FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Efficient IoT Application Delivery and Management in Smart City Environments

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Michael Vögler
Matrikelnummer 0625617

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.Prof. Schahram Dustdar

Diese Dissertation haben begutachtet:

(Univ.Prof. Schahram Dustdar) (Prof. Frank Leymann)

Wien, 11.04.2016

(Michael Vögler)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Efficient IoT Application Delivery and Management in Smart City Environments

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Michael Vögler
Registration Number 0625617

to the Faculty of Informatics
at the TU Wien

Advisor: Univ.Prof. Schahram Dustdar

The dissertation has been reviewed by:

(Univ.Prof. Schahram Dustdar) (Prof. Frank Leymann)

Wien, 11.04.2016

(Michael Vögler)

Erklärung zur Verfassung der Arbeit

Michael Vögler
Macholdastraße 24/4/79, 1100 Wien

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

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

This work was partly supported by the Pacific Controls Cloud Computing Lab (PC^{3}L)[1], a joint lab between Pacific Controls L.L.C., Scheikh Zayed Road, Dubai, United Arab Emirates and the Distributed Systems Group of the TU Wien.

[1]http://pc3l.infosys.tuwien.ac.at/

Danksagung

Mit dem Schreiben dieser Zeilen beende ich nun nicht nur diese Dissertation, sondern schließe auch mein Informatikstudium ab, an dem ich mittlerweile fast eine Dekade gearbeitet habe. Rückblickend betrachtet, wäre diese Arbeit nie ohne das Zutun und die Unterstützung zahlreicher Personen möglich gewesen, denen ich in Form dieser Zeilen danken möchte.

Als Erstes möchte ich mich bei meinem Betreuer, Prof. Schahram Dustdar, für die Unterstützung und die Chance bedanken, meine Dissertation in der Distributed Systems Group (DSG) zu schreiben, in der ich meine Forschungsideen großteils eigenständig verfolgen und umsetzen konnte. Des Weiteren danke ich Prof. Frank Leymann für seine interessanten Vorträge über die Grundsätze der Cloud und für die Zweitbegutachtung dieser Arbeit.

Natürlich möchte ich mich auch bei meinen Kollegen der DSG bedanken, die mich auf meinem Weg mit guter Zusammenarbeit und hilfreichem Feedback begleitet haben. Insbesondere möchte ich mich bei Christian Inzinger und Johannes M. Schleicher bedanken, die an allen meinen Forschungsarbeiten beteiligt waren, mir immer mit Rat und Tat zur Seite standen und dadurch die vorliegende Arbeit erst möglich gemacht haben.

Mein Dank gilt natürlich auch meinen Eltern, die mir das Studium ermöglicht und mich immer in meinen Vorhaben und Träumen unterstützt haben. Weiters möchte ich mich bei meiner gesamten Familie und natürlich auch meinen Freunden für die Unterstützung bedanken.

Darüber hinaus, gebührt der meiste Dank Kara Wieland, die mich im kompletten Verlauf meines Studiums begleitet hat, Höhen und Tiefen mit mir überstanden hat, es mit Engelsgeduld ertragen hat, wenn ich Abende und Wochenenden vor dem Laptop verbracht habe und mir immer Mut zugesprochen hat, auch wenn sie selbst gerade eine stressige Zeit durchmachte.

Abstract

The smart city concept initially emerged as an umbrella term for the use of information and communication technology (ICT) in cities with the goal of delivering additional services to their citizens and generally becoming more efficient in terms of resource utilization. Traditionally, these resources were mainly limited to energy and mobility systems. However, with the evolution and ubiquitous availability of information technology, potential target domains and resources that are addressable in a smart city changed significantly. New areas like smart buildings or smart traffic systems can now be tackled. With the recent advent of the Internet of Things (IoT), more and more stakeholders in the smart city domain start to deploy connected IoT devices that allow for sensing and controlling the physical environment they are residing in. Based on the deployed IoT devices and the available smart city infrastructure, IoT applications emerged as a central enabler for stakeholders to build new innovative smart city services for citizens. Such IoT applications need to efficiently manage large amounts of data provided by connected devices, which in combination with the rapid growth of IoT, is challenging. Furthermore, deployed IoT applications need the ability to fully utilize the underlying smart city infrastructure resources to optimally fulfill their requirements at all times. Apart from the intrinsic challenges of operating and managing IoT applications in the smart city domain, such applications must also support the seamless integration of stakeholders and data from different domains to help building new applications that are able to tackle the increasingly complex challenges of today's smart cities.

