University of Ljubljana
Faculty of Computer and Information Science

Naum Gjorgjeski

Complex event processing for integration of Internet of Things devices

BACHELOR'S THESIS

UNDERGRADUATE UNIVERSITY STUDY PROGRAMME
COMPUTER AND INFORMATION SCIENCE

Mentor: prof. dr. Matjaž B. Jurič

Ljubljana, 2016

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Naum Gjorgjeski

Kompleksna obdelava dogodkov za integracijo naprav v Internetu stvari

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž B. Jurič

Ljubljana, 2016

The text is formatted with the LaTeXeditor.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Faculty of Computer and Information Science issues the following thesis:

Complex event processing for integration of Internet of Things devices

Subject of the thesis:

Analyze the Internet of Things field, identify applications and characteristics of the things and challenges in IoT network architecture. Describe the integration of the Internet of Things and cloud computing, identify the effects of integration and analyze scalability and elasticity implications. Describe the microservice architecture and identify the impact of microservices on IoT solutions with special emphasis on scaling. Identify the need for complex event processing, analyze and describe key characteristics of complex event processing. Develop an application, which follows the principles of microservices architecture and demonstrates the integration of the Internet of Things and cloud computing. Demonstrate how to achieve scalability and elasticity of the application and implement real-time processing of IoT events.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kompleksna obdelava dogodkov za integracijo naprav v Internetu stvari

Tematika naloge:

Analizirajte področje Interneta stvari, opredelite aplikacije, lastnosti stvari in izzive v arhitekturi IoT omrežij. Opišite integracijo Interneta stvari in računalništva v oblaku, opredelite posledice integracije in analizirajte implikacije skalabilnosti in elastičnosti. Opišite mikrostoritveno arhitekturo in identificirajte vpliv mikrostoritev na IoT rešitvah s posebnim poudarkom na skaliranje. Opredelite potrebo po kompleksni obdelavi dogodkov, analizirajte in opišite ključne lastnosti kompleksne obdelave dogodkov. Razvijte aplikacijo, ki sledi načelom mikrostoritvene arhitekture in demonstrira integracijo Interneta stvari in računalništva v oblaku. Demonstrirajte kako doseči skalabilnost in elastičnost aplikacije in implementirajte procesiranje IoT dogodkov v realnem času.

I would like to thank my mentor prof. dr. Matjaž B. Jurič for the guidance and assistance I have received during the preparation of my Bachelor's thesis.

I would like to express my deepest and biggest gratitude to my family for being patient and supportive throughout my studies.

За моето семејство.

Contents

Abstract

Povzetek

1 Introduction **1**

2 Overview of the Internet of Things **5**

2.1 Applications 5

2.2 Characteristics of the "things" 6

2.3 Challenges in IoT networks 7

3 Integration of the Internet of Things and cloud computing **11**

3.1 Cloud requirements for IoT solutions 11

3.2 Types of clouds 12

3.3 Cloud services 14

3.4 Effects of the integration 14

3.5 Scalability and elasticity 15

4 Microservices **21**

4.1 Definition of microservices 21

4.2 Comparison with monolithic architecture 22

4.3 Impact of microservices architecture on IoT solutions 25

4.4 Scaling of microservices 29

5 Complex event processing **33**
5.1 The need for CEP 33
5.2 Evolution of information flow processing systems 34
5.3 Constructs and operators of CEP languages 37

6 A practical example **43**
6.1 Goals of the practical example 43
6.2 Structure of the practical example 43
6.3 Scenarios 47
6.4 Apache Kafka 48
6.5 Esper 54
6.6 Deployment and management of the microservices using Kubernetes 62

7 Conclusion **79**

Bibliography **82**

List of abbreviations

Abbreviation	Meaning
IoT	Internet of Things
CEP	Complex event processing
IIoT	Industrial Internet of Things
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
REST	Representational state transfer
SOAP	Simple Object Access Protocol
RPC	Remote procedure call
SLA	Service-level agreement
DBMS	Database management system
DSMS	Data stream management system
IFP	Information flow processing
JAX-RS	Java API for RESTful Web Services
EPL	Event Processing Language
CDI	Contexts and Dependency Injection
CORS	Cross-origin resource sharing
GUI	Graphical user interface
QoS	Quality of service

Abstract

Title: Complex event processing for integration of Internet of Things devices

Author: Naum Gjorgjeski

As a relatively new technology, the Internet of Things (IoT) faces many challenges. IoT networks are characterized by a big number of devices. Each of the devices produces huge amount of events. Therefore, scalability is one of the key requirements of IoT applications. Cloud computing could help us achieve scalability by providing virtually unlimited resources. The microservices architecture is becoming increasingly popular for cloud deployments of applications. We often want to extract real-time information from the events that are coming from IoT devices. It would be harder to infer useful information from the enormous amount of raw events, if we store them in a database. Complex event processing enables us to analyze the events as the stream of events flows and to infer meaningful information from them in real time. To demonstrate all of this in practice, we developed an IoT application, which follows the principles of microservices architecture. It is able to simulate events, consume them, do complex event processing and display visualizations. In order to balance the load between the instances and achieve scalability and elasticity, the microservice which is consuming the events can be scaled up and scaled down.

Keywords: Internet of Things, cloud computing, microservices, complex event processing, Esper, Kubernetes.

Povzetek

Naslov: Kompleksna obdelava dogodkov za integracijo naprav v Internetu stvari

Avtor: Naum Gjorgjeski

Internet stvari (IoT) se kot relativno nova tehnologija sooča s številnimi izzivi. Za IoT omrežja je značilno, da jih sestavlja veliko število naprav. Vsaka od teh naprav generira ogromno količino dogodkov. Zato je skalabilnost ena od ključnih zahtev IoT aplikacij. Računalništvo v oblaku nam lahko pomaga doseči skalabilnost tako, da nam zagotavlja virtualno neomejene količine virov. Arhitektura mikrostoritev postaja vse bolj popularna za namestitev aplikacij v oblaku. Pogosto hočemo iz dogodkov, ki prihajajo iz IoT naprav, pridobiti informacije v realnem času. Težje bi bilo pridobiti uporabne informacije iz enormne količine neobdelanih dogodkov, če bi dogodke shranjevali v podatkovno bazo. Kompleksna obdelava dogodkov nam omogoča, da analiziramo dogodke in iz njih pridobivamo uporabne informacije v realnem času. Da bi vse to demonstrirali, smo razvili IoT aplikacijo, ki sledi načelom mikrostoritvene arhitekture. Aplikacija lahko simulira dogodke, jih sprejema, izvaja kompleksno obdelavo dogodkov in prikazuje vizualizacije. Mikrostoritev, ki sprejema dogodke, lahko skaliramo navzgor in navzdol s ciljem, da uravnotežimo obremenitev med instancami in dosežemo skalabilnost in elastičnost.

Keywords: Internet stvari, računalništvo v oblaku, mikrostoritve, kompleksna obdelava dogodkov, Esper, Kubernetes.

Chapter 1

Introduction

Nowadays, we are surrounded by a wide range of electronic devices used for collecting and generating data. These devices are usually interconnected and form a network, which also includes information systems and sometimes other entities too. That is a simplistic and minimal description of what we call the Internet of Things (IoT) [18]. In a broader sense, the Internet of Things has many meanings and by using that term we are essentially referring to one or many of the following [45]:

- the underlying network connecting these devices (usually wireless technologies and the Internet);
- the diverse devices themselves (e.g. sensors, actuators, RFIDs);
- applications and services that enable processing of devices' data.

In an Internet of Things network we have numerous devices, each generating a massive number of events. Event could be defined as anything that happens, or is contemplated as happening [12]. Broadly, event could be an incoming email, a financial trade, a tire puncture, an indication that a light is turned on or off, etc. In IoT networks, an event could be and often is a reading from a sensor.

Integration and analysis of the generated data are essential. Thereby, the data could give us valuable insights, so we can reach conclusions and make data-based decisions. One would immediately point to Big data, as we use this term to denote large and complex quantities of data. The increases in storage capacities enabled storage of huge amounts of data. At the same time, advancements in computing power and artificial intelligence contributed for more efficient analysis of the data. However, the data generated by IoT networks is not only quantitatively big, but also fast, since it is being created at enormous rates. Therefore, the term Fast data emerges. If we store the data in order to process it later, we are missing precious opportunities to react as the data is being generated. Consequently, not being able to act in real-time causes the data to lose some of its value [13].

To process events in terms of Fast data, we use complex event processing (CEP) techniques. Complex event processing is processing of events from multiple sources in order to identify meaningful events and respond to them as quickly as possible [4]. A simple example might be a building's fire detection system. We can place a set of sensors which measure temperature across the whole building. The sensors send their measurements to an information system which might do simple pattern matching and check if few recent measurements are much higher than the previous ones. We can use more complex CEP techniques as well. If the system detects unusual increase of temperature in a short amount of time, it raises an alert. The main task of CEP is to transform a set of base events (e.g. measurements) into one or more complex events carrying logical semantic content. The set consists of a different number of base events, varying from a few to thousands of them. Furthermore, the extracted complex events can be used to derive even more complex events [4].

Complex event processing systems must enable high throughput of events and on-the-fly event processing. In order to fulfill these demands, CEP systems must be extremely scalable. An increase in scalability can be achieved by distributed systems. In the current era of cloud computing, cloud-based systems are becoming a crucial architectural part and are a perfect fit for manipulation of events generated by IoT networks. However, not all architectures utilize the potential of cloud environments maximally. We shall see why microservices architecture might be a better choice than the traditional monolithic architecture for deployment of applications in cloud environments.

The structure of the thesis is as follows. In chapter 2 we review the IoT and the challenges that arise in IoT networks. Then in chapter 3 we describe the effects of the integration of IoT and cloud computing. We shall see which issues of the IoT architecture are addressed by cloud computing, with special consideration of scalability and elasticity. Chapter 4 defines the microservices architecture. We compare it with the monolithic architecture and explain why the microservices architecture is a better fit for deployment in cloud environments. As scaling is a key requirement in IoT solutions, the strategies of scaling are also discussed. Chapter 5 discusses the need for CEP systems, their evolution and the most common constructs and operators of CEP languages. In chapter 6 we develop a practical example of a scalable and elastic IoT application, which is able to produce and consume events, do complex event processing and display visualizations. Chapter 7 provides a conclusion of the thesis.

Chapter 2

Overview of the Internet of Things

To feel the added value brought by the Internet of Things, it would be useful to look at some applications first. Then we will go through the challenges we face in IoT networks. Potential solutions of some of the challenges are also briefly reviewed. Since the IoT paradigm has grown recently, in practice most of the solutions are either newly developed and still under research or are enhancements and adaptations of existing solutions from other more established technologies.

2.1 Applications

Internet of Things finds useful scenarios everywhere around us. Some examples are: home automation, supply chain management, assisted living, environmental monitoring, traffic monitoring. Therefore, it makes our lives more comfortable and protects our environment. The industry could also utilize the potential of the IoT. As the main objective of companies is to maximize their profits, IoT could play an important role in helping them reduce costs.

In most cases, when people think about applications that are made possible by the ubiquitous devices, they initially recall of home automation, i.e. smart homes. For instance, in our homes we can install sensors, which sense and measure some physical conditions and actuators, which act so that they change those conditions. Then, through an application on our mobile phone or laptop we can remotely control many parameters. We can check if we have forgotten to turn off the light bulbs, command home appliances, use temperature sensors to get data about air conditions, use actuators to adjust air conditions, use motion sensors to detect unwanted visitors, etc.

Another important application of the Internet of Things are healthcare systems. For example, sensors which monitor various vital signs can be especially helpful to older people. Sensors could sense if something undesirable happens and alarm the responsible person.

Also, a subject of current broad interest is traffic monitoring. Many developed cities struggling with traffic bottlenecks have already tried to solve this problem by investing in sensors that monitor the amount of traffic and information systems that process the data, so that the traffic flow is optimized or drivers are informed about the current conditions through an application.

The IoT might be the next big thing in the industry as well. Industrial Internet of Things (IIoT) is emerging as a segment of the IoT which is engaged in the industry only. Businesses find creative ways to use the IoT and consequently improve their performance. The IoT could help them reduce the costs and increase the productivity of their employees. Common examples that have been adopted in practice are supply chain and energy management. The IIoT also opens lots of possibilities for development of new business models [18].

2.2 Characteristics of the "things"

In order to be aware of the difficulties in IoT networks, one must know what the "things" actually are and which are their characteristics and limitations. A "thing" in an IoT network is an object which possesses most or all of the following functionalities [45]:

- It must represent a physical object;
- It must be uniquely identifiable;
- It must be able to communicate and interact with other "things" or other entities in the network. In order to be reachable in the network and support communication a "thing" must have an address;
- It must have at least minor computing capabilities;
- It may have an ability to sense and measure some physical conditions (e.g. temperature, humidity) and act so that it changes those conditions.

2.3 Challenges in IoT networks

At the moment, the IoT is gaining momentum and is becoming increasingly popular. The estimates of analysts about the growth of the number of devices varies, but they all agree it would continue to increase at very fast pace in the upcoming period. We could credit its recent popularity and viability to the advancements in wireless technologies, standardization of communication protocols, the lowering prices of devices and increases in storage and computing power [37].

However, when we set up an architecture for an IoT network, we face different challenges, varying from low level limitations with the wireless network itself to obstacles on the semantic level. During the process of construction of IoT architectures, we must be aware of these challenges and choose optimal solutions which will best suit our needs. The challenges are the following [45] [35]:

Devices heterogeneity In an IoT network we usually have plenty of different devices, which may use different protocols for communication.

Because the IoT emerged recently, there is still no standardization for the communication between entities in an IoT network. Consequently, similar devices made by different vendors often use different protocols. They also have different computing capabilities, ranging from devices only able to do some minor computing to powerful devices that can do a lot more. Depending on the level of heterogeneity, enabling communication in such network and integration of the data from different devices may be a problem.

Scalability IoT networks often consist of an enormous number of devices, which are producing an enormous amount of events, therefore exposing scalability as a central issue. Cloud computing establishes itself as an optimal solution, providing virtually unlimited resources. Achieving scalability is one of the key points in IoT networks and is further discussed in the following chapters.

Wireless networking technologies In an IoT network the data is being generated at high rates and the data flow is often constant. However, the available spectrum (e.g. the ISM radio band) is limited and is being used by other technologies at the same time. For that reason, further research for solutions like cognitive radio and dynamic spectrum management is carried out. For instance, cognitive radio detects the most suitable and currently unused frequencies (including reserved frequencies - TV, FM radio etc.). We can use these frequencies for our own needs as long as they are free. Still, we should be aware that the usage of reserved frequencies is often regulated by local laws.

Energy efficiency If the devices in an IoT network operate on battery, they have low energy capacity. Therefore, resource efficiency for the devices is compulsory in order to minimize the consumption. Also, new substitutes for battery like micro solar panels can mitigate this problem.

Autonomy An important challenge imposed by the complexity and the size of IoT networks is to make the "things" smarter, so that they could be as autonomous as possible, be able to respond to different situations and circumstances and work without or with minimal human intervention.

Privacy and security Privacy and security are amongst the biggest obstacles preventing the IoT from expanding even faster. As we said, the devices often have limited computing capabilities. Energy is a scarce resource as well. However, we know that security depends heavily on energy and computing capabilities (because of encryption). As more complex security schemes cannot be used, security is often neglected by manufacturers. To ensure privacy in an Internet of Things network, authentication and data integrity have to be addressed. Since the collected data is sometimes personal, we should have control of who collects the data, when and what data is being collected. Another possible attack is eavesdropping. Since in some cases the "things" are left unsecured, it is also possible to break them down physically.

