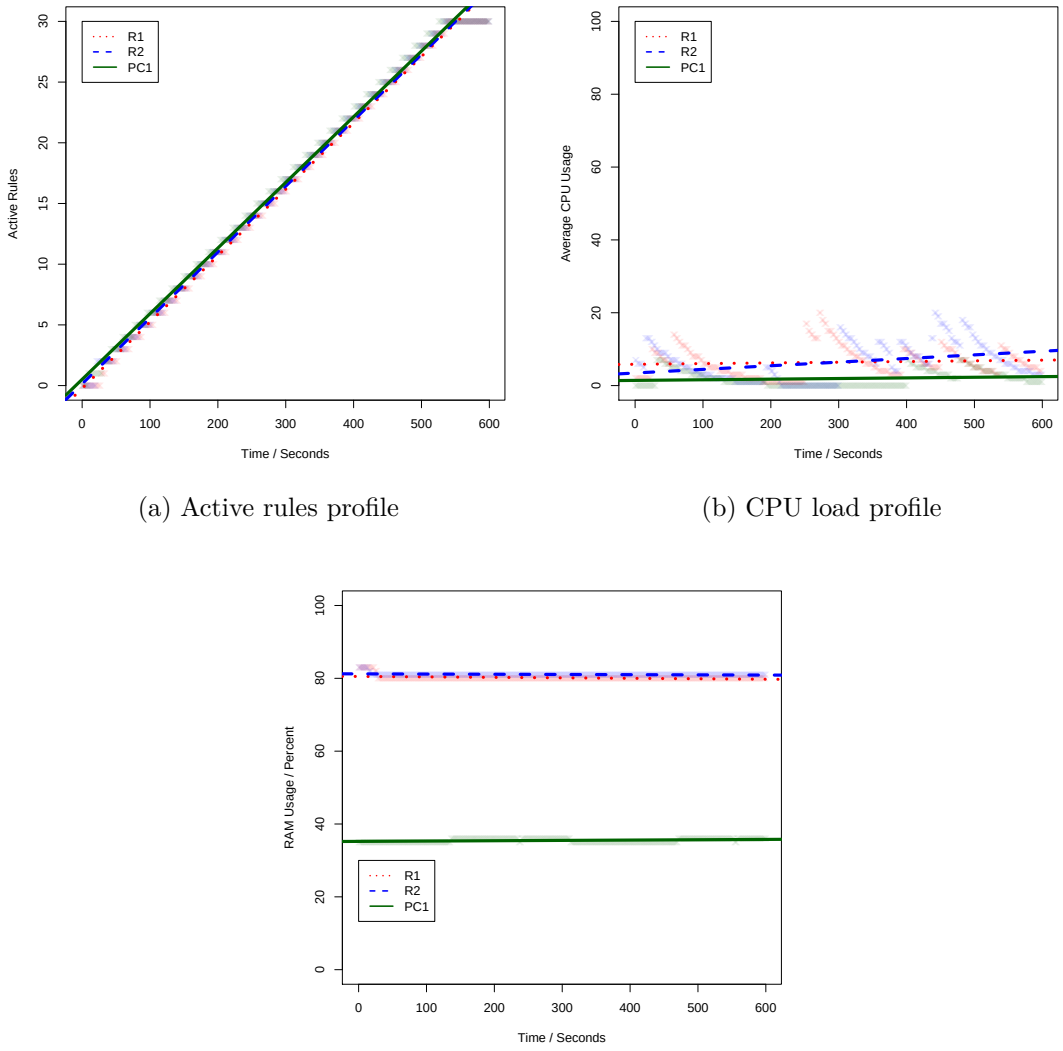
Listing 7.15: Test event

In the following, the results of the individual test runs are discussed. All depicted figures follow the same structure and use linear regression to describe the measurement data. In simple linear regression, scores are predicted on one variable from the scores on a second variable. The predicting variable is always depicted on the y-axis and is named criterion variable. The second variable on which the prediction is based on is called the predictor and is located on the x-axis. The following figures use always the physical

unit time measured in seconds as predictor. The results are single lines defined by the linear formula $y = c + b * x$ fitting through a cloud of data points. For purposes of clear presentation, measured data points are pictured in a transparent manner, while the regression is at the forefront.

MinNumberOfActiveRules - Without Load

Figure 7.7 depicts the distribution using the strategy *MinNumberOfActiveRules*. As



(a) Active rules profile

(b) CPU load profile

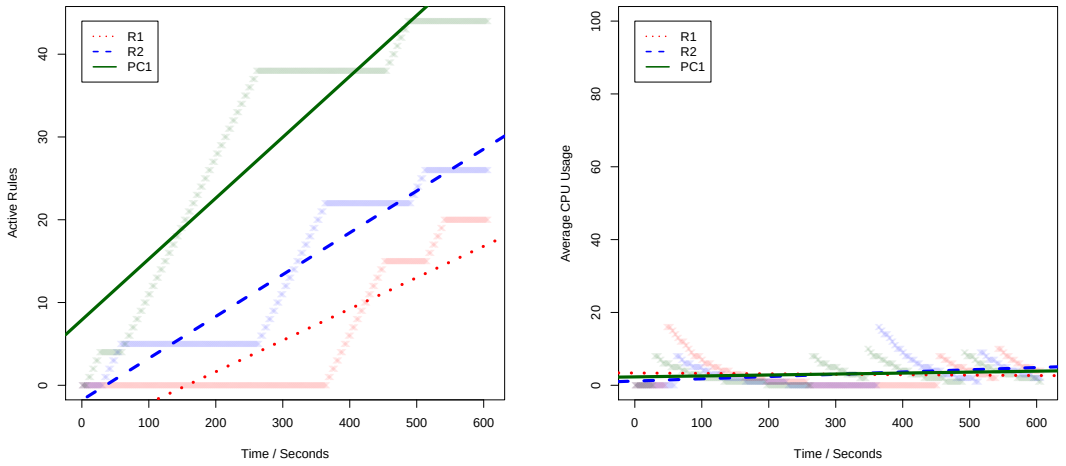
(c) RAM load profile

Figure 7.7: Results of distributing 90 rules using the strategy MinNumberOfActiveRules without load.