In this thesis we present a set of novel approaches that allow for efficient operation and management of IoT applications in a smart city ecosystem. We first introduce a methodology that makes IoT devices first class citizens in the design, development, and operation of IoT applications, which allows for leveraging the available capabilities of these resources to build more resilient and performant applications. We present an approach for elastic provisioning of software and application capabilities on resource-constrained IoT devices that explicitly considers the significant heterogeneity in terms of available storage and processing power of these devices. Next, we introduce a declarative, constraint-based model to describe IoT applications as a set of clearly separated components. Based on this model, we derive an approach to dynamically generate optimized deployment topologies for IoT applications that are tailored to the currently available physical infrastructure. Since the monitoring of IoT applications is an essential part of application operation, we introduce a non-intrusive monitoring approach that supports in-depth analysis of data-intensive IoT applications independent of the underlying execution environment.

Finally, to ensure the efficient execution of IoT applications, we present an approach for analyzing monitored infrastructure data to optimize the overall IoT application deployment. By using a set of illustrative scenarios, we extensively evaluate the results of our investigations and show that our contributions support the efficient delivery of robust and flexible IoT applications by allowing them to fully utilize the complete range of infrastructure resources available in a smart city ecosystem.

Kurzfassung

Das Konzept intelligenter Städte entstand ursprünglich als Oberbegriff für die Nutzung von Informations- und Kommunikationstechnologie in Städten, mit dem Ziel Bürgern zusätzliche Dienstleistungen anzubieten und im Allgemeinen effizienter mit vorhanden Ressourcen umzugehen. Zu Beginn waren diese Ressourcen in erster Linie auf Energie- und Mobilitätssysteme beschränkt. Mit der raschen technologischen Entwicklung und der damit einhergehenden allgegenwärtigen Verfügbarkeit von Informationstechnologie vergrößerten sich jedoch auch die potenziellen Zieldomänen und Ressourcen, die man in intelligenten Städten ansprechen konnte. Dementsprechend ist es nun möglich, neue Bereiche, wie etwa intelligente Gebäude oder intelligente Verkehrssysteme, zu erschließen. Mit der Entstehung des Internet der Dinge (IoT), haben immer mehr Akteure begonnen vernetzte IoT Geräte in intelligenten Städten einzusetzen, die es wiederum ermöglichen, die physische Umgebung, in der sie sich befinden, zu erfassen und zu steuern. Basierend auf den installierten IoT Geräten und der verfügbaren Infrastruktur von intelligenten Städten, sind sogenannte IoT Anwendungen zu mächtigen Mechanismen für Akteure geworden, um neue und innovative Dienstleistungen für Bürger zu entwickeln und diesen bereitzustellen. Allerdings müssen solche Anwendungen effizient mit großen Datenmengen umgehen können, was in Kombination mit dem rasanten Wachstum des IoT eine große Herausforderung darstellt. Darüber hinaus brauchen laufende IoT Anwendungen die Möglichkeit, die zugrunde liegenden Infrastrukturressourcen von intelligenten Städten im vollen Umfang zu nutzen, da sie nur so die an sie gestellten Anforderungen jederzeit optimal erfüllen können. Neben den intrinsischen Herausforderungen, die durch den Betrieb und die Verwaltung von IoT Applikationen in intelligenten Städten entstehen, müssen Anwendungen auch die nahtlose Integration von Interessenvertretern und Daten aus den unterschiedlichsten Bereichen unterstützen. Erst dadurch ist es möglich, neue Anwendungen zu entwickeln, die in der Lage sind, die immer komplexer werdenden Herausforderungen intelligenter Städte zu bewältigen.