Semantic interoperability When we collect data from IoT networks, we might do real-time processing (i.e. CEP) or store the data in a database and process it later. We want to infer something from the data, i.e. turn the raw data into useful information. Since the amount of data is enormous, we must support automated reasoning. In order to do that, the data must be in a standardized format. Furthermore, it is recommended that the raw data is equipped with meta-data, i.e. additional data carrying information about the content of the raw data.

Chapter 3

Integration of the Internet of Things and cloud computing

Although the Internet of Things and cloud computing are technologies that have been developing independently from one another, there are strong reasons why we should integrate them. In the previous chapter we reviewed the difficulties we face in an IoT network. Most of them arise because of the huge number of devices and their limited capabilities.

Offering virtually unlimited storage and computing capabilities, cloud computing could help us tackle many of them successfully. On the other hand, IoT is a promising technology that could alter the Internet, shifting its main focus from communication between user devices to machine-to-machine communication (e.g. communication between a sensor and a node that does data aggregation). Being an integral part of such revolution would further expand the usage of the already widely adopted cloud computing [36].

3.1 Cloud requirements for IoT solutions

In order for the integration to be successful, the cloud is expected to meet the following IoT application requirements [41]:

Device management When a device is initialized, it must be able to connect to the cloud and identify itself by its unique identifier. That way, we know that the device is active and we also know which data is provided by the device. The communication is not one-way, as when the device is active, it is important for the device to be managed by the cloud (e.g. provide software updates).

Data ingestion Each device in an IoT network generates immense amount of data. When this is multiplied by the number of devices in IoT networks, we get unprecedented data volumes. Scalability is one of the most important sore points in IoT solutions and the cloud is expected to handle such incoming amounts of data.

Transformation and storage When the messages arrive in the cloud, the cloud must enable selection and transformation of the raw data into useful information. The transformation is done on-the-fly to get real-time information or later to get retrospective information. CEP is the most important player in on-the-fly processing and is described in detail in chapter 5. Cloud should be able to provide computing resources for CEP. If we want to do further processing later, the huge amounts of data must be stored. In that case, the cloud must ensure reliable data storage.

Real-time notifications By using CEP techniques, the raw data is continuously analyzed and complex events are produced. The IoT application must be able to report the complex events in real-time in form of notifications.

Visualization The cloud must serve as a host for visualizations of the IoT network's status.

3.2 Types of clouds

After the decision to use the cloud infrastructure, we can choose between a private, public and hybrid cloud. Each one has some advantages and drawbacks.

A private cloud offers us better control of the privacy because we use our own servers. However, we have to maintain the infrastructure by ourselves, which is not an easy task. Private clouds are not intended to sell services to external customers, but to have full control of the data. It is an expensive option usually chosen by big companies or even countries and isn't a viable solution for the others.

If we decide to use a public cloud, we won't bother with the infrastructure management. Since the data is on a public cloud, the cloud provider ensures limited access and employs many security mechanisms for isolation. If the data generated by IoT devices is not confidential, public cloud is suitable for storage of that data and deployment of our applications.

Hybrid cloud is a combination of the two approaches. There are various reasons for usage of hybrid clouds - e.g. we might want to control confidential data in private clouds, while the rest is on public clouds. Some use public clouds to maintain scalability of their applications during peak periods or as a prevention from natural disasters and electricity blackouts, securing their application availability in case they have their data center in one location only.

The dominant Pay-As-You-Go pricing in public clouds is favorable, as it allows us to pay only for the resources and services we actually use. Then, if we decide to offer our services or applications to consumers, the same pricing model could be applied too, enabling them to pay depending on which services or applications they use and how much they use them. For instance, we might get access to sensors' data, either by setting up our own network of IoT devices or through payable services offered by sensors' data providers. We can use that data to make an application which does CEP and pushes real-time notifications to a data-based visualization application. CEP provides additional insights by extracting information from the raw data, while the visualization application provides nice intuitive visualizations. We can then offer the application on a subscription basis. Thereby, the pricing model also fits perfectly in the plan for integration of cloud computing and the IoT [30] [3].

3.3 Cloud services

Cloud environments offer the following resources as on-demand services [36]:

Storage Cloud offers storage and enables us to store the raw data from sensors and the higher level data gathered from analysing the raw data (e.g. by CEP). We practically get storage on-demand, anytime, as much as we need it. As we saw, if we are using public clouds, the payment amount is proportional to the amount of storage we use.

Computational resources It is impossible to use IoT devices for complex computations because of their low computing capabilities. Therefore, they must transfer their data to a node which is able to do proper complex computations (e.g. sensors must transfer their measurements). Cloud is a favorable solution as the data from the devices can be transferred in the cloud, where we have practically unlimited computational resources. In the cloud, we can choose whether we want to process the data in real-time (e.g. use CEP techniques) or store the data and process it later. To sum up, computational resources can be used for computations, analysis and visualization of the data we get from IoT devices.

3.4 Effects of the integration

The integration of IoT and cloud computing solves or diminishes some of the challenges in IoT networks we pointed out in section 2.3 and satisfies the needs of IoT applications discussed in section 3.1.

As we saw, we can completely move computations to the cloud and maximize the energy efficiency of IoT devices. Their primary task would be to transfer the data to the cloud [36].

As illustrated in figure 3.1, the cloud serves as a mediator between the "things" and the applications [36]. The "things" are service-oriented and offer their data through web services (e.g. REST, SOAP). In this way, the low-level "things" world is abstracted. We are able to develop applications only by knowing what the low-level networks of "things" offer through their services. The abstraction eases the process of application development.

This results in a loosely coupled system where different individuals can contribute by offering their Infrastructure, Platform or Software services in the cloud (i.e. IaaS, PaaS, SaaS). Separate individuals or companies could offer: storage and computational resources, sensors' data, algorithms for data processing, visualization tools, etc [43].

Many standard problems in distributed systems (e.g. virtual-machine escape) have to be addressed. In case we are using public clouds, this is done by providers. However, depending on the sensitivity of the data, the trust in the service provider may be questionable [36]. If the data from IoT devices is not confidential, this issue doesn't outweigh the benefits from the integration of IoT and cloud computing (by using public clouds).

3.5 Scalability and elasticity

Although scalability and elasticity might look like synonyms, they represent related concepts with a few differences between them.

Scalability is the capability to withstand increasing workloads. Scalability doesn't take into consideration the amount of resources we use. Basically, if an application is scalable by using 5 servers, it will still be scalable if we use 10 of them. As long as the application is able to deal with the workload at any time, it is scalable. Speed, frequency and granularity of the scaling actions are not taken into account either. In figure 3.2a we can see an example of a scalable and non-elastic application. In figure 3.2b the application is not scalable, as it cannot handle the peaks of increased workload. In order for an application to be elastic, it must be scalable.

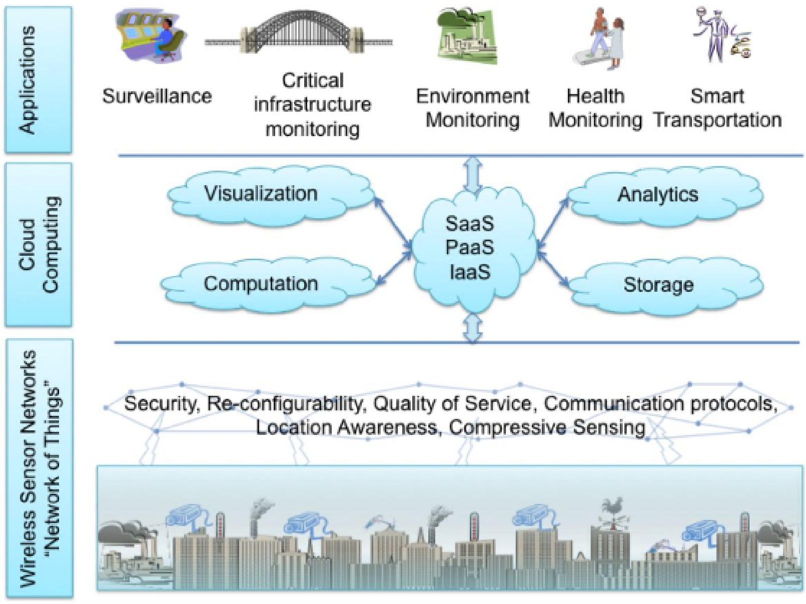


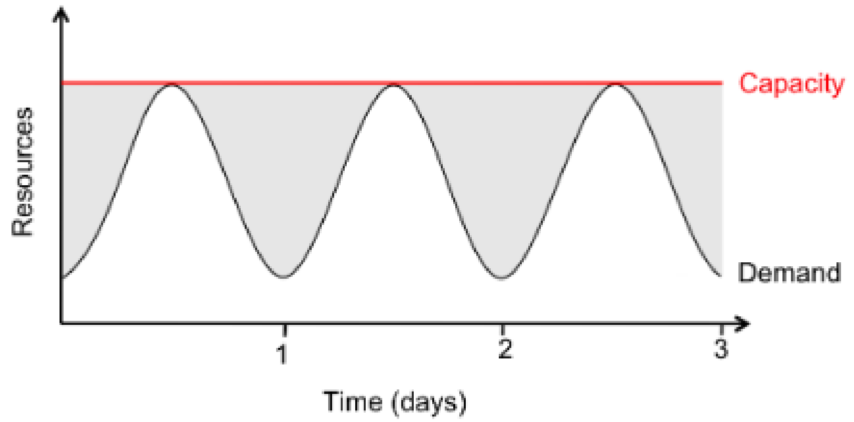
Figure 3.1: The cloud enables easier development of IoT applications and satisfies most of their requirements [43].

Elasticity reflects the level of adaptivity to the current workload by dynamic provisioning and deprovisioning of resources, either manually or automatically. The level of adaptivity depends on the precision, i.e. the standard deviation of the provisioned resources compared to the current resource demand. The precision is influenced by the speed of scaling of applications, i.e. the time needed to get from underprovisioned to optimal or overprovisioned amount of resources or to get from overprovisioned to optimal or underprovisioned amount of resources [44]. An example of elastic application is illustrated in figure 3.2c.

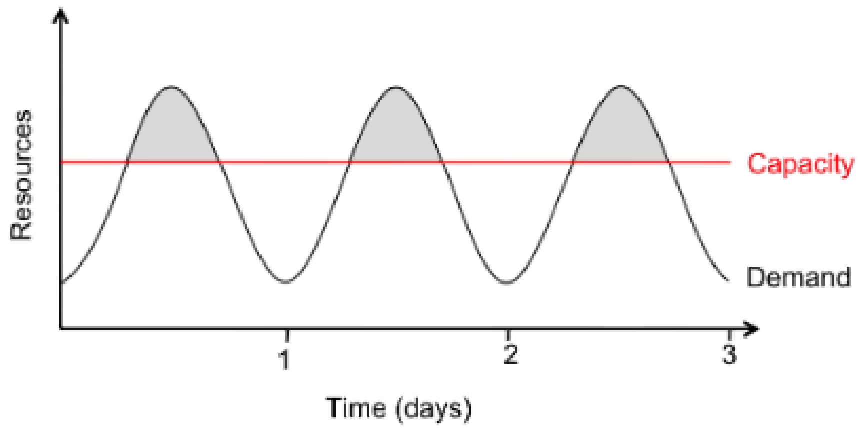
As we have seen before, IoT networks are generating huge amounts of data. Therefore, scalability needs special attention. For that purpose, cloud computing offers on-demand services. In cloud environments we get dynamically allocated storage and computational resources at run-time. The amount of resources could be set either by us or dynamically depending on the load and the maximum budget, if we have chosen one. For example, if the number of events produced by sensors is too high and the current instances are not capable to process all of the events, additional instances could be deployed manually or automatically. In this way, high Quality of Service (QoS) is sustained [43].

Although IoT applications require scalability in the first instance, a scalable and non-elastic IoT application is an impractical solution. Scalable and non-elastic application wastes resources unnecessarily. A waste of resources results in a waste of money and lower performance, which consequently leads to uncompetitive IoT solution. The demand of resources in an IoT application might fluctuate due to: addition of new sensors, fail of a group of sensors, group of sensors only operating during certain period of the day, increased or decreased amount of users of the visualization application etc.

Thus, we want both a scalable and elastic IoT solution. Whether we achieve that objective depends heavily on the choice of architecture. Not all architecture approaches combined with cloud computing guarantee both scalability and elasticity. Therefore, in the next chapter we will look at a favorable and suitable architecture for IoT solutions — the microservices architecture.

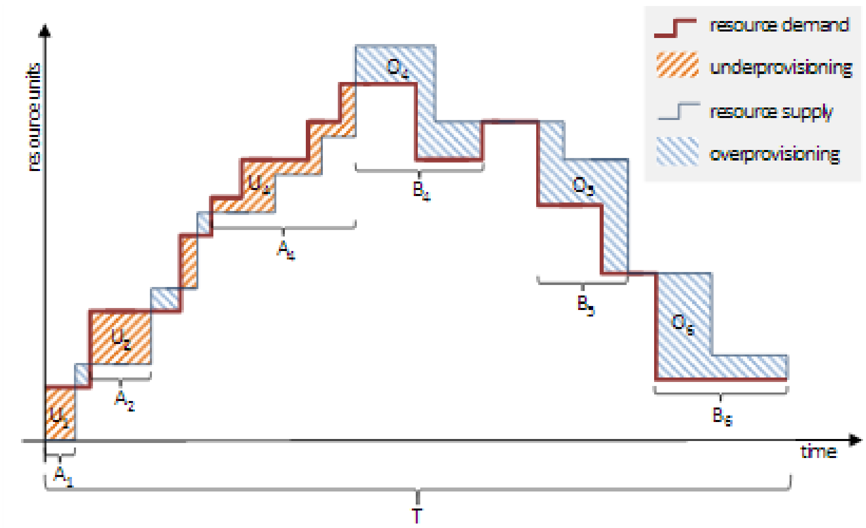


(a) The amount of resources is enough for the application to handle peaks and the application maintains scalability. However, a lot of the resources remain unused when the workload is low [34].



(b) If we decide to use lower amount of resources, it is not enough for the application to handle peaks. Therefore, the application is not scalable [34].

Figure 3.2: Illustrations of the concepts of scalability and elasticity of an application.



(c) It is not possible to fully adapt to resource demands. Elastic applications try to minimize the difference between the provisioned resources and the demand. As deviation is minimized, QoS and resource efficiency are maximized [44].

Figure 3.2: Illustrations of the concepts of scalability and elasticity of an application.

Chapter 4

Microservices

In order to get the most of the integration of IoT and cloud computing, the use of microservices is advisable. This chapter provides answers to why this is so.

4.1 Definition of microservices

Microservices are an architectural approach to developing applications as a set of small services, where each service is running as a separate process, communicating through simple mechanisms [29].

Most of the advantages of the microservices architecture stem from its main feature — decomposition of a service or an application into smaller components, i.e. microservices. The result of the decomposition should be many independent units (i.e. components). Each of these components should implement a specific functionality. As a result, we can develop, deploy, upgrade and scale every microservice independently. Since not every microservice has equal workload, each microservice is scaled separately. That enables us to use optimal amount of resources and makes microservices architecture a natural fit for achieving both scalability and elasticity. We control every microservice separately. Since they are smaller, they are easily manageable. Developing them separately enables us to use different technologies (e.g. different programming languages) for each microservice. When we want to release an update for a part of our application or service, we don't redeploy the whole application, but only the corresponding microservice.

Microservices communicate through web services (such as REST) or use remote procedure calls (RPC). However, communication between processes is costly. We should prefer to avoid it and reduce it to the minimum.

All of the mentioned benefits and drawbacks coming from the microservice architecture are described in detail in the following sections.

4.2 Comparison with monolithic architecture

The advantages of microservices architecture are best identified when we compare it to the traditional monolithic architecture. This chapter explains why the microservices architecture is a better fit for deployment in cloud environments.