expected, the number of rules (Figure 7.7a) is divided equally among all three EPNs (R1, R2, PC1). The CPU utilization (Figure 7.7b) shows that the resource constrained devices are slightly affected by the growing number of registered rules, whereas PC1 shows no conspicuities. The 30 registered rules are computed effortless and without perceptible rise of CPU load. The figure regarding RAM utilization (Figure 7.7c) presents a constant RAM utilization for all devices. This means that all memory resources needed to handle the registration process are already allocated at startup time of the JVM and no additional memory is demanded. The same behavior is determined for the remaining test runs and therefore the figures for RAM utilization are waived in the following discussions, because they do not provide any new information.

MinCPUUtilization - Without Load

Figure 7.7 depicts the distribution using the strategy *MinCPUUtilization*. The spreading of rules (Figure 7.8a) demonstrates clearly that the evenness does no longer exist. PC1 receives approximately twice as many rules as the resource constrained devices, while having the same CPU utilization (Figure 7.8b).



(a) Active rules profile

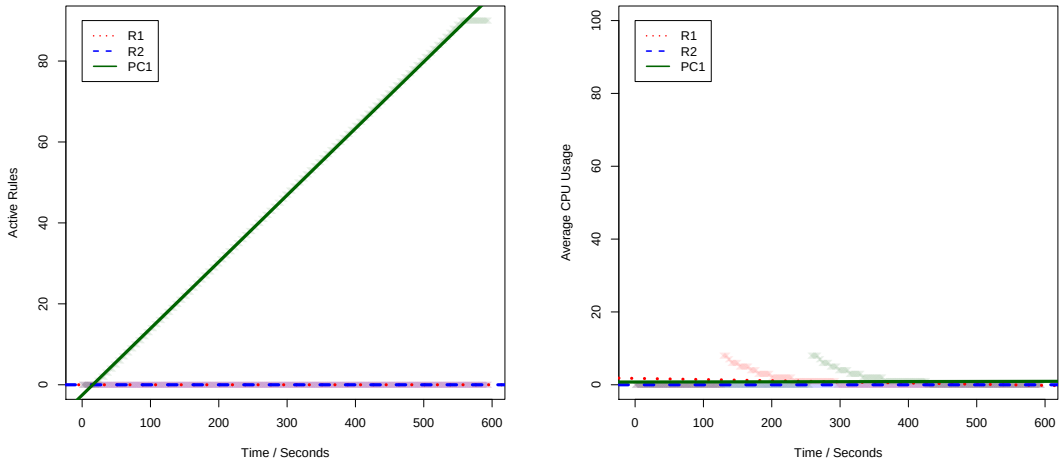
(b) CPU load profile

Figure 7.8: Results of distributing 90 rules using the strategy MinCPUUtilization without load.

MinRAMUtilization - Without Load

The result of the strategy MinRAMUtilization is depicted in Figure 7.9. As already discussed, the selected test setup does not lead to an expansion of the JVM heap space. This is why a constant RAM utilization for all three EPNs is measured. In absolute numbers, PC1 has the smallest percentage of RAM utilization and obtains all rules

exactly as expected (Figure 7.9a). The outcome for the CPU load (Figure 7.9b) is not surprising, while R1 and R2 remain in idle state due to lacking work, PC1 processes the rules without notable effort.



(a) Active rules profile

(b) CPU load profile

Figure 7.9: Results of distributing 90 rules using the strategy MinRAMUtilization without load.

MinNumberOfActiveRules - With Load

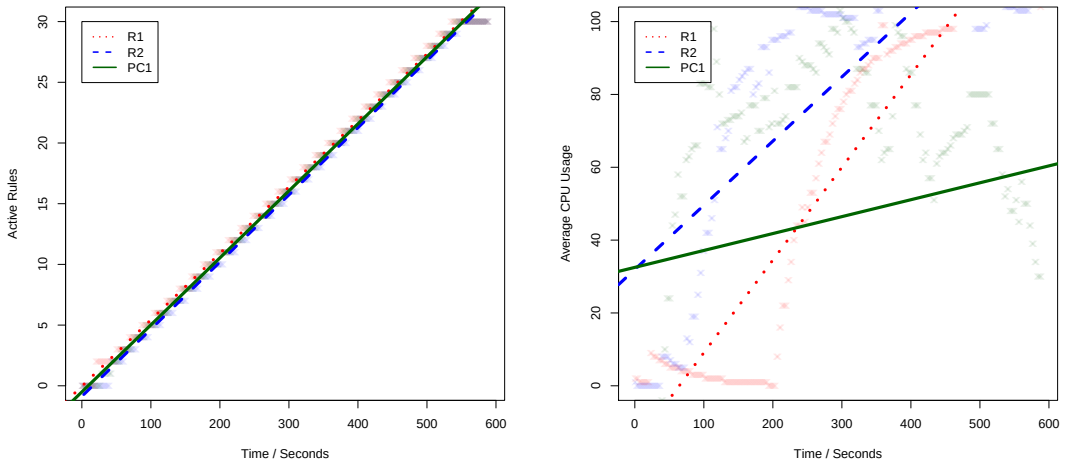
Figure 7.10a exactly meets the specification by distributing all rules equally among the three EPNs. The CPU load profile (Figure 7.10b) shows that the slope of the lines is indirect proportional to the available means: While the resource constrained EPNs are fully engaged, PC1 merely reaches about fifty percent of its CPU utilization.