In dieser Arbeit stellen wir eine Reihe von neuartigen Ansätzen vor, die den effizienten Betrieb und eine ebensolche Verwaltung von IoT Anwendungen im Ökosystem intelligenter Städte ermöglichen. Wir beginnen mit einer Methodologie, die IoT Geräte zu wichtigen Elementen in der Konzeption, der Entwicklung und dem Betrieb von IoT Applikationen macht. Dieser Ansatz ermöglicht es, die verfügbaren Ressourcen dieser Geräte zu nutzen, um belastbarere und performantere Applikationen zu bauen. Als Nächstes präsentieren wir einen Ansatz zur elastischen Installation von Software- und Applikationskomponenten auf IoT Geräten, die eingeschränkte Ressourcen zur Verfügung stellen. Der Ansatz berücksichtigt dabei explizit die signifikante Heterogenität dieser Geräte im Bezug auf den verfügbaren Speicher oder die vorhandene Rechenleistung. Weiters stellen wir ein deklaratives Modell vor, mit dem IoT Applikationen beschrieben werden können, die aus klar getrennten und eigenständigen Komponenten bestehen. Basierend auf diesem Modell leiten wir einen Ansatz zur dynamischen Erzeugung von optimierten Verteilungstopologien für IoT Applikationen ab, die auf die verfügbare physische Infrastruktur zugeschnitten sind. Da die Überwachung von laufenden IoT Anwendungen ein wesentlicher Bestandteil des Anwendungsbetriebes ist, führen wir einen eingriffsfreien Überwachungsansatz ein. Dieser Ansatz unterstützt die gründliche Analyse von datenintensiven IoT Anwendungen unabhängig von der zugrunde liegenden Ausführungsumgebung. Um eine effiziente Ausführung von IoT Anwendungen zu gewährleisten, wird schließlich ein Verfahren zur Analyse von verfügbaren Infrastrukturdaten vorgestellt, das es ermöglicht, Verteilungstopologien von IoT Applikationen zu optimieren. Durch Verwendung repräsentativer Beispielszenarien werden die vorgestellten Ergebnisse unserer Untersuchungen ausführlich evaluiert und zeigen, dass unsere Ansätze die effiziente Bereitstellung von robusten und flexiblen IoT Anwendungen unterstützen, indem Anwendungen die Möglichkeit gegeben wird, die komplette Bandbreite an verfügbaren Infrastrukturressourcen im Ökosystem intelligenter Städte zu nutzen.

Contents

List of Figures

List of Listings

List of Publications

The work presented in this thesis is based on research that has been published in the following conference papers and journal articles. For a full publication list of the author please refer to the website at http://dsg.tuwien.ac.at/staff/mvoegler.

- Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Efficient and Scalable IoT Service Delivery on Cloud. In *Proceedings of the 6th International Conference on Cloud Computing*, CLOUD'13, pages 740–747, 2013. doi:10.1109/CLOUD.2013.64
- Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Towards Automated IoT Application Deployment by a Cloud-Based Approach. In *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications*, SOCA'13, pages 61–68, 2013. doi:10.1109/SOCA.2013.12
- Michael Vögler, Johannes M Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments. In *Proceedings of the 9th Symposium on Service-Oriented System Engineering*, SOSE'15, pages 78–87. IEEE, 2015. doi:10.1109/SOSE.2015.23
- Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. DIANE – Dynamic IoT Application Deployment. In *Proceedings of the 4th International Conference on Mobile Services*, MS'15, pages 298–305, 2015. doi:10.1109/MobServ.2015.49
- Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. A Scalable Framework for Provisioning Large-scale IoT Deployments. *ACM Transactions on Internet Technology*, 16(2):11:1–11:20, March 2016. doi:10.1145/2850416
- Michael Vögler, Johannes M Schleicher, Christian Inzinger, Bernhard Nickel, and Schahram Dustdar. Non-Intrusive Monitoring of Stream Processing Applications. In *Proceedings of the 10th International Symposium on Service-Oriented System Engineering*, SOSE'16, pages 190–199. IEEE, 2016. doi:10.1109/SOSE.2016.11
- Johannes M Schleicher, Michael Vögler, Schahram Dustdar, and Christian Inzinger. Enabling a Smart City Application Ecosystem: Requirements and Architectural