Monolithic application has all of its components packed together. For instance, monolithic web applications have the client-side, the server-side and the database in a single logical executable. Similarly, monolithic IoT applications have the whole logic for: communication with IoT devices, processing of devices' data, eventual communication with databases and visualization, in a single logical executable as well. Although we might develop an application in a modular way (e.g. have separate Maven modules for different tasks), at the end it is still packed as a single logical executable.

In figure 4.1 we can spot the differences between the monolithic and the microservices architecture. The application illustrated on the left side is developed by the monolithic approach and has all of its components packed into one container. To achieve scalability and elasticity in monolithic applications, more instances of the whole application must be deployed or terminated. However, different application functionalities rarely have an equal share of the workload. In IoT solutions, we cannot expect the workload to be continuously equal both in the component communicating with sensors and the component for visualization. Therefore, we mustn't scale them together. Similarly, in a typical web application for online sale of goods, we can predict that the catalogue lists will be used more frequently than a purchase completion functionality because most of the people go through catalogues first and then decide to checkout. With the microservices approach, as shown on the right side of figure 4.1, every microservice is packed as an independent component. We can scale every microservice independently and change the number of instances for each microservice separately. Since microservices are fine-grained components, it is the essential approach if we want to have a fully elastic application. We scale the application according to the workload of each of the microservices, instead of scaling all of the components together like we have to do in monolithic applications. In this way, we get a scalable, elastic and resource efficient application.

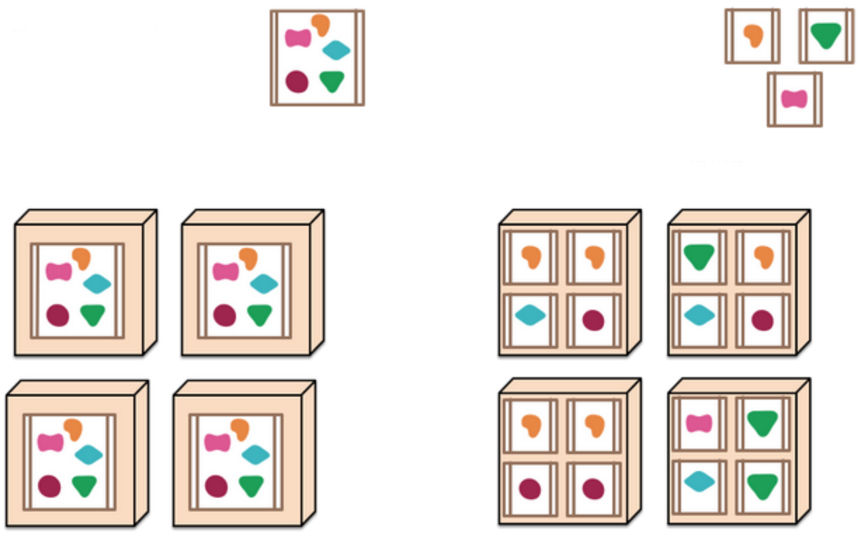


Figure 4.1: The application illustrated on the left side adopts the monolithic approach. It has all of its components packed in a single logical executable and runs in a single process. On the right side, we see the same application built using microservices. Each component is a separate process and can be developed, deployed, upgraded and scaled independently [29].

Figure 4.1 shows another difference of the monolithic and the microservices architecture. Monolithic applications usually run as a single process. If we want to release an update of the application, the whole application must be redeployed for the changes to take effect. It doesn't matter which component we have changed. With the microservices approach, we would redeploy the appropriate microservice, where we have made a change. The update of one microservice should cause no or minor changes to the other microservices. Here we can spot a major challenge of the microservices approach. We mentioned that the communication between components is expensive and should be minimized. If a change in one microservice imposes many changes in other microservices, the benefits of the microservices architecture are lost. For instance, a change in the way the application communicates with IoT devices and gets data from them must have no or minimal impact on how we process the data. Therefore, we should try to componentize the application into microservices in a way that would allow for the communication between the microservices to be minimal. The approaches to componentization of IoT applications are discussed in section 4.3.1.

Monolithic applications could become so big that managing them could be a nightmare. If a monolithic application is big, it is more vulnerable and harder to update. With that in mind, it is more likely to make a mistake. Bugs in monolithic applications could be really expensive, as they cause the whole application to crash, whereas in microservices applications only the corresponding microservice collapses. That means the application is still running and only the specific functionality implemented by the microservice is unavailable. The importance of this behavior is even more apparent in IoT applications. For instance, if a microservice which communicates with a certain group of sensors crashes, that won't affect or stop the processing of the data provided by microservices which communicate with other sensors. The other components of the application will still be up and running.

Nonetheless, the monolithic architecture finds its applications, where it might actually be better than the microservices architecture. Simple and small applications are both easily manageable and easily scalable. In that case, we wouldn't bother with the adoption of the microservices architecture since it is an overkill for simple applications. However, we know that applications in the IoT world are rarely simple. The advantages of the microservices architecture are revealed when the complexity starts to grow [29] [40].

4.3 Impact of microservices architecture on IoT solutions

IoT applications have high requirements regarding scalability, what fundamentally pushes us towards distributed architecture. Even though we have virtually unlimited storage and computing resources in the cloud, we saw that IoT applications still require a different approach than the monolithic one.

The microservices architecture has emerged in recent years and consequently is not fully defined. However, most of the microservices applications share the same characteristics. In general, the microservices architecture is adaptable to the requirements of IoT applications. In this section we will analyze the characteristics of the microservices architecture that directly affect IoT applications.

4.3.1 Componentization via services

When developing applications, it is a good practice to break down the application into several components. Such components that programmers frequently use are libraries. The concept of services in microservices architecture is similar to the libraries concept. However, there is one big difference. Libraries are linked to the program and when the program is running, there is only one process. On the other hand, the microservices architecture tends to componentize a project into services, where each service is running in its own separate process. Thereby, each microservice could be deployed and scaled independently [29].

By componentization into (micro)services the problem with the vast heterogeneity of devices could be addressed. We can have distinctive microservices for devices that communicate using different protocols. These microservices might act as a proxy. The issue of adding new devices, which communicate using a protocol we don't support, is usually resolved by adding a microservice acting as a proxy between protocols [47].

In general, there are two approaches for decomposition of applications into microservices: verb-based and noun-based [33]. Verb-based is the decomposition of an application around single use cases. In IoT applications, such scenarios are rare. If we are dealing with multiple groups of devices, we might group the logic for communication with certain type of devices (e.g. temperature sensors), the data processing logic and the visualization logic for this certain group of sensors in one microservice. However, that approach is not a natural fit for IoT applications, as the scaling of different modules is dependent upon different factors. For instance, the communication with devices is most dependent upon the number of devices and the amount of data they generate, while the visualization application must take into account the number of users which access it simultaneously. On the other hand, noun-based is the decomposition in which a microservice is responsible for every operation related to a certain functionality. That is the approach we have mentioned throughout the chapter. We can have one microservice which communicates with the devices and exchanges data with them, other microservice which processes the "hot" data (e.g. CEP), third microservice might store the data in a database for eventual later processing, while fourth microservice might be responsible for visualization. This leads to a dynamic application, where we can orchestrate and scale each functionality separately. A combination of the verb-based and noun-based approach is also possible. First, we do a verb-based decomposition. Then, as we described before, we might decomponentize the microservice responsible for the communication with devices into many microservices in a noun-based way. In this manner, each microservice is acting as a proxy and is responsible only for a group of devices.

Microservices are reusable components. Once developed, a microservice can be integrated into other applications [28]. For instance, if we develop a microservice for communication with a certain type of sensors, it can be integrated in other applications that need to communicate with the same type of sensors.

4.3.2 Decentralized governance

Monolithic applications are usually written in one programming language. The approach to write them in different languages is possible, but that might bring additional problems. Lots of companies decide to use the same technologies for most of their projects. However, we know that a specific language or technology doesn't fit all projects and specifications. That often leads to poor and slow applications with many bugs.

On the other hand, microservices enable a decentralized approach. In fact, the microservices architecture is encouraging developers to write different microservices using different technologies. This characteristic is especially helpful to IoT applications. For instance, it enables us to use different technologies for communication with devices and for data processing. We can use an alternative technology for visualizations as well. Microservices enable all of the technologies to be integrated without having to worry about compatibility issues.

Microservices are completely independent. If a microservice wants to communicate with other microservice (e.g. the data processing component communicates with the component for communication with devices), it does that through standardized protocols (e.g REST, SOAP). Microservices are agnostic. A microservice is unaware of what the core of an other microservice consists of, as it is a different process.

Decentralized governance results in faster adoption of newer and better technologies, which is not the case with monolithic applications. This is especially important for fast developing fields like the IoT. However, having freedom of choice doesn't mean that we should use a different technology for every microservice. We should use a different technology if it brings enhancements to our application [29] [40].

4.3.3 Design for failure

Since microservices are independent components, they should be tolerant of failure of other services. A service call failing because of unavailability of other service occurs often.

This is one of the rare drawbacks of using microservices instead of the opposite monolithic architecture. Since in microservices architectures we must be particularly aware of failures of other services, a slight increase in complexity is necessary.

In IoT applications, quick detection of failures is needed. Microservices applications tackle this problem by real-time semantic monitoring, which enables us to quickly detect a problem and mobilize the staff to resolve it. Useful parameters for semantic monitoring in IoT applications might be current throughput or latency of sensor events [29].

4.3.4 Evolutionary design

The IoT is a fast developing technology and we can't know what the future brings. Probably we would want to integrate new devices in the system. Due to the advancements in artificial intelligence, in the future we might want to add new analyses, so that we get additional insights. As monoliths get big and complex, this is not an easy task. Therefore, microservices serve as a tool to control changes in a faster pace [29].

4.4 Scaling of microservices

The microservices architecture brings more freedom for deployment and management of applications in cloud infrastructures. In order for an application to be elastic, we should ensure that every microservice uses an optimal amount of resources.

For scaling of microservices both IaaS and PaaS can be used. However, PaaS environments are particularly interesting for us. They offer a platform, which is responsible for low-level operations like management of virtual machines, application deployment, load balancing, etc. PaaS enables easier management of applications and is especially suitable for microservices. It enables us to focus on the IoT aspect and not on the essential low-level part of the application. On the contrary, we could hardly find a PaaS provider that supports monolithic applications. In case of a monolithic application, we must do the job of configuration and management of the application. In the following sections, the different strategies of scaling are explained with special consideration of microservices [40].

4.4.1 Vertical scaling

Vertical scaling (or scaling up) is an addition of resources (i.e. memory, CPU) to a single specific node. That gives the node additional capabilities to handle application requirements successfully. However, this approach is very limited in its nature, as we can't add resources forever because nodes hit some physical constraints. IoT applications are often dealing with a huge number of events and require lots of resources, which a single node often can't provide. Therefore, this approach is not feasible if it is not combined with other horizontal scaling approaches [17].

4.4.2 X-axis scaling

The following approaches to scaling are actually the three dimensions of the scale cube [42] [40]. The scale cube is illustrated in figure 4.2.

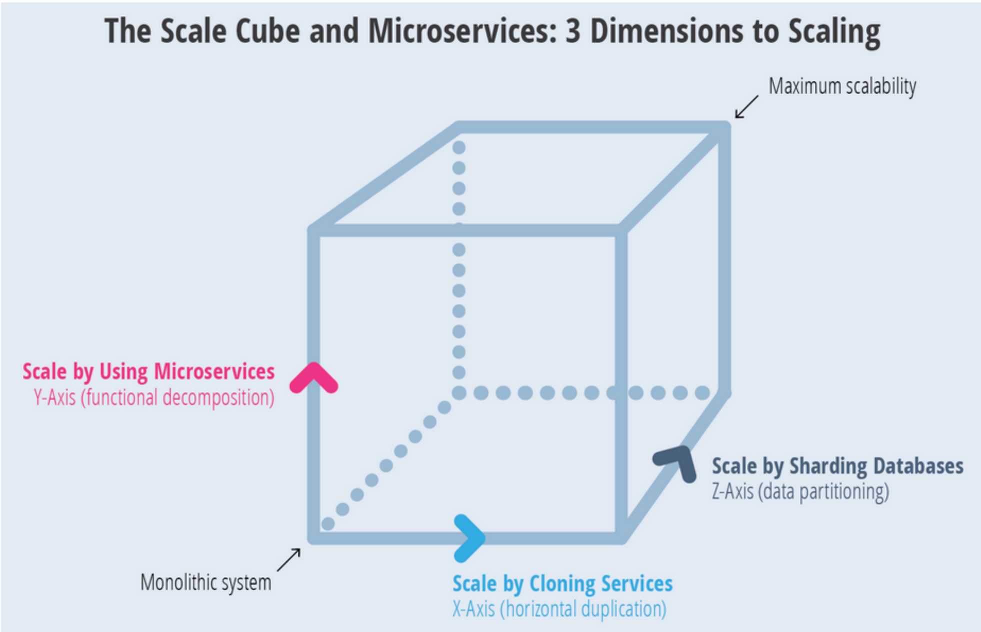


Figure 4.2: The scale cube and its dimensions [14]. X-axis scaling is classic horizontal scaling by adding more instances. Y-axis scaling is scaling by decomposition of the application to microservices. Z-axis scaling introduces the concept that one instance is responsible for a subset of the data or the requests.

X-axis scaling is a typical horizontal scaling (or scaling out). We use several application instances in order to distribute the workload evenly. It is most common for monolithic applications to use this kind of scaling. Having N instances of an application, each instance gets $1/N$ of the workload. Since monolithic applications often maintain user state, the same server must process requests from a certain user.

That is not the case with microservices, which are stateless. Each microservice is scaled separately. PaaS environments further facilitate the scaling of microservices, as they allow us to specify the number of instances for a specific microservice. Some PaaS providers also offer automatic scaling, i.e. microservices are dynamically scaled depending on the current workload.

If we use automatic scaling, the degree of elasticity depends on the scaling rules we set. Otherwise, we must monitor the state of the application and try to achieve scalability and elasticity manually.

4.4.3 Y-axis scaling

Y-axis scaling is defined as the functional decomposition of an application into services. If we look back at section 4.1, we see that microservices represent this concept by definition. In section 4.3.1 we discussed the functional decomposition of IoT applications as well. In practice, a combination of both X-axis and Y-axis scaling is most commonly used for scaling of IoT applications. It is also the most popular scaling option for microservices applications overall.

4.4.4 Z-axis scaling

Z-axis scaling is based on the sharding principle. It is similar to X-axis scaling, except for one variation. An instance is responsible only for a subset of the requests or the data (in case the microservice uses and communicates with a database). A common use case where Z-axis scaling is applicable is division according to the user category. For example, requests of users who have payed for a service are routed to different, more powerful servers. Premium users usually have higher Service Level Agreements (SLA) than non-premium users. Therefore, their requests are not routed together with the requests of non-premium users, as higher performance must be ensured. Better performance is usually provided by less restrictive policy on the creation of new instances or redirection to more powerful instances. Premium users might also have access to additional services. For example, we might process and visualize data from additional sensors, available to premium users only. Therefore, they will be redirected to servers which are hosting these additional services. A component for routing might be implemented as an additional microservice. However, Z-axis scaling is adding complexity to an application and should be used only if it is crucial for user division or achievement of higher performances.

In figure 4.3 we can see the X-axis, Y-axis and Z-axis scaling in a practical example.