MinCPUUtilization - With Load

The significant performance advantage of PC1 compared to R1 and R2 is clearly evident in Figure 7.11a. It handles more than two thirds of all rules under approximately the same CPU load (Figure 7.11a). The regression lines have approximately the same slope, indicating that their utilization is equal according their resource proportions.

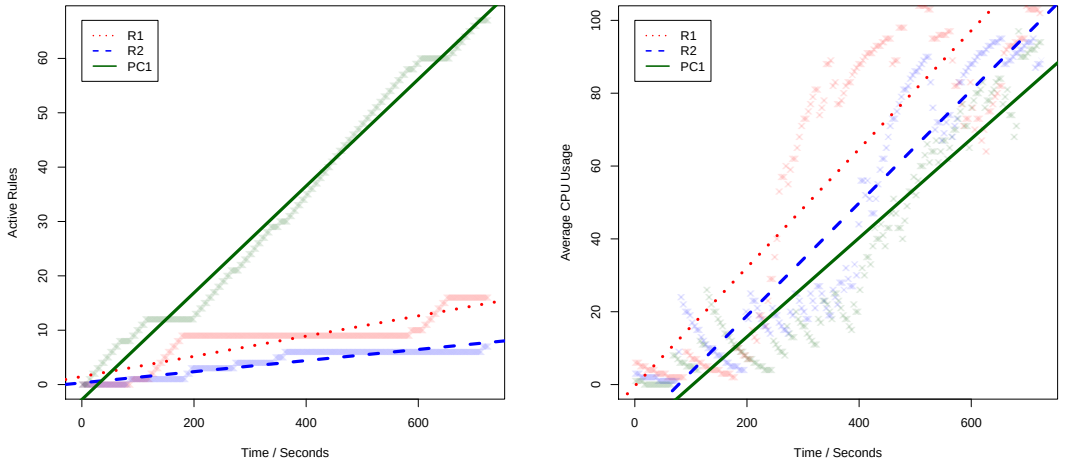
MinRAMUtilization - With Load

The distribution of rules (Figure 7.12a) corresponds exactly to the result from the test run without load. All rules are assigned to PC1, because it retains the lowest RAM utilization throughout the entire test run. Figure 7.12b shows also a similar picture for



(a) Active rules profile (b) CPU load profile

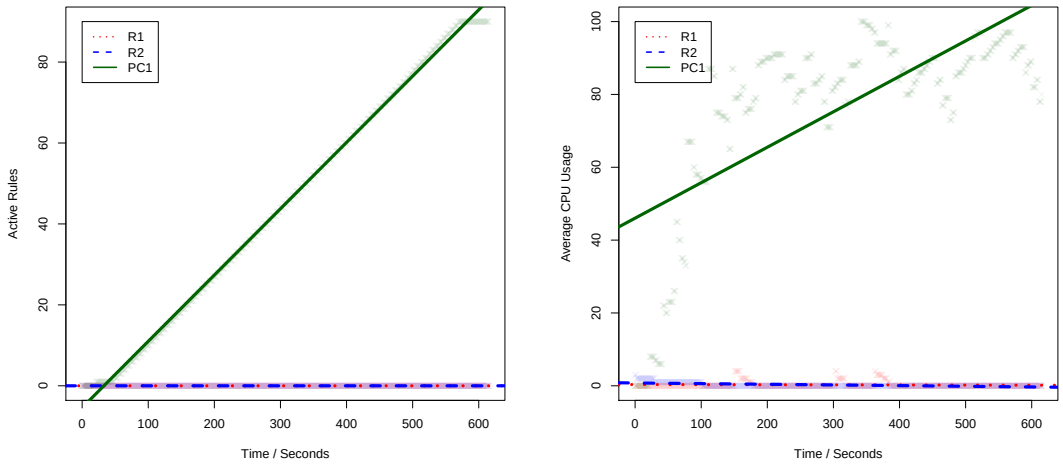
Figure 7.10: Results of distributing 90 rules using the strategy MinNumberOfActiveRules with load.



(a) Active rules profile (b) CPU load profile

Figure 7.11: Results of distributing 90 rules using the strategy MinCPUUtilization with load.

the CPU utilization, however due the increased computation load the line for PC1 moves upwards.



(a) Active rules profile

(b) CPU load profile

Figure 7.12: Results of distributing 90 rules using the strategy MinRAMUtilization with load.

Conclusion

The analysis above shows that the framework allows distributing rules based on different strategies. Consequently, the framework is suited to balance system load among available resources in a manner that improves overall system performance and maximizes equal resource utilization.