Aspects. *IEEE Internet Computing*, 20(2):58–65, Mar 2016. doi:10.1109/MIC.2016.39

- Michael Vögler, Johannes M Schleicher, Christian Inzinger, Schahram Dustdar, and Rajiv Ranjan. Migrating Smart City Applications to the Cloud. *IEEE Cloud Computing*, page to appear, Mar-Apr. 2016

- Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar. Ahab: A Cloud-based Distributed Big Data Analytics Framework for the Internet of Things. *Software: Practice and Experience*, page to appear, 2016

CHAPTER 1

Introduction

The smart city concept has emerged as an umbrella term for the pervasive implementation of information and communication technology (ICT) systems designed to improve various areas of today's cities, such as citizen well-being, infrastructure, industry, and government. Predominantly, these areas are seen as vertical applications that operate on highly heterogeneous sets of infrastructures that are managed by different providers in order to serve multiple stakeholders. Currently, the fundamental stakeholders in a smart city are energy and transportation providers, as well as government agencies, which produce large amounts of data about certain aspects (e.g., public transportation) of a city and its citizens. Increasingly, these stakeholders deploy connected Internet of Things (IoT) devices that deliver large amounts of near real-time data and allow to enact changes in the physical environment. On top of the deployed IoT devices and available smart city infrastructure, more and more IoT applications emerge that operate in this dynamic environment to better integrate the large number of stakeholders that not only provide data for applications, but can also contribute functionality or impose (possibly conflicting) requirements. For IoT applications the efficient management of large volumes of data produced by a quickly growing number of connected resources is challenging, especially since data gathered by IoT devices might have critical security and privacy requirements that must be respected at all times. Nevertheless, utilizing IoT applications as the key enabler for smart cities presents a significant opportunity to closely integrate stakeholders and data from different domains to create new types of applications that are able to tackle the increasingly complex challenges of today's cities, such as autonomous traffic management, efficient building management, and emergency response systems. Furthermore, future IoT applications must also be able to operate across city infrastructure boundaries to create a global interconnected system of systems for the future Internet of Cities [109]. Thus, in this thesis we argue that such applications will be designed, implemented, and operated as cloud-native applications allowing them to elastically respond to changes in request load, stakeholder requirements, and unexpected changes in their environment.