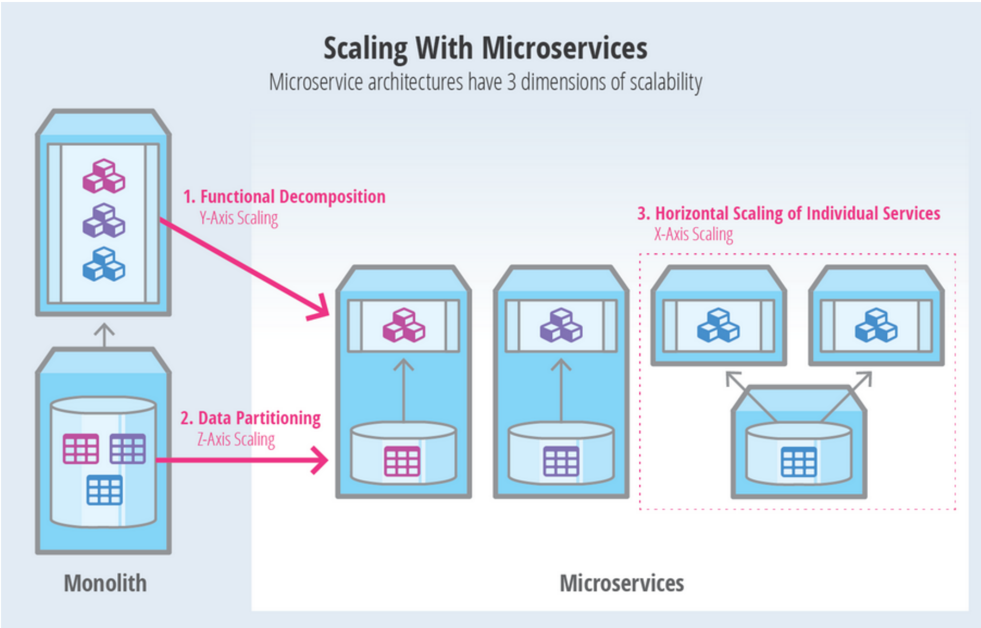


Figure 4.3: The dimensions of the scale cube in a practical example. We first componentize the application into microservices (Y-axis scaling). Which instance would handle a request depends on which subset of the data is needed (Z-axis scaling). Routing of requests in order to achieve user division is not illustrated in this figure. We then replicate each microservice according to our needs (X-axis scaling) [14].

Chapter 5

Complex event processing

Event-Driven architecture (EDA) is an architecture based on production, detection, consumption of and reaction to events [11]. CEP is a subset of EDA. CEP's task is to filter, aggregate and match low-level events to generate higher-level, i.e. complex events [46].

5.1 The need for CEP

We collect events (i.e. raw data) from IoT devices in order to turn them into useful information. One way to do that is to store the events in a database and subsequently use algorithms to get data-based insights. However, this approach has a significant shortcoming. If we store the data in a database in order to process it later, we lose the opportunity to analyze the data while it is still fresh. In most cases we want to turn the raw events from IoT devices into meaningful information in real time. We know that events in IoT networks are being created at unprecedented velocity. That is why the term Fast data is so popular nowadays. Even though we might use a database for near real-time event processing, it is not a favorable approach. The queries against a database are significantly affecting processing performance. This issue is even more obvious in IoT networks because of the high number of events and the high velocity of event creation. Therefore, this approach doesn't fit the needs of IoT networks.

Let's take the example of a fire detection system we mentioned in chapter 1. We want to have a notification of a fire as soon as it starts. Time in such situations is critical and the applicability of a fire detection system depends upon it. In fire detection systems there is no need to store any of the raw events at all. If a fire is detected, a complex event is fired. The complex event includes all the relevant information for the detected fire (e.g. location of the fire) [38]. Therefore, use of a database in such scenarios is redundant. We can conclude that a different approach is needed.

CEP helps us transform raw events into relevant information on-the-fly. IoT is not the only field where CEP is used. CEP can be used to analyze: streams of transactions to detect credit card frauds, financial markets to detect trends, network traffic to detect intrusions, etc [38]. In this chapter, we are going to describe its evolution, comparing it to its predecessors. Then, we will go through a list of the most common constructs and operators used in CEP languages. By using combinations of the constructs and the operators, we can analyze an event stream in real time and extract various information.

5.2 Evolution of information flow processing systems

The need to process huge amount of data in real time has resulted in development of systems for information flow processing (IFP). The general high-level structure of IFP systems is illustrated in figure 5.1.

Sources are entities that produce data. The data from the sources flows forward to the IFP engine. Sources might be various entities. In IoT networks, sources are the IoT devices, which produce events. Source might be other IFP engine as well. As the stream of data flows into the IFP engine, the data is being processed (e.g. filtered, aggregated, matched, etc.) according to the defined processing rules. Processing rules are added, managed and removed by rule managers. The output, produced by the IFP engine in

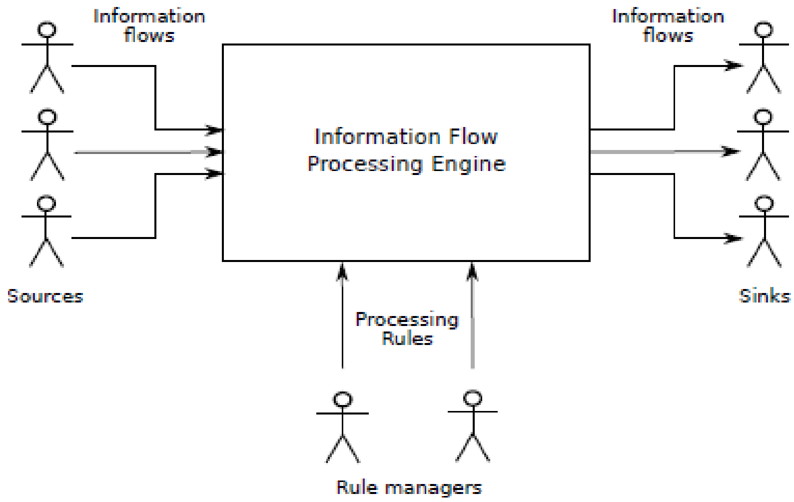


Figure 5.1: A high-level view of a typical IFP system. The data is produced by producers, i.e. sources. Then it is being processed according to the defined processing rules. The produced output flows forward and is transferred to the recipients, i.e. sinks [38].

accordance with the processing rules, is transferred to the sinks. Amongst them, there might be an IFP engine which does further processing of the output.

We can see that IFP systems must satisfy some key requirements of IoT networks. First, they must be able to process the data in real-time and be able to cope with enormous data amounts. Second, they must offer a language for data processing, which will enable us to specify complex relationships between the data in an efficient manner [38].

In the following subsections we present the evolution of IFP systems: active database systems, data stream management systems and the most popular — complex event processing systems.

5.2.1 Active database systems

We get data from the traditional database management systems (DBMS) in a Human-Active Database-Passive way, i.e. we write queries and get data. Active database systems complement DBMS, trying to move the active behavior on the database level instead. However, the basis for these systems is still persistent storage. That means they still have problems dealing with huge amounts of data and a big number of processing rules [38].

5.2.2 Data stream management systems

Data stream management systems (DSMS) try to overcome the issues that active database systems have. We can install continuous (standing) queries which are deployed on database systems and provide results until we remove them. Therefore, we get notifications as the stream is processed, without having to run many queries. We use these systems in a Human-Passive Database-Active way. However, DSMS are not able to use sequencing and ordering relations, as they are still a database extension [38].

5.2.3 Complex event processing systems

Unlike DSMS, which is a database extension, CEP is an extension of publish-subscribe messaging systems. In publish-subscribe systems, when we subscribe to a topic, we receive single events, as publish-subscribe systems don't maintain history of events and don't discover connections between them. CEP extends this behavior and enables us to subscribe to complex events. In order to offer us subscription to complex events, CEP addresses the main shortcomings of DSMS systems. DSMS systems don't have the ability to specify complex relationships between the data, as time-consuming operations such as ordering are very slow in a database extension. On the other hand, CEP systems can do complex pattern matching, filtering, aggregation, sequencing, ordering, etc. CEP gives us much more freedom in specifying complex processing rules. CEP tries to find a relationship between the incoming events according to the processing rules we defined. If it does find a relationship, a complex event is fired [38].

CEP may take into account event hierarchies as well. Complex events can be processed further and higher-level complex events may be generated. We can have as many levels of hierarchy as we want [46].

5.3 Constructs and operators of CEP languages

Almost every CEP engine has its own pattern-based event processing language. Pattern-based languages specify conditions and actions to be performed if the conditions are met. Conditions are usually represented by patterns, which process stream partitions using various constructs and operators. Transforming rules (e.g. produce two streams from one stream) are specific for transforming languages, which are mostly used in DSMSs. Modern CEP languages (e.g. Esper's Event Processing Language) enable us to specify transforming rules as well [38]. Most of the languages do not possess all of the language constructs and operators described in this section, but only a subset of them. We will try to cover the most common constructs and operators in CEP languages (adapted from [38]).

5.3.1 Single-item operators

Single-item operators are amongst the simplest operators in CEP systems. As their name suggests, they process single events independently, one after another. There are two classes of single-item operators:

Selection operators These are classic filtering operators, throwing away events that don't satisfy specified criteria. For instance, in a fire detection system, selection operators might filter events which indicate that the temperature is higher than a certain threshold.

Elaboration operators These operators are used for transformation of events. Some examples are projection and renaming operators. Projection operators extract only a part of the properties of the events (e.g. extract only the location of sensor readings). Renaming operators are used to change the name of an event property.

5.3.2 Logic operators

Logic operators are used for detection of event combinations. The order of events is not important for these operators. There are four groups of logic operators:

Conjunction A conjunction of events $E_1, E_2, ..., E_n$ means that all of the events $E_1, E_2, ..., E_n$ have occurred.

Disjunction A disjunction of events $E_1, E_2, ..., E_n$ means that any of the events $E_1, E_2, ..., E_n$ have occurred.

Repetition A repetition of event E with degree (m, n) means that the event has occurred between m and n times.

Negation A negation of event E means that the event E didn't occur.

We can combine these operators (e.g. use both conjunction and disjunction) to form more complex patterns. Also, logic operators are frequently used in combination with windows, which we cover in one of the following subsections. Windows are used in order to set boundaries, so that logic operators take into account only a partition of the stream (e.g. conjunction of two events in the last three seconds).

5.3.3 Sequences

Sequences are very similar to logic operators. The difference is that the order of events is vital for sequences. A conjunction sequence of events $E_1, E_2, ..., E_n$ means that all of the events $E_1, E_2, ..., E_n$ have occurred in the stated order.

5.3.4 Iterations

In iterations we specify an iteration condition. The sequences of events must satisfy the specified iteration condition. The ordering of events is important for iterations as well. Iterations are not included in all event processing languages because they can be expressed with other event processing constructs (e.g. combination of sequences and recursion).

5.3.5 Windows

Windows are language constructs which enable operators to process only a part of the event stream. They are frequently used with various operators. Their purpose depends on whether the operators are blocking or non-blocking. Blocking are the operators which have to consume the whole stream of events before they generate output (e.g. negations and repetitions with specified higher bound). On the other hand, non-blocking operators generate output as the stream of events flows and they can stop generating output as soon as they detect the needed events (e.g. conjunctive and disjunctive operators). Therefore, windows are vital for usage of blocking operators because they specify a finite part of a stream, so that blocking operators are able to produce output. As far as non-blocking operators are concerned, windows make them more powerful by specifying which part of the stream they should look at.

There are two types of windows classifications. According to the first one, windows are divided into logical (i.e. time-based, such as the events in the last three seconds) and physical (i.e. count-based, such as the last three events).

The second type of classification depends upon the way boundaries of a window move. According to this classification, windows can be:

Fixed windows Bounds of fixed windows do not move. We should specify both the lower and the upper bound.

Landmark windows The lower bound of landmark windows is specified in advance and does not move. The upper bound moves as new events enter the CEP engine.

Sliding windows They are the most frequently used type of windows. Like in landmark windows, the upper bound moves as events flow. In sliding windows, the lower bound moves with the same tempo as the upper bound. Thus, standard sliding windows enable us to look at the last three events or the events in the last three seconds, as the lower bound moves together with the upper bound. Sliding windows also have some variations (e.g. tumbling windows).

5.3.6 Stream management operators

As their name suggests, stream management operators are used to manage the streams. Stream management operators are:

Join The join operator is a blocking operator used to merge two streams (or parts of two streams) into one.

Bag operators Bag operators use the standard set operations to manipulate streams. The union operator merges many streams into one, which has unique events from all of the streams. The intersection operator merges many streams into one, which has events detected in each of the streams. The except operator accepts two streams and removes events from the first stream that are contained in the second stream. The remove-duplicate operator takes one stream and removes the duplicate events.

Duplicate It copies the same stream, so that a copy is provided to each distinctive sink (e.g. for further processing).

Group-by Groups events of a stream partition according to an event property.

Order-by Orders events of a stream partition according to an event property.

5.3.7 Parametrization

A language must be able to offer parametrization in one way or another. We often need to process events in one stream in relation with the events in other stream. For instance, in a fire detection system, we might produce a fire alert if we detect both high temperature and smoke. Therefore, we need to filter the temperature sensors' stream in correlation with the smoke sensors' stream. Different CEP languages handle this issue differently. Most of the modern languages offer parametrization through the conjunction operator. For instance, in order to detect a fire, we can specify that the temperature must be above 45°C *and* there must be smoke in the last 10 seconds. Then, the CEP engine does parametrization internally. In some older languages, we have to use the join operator and merge the streams.

5.3.8 Aggregates

Aggregates are used for aggregation of events in a stream partition. Typical examples of aggregates are minimum, maximum and average. There are two types of aggregates:

Detection aggregates They are used during the evaluation of the conditions of the processing rules. For instance, we might want to detect the events that are below the average temperature of the last ten sensor readings.

Production aggregates These aggregates are used for production of higher-level events. A simple example might be generation of events containing an average of the last ten temperature readings.

Besides the typical aggregates (minimum, maximum, etc.), some languages also enable us to define custom aggregates.

Chapter 6

A practical example

The complete source code of the practical example is available at `https://github.com/NaumGj/diploma`.

6.1 Goals of the practical example

Our goal is to build both a scalable and elastic IoT application. It must be able to consume high amount of IoT events and adjust the amount of allocated resources to the amount of data. It must be able to process the stream of events in real time, i.e. infer useful knowledge from it. At the end, it should visualize both some of the stream's properties and the inferred knowledge from the stream. In order to demonstrate this, we introduce three different scenarios, described in section 6.3.

6.2 Structure of the practical example

In this section we are going to make a high-level overview of the structure of the practical example. Every component of the practical example is described in detail in the following sections.

In chapter 4 we discussed the microservices architecture and the benefits arising from its adoption. Our IoT application follows the principles of the microservices architecture. It consists of six microservices:

Apache Kafka producer In our case, Kafka producers simulate an event stream. We need lots of devices to generate amounts of data such as the amounts generated in IoT networks. Since we don't have so many IoT devices, we will need to simulate an event stream. In order to do that, we use Kafka producers for event production.

Apache Kafka message broker Message brokers are delivering messages from the producers to the consumers.

Apache Kafka consumer Kafka consumers consume the messages produced by producers (through message brokers).

Apache ZooKeeper ZooKeeper is used by producers, message brokers and consumers for managing and sharing state [31]. It is also used by the CEP adapter. The CEP adapter gets events from the consumers through REST services. It uses ZooKeeper in order to discover every consumer instance and get all of the events.

Complex event processing microservice The CEP microservice consists of adapter, statements and listeners. The adapter sends events into the CEP engine. Statements process the events and produce output. The output is then handled by listeners. For CEP, we use the Esper engine for Java.

Visualization (front-end) web application The inferred knowledge from the events is visualized. The event throughput and the latency of events between Kafka producers and consumers are visualized as well.