7.2.4 Failover mechanism

The failover mechanism of the framework guarantees that activated rules are always executed on healthy nodes. Thereby, a node is assumed to be healthy as long as it sends heartbeat messages.
In the following this mechanism is evaluated in practice. For that reason, exactly one rule is installed on the evaluation system. The framework selects an EPN and takes care that the rule is installed on it. All steps performed so far comply to the normal behavior of the framework. Subsequently to this, this test run wants to simulate the failover process by switching off the EPN with the test rule running on it. The shutdown is carried out without preannouncement and rather abruptly by pulling out the power cord. The expected reaction should be that the breakdown of the EPN is detected and eventually the failover mechanism is provoked.
Besides the fundamental assessment of the functionality, a quantitative evaluation is conducted. It should be examined whether the failover mechanism works within reasonable time limits. The time span between last occurrence of the rule on the faulty node and first occurrence of the rule on the healthy is measured. The rule is periodically provoked

by emitting the test event every second. Clearly, the measurement result is significantly influenced by the heartbeat message period. For the test run the interval is set to 5000 milliseconds. Table 7.2 sums up the result of the test runs. On average, it takes

No	Faulty node	Healthy node	Duration in ms
1	R1	R2	6567
2	R1	R1	4352
3	R1	PC1	6874
4	R1	R1	5608
5	R1	PC1	5574
6	R1	PC1	5553
7	R1	R1	8892
8	R1	PC1	6937
9	R1	R1	6517
10	R1	PC1	7566

Table 7.2: Measurement result of failover mechanism

approximately 6.5 seconds till a rule is successfully redistributed. The concrete figure is not particular meaningful, because it strongly depends from various configuration properties like the delay between two heartbeat messages and the overall structure of the corresponding rule. Nonetheless, it shows that the redistribution can be performed in reasonable timeframes when necessary.

7.2.5 Registration process

A feature of the framework is the ability to reinitiate the registration handshake, in case it does not succeed promptly. The nodes start the registration process after a configurable delay again, whereby the time interval is consequently increased with every unsuccessful attempt.
In the following a test scenario wants to quantify the advanced registration process by analyzing the dynamic behavior. Both types of nodes (DN and EPN) share the same implementation regarding registration, hence it does not matter which type is chosen. For conducting the test, one instance of EPN is picked and started without running a CMU. The recorded registration attempts are depicted in Figure 7.13. As in the specification stated, the time interval is consequently increased with every unsuccessfully attempt. This approach guarantees that a first failed attempt will be tried again in a reasonable short time, while a fundamental problem burdens the system minimally.

7.2.6 Dynamic change of DN configuration

The use case discussion in the first part of this chapter generally stops at the point the rule is matched, because further processing was not primarily in the focus of the investigation. As reaction either some sort of notification message was specified (“Keep an eye on your equipment“, “DoS warning“) or a general system setting was

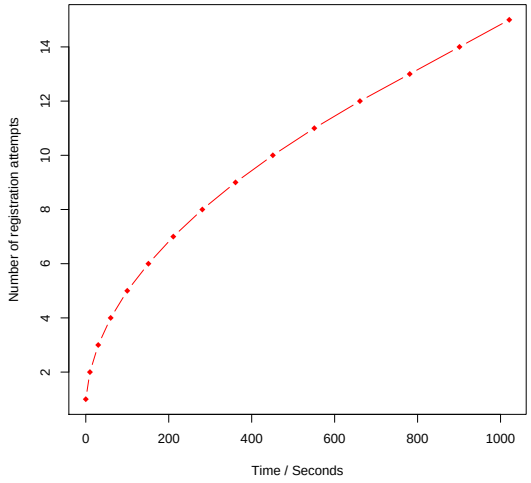


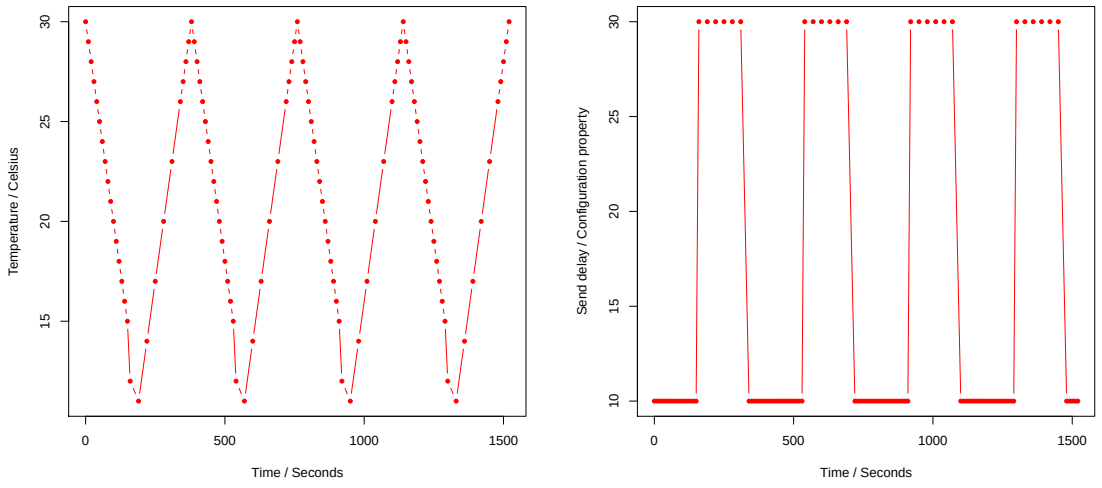
Figure 7.13: Time behavior of registration attempts