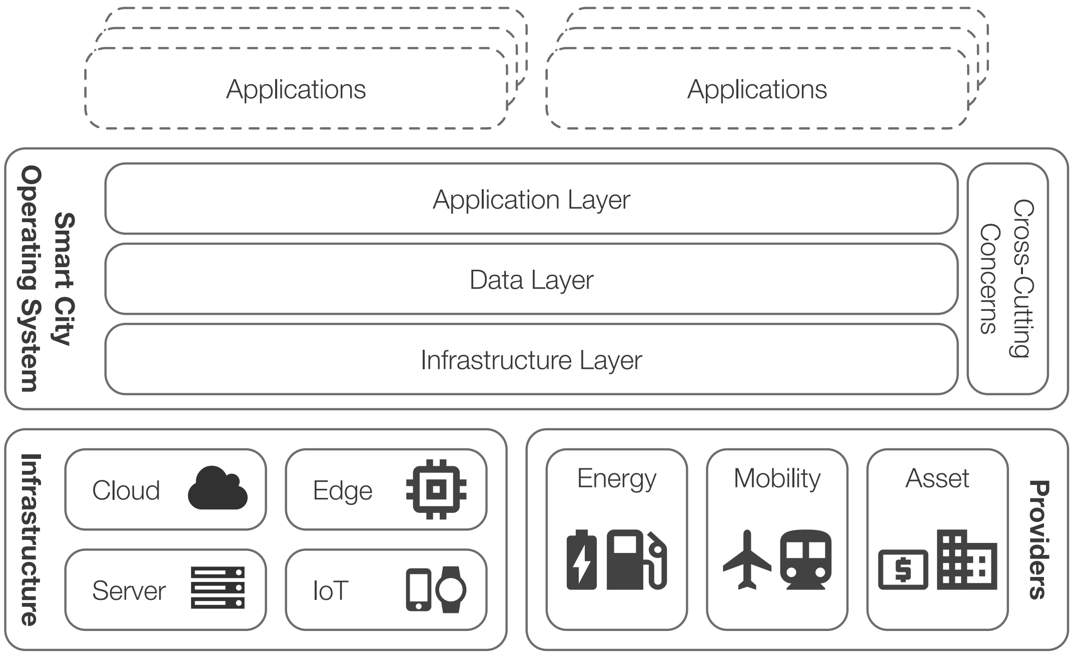


Figure 1.1: Smart City Application Ecosystem

To address the intrinsic complexities of delivering applications in such environments, we envisioned a Smart City Application Ecosystem [112], which is depicted in Figure 1.1. By introducing such an ecosystem we aim to closely integrate requirements, functionality, and capabilities of multiple stakeholders in a seamless and manageable way in order to enable the full potential of IoT applications in the smart city domain. To implement such an ecosystem, in this thesis we will provide novel approaches to efficiently operate and manage IoT applications, which will build the foundation for a Smart City Operating System, a central element of future smart city application ecosystems.

1.1 Problem Statement

To enable the creation of IoT applications that run in the previously introduced smart city application ecosystem, it is necessary to provide sensible abstractions that hide the underlying complexities of operating, analyzing, and managing complex IoT applications in the smart city domain. Much like today's mobile application ecosystems, a smart city application ecosystem must enable stakeholders to focus on the application development itself without the necessity to think about infrastructure, data, or application management. In order to achieve this ease of development, such an ecosystem must address several challenges that arise from the highly dynamic and complex nature of the smart city domain itself.

First, in order to incorporate and enable the heterogeneous sets of available infrastructures in a smart city, an application ecosystem needs the ability to manage and operate the large number of devices that emerge in today's IoT infrastructures in order to integrate them as vital information providers and as a mechanism to enact changes in their physical environment. This calls for novel means for staging, deploying, and organizing such IoT resources to ensure that their full potential can be realized in IoT applications.

Second, IoT applications need to be able to run on and integrate a plethora of different infrastructure types to fully utilize the available resources in order to operate with maximum performance. To enable this the application ecosystem must be able to handle a large number of different resources, ranging from traditional servers and hosted cloud solutions to the dormant computational potential of edge resources. Moreover, the large number of providers with different infrastructures along with the rapid pace of infrastructure evolution, call for means to move applications and their respective deployment topologies seamlessly between providers in the smart city domain (e.g., from a dedicated server setup to a hosted cloud to the edge of the infrastructure itself).

Third, IoT applications need the ability to deal with and facilitate the massive amounts of data that are emitted by the smart city. Therefore, to enable data-driven IoT applications that are essential for every aspect of smart city development, it is vital to provide smart data management mechanisms. Among other things, these mechanisms need to handle high-volume data streams as well as large batches of data in structured and unstructured formats, which calls for novel integrated processing approaches.

Finally, to enable IoT applications that are able to seamlessly incorporate data and infrastructure resources made available in this ecosystem, it is essential to provide novel means for designing and developing such applications. To support practitioners in developing applications for this complex domain, a comprehensive toolset is required that hides the intricacies of the heterogeneous physical infrastructure in order to provide simple and transparent access to the underlying resources. This not only calls for novel design and development methodologies, but also for a sophisticated runtime environment that enables adaptive execution of IoT applications by utilizing the available heterogeneous infrastructures.

1.2 Research Questions

The problems identified in Section 1.1 serve as motivation for the research conducted throughout this thesis. Specifically, this work addresses the following research questions.

Research Question I:
How can resource-constrained edge devices be seamlessly incorporated in the provisioning process of IoT applications in smart city ecosystems?

As discussed in Section 1.1, the rise of IoT with its constantly