The architecture of the IoT application is illustrated in figure 6.1. We can have one or many Kafka producers, simulating an event stream. These events are transferred to the consumers by message brokers. A Kafka message broker can transfer an enormous amount of events to many consumers. In most cases, we only need a few of them. However, that is not the case with the consumers. This is the bottleneck for achieving scalability and the key point for achieving elasticity. We should be able to change the number of consumers dynamically according to the amount of events. Message brokers and consumers use ZooKeeper to agree which consumers should read from a particular broker. Since every consumer receives unique events, the CEP adapter uses ZooKeeper to discover all of the consumers and receive events from all of them. This is particularly important because the number of consumers changes as we dynamically deploy and terminate consumer instances. The events from the adapter are processed by statements, whose output is then handled by the listeners. The visualization component communicates with the listeners through REST services and consumes the meaningful output produced by the statements. We can also check how our fire detection and shoplifting scenarios work, by sending custom events through front-end forms.

The visualization microservice is written using JavaScript, the AngularJS framework and HTML. We won't discuss its source code in the thesis, as it is not complex and every programmer who has used these technologies will find it easy to understand. We will only see how we used the forms and how the visualizations look like. All of the other microservices are written in Java EE and are using Maven for build automation. The KumuluzEE framework enables us to pack the Java EE components into microservices as standalone JARs. A Docker image is built for every microservice. The Docker images are then pushed to the Google Cloud Engine, where we can deploy, scale and manage our containerized microservices using Kubernetes.

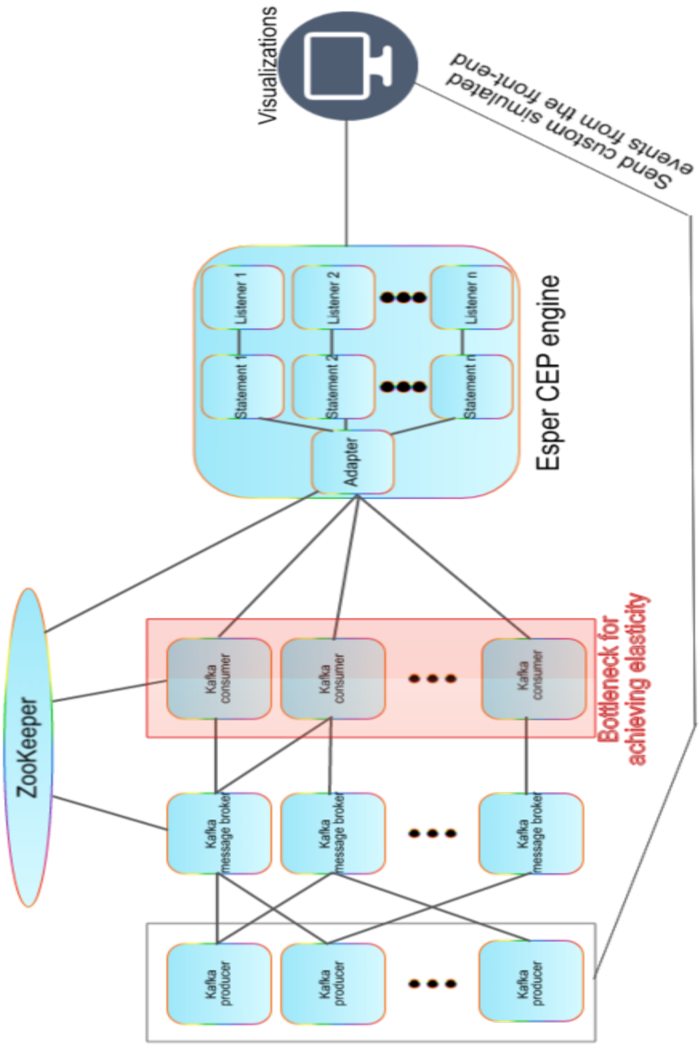


Figure 6.1: The architecture of our application consists of various components. Kafka producers simulate event production. The events are transferred to consumers through one or many message brokers. The adapter then collects the events and sends them to the CEP engine, which processes the events according to the processing rules (i.e. statements) and produces meaningful output. Then, the output is visualized in the front-end component of the application.

6.3 Scenarios

6.3.1 Test of the throughput and the latency of events in Apache Kafka

The goal of this scenario is to test the bottleneck for achieving scalability and elasticity. We use producers to generate events. As we dynamically deploy and terminate producer and consumer instances, we observe the number and latency of events in the last 10 seconds in real time. The latency is computed in the bottleneck only, i.e we measure the time from the production of an event in one of the producers to its arrival in one of the consumers. In order to get the throughput and the latency, we will create two Esper statements.

6.3.2 Fire detection

In this scenario, we try to detect a fire as soon as it breaks out. We receive data from two types of sensors: temperature and smoke sensors. We detect a fire if:

- The temperature is above 40 °C;
- The level of smoke is above 5 % obscuration per meter [1];
- The abnormal temperature and smoke sensor readings are coming from the same room;
- The abnormal temperature and smoke readings have both been detected in the last 25 seconds.

We can observe the active fires in the front-end component. Also, through forms on the front-end, we can send custom temperature and smoke sensor events to one of the producers using POST (REST) method. We should

[1] The unit indicating level of smoke depends on the type of the smoke detector. We obtained this unit from [32]. Although it might not be the most accurate one, the purpose of this scenario is to show how to combine data from different types of sensors.

specify the room number and the sensor reading value (temperature or level of smoke). This way, we can easily test the scenario.

6.3.3 Shoplifting

In this scenario, we demonstrate pattern matching. We receive 3 types of events: events indicating that a product is removed from the shelf, events indicating that a product is paid and events indicating that a product exits the store. We detect that a product is stolen if we have received an event indicating that the product is removed from the shelf and then receive another event indicating that the product exits the store, without receiving an event indicating that the product is paid. In the front-end component, we can observe products that are shoplifted. Also, we can send custom events indicating that a product is removed from shelf, is paid or exits the store.

6.4 Apache Kafka

6.4.1 Topics, partitions and replication factor

Apache Kafka is a distributed, partitioned, replicated commit log service [1]. In Kafka, we publish messages in categories called topics. Topics are divided into many partitions. The number of partitions is specified by us. The messages can be assigned to a partition based on some semantic properties of the message or simply in a round-robin fashion. Replication factor is the number of replicas of each partition. When the replication factor is bigger than one, the replicas of the partition are spread across different message brokers (if there are multiple message brokers). The replicas of a partition are completely identical, as their purpose is to make a topic resilient to broker fails. Data in a partition is lost only if all brokers holding replicas of a certain partition fail. In order for the replicas to be identical, a leader broker for each partition is elected. Replicas of a certain partition are synchronizing with the replica that is hosted by the leader of that partition [31]. The relation between topics, partitions and replicas is best illustrated in figure 6.2.

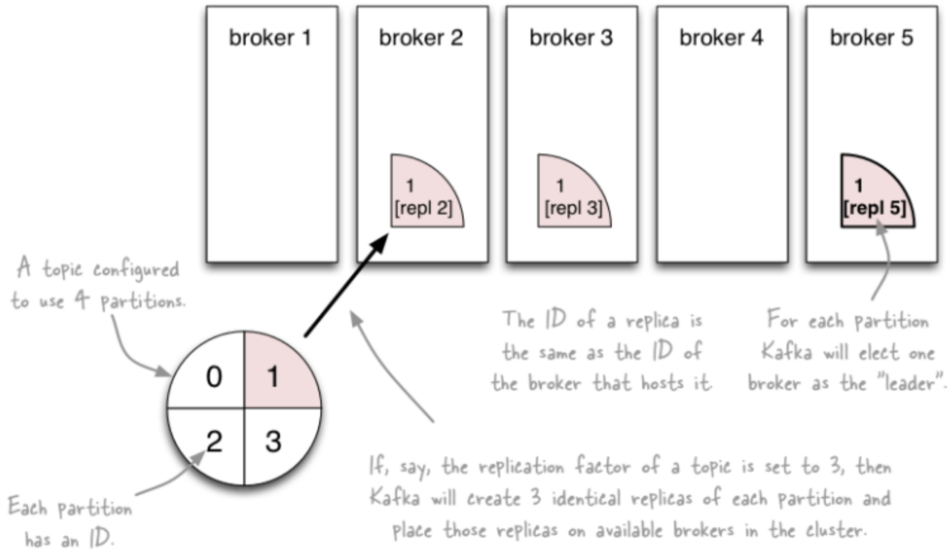


Figure 6.2: Topics, partitions and replication factor [31].

6.4.2 Consumers

In general, there are two messaging patterns: message queue and publish-subscribe. In the message queue pattern, a message is consumed by one of the consumers. In the publish-subscribe pattern, a message is received by all of the consumers that have subscribed to a topic. Kafka enables us to use both of these patterns by the consumer group concept, illustrated in figure 6.3.

In a consumer group partitions are consumed by one consumer only. If there is one consumer group only, we get a traditional message queue. This characteristic is particularly important for achieving scalability because the load is balanced among the consumers belonging to a certain consumer group. While a partition is consumed by one consumer only, the consumer can still consume many partitions at once. In this way, we use partitions to achieve parallelism and increase throughput. This means that the maximum number

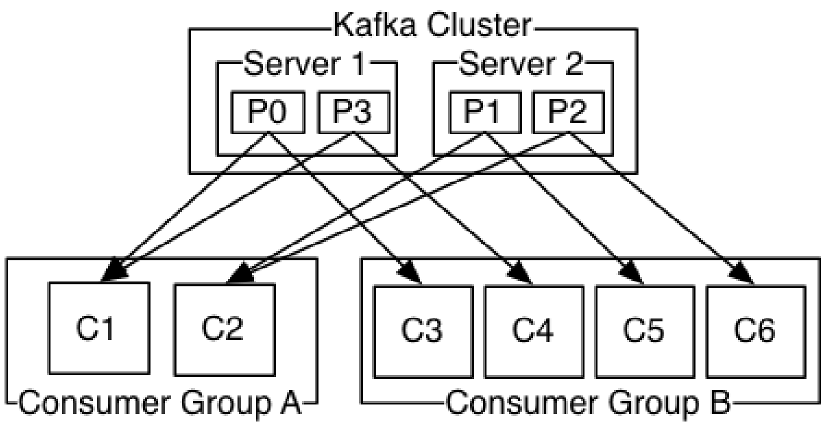


Figure 6.3: Kafka's consumer group concept. A consumer group consumes each of the partitions, but only one consumer in a consumer group consumes a certain partition [1].

of useful consumers is equal to the number of partitions of a topic. In that case, a consumer consumes messages from one partition only. On the other hand, if every consumer belongs to a separate consumer group, we get the publish-subscribe pattern. The consumer group concept enables combinations of the two messaging patterns as well. Such example is illustrated in figure 6.3.

6.4.3 Implementation of the producers

For the implementation of the producers and the consumers we used the Kafka API for Java, its documentation [1] and the help of a web article [16].

We have three different producers for the three different scenarios described in 6.3. Although we might have implemented one producer that generates all types of events, it seemed more natural to have separate producers. In this section, we describe only the producer of events for the event throughput and latency scenario. The other producers differ in the types of events they send. Also, the producers for the fire detection and shoplifting scenarios offer injection of custom events (from the front-end application) through REST services implemented using JAX-RS.

First, among the other needed dependencies, we should specify the dependencies for Kafka and KumuluzEE in module's *pom.xml* file:

```
<dependency>
    <groupId>com.kumuluz.ee</groupId>
    <artifactId>kumuluzee-core</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
</dependency>
```

KumuluzEE will pack our producer component in a microservice. The versions of the dependencies are specified in the parent, i.e. the root *pom.xml*. The producers are implemented as application scoped beans. The production of events is initialized by the following method:

```
public void init(@Observes @Initialized(
    ↪ ApplicationScoped.class) Object init) {
    initTimer();
}
```

The *initTimer()* method contains initialization logic. In this method we schedule the production of events. Before starting event production, we need to initialize the Kafka producer through the Kafka API. To do that, we placed a configuration file named *producer.props* in the *resources* map. The most important configuration setting is the specification of the *bootstrap.servers* property. Here we specify the IP addresses of the message brokers in a Kafka cluster and the ports they listen on. As we will see later, passing a fixed IP is not a problem thanks to the Kubernetes services. Once configured, the producer can start sending events:

```
producer.send(new ProducerRecord<String, String>("fast-
    ↪ messages", mapper.writeValueAsString(event)));
```

The first parameter (in our case "fast-messages") is the name of the topic we send messages to. The second parameter is an event serialized to the String class. Here, *mapper* is Jackson's Object Mapper, which helps us serialize and deserialize event classes to JSON or String and vice versa. Since we haven't specified to which partition the message should be sent, the producer will send our messages in a round-robin fashion.

6.4.4 Implementation of the consumer

Unlike the producers, we have one universal consumer which consumes each event type we have defined. We should include the same dependencies in the *pom.xml* file of the consumer module, as we did in the *pom.xml* files of the producers.

Similar to the producers, we first need to read consumer's configuration file (named *consumer.props*). We will have a look at the two most vital properties:

bootstrap.servers This property is having the same role as in the producer's configuration file.

group.id Here we specify the ID of the consumer group. Later, when we will deploy many consumer instances, they will belong to the same consumer group. That means the instances will balance the load among themselves in a message queue manner and enable us to achieve scalability.

Then, we should subscribe to the topics we want the consumer to consume (in our case the "fast-messages" topic):

```
consumer.subscribe(Arrays.asList("fast-messages"));
```

When we have subscribed to one or more topics, we can poll events from them:

```
ConsumerRecords<String, String> records = consumer.poll
    ↪ (200);
```

If the consumer doesn't receive any events, it sleeps for 200 milliseconds and tries to poll again. We can then iterate through the records and deserialize them:

```
SimpleEvent event =
    mapper.treeToValue(msg, SimpleEvent.class);
```

The events are added to an array which is emptied every time the CEP adapter consumes the events from it.

We can see above that we deserialize and convert each event type to the *SimpleEvent* class. We can do this thanks to Jackson's polymorphism. As we have mentioned before, we have a root event type, implemented in the *SimpleEvent* class which is located in the *module-models* Maven module. All of the other event types extend this class, inherit its properties and add new event-specific properties. Jackson deserializes each event to its actual type. It knows to which type an event should be deserialized because we registered each event type through the following annotations in the *SimpleEvent* class:

```
@JsonSubTypes({
    @JsonSubTypes.Type(
    value = TemperatureSensorEvent.class,
    name = "temperature"),
    @JsonSubTypes.Type(value = SmokeSensorEvent.class,
    name = "smoke"),
    @JsonSubTypes.Type(value = ShelfEvent.class,
    name = "shelf"),
    @JsonSubTypes.Type(value = PaidEvent.class,
    name = "paid"),
    @JsonSubTypes.Type(value = ExitEvent.class,
```

```
    name = "exit")
})
```

One more notable thing in the implementation of the consumer is the use of ZooKeeper. The consumer implements *@PostConstruct* and *@PreDestroy* methods, invoked just after the bean is constructed and before it is destroyed respectively. In the *@PostConstruct* method, the consumer registers itself as a service in ZooKeeper, while in the *@PreDestroy* method it unregisters the service. The CEP adapter uses ZooKeeper to discover each consumer instance which is currently deployed and consume events from each of them. The module implementing the communication with the ZooKeeper server (*module-utils*) is taken from [39].

6.5 Esper

In chapter 5 we discussed the need to analyze the events as the stream of events flows, instead of storing them in a database and then running queries against a database. In order to be able to analyze events with CEP techniques, we decided to use Esper.

Esper is a standalone tool for CEP and event series analysis. It is an open-source tool, which comes with API for Java and .NET. It is designed to enable complex computations in event-driven architectures in real time, while maintaining high throughput and low latency. It offers its own language for event processing called Event Processing Language (EPL). This language has most of the constructs and operators we discussed in section 5.3: aggregators, filtering operators, windows, etc. In general, Esper processes events through event patterns or event stream queries [9]. We will use both of them: an event pattern in the shoplifting scenario and event stream queries in the other scenarios.