set (“KEEP_DOOR_OPEN_IN_MS = 5000“, “ON_OFF = 1“), assuming that these reactions are processed by superordinate instances (machines or human beings).
The design of the framework enables dynamic configuration of DN instances based on reactions. Just as other features of the framework were evaluated, this aspect should be assessed with the help of a scenario, where the tactic “Manage Sample Rate“ [BCK12a] is applied. The overall volume of data should be minimized in order to increase performance and scalability. This scenario employs configurable DNs to control the frequency events are emitted. For that reason, two different working modes are introduced, called normal respective verbose. The main difference is the frequency at which events are transmitted. While the normal mode reduces bandwidth consumption by using longer periods, the verbose mode sends events with a much higher frequency to inform the system about critical conditions in order to achieve shorter response times. The switching between the two modes should be controlled from outside.
The evaluation setup described in Section 7.2.1 is slightly adopted for this purpose. R2 does no longer act as EPN, but instead it hosts an instance of EPN and simulates sensor values. These values are changed every second, commuting constantly between two border values.
The statements deployed on an arbitrary EPN node are depicted in Listing 7.16. Their role is to monitor sensor values regarding their ranges. If they exceed (Query1) or fall below a certain limit (Query2), they should trigger corresponding configuration changes. In our scenario, the sensor values are temperature measurements. Above a certain limit (25 degree Celsius), the overall system enters a critical state and sensor values should be emitted at shorter intervals. On the other side, if sensor values fall below a certain limit (15 degree Celsius), the frequency is reduced in order to discharge the system and cut the overall data volume.

```
1  /* Query1: detectTemperatureAbove25 */
2  CONDITION name = "Temperature" AND value >= 25 FROM StorageFloor
3
4  /* Query2: detectTemperatureUnder15 */
5  CONDITION name = "Temperature" AND value <= 15 FROM StorageFloor
6
7  /* Rule1 */
8  detectTemperatureUnder15 -> detectTemperatureAbove25 TRIGGERS Temperature,
       StorageFloor, TASK_INTERVAL_S = 10
9
10 /* Rule2 */
11 detectTemperatureAbove25 -> detectTemperatureUnder15 TRIGGERS Temperature,
       StorageFloor, TASK_INTERVAL_S = 30
```

Listing 7.16: Implementation for dynamic change of DN configuration.

The results of the evaluation run are depicted in Figure 7.14. As mentioned above the DN supplies temperature values, which oscillate between 10 and 30 degree of Celsius. Figure 7.14a illustrates that recording, whereby the period between two values is relevant. It can be observed a behavior exactly as expected: At the time the value exceeds the upper limit, the DN switches to verbose mode and sends events at higher rate. On the other side, the DN changes the mode back to normal as soon as the values are under the lower limit. Figure 7.14b shows the course of the configuration property *send delay*. It defines the delay between two event transmissions. The rising edge (10 -> 30) is exactly the time *Rule2* hits, respectively the falling edge (30 -> 10) the time *Rule1* triggers.



(a) Sensor values

(b) Profile of configuration property *send delay*

Figure 7.14: Dynamic change of DN configuration

CHAPTER 8

Conclusion & future work

8.1 Conclusion

IoT promises a fully interconnected world, where objects and people are tightly intertwined. While the potential of new applications and gadgets is enormous, the rise of IoT also yields technology challenges that could stand in the way of realizing its full value. Particularly in the areas of security, privacy, interoperability, and standards, research works are needed to meet the challenges accompanying the vision of IoT.
This thesis has addressed the last two mentioned areas, and it has proposed a framework that reduces the time and effort in developing IoT applications. In particular, this work has defined a distributed solution for applications that apply CEP as their main tool for detecting meaningful patterns. In comparison to centralized approaches, the distributed architecture of the framework is better suited to address performance and scalability properties appearing in scenarios, where large numbers of events are generated from geographically widespread sources. The framework follows a modular approach and is composed of three main components, namely Configuration Management Unit (CMU), Event Processing Node (EPN), and Device Node (DN). Each component is entrusted with a special task and represents an autonomous unit, which can be deployed independently. CMU acts as data supplier, EPN incorporates the CEP, and CMU represents the central registry and agent between the two other components. The number of nodes (EPN and DN) may vary, depending on the requirements the application on top of the framework is currently facing. Additional nodes are added to the system dynamically. Especially, the expansion of EPN instances should disburden system load and facilitates scalability.
The usefulness of IoT frameworks is also based on the possibility of interconnecting a broad number of devices belonging to a heterogeneous set of technologies. When it comes to implementing such heterogeneous scenarios, several issues concerning the interactions among those elements arise, which the framework has addressed with a comprehensive communication model and well-defined APIs. For the top-level abstraction,

a predominate part of the communication flow follows the REST architecture style in combination with HTTP. The assumption that this choice of technology is easier to use and more flexible compared to the WS* specification and SOAP was confirmed during conducting the evaluation. JMeter, the test tool, could be hooked in with little expense and effort, exactly as desired for good interoperability. The second communication type, the publish-subscribe pattern, is applied for the exchange of events between EPNs and DNs. A typical IoT application will comprise a large number of DN instances emitting events at an even higher rate, such that a N-to-N approach in a sort of server-client pattern does not scale for this communication path.
Another challenge regarding interoperability is that every event exchanged by any node, must be interpretable and understandable by every other node within the application. The framework addresses this issue by applying a lightweight data & event model that incorporates the basic characteristics of IoT data. It provides a basic structure for events and is extendable by custom properties to consider individual requirements. Experiences during the prototype implementation show that the structure could be adapted without great expenditure.
A prototype was implemented to demonstrate the framework's feasibility. In this context, Spring Boot has proved to be a very great tool for establishing a production-grade application. A number of low level tasks (data mapping, data serialization, etc.) were only a matter of configuration and substantially shortening realization time. In general, the implementation of the prototype attempts to reuse structures in form of patterns, which have been developed and proved over years. For this reason, the chain of responsibility design pattern was applied to implement parts of the logic triggered by API calls. The advantage of this pattern lies in the enhanced reusability. For example, components for validation purposes are repeatedly used among multiple API methods.
The openness towards different event processing engines was another goal of the framework. With the help of the design pattern abstract factory, an interface has been build that makes the framework greatest possible independent on how its engines are initialized and employed. For the prototype, Esper was selected as CEP engine. The integration was straightforward, merely the transformation between our language and EPL meant some implementation effort. However, the overall conclusion is that the integration of further CEP engines should be possible without great problems and within a reasonable period of time.
Besides the distributed fashion of the framework for supporting scalability and other requirements encountered in the IoT context, the user-friendly language is a main contribution of this thesis. It allows less experienced users to express uses cases in an intuitive way. The separation of the language into a query and rule part has been proved as useful in two respects: Firstly, it gives the user the most succinct notation for expressing patterns of events (queries) and configuration changes (rules). The advantage of a succinct notation is that it lets users express patterns correctly and judge instantly that it expresses what was intended. During evaluation, statements and their corresponding equivalences translated into EPL were presented. The comparison demonstrates that both a qualitative (complexity) and quantitative (number of statements) improvement