A simple CEP architecture is shown in figure 6.4. The adapter sends raw events to the CEP engine, which processes them using a user-defined statement. The statement's output are complex events, which are transferred to the listener. This is a traditional CEP architecture in its simplest form. We can then add many statements vertically or horizontally. For instance, we might have many parallel statements or the output of one statement is then the input of another statement. Such situations are demonstrated in our scenarios as well. Also, a statement might have many update listeners or have no update listener at all. For instance, one listener of a statement might do logging only, while another one might offer the complex events as REST services.

Figure 6.4: Simple CEP architecture. The raw events from the adapter are processed using statements. The output of the statements is then reported to listeners.

6.5.1 Implementation of the adapter

The microservice for CEP is having a common *pom.xml* file for the adapter, the statements and the listeners. In addition to the Esper dependency, we should also include some additional dependencies: ANTLR, CGLIB and Apache Commons logging. These libraries are required by Esper. More information on their usage is provided in [9].

In the initialization part, we should first define all of the event types. For example, the event generated by smoke sensors is added with the following code:

```
configuration.addEventType("SmokeEvent",
    ↪ SmokeSensorEvent.class.getName());
```

The other event types are added in the same way. Then, we get an instance of Esper's service provider:

```
EPServiceProvider epService = EPServiceProviderManager.
    ↪ getProvider("RestAdapter", configuration);
```

With the help of Esper's service provider, we can now register our statements and attach listeners to them. Here is an example of the registration of the statement that computes latency and the listener that receives updates from the statement:

```
LatencyStatement latencyStmt = new LatencyStatement(
    ↪ epService.getEPAdministrator());
latencyStmt.addListener(new LatencyListener());
```

Once we have initialized Esper, we can schedule the adapter to call the REST services offered by the consumers:

```
ScheduledFuture<?> runnableHandle = scheduler.
    ↪ scheduleAtFixedRate(runnable, 5000, 500, TimeUnit
    ↪ .MILLISECONDS);
```

After the initial delay of 5000 milliseconds, the REST services are called every 500 milliseconds. As we said, the adapter uses ZooKeeper to discover every consumer instance and get its events through GET requests. The instance of ZooKeeper's service registry is injected in the adapter's bean through Contexts and Dependency Injection (CDI).

When the adapter receives some events, we can send each of them to the CEP engine:

```
epService.getEPRuntime().sendEvent(event);
```

6.5.2 Implementation of the statements

The creation of statements in Esper is pretty straightforward. Thus, the four statements we have are created in this manner:

```
public Statement(EPAdministrator admin) {
    String stmt = "<The statement>";
    EPStatement statement = admin.createEPL(stmt);
}
```

Hereinafter we will discuss the content of every statement.

Number of events in the last 10 seconds

In Esper, we can almost always get the desired results in many ways. This is also the case with the computation of the number of events in the last 10 seconds. In our application, we have first registered the following context:

```
create context Ctx10Seconds initiated @now and
pattern [every timer:interval(10)]
terminated after 10 sec
```

That means Esper will do an action every 10 seconds, without keeping events in memory. Then, we can get the actual number of events with the following statement:

```
context Ctx10Seconds select count(*) as cnt
from SimpleEvent output snapshot when terminated
```

In this statement, we used Esper's *count* aggregation function. The output is sent to the listener when the pattern terminates, i.e. every 10 seconds.

We could have adopted a simpler approach and could have used a sliding window:

```
select count(*) as cnt
from SimpleEvent.win:time(10 sec)
```

However, this statement would produce output every time an event enters or leaves the sliding window. Furthermore, Esper would keep all of the events that arrived in the last 10 seconds in its memory.

Using a batch looks simpler than our solution as well:

```
select count(*) as cnt
from SimpleEvent.win:time_batch(10 sec)
```

According to this statement, Esper would accumulate events and send the output to the listener every ten seconds, just like above. However, it would still keep events in memory. Therefore, the solution we chose fits our needs better than the other solutions [10].

Average latency (between the producers and the consumers) in the last 10 seconds

All of the events are timestamped when they are sent from one of the producers and when they arrive at one of the consumers. Since each event has these two properties, we can easily calculate the average latency of the events that arrived into the CEP engine in the last 10 seconds:

```
context Ctx10Seconds
select avg(timestampConsumed - timestamp) as avgLatency
from SimpleEvent output snapshot when terminated
```

Note that we use the same context we already defined to count the number of events. Esper can register a given context only once and we mustn't redefine it.

Fire detection

In section 6.3.2, we described the fire detection scenario and the conditions that need to be met for a fire to be detected. At first, we might go straightforward and define the following statement:

```
select temperature.roomNumber as room,
temperature.celsiusTemperature as temp,
smoke.obscuration as obscur
from TemperatureEvent.win:time(25 sec) as temperature,
SmokeEvent.win:time(25 sec) as smoke
```

```
where temperature.roomNumber = smoke.roomNumber
and temperature.celsiusTemperature > 40.0
and smoke.obscuration > 5.0
```

The statement actually does detect fire. However, there is one problem. For example, if the CEP engine receives two abnormal temperature and two abnormal smoke events, it outputs the four possible combinations of these events, i.e. we get four complex events. When a fire really breaks out, we might get thousands of abnormal events in 25 seconds. So, if we have T abnormal temperature events and S abnormal smoke events in the last 25 seconds, the output would be $T * S$ complex events. We can see that a different approach is needed.

Granularity of statements is recommended in Esper's documentation. If a complex statement can be partitioned into many smaller and more comprehensible statements, we should do that. Therefore, we will define three statements. As the events flow, the first statement inserts the abnormal temperature events into another stream, called *HighTemperature*:

```
insert into HighTemperature
select * from TemperatureEvent as temperature
where temperature.celsiusTemperature > 40.0
```

Similarly, the second one inserts the abnormal smoke events in a separate stream, called *LotSmoke*:

```
insert into LotSmoke
select * from SmokeEvent as smoke
where smoke.obscuration > 5.0
```

At the end, we combine the two new streams in order to detect a fire:

```
select temperature.roomNumber as room,
temperature.celsiusTemperature as temp,
smoke.obscuration as obscur from
```

```
HighTemperature.std:groupwin(roomNumber).win:time(25
    ↪ sec) as temperature,
LotSmoke.std:groupwin(roomNumber).win:time(25 sec) as
    ↪ smoke
where temperature.roomNumber = smoke.roomNumber
```

In this statement, we take advantage of Esper's *std:groupwin* function. It enables us to create sub-views [9]. By specifying the room number as a key, we will get only the last abnormal temperature and smoke events from a specific room for the last 25 seconds. If there are no abnormal events for a certain room in the last 25 seconds, no events are taken into account for that room. In this way, only combinations of the last abnormal events in a certain room are created.

Shoplifting detection

In section 6.3.3, we described the shoplifting scenario. In this scenario, we will demonstrate the use of pattern matching. We detect that a product is stolen by the following statement:

```
select * from pattern
[every s=ShelfEvent ->
(ExitEvent(productId = s.productId)
and not PaidEvent(productId = s.productId))
where timer:within(12 hours)]
```

We detect every event where a product is removed from the shelf, which is then followed by an event indicating that the product exits the store without being paid. We keep events for the last 12 hours.

6.5.3 Implementation of the listeners

The listeners are similar to each other. In our case, they all gather the complex events produced by the statements and offer them to the front-end component through REST services. For example, the fire detection listener adds the active fires in an array. Then, we inject the listener and offer its events through REST services:

```
@RequestScoped
@Path("fires")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class RestServiceFires {

    @Inject
    private FireDetectionListener fireListener;

    @GET
    public Response getFires(){

        List<FireObject> fires = fireListener.getFires
           ↪ ();

        return Response.ok().entity(fires).build();
    }

}
```

We must also take care of Cross Origin Resource Sharing (CORS), so that we can access the REST services from the front-end. This is done by specifying a CORS filter in the *webapp/WEB-INF/web.xml* file located in the *resources* map.

6.6 Deployment and management of the microservices using Kubernetes

6.6.1 Project setup and build of images

The process of deployment and management of microservices with Kubernetes is adapted to our needs from the tutorial [27] and the documentation [20] of Kubernetes.

In order to be able to use the Google Cloud Platform, we must first create a Google account. For the purposes of this thesis, we used the *Google Cloud SDK* command-line interface. Also, we had to install *Docker* and the *kubectl* Kubernetes component, as described in [27].

An illustration of what we should achieve is displayed in figure 6.5. We must build a Docker image for each of our microservices or use a ready prebuilt Docker image from the Docker Hub. Then, we will deploy these images in a Kubernetes cluster, where we can manage and replicate them as needed. We explain how to do this in the rest of the section.

We use our Google account to log into Google's cloud console (`https://console.cloud.google.com/`). Then, we create a new Google Cloud Platform project and we assign a unique ID to it, as shown in figure 6.6. The ID is particularly important as we will be using it to push and deploy our microservices in the cloud.

Docker helps us containerize our microservices. In order to be able to build Docker images, we should start a Docker machine. We use Google Cloud SDK to run Docker commands. However, Docker commands can be run in other shells as well (e.g. Docker Quickstart Terminal). If a Docker machine is not created, we should first create it, as described in [15]. To start a Docker machine named *default*, we should run the following command:

```
docker-machine start default
```

In order to get and run the commands to set up the Docker environment for the machine named *default*, we run the following command [5]:

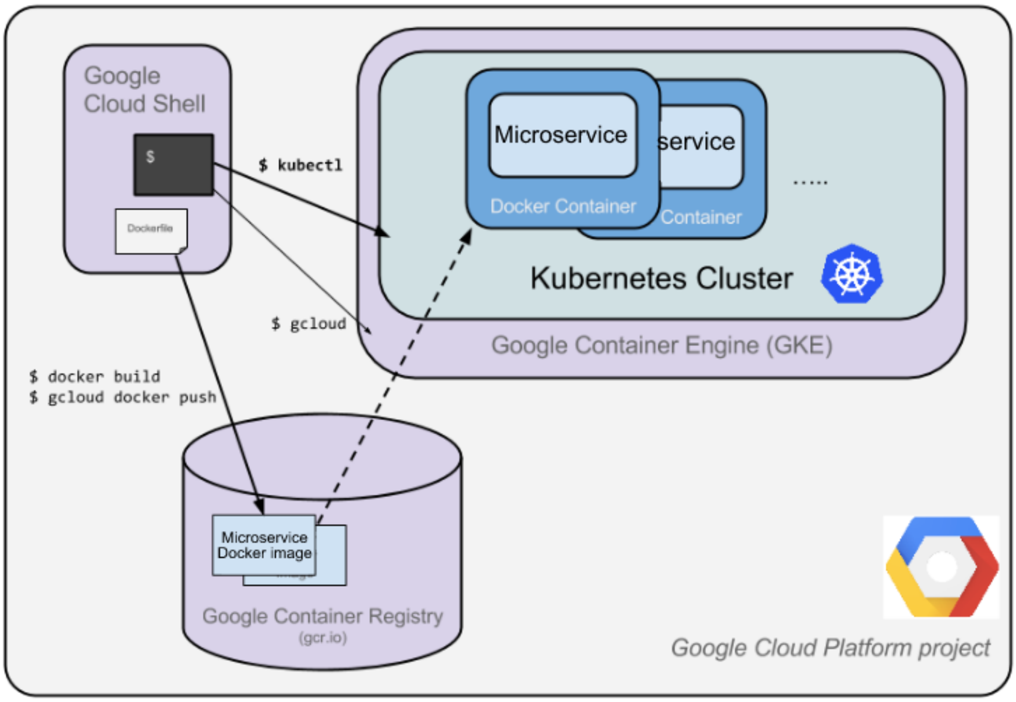


Figure 6.5: The process of deployment of our microservices in a Kubernetes cluster. We should build an image or use a prebuilt image for each of our microservices. Then, each image is pushed to the Google Container Registry. We can then manage and scale the containerized images in the Google Container Engine using Kubernetes [27].

```
docker-machine env default
```

At this point, we have a configured Docker machine and we can build images for our microservices. In order to build images automatically, we need a Dockerfile, which is actually "a Docker script". It contains the commands we could have run directly, one by one, in a command line interface [8]. The Dockerfile shown below is taken from [39] and is similar for each microservice, except for the front-end. The Dockerfile in the *module-producer* map that should automate the build of our producer looks like this:

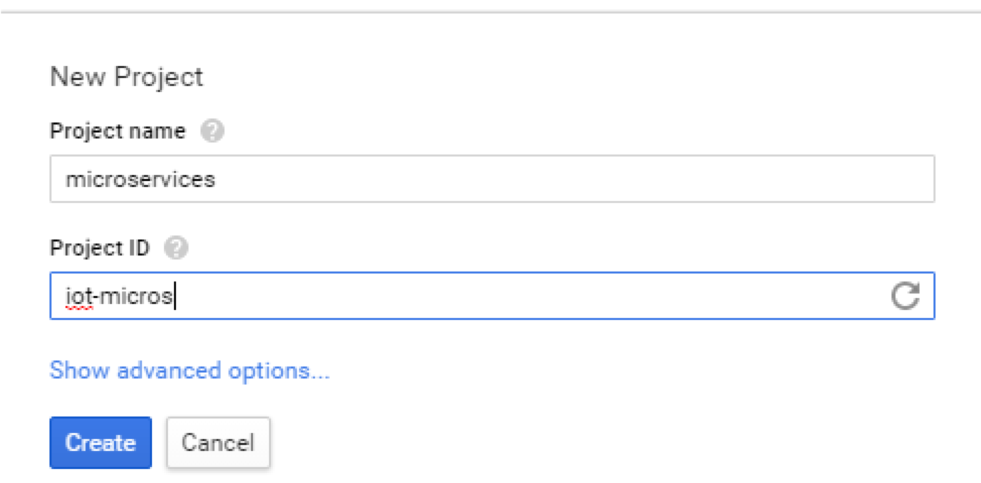


Figure 6.6: Creation of a new project in Google's cloud console.

```
FROM java:openjdk-8u45-jdk
MAINTAINER tilen.faganel@me.com
RUN apt-get update -qq && apt-get install -y wget git
RUN wget http://mirrors.sonic.net/apache/maven/maven
    ↪ -3/3.3.9/binaries/apache-maven-3.3.9-bin.tar.gz
    ↪ && \
        tar -zxf apache-maven-3.3.9-bin.tar.gz && rm
            ↪ apache-maven-3.3.9-bin.tar.gz && \
        mv apache-maven-3.3.9 /usr/local && ln -s /usr/
            ↪ local/apache-maven-3.3.9/bin/mvn /usr/bin/
            ↪ mvn
RUN mkdir /app
WORKDIR /app
ADD . /app
RUN mvn clean package
ENV JAVA_ENV=PRODUCTION
EXPOSE 8080
```

```
CMD ["java", "-server", "-cp", "module-producer/target/
    ↪ classes:module-producer/target/dependency/*", "
    ↪ com.kumuluz.ee.EeApplication"]
```

We take the Java JDK image and install git and Maven. Then, we add the application in the *app* folder. Maven cleans, compiles and packages our project. Before the final CMD command, we expose port 8080. We can have only one CMD command in a Dockerfile. In our case, we run our microservice, which is packaged by KumuluzEE. Dockerfiles for other microservices differ only in the last line, where the respective module is specified instead of the *module-producer* module.

The Dockerfile for the front-end microservice looks differently:

```
FROM ubuntu
RUN apt-get update -qq
RUN apt-get install -y build-essential nodejs npm
    ↪ nodejs-legacy vim
RUN mkdir /myangularapp
ADD www /myangularapp
WORKDIR /myangularapp
RUN npm install -g http-server
EXPOSE 8080

CMD ["http-server"]
```

It starts from the latest Ubuntu image, installs *npm* and its lightweight *http-server*, exposes port 8080 and runs the HTTP server.