was achieved. In conclusion, the syntax of the custom language is more straightforward and target-oriented compared to the native language provided by Esper. For sake of completeness, it should be mentioned that the succinct notation comes not for free and results in limited expressiveness. Secondly, queries are reusable among different rules. In general, reusability is a great mantra in software engineering, because it leads to greater productivity, easier maintenance and better stability.
Another notable fact of the query language is that the domain information is treated like a first-class citizen. That means, the basic structure of the query contains an autonomous part for defining provenance information, syntactically reassembled by the "FROM" clause. An alternative approach could have been to treat spatial information like any other property. The chosen approach has been proven useful in the proposed reference building. Different floors or types of rooms are stated within the query in an intuitive and succinct way. Another important feature is the proficiency to reason about causal dependencies among events. That language feature was leveraged in all use cases, which were proposed in connection with the reference building.
The results of the evaluation part confirm the functional capabilities of the proposed framework. Apart from one small exception, all test cases from the use case scenario have been satisfied to the full contentment of the assignment. The exception will be discussed in Section 8.2. Besides testing the functional scope, so called quantitative evaluations have been conducted to underpin the feasibility of certain features. Rule distribution, the feature distributed applications are most interested in, has been tested most extensively. The results show that the framework is suited to balance system load among available resources in a manner that improves overall system performance and maximizes equal resource utilization. Another evaluation focuses on the redistribution capabilities of the framework. Redistribution describes the capacity to detect faulty nodes and redistribute rules running on them, so guaranteeing that installed rules always run on healthy nodes. The outcome confirms that redistribution can be performed in reasonable timeframes. Finally, capabilities towards a self-reconfigurable framework were proven. For this purpose, a test scenario was created, in which devices adjust their event emitting frequency as direct response to triggered rules.

8.2 Future work

Even though the framework's suitability has been demonstrated by implementing use cases in the domain of smart buildings, some new challenges can be envisaged. A list of extensions and improvements have been identified as future work, whereby only the topic security is a real show stopper toward practical usage.

Extension of security measures

In fact everybody who knows how to access the APIs can retrieve all relevant information. All considerations regarding security were not part of this thesis. As stated at the

beginning of this chapter, security is another major challenge that must be considered in all its facets. Main topics of interest are authentication, authorization, encryption, privacy concerns, and much more.

Extension of language expressiveness

The evaluation shows that the expression strength of the language covers a wide range of use cases, but still leaves room for improvement. The ability to reason about the absence of events would be a desirable feature and would increase the universal applicability of the framework. While the query part of the language allows to define the absence of properties, there is currently no equivalent on event level. It is presumable that in practice the ask for the absence of events will be just as important as the detection of the presence of events. In the particular case, a workaround was applied, but this is not entirely satisfactory in practice. The overall goal here should be to achieve a language, which has a good balance between simplicity and expressiveness power, and minimizes ambiguities and misunderstandings.

Improvement of language syntax

For identification of events the language uses the specific property *name*. An ordinary query therefore reads as follows:

```
1 CONDITION name=Event
```

The indention of the user is plain to see. A slight improvement of the syntax could be achieved by omitting the property information. The result would look like as stated below:

```
1 CONDITION Event
```

The outlined improvement is just a minor adjustment, which could be classified as syntax sugar. It should serve as indicative value that further efforts could make the language even more easier to read and to express.

Providing tools for rule and query management

The management of rules and queries for less technically experienced users could be simplified by providing a visualized editor. The advantage would be that users could add and remove statements without writing them textual. Drag and drop techniques could further support users by creating queries and rules.

Extension of evaluation measures

Besides usability improvements, the distribution analysis discussed in Chapter 7 could be intensified to underpin the results in a broader context. The evaluation structure used in this thesis was suitable for proving distribution capabilities, but the setup should

be classified as rather simple regarding various parameters like number of participating devices and number of exchanged events. A more comprehensive evaluation with a broader setup consisting of a wide variety of devices could intensify the results and testify the usefulness of the publish-subscriber pattern between DNs and EPNs under high load.

Extension of distribution strategies

Other distribution strategies that investigate metrics beyond CPU utilization, RAM utilization, and number of active rules may be of interest. The three built-in strategies make their decisions based on measurements captured in the past. For example, machine learning procedures could help to predict future utilization, by analyzing the application in more intense and classifying new rules according their complexity.