Since we have the Dockerfiles, we can finally build our images. We should build six images: three for the producers (one for each scenario), one for the consumer, one for CEP and one for the front-end. In order to deploy Apache ZooKeeper and Apache Kafka message brokers, we will use existing images from the Docker Hub, as we will see later. Therefore, we won't build these images ourselves. We run the following command six times to build the six images we need:

```
docker build -t gcr.io/iot-micros/<name-of-the-
    ↪ microservice> -f <map-containing-the-Dockerfile-
    ↪ for-the-microservice>/Dockerfile .
```

The *-t* parameter specifies the image tag, while the *-f* parameter specifies the location of the Dockerfile. Note that *iot-micros* is the ID of the Google Cloud Platform project we created. The name of a microservice is specified by us. The names should be unique within a project. Once we have built the images, we can push all of them to the Google Container Registry (we use the tag that we have specified in the previous step):

```
gcloud docker push gcr.io/iot-micros/<name-of-the-
    ↪ microservice>
```

Then, we should create a Kubernetes cluster. This can be done either through the console with the *gcloud* command, as described in [27], or through the GUI in Google's cloud console (in the *Container Engine* section). Amongst other parameters, we specify the name of the cluster, the number of nodes and the machine types in the cluster. When the cluster is created, we should get credentials to be able to access and manage it from the Google Cloud SDK:

```
gcloud container clusters get-credentials <cluster-name
    ↪ >
```

6.6.2 Pods, replication controllers and services in Kubernetes

Before we deploy our microservices, let's take a look at the building blocks of Kubernetes that we need. As described in the Kubernetes documentation [22], pods are the smallest deployable unit in Kubernetes. They can contain one or more containers. However, if we put many containers in a pod, we must scale them together. Therefore, we decided to have one container in each of our pods. However, if deploy our applications as "naked" pods, in case of a pod, node or some other kind of failure, the pod is not redeployed. That is where replication controllers come in handy [23]. In replication controllers, we can specify the number of pod replicas we want to maintain. If a pod fails for some reason, a new pod is automatically deployed.

Every pod has its own IP address, visible only in the cluster. It is also possible to expose the pod and get an external IP and external access to the pod. The ZooKeeper server and the Kafka brokers run in pods. In order to connect to ZooKeeper or the brokers, we need their IPs. The brokers and the microservice for CEP need the IP of ZooKeeper, while the producer and the consumer microservices need the IPs of the brokers. However, we mentioned that a pod might fail. The replication controller ensures that a number of pod replicas are running and starts a new pod if some pod fails. The newly created pod would have a different IP from the pod that failed. A good practice in order to connect to ZooKeeper and the brokers is to use services. Kubernetes services expose a set of pods using a policy we define. Services get a stable IP, regardless of changes in the set of pods [26].

To run Apache Kafka and Apache ZooKeeper in a Kubernetes cluster, we used the service and the replication controller files from an open-source repository [2]. In these replication controllers, ZooKeeper and the brokers are deployed from existing images in the Docker Hub [6] [7]. We will also have both a service and a replication controller for our CEP and front-end microservices. The consumer doesn't need a service, since the CEP adapter implements custom logic to discover all of the consumer instances programatically (still through ZooKeeper, as we explained at the end of section 6.4.4). Therefore, it will only have a replication controller. As far as producers are concerned, we could have written replication controllers for them as well. However, it is sufficient to deploy them as "naked" pods since we deploy and terminate them regularly in order to test how our application works. In order to inject custom events from the front-end, we also expose the producers for the fire detection and the shoplifting scenarios, as shown later. All service and replication controller files are located in the *Services and Replication Controllers* map of our GitHub project. They are implemented similarly as the replication controllers and services in [2]. We should have a look at consumer's replication controller, as it is a little specific:

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    app: kafka-entities
    component: consumer
  name: consumer
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: kafka-entities
        component: consumer
    spec:
      containers:
      - name: consumer
        image: gcr.io/iot-micros/consumer:latest
        env:
          - name: MY_POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP
```

The *kind* object tells Kubernetes to which building block this file refers to. In the *metadata* object we specify the name of the replication controller and the *app* and *component* objects, used for targeting by other resources. Then, in the *spec* object we specify the number of replicas we want to maintain. The *spec* object nested in the *template* object specifies the pod. It tells Kubernetes it should build the pod from the latest version of the *gcr.io/iot-micros/consumer* image. The name of the image is actually the tag that we used in order to build the image and then push it to the Google Container Registry [24]. To this point, the other replication controllers are similar. The environment variable *MY_POD_IP* is specific for the consumer replication controller. It contains the IP of the pod in which an instance of the consumer is deployed. We remember that the consumer has a *@PostConstruct* method, in which it should register itself in ZooKeeper, so that the CEP microservice can discover every available consumer instance. What the consumer actually registers, is the IP of the pod in which the consumer instance is deployed. In the *@PostConstruct* method, we take the value of this environment variable and store it in ZooKeeper:

```
String endpointURI = System.getenv("MY_POD_IP");
// store the endpointURI in ZooKeeper
```

Finally, let's deploy our microservices. We should first deploy the ZooKeeper server and an Apache Kafka broker. Then, we can deploy the CEP and front-end microservices. In order to do that, we move to the *Services and Replication Controllers* map and run the following commands for each of the four mentioned microservices:

```
kubectl create -f <service-file>
kubectl create -f <replication-controller-file>
```

We can then deploy a consumer instance. We run only the second command, as the consumer doesn't have a service file. At the end, we can deploy a producer as a "naked" pod. We will show how to deploy and expose the producer which is generating events for the fire detection scenario. The procedure is the same for the other two producers. Assuming the image of the producer is called *gcr.io/iot-micros/fire-producer*, we run the following commands:

```
kubectl run fire-producer --image=gcr.io/iot-micros/
    ↪ fire-producer
kubectl expose deployment fire-producer --type="
    ↪ LoadBalancer" --port=8080
```

We are exposing the producer as a load balancer, since it is not important in which producer instance (in case we deploy many producer instances) we inject the custom events from the front-end. Instead of exposing it in this way, we can also build a service from a file, as we did for some of the other microservices.

6.6.3 The scenarios and the front-end

In the previous section, we created a service for the front-end component. The following command would produce a list of the services we created:

```
kubectl get services
```

In the output of this command, we can see the service for our front-end microservice and we can read its external IP. We can then access the front-end in a browser at *<external-ip>:8080*.

Test of the throughput and the latency of events

The front-end offers us visualizations of the number and the latency of events in the last 10 seconds. We will see how to scale the consumer both manually and automatically. First, we will scale the consumer manually, so that we can see better how the number of consumer instances affects the scalability of our application. Then, we will automate this process and the consumer instances will be deployed or terminated automatically, depending on the current load.

We can scale the consumer and the producer manually by running the respective command:

```
kubectl scale rc consumer --replicas=<number-of-
    ↪ instances>
kubectl scale deployment <name-of-the-producer> --
    ↪ replicas=<number-of-instances>
```

An alternative way to scale the consumer would be to change the number of replicas in the replication controller file and then run a rolling update, as described in the Kubernetes documentation [25].

In figure 6.7, we can see the front-end view. The red arrows represent a change in the number of consumer instances, i.e. we deployed new or terminated some of the existing consumer instances. The blue arrows represent a change in the number of producer instances. Above the arrows, we can see the number of instances for the respective deployment. The black arrows represent that around that point of time the Kafka broker was rebalancing the partitions of the topic, consequently causing a bigger latency. So, the topic to which we were sending events has six partitions. This is specified in the replication controller of the message broker. Since we wanted to test the bottleneck for achieving scalability and elasticity, we wanted to keep the other things simple. We have deployed one message broker. The partitions of the topic were not replicated, i.e. they had a replication factor of one. We started with three consumers and one producer. We tried to achieve elasticity, so we destroyed two consumer instances. Meanwhile, we launched one more producer instance. The first peak happened when we launched an additional producer. At the same moment, the broker was rebalancing the partitions of the topic and that caused the latency to grow. At that point, the broker assigned all of the six partitions to our single consumer. Then, we launched a third producer. Since one consumer couldn't handle the large amount of events, the latency started to grow. Therefore, we quickly deployed two more consumer instances in order to handle the increased load and achieve scalability. We get the second peak as a new rebalance of partitions was needed. This time, each consumer instance got two partitions of the topic. However, we weren't sure that we have achieved elasticity. Therefore, when the latency decreased, we destroyed one consumer instance. After the third rebalance, where each of our two consumer instances got three partitions to consume, we see that the latency decreases again. This time, we have achieved both scalability and elasticity for the current load of events. As we can see on the throughput graph below, the number of received events increased each time we have deployed a new producer.

Now that we have seen how the number of consumer instances affects the scalability of our application, we can automate the process of deployment and termination of consumer instances. Assuming the name of the consumer replication controller is *consumer* (it is specified in the replication controller file of the consumer), we can run the following command:

```
kubectl autoscale rc consumer --max=6 --cpu-percent=85
```

Since the minimum number of instances is by default one, we don't have to specify it explicitly. We specify that the maximum number of consumer instances is six, as our topic has six partitions. Adding more consumers wouldn't help us, since our consumers follow the message queue pattern (each partition is consumed by one consumer at a time). We also specify that a new instance should be deployed, if the average CPU utilization in our consumer pods exceeds 85% [21]. In general, the formula Kubernetes uses to calculate the target number of pods at a certain moment is the following [19]:

$$TargetNumberOfPods = \\ ceil(sum(CurrentPodsCPUUtilization)/TargetCPUUtilization) \quad (6.1)$$

So, our consumer will be scaled automatically depending on the current load. In order to see the current autoscalers and the average CPU utilization for each of them we can run the following command:

```
kubectl get hpa
```

A sample output is shown in figure 6.8.

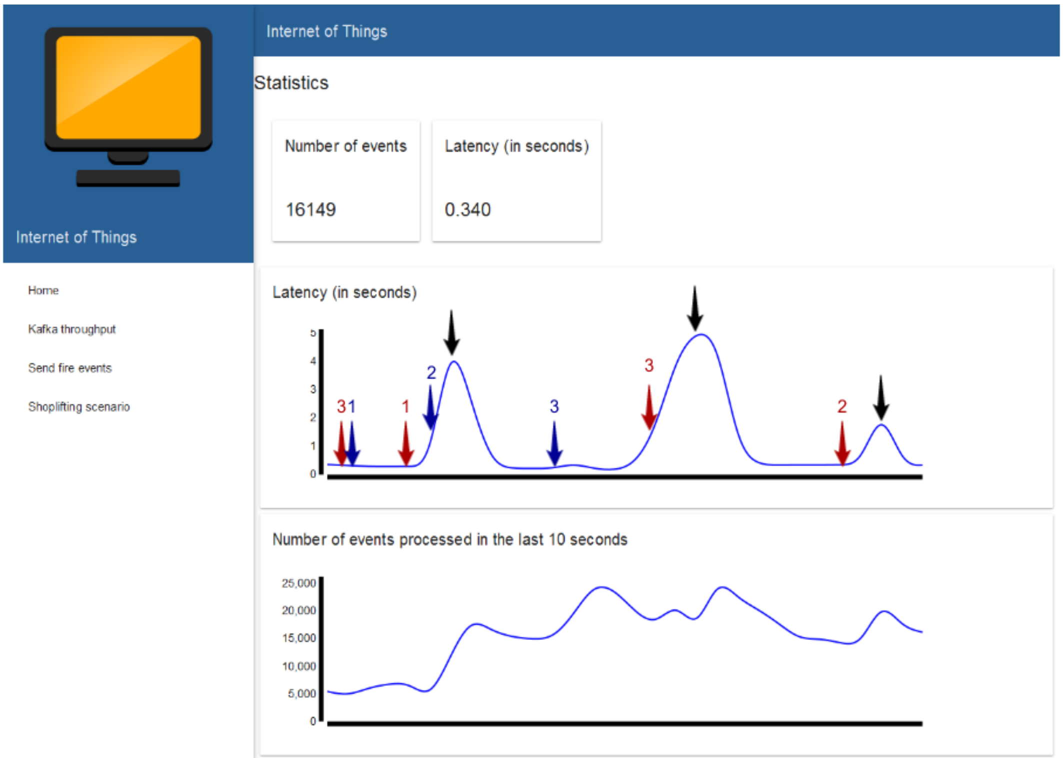


Figure 6.7: Front-end view of the test of throughput and latency of events between the producers and the consumers. The arrows are added additionally and are not part of the visualization. The red arrows represent a change in the number of consumer instances. The blue arrows represent a change in the number of producer instances. Above the arrows we can see the current number of consumer or producer instances. The black arrows represent a rebalance of the partitions, which occurs shortly after new consumer instances are deployed or some of the existing consumer instances are terminated.

```
NAME        REFERENCE                          TARGET    CURRENT   MINPODS   MAXPOD
S   AGE
consumer    ReplicationController/consumer     85%       70%       1         6
    4m
```

Figure 6.8: A sample output of the command *kubectl get hpa*.

In order to see the current number of consumer instances (assuming that the replication controller is named *consumer*), we run the following command:

```
kubectl get rc consumer
```

We tested how autoscaling works in our application. We deployed two producer instances, one by one. Since the CPU utilization got too high, our autoscaler deployed two additional consumer instances, as shown in figure 6.9. In order to check the number of consumer instances we used the above-mentioned command. Let us note that sometimes we have to wait for a few minutes for the CPU utilization to stabilize. Therefore, after the deployment of the second producer, we allowed the CPU utilization to stabilize. The autoscaler maintained three consumer instances, which were capable of handling the load from the two producers. Our application maintained low and constant latency of the events coming from the two producer instances. At the same time, the current CPU utilization didn't allow the autoscaler to try to terminate one consumer instance and scale down.

```
C:\Users\Naum\workspaceEE\iot\Services and Replication Controllers>kubectl get r
c consumer
NAME        DESIRED   CURRENT   AGE
consumer    3         3         14m
```

Figure 6.9: In order to handle the amount of events from two producer instances, the autoscaler maintained three consumer instances.

So, the autoscaler maintained sufficient number of consumer instances and achieved scalability. Then, we decided to terminate one producer instance in order to see whether the autoscaler will try to achieve elasticity. We checked the number of consumer instances regularly and after a short time the autoscaler terminated one of the three consumer instances. The front-end view is shown in figure 6.10. We can see that the latency is constant, except for the time when one of the consumer instances was terminated and a rebalance of the partitions was needed.

Then, we terminated the one producer instance that was left. The autoscaler terminated one consumer instance as well, so that it maintained only a minimum of one consumer instance.

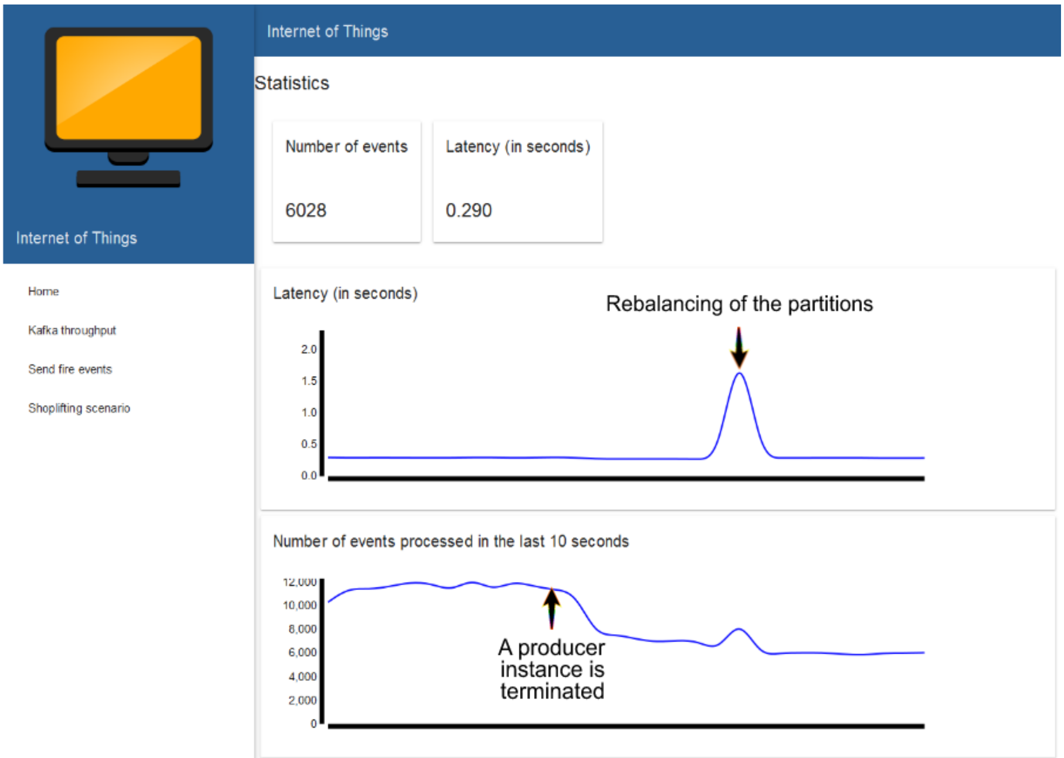


Figure 6.10: After one producer instance was terminated, the autoscaler terminated one consumer instance as well. When the consumer instance was terminated, the broker had to do rebalancing. At that point, we see a peak in the latency of events. The arrows and the text above and beyond them are added additionally.

Fire detection

We can send both temperature and smoke sensor events through the front-end, by specifying the room number and the respective measurement. Below we get real-time notifications of the active fires.

The front-end view for the fire detection scenario is shown in figure 6.11. For the rooms with IDs 3 and 7, we have sent events that contain high temperature measurements and then events that contain a high level of smoke. Since the time between the respective temperature and smoke sensor events was less than 25 seconds, we got a fire alert for these two rooms. For example, if we have sent an additional event for room 7, indicating a temperature of 60 °C, the temperature field in the existing notification would have changed.

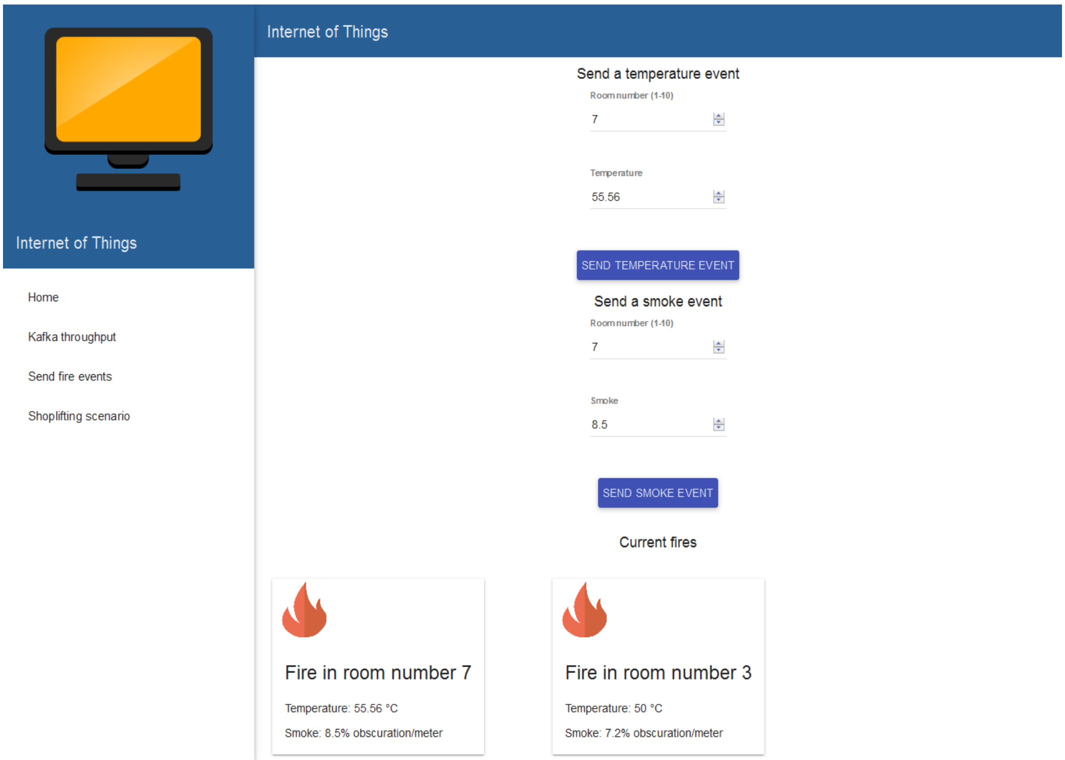


Figure 6.11: Front-end view of the fire detection scenario. Through the forms, we can send custom temperature and smoke sensor events. At the bottom, we can see the current active fires.

Shoplifting

Through the front-end, we can also send each of the event types for the shoplifting scenario, by specifying the ID of a product. Below we get real-time notifications of the stolen products. In figure 6.12, we can see how this looks. We have sent an event that the product with id *perfume-12345* is removed from the shelf and an event indicating that the same product exits the store. Because we didn't send an event that the product is paid, we got a real-time warning that the product is stolen.

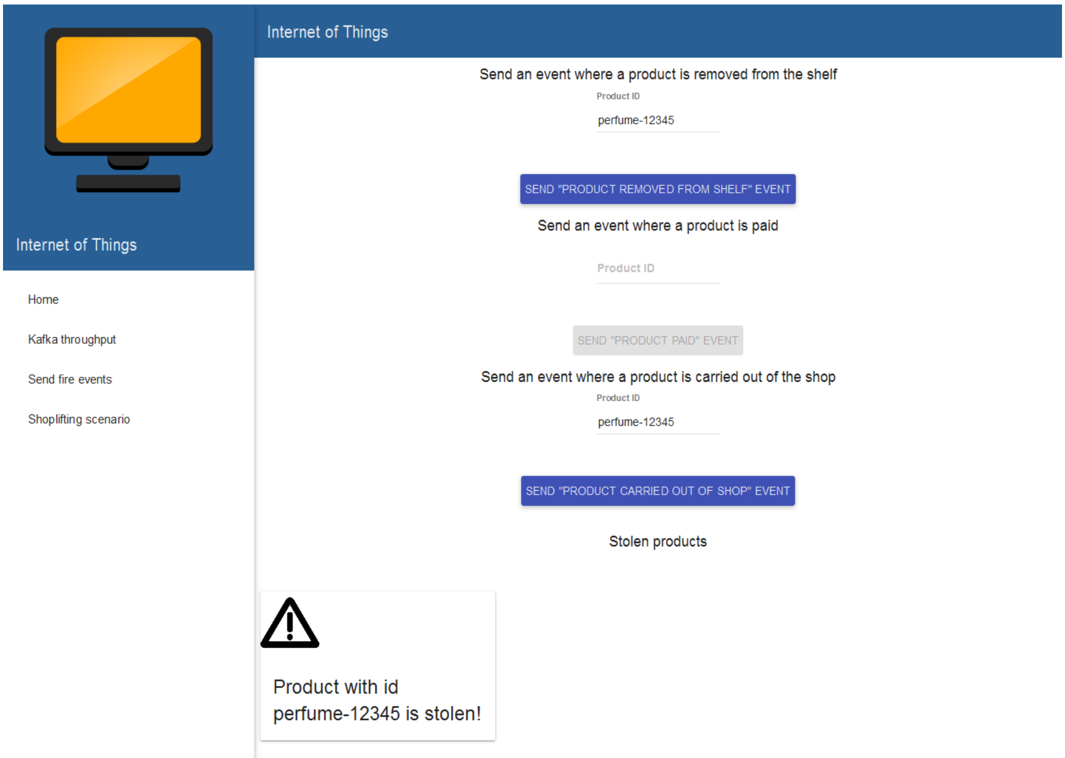


Figure 6.12: Front-end view of the shoplifting scenario. Through the forms, we can send each of the event types for the shoplifting scenario. At the bottom, we can see the stolen products.

Chapter 7

Conclusion

IoT is a promising technology that may change the way we live. Since it is relatively new, it faces many challenges. In order to realize the IoT vision, achieving scalability is one of the key challenges to overcome. We can achieve scalability if we integrate IoT solutions and cloud computing. Cloud computing offers virtually unlimited storage and computational resources and is a perfect fit for IoT applications.

Throughout the thesis, we saw that not all architectures are suitable for cloud deployments of IoT applications. The traditional monolithic architecture is resource inefficient, as the workload for different functionalities of IoT applications depends on different factors. We want to achieve elasticity and deploy, scale and manage the functionalities of an IoT application separately. The microservices approach to building IoT solutions does exactly what we want. We concluded that the microservices architecture is suitable for cloud deployments of IoT applications.

Still, achieving scalability only in order to be able to store the stream of events in a database is not sufficient. Most of the real-world scenarios (such as the fire detection and shoplifting scenarios we introduced) require real-time actions. Complex event processing helps us process huge amount of events on-the-fly. As the stream of events flows, we are able to extract information from the events in real time. We saw that different CEP techniques can be used for event analysis.

In order to show all of this in practice, we developed a practical example. We showed how we can achieve scalability by changing the number of consumer instances, both manually and automatically. We developed three scenarios and used Esper to analyze the events for all of the scenarios in real time. We used Google's public cloud and deployed our application in a Kubernetes cluster in the Google Container Engine. We described how the microservices of our IoT application are developed and how to build Docker images in order to containerize our microservices. At the end, we managed our microservices through Kubernetes and tested our scenarios through the front-end.

Further work might be the integration of Esper with Apache Storm, which would distribute the complex event processing done by Esper. Some scenarios require later processing of events as well. Therefore, another upgrade could be to introduce an option to store the events in a database and extensively use cloud's storage capabilities. In this way, we would be able to get both real-time and retrospective information.

Bibliography

[1] Apache Kafka documentation. https://kafka.apache.org/documentation.html. [Accessed: 20. 7. 2016].

[2] Apache Kafka in a Kubernetes cluster. https://github.com/CloudTrackInc/kubernetes-kafka. [Accessed: 31. 7. 2016].

[3] Cloud types: private, public and hybrid. http://www.asigra.com/blog/cloud-types-private-public-and-hybrid. [Accessed: 18. 7. 2016].

[4] Complex event processing. https://en.wikipedia.org/wiki/Complex_event_processing. [Accessed: 10. 3. 2016].

[5] Docker env. https://docs.docker.com/machine/reference/env/. [Accessed: 28. 7. 2016].

[6] Docker image for Apache Kafka. https://hub.docker.com/r/wurstmeister/kafka/. [Accessed: 31. 7. 2016].

[7] Docker image for Apache ZooKeeper. https://hub.docker.com/r/digitalwonderland/zookeeper/~/dockerfile/. [Accessed: 31. 7. 2016].

[8] Dockerfile reference. https://docs.docker.com/engine/reference/builder/. [Accessed: 29. 7. 2016].

[9] Esper reference. http://www.espertech.com/esper/release-5.4.0/esper-reference/html_single/index.html. [Accessed: 15. 4. 2016].

[10] Esper: Solution patterns. http://www.espertech.com/esper/solution_patterns.php. [Accessed: 29. 7. 2016].

[11] Event-driven architecture. https://en.wikipedia.org/wiki/Event-driven_architecture. [Accessed: 20. 7. 2016].

[12] Event processing glossary. http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2/. [Accessed: 10. 3. 2016].

[13] Fast data: the next step after Big data. http://www.infoworld.com/article/2608040/big-data/fast-data--the-next-step-after-big-data.html. [Accessed: 5. 3. 2016].

[14] From monoliths to microservices: An architectural strategy. http://thenewstack.io/from-monolith-to-microservices/. [Accessed: 11. 7. 2016].

[15] Get started with Docker Machine and a local VM. https://docs.docker.com/machine/get-started/. [Accessed: 27. 6. 2016].

[16] Getting started with sample programs for Apache Kafka 0.9. https://www.mapr.com/blog/getting-started-sample-programs-apache-kafka-09. [Accessed: 21. 7. 2016].

[17] Horizontal and vertical scaling. https://en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling. [Accessed: 11. 7. 2016].

[18] Internet of things. https://en.wikipedia.org/wiki/Internet_of_things. [Accessed: 3. 3. 2016].

[19] Kubernetes: Autoscaling algorithm. https://github.com/kubernetes/kubernetes/blob/master/docs/design/horizontal-pod-autoscaler.md#autoscaling-algorithm. [Accessed: 4. 9. 2016].

[20] Kubernetes documentation. http://kubernetes.io/docs/. [Accessed: 2. 8. 2016].

[21] Kubernetes: Horizontal Pod Autoscaling. http://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/walkthrough/. [Accessed: 4. 9. 2016].

[22] Kubernetes: Pods. http://kubernetes.io/docs/user-guide/pods/. [Accessed: 3. 7. 2016].

[23] Kubernetes: Replication controller. http://kubernetes.io/docs/user-guide/replication-controller/. [Accessed: 3. 7. 2016].

[24] Kubernetes: Replication controller operations. http://kubernetes.io/docs/user-guide/replication-controller/operations/. [Accessed: 7. 8. 2016].

[25] Kubernetes: Rolling updates. http://kubernetes.io/docs/user-guide/rolling-updates/. [Accessed: 15. 8. 2016].

[26] Kubernetes: Services. http://kubernetes.io/docs/user-guide/services/. [Accessed: 4. 7. 2016].

[27] Kubernetes walkthrough. http://kubernetes.io/docs/hellonode/. [Accessed: 5. 8. 2016].

[28] Microservice: Cloud and IoT applications force the CIO to create novel IT architectures. http://analystpov.com/cloud-computing/microservice-cloud-and-iot-applications-force-the-cio-to-create-novel-it-architectures-25117. [Accessed: 10. 7. 2016].

[29] Microservices. http://martinfowler.com/articles/microservices.html. [Accessed: 1. 7. 2016].

[30] Private vs. public cloud: What's the difference? https://www.expedient.com/blog/private-vs-public-cloud-whats-difference/. [Accessed: 5. 7. 2016].

[31] Running a multi-broker Apache Kafka 0.8 cluster on a single node. http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/. [Accessed: 24. 7. 2016].

[32] Smoke detectors. https://en.wikipedia.org/wiki/Smoke_detector. [Accessed: 16. 8. 2016].

[33] The scale cube. http://microservices.io/articles/scalecube.html. [Accessed: 15. 7. 2016].

[34] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A Berkeley view of cloud computing. 2009.

[35] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[36] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. On the integration of cloud computing and Internet of Things. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 23–30. IEEE, 2014.

[37] Michael Chui, Markus Löffler, and Roger Roberts. The Internet of Things. *McKinsey Quarterly*, 2(2010):1–9, 2010.

[38] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[39] Matjaž B. Jurič, Tilen Faganel. KumuluzEE: Building microservices with Java EE. http://www.javamagazine.mozaicreader.com/JanFeb2016#&pageSet=80&page=0. [Accessed: 20. 6. 2016].

[40] Tilen Faganel. Ogrodje za razvoj mikrostoritev v Javi in njihovo skaliranje v oblaku. Diploma thesis, Faculty of Computer and Information Science, University of Ljubljana, 2015.

[41] Bob Familiar. *Microservices, IoT and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions.* 2015.

[42] Martin L. Abbott, Michael T. Fisher. *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise.* Addison-Wesley, 2015.

[43] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[44] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, 2013.

[45] 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.

[46] D Robins. Complex event processing. In *Second International Workshop on Education Technology and Computer Science. Wuhan.* Citeseer, 2010.

[47] Tomislav Vresk and Igor Čavrak. Architecture of an Interoperable IoT Platform Based on Microservices.