Replication of CMU

The CMU acts as central registry agent between the two other types of components. It holds crucial information such as registered components, and provides a basis for functionalities like rule distribution and configuration change propagation. Currently, the framework includes no instruments for scaling out the CMU component. This is because the CMU instance is not identified as bottleneck in the first place. From the technical point of view, scaling out would mean to replicate all information to further instances.

List of Figures

List of Tables

Listings

Acronyms

6LoWPAN IPv6 over Low Power Wireless Personal Areas Network. 6

AMWR Adaptive Moving Window Regression. 22

ANTLR ANother Tool for Language Recognition. 60

BNF Backus Naur Form. 39

CEP Complex Event Processing. 2, 5

CoAP Constrained Application Protocol. 6

CoR Chain of Responsibility. 50, 51

DAO Data Access Object. 49

DI Dependency Injection. 48

DSBMSs Database Management Systems. 12

DSMS Data Stream Management System. 19

EDA Event Driven Architecture. 9

EJB Enterprise JavaBeans. 47

EPC Electronic Product Code. 22

EPL Event Processing Language. 14

ESP Event Stream Processing. 10

GWT Google Web Toolkit. 51

IFP Information Flow Processing. 19

IoC Inversion Of Control. 47

IoT Internet of Things. 1, 5

IP Internet Protocol. 6

IPSO IP over Small Objects. 6

JDBC Java Database Connectivity. 49

JMS Java Messaging Service. 53

JNA Java Native Access. 78

JSON JavaScript Object Notation. 50

JVM Java Virtual Machine. 66

NFA Nondeterministic Finite Automaton. 19

NFC Near-Field Communication. 6

ORM Object-relational mapping. 49

OSHI Operating System and Hardware Information. 78

RFID Radio Frequency IDentification. 5, 6

SEP Stream Event Processing. 9

SNMP MIB Simple Networking Management Protocol Management Information Base. 5

SOA Service Oriented Architecture. 8

SQL Structured Query Language. 11

WSAN Wireless Sensor & Actuator Networks. 6

XML Extensible Markup Language. 55

XSD XML Schema Definition. 56

Bibliography

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[ACMZ15] A. Akbar, F. Carrez, K. Moessner, and A. Zoha. Predicting complex events for pro-active iot applications. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 327–332, Dec 2015.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[Ash09] Kevin Ashton. That 'internet of things' thing. *RFiD Journal*, 22(7):97–114, 2009.

[ATR16] Mohammed Riyadh Abdmeziem, Djamel Tandjaoui, and Imed Romdhani. Architecting the internet of things: state of the art. In *Robots and Sensor Clouds*, pages 55–75. Springer, 2016.

[BCK12a] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[BCK12b] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[BDG+07] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, New York, NY, USA, 2007. ACM.

[BWDW13] Payam Barnaghi, Wei Wang, Lijun Dong, and Chonggang Wang. A linked-data model for semantic sensor streams. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 468–475. IEEE, 2013.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[CFS+14] C. Y. Chen, J. H. Fu, T. Sung, P. F. Wang, E. Jou, and M. W. Feng. Complex event processing for the internet of things and its applications. In *Automation Science and Engineering (CASE), 2014 IEEE International Conference on*, pages 1144–1149, Aug 2014.

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[Cor16a] Coral8. Complex event processing: Ten design patterns. `http://complexevents.com/wp-content/uploads/2007/04/Coral8DesignPatterns.pdf`, Accessed: 08/2016.

[Cor16b] Oracle Corporation. Enterprise javabeans technology. `http://www.oracle.com/technetwork/java/javaee/ejb/index.html`, Accessed: 10/2016.

[DZJ02] AiLing Ding, XiangMo Zhao, and LiCheng Jiao. Traffic flow time series prediction based on statistics learning theory. In *Intelligent Transportation Systems, 2002. Proceedings. The IEEE 5th International Conference on*, pages 727–730, 2002.

[EFGK03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[FJL+01] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30(2):115–126, May 2001.

[Fla05] David Flanagan. *Java in a Nutshell*. " O'Reilly Media, Inc.", 2005.

[Fou16a] Apache Software Foundation. `http://activemq.apache.org/`, Accessed: 10/2016.

[Fou16b] Apache Software Foundation. `http://activemq.apache.org/how-do-i-embed-a-broker-inside-a-connection.html`, Accessed: 10/2016.

[Fou16c] The Apache Foundation. Apache jmeter. http://jmeter.apache.org/, Accessed: 11/2016.

[Fou16d] The Raspberry Pi Foundation. Raspberry pi - teach, learn and make with raspberry pi. https://www.raspberrypi.org/, Accessed: 11/2016.

[Fow16] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html, Accessed: 10/2016.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GRF15] B. Gaunitz, M. Roth, and B. Franczyk. Dynamic and scalable real-time analytics in logistics combining apache storm with complex event processing for enabling new business models in logistics. In *Evaluation of Novel Approaches to Software Engineering (ENASE), 2015 International Conference on*, pages 289–294, April 2015.

[Grö11] Marko Grönroos. *Book of Vaadin*. Lulu.com, 2011.

[Grö16] Marko Grönroos. Model-view-presenter pattern with vaadin. https://vaadin.com/web/magi/home/-/blogs/model-view-presenter-pattern-with-vaadin, Accessed: 10/2016.

[GSJM14] Nithyashri Govindarajan, Yogesh Simmhan, Nitin Jamadagni, and Prasant Misra. Event processing across edge and the cloud for internet of things applications. In *Proceedings of the 20th International Conference on Management of Data*, COMAD '14, pages 101–104, Mumbai, India, India, 2014. Computer Society of India.

[HM11] H. Hada and J. Mitsugi. Epc based internet of things architecture. In *RFID-Technologies and Applications (RFID-TA), 2011 IEEE International Conference on*, pages 527–532, Sept 2011.

[Inc16a] EsperTech Inc. About esper and nesper. http://www.espertech.com/esper/, Accessed: 08/2016.

[Inc16b] EsperTech Inc. Tutorial. http://www.espertech.com/esper/tutorial.php, Accessed: 08/2016.

[IV15] A. Ilapakurti and C. Vuppalapati. Building an iot framework for connected dairy. In *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, pages 275–285, March 2015.

[jbo16] jboss.org. Chapter 8. complex event processing. https://docs.jboss.org/drools/release/6.2.0.CR4/drools-docs/html/DroolsComplexEventProcessingChapter.html, Accessed: 07/2016.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Lee14] I. J. Lee. Big data processing framework of road traffic collision using distributed cep. In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pages 1–4, Sept 2014.

[Lon16] Josh Long. Deploying spring boot applications. https://spring.io/blog/2014/03/07/deploying-spring-boot-applications, Accessed: 10/2016.

[LST02] R.C. Luo, K.L. Su, and K.H. Tsai. Fire detection and isolation for intelligent building system using adaptive sensory fusion method. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 2, pages 1777–1781 vol.2, 2002.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Luc07] David C Luckham. A short history of complex event processing. part 1: Beginnings. *Online only (http://complexevents. com/wp-content/uploads/2008/02/1-a-short-history-of-cep-part-1. pdf)*, 2007.

[MCT14] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. Learning from the past: Automated rule generation for complex event processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 47–58, New York, NY, USA, 2014. ACM.

[Mic06] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.

[Mol09] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[MQT16] MQTT. A lightweight messaging protocol for small sensors and mobile devices, optimized for high-latency or unreliable networks. http://mqtt.org, Accessed: 05/2016.

[MSPC12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012.

[Ora16a] Oracle. Oracle complex event processing (cep) 10g release 3. http://docs.oracle.com/cd/E13157_01/wlevs/docs30/, Accessed: 08/2016.

[Ora16b] Oracle. Basic jms api concepts. http://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html, Accessed: 10/2016.

[Ora16c] Oracle. The jms api programming model. https://docs.oracle.com/javaee/7/tutorial/jms-concepts003.htm, Accessed: 10/2016.

[Par16] Terence Parr. http://www.antlr.org/, Accessed: 09/2016.

[PBEV09] M. Presser, P. M. Barnaghi, M. Eurich, and C. Villalonga. The sensei project: integrating the physical world with the digital world of the network of the future. *IEEE Communications Magazine*, 47(4):1–4, April 2009.

[QSF+14] Yongrui Qin, Quan Z. Sheng, Nickolas J. G. Falkner, Schahram Dustdar, Hua Wang, and Athanasios V. Vasilakos. When things matter: A data-centric view of the internet of things. *CoRR*, abs/1407.2704, 2014.

[RSL99] V Ryan, S Seligman, and R Lee. Schema for representing java (tm) objects in an ldap directory. Technical report, 1999.

[Sci16] The Carnegie Mellon University Computer Science. The "only" coke machine on the internet. https://www.cs.cmu.edu/~coke/history_long.txt, Accessed: 08/2016.

[SGLN+11] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

[SL16] W. Roy Schulte and David Luckham. Real-time intelligence and how it uses complex-event processing (cep). http://complexevents.com, Accessed: 07/2016.

[SLW10] G. Schmutz, D. Liebhart, and P. Welkenbach. *Service-oriented Architecture: An Integration Blueprint : a Real-world SOA Strategy for the Integration of Heterogeneous Enterprise Systems : Successfully Implement Your Own Enterprise Integration Architecture Using the Trivadis Integration Architecture Blueprint*. Professional expertise distilled. Packt Pub., 2010.

[Sof16a] Pivotal Software. https://projects.spring.io/spring-boot/, Accessed: 10/2016.

[Sof16b] Pivotal Software. Scheduling tasks. https://spring.io/guides/gs/scheduling-tasks/, Accessed: 10/2016.

[SS13] O. Saleh and K. U. Sattler. Distributed complex event processing in sensor networks. In *2013 IEEE 14th International Conference on Mobile Data Management*, volume 2, pages 23–26, June 2013.

[Ste16] William Stewart. The internet toaster. `http://www.livinginternet.com/i/ia_myths_toast.htm`, Accessed: 08/2016.

[TIB16] TIBCO. Tibco businessevents. `http://www.tibco.com/products/event-processing/complex-event-processing/businessevents`, Accessed: 08/2016.

[VBD01] E. Vargas, J. Bianco, and D. Deeths. *Sun Cluster Environment: Sun Cluster 2.2*. Sun Microsystems Press Series. Sun Microsystems Press, 2001.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[Wik16] Wikipedia. Internet digital dios. `https://en.wikipedia.org/wiki/Internet_Digital_DIOS`, Accessed: 08/2016.

[XWP14] Teng Xu, J.B. Wendt, and M. Potkonjak. Security of iot systems: Design challenges and opportunities. In *Computer-Aided Design (ICCAD), 2014 IEEE/ACM International Conference on*, pages 417–423, Nov 2014.

[YCL11] Wen Yao, Chao-Hsien Chu, and Zang Li. Leveraging complex event processing for smart hospitals using rfid. *J. Netw. Comput. Appl.*, 34(3):799–810, May 2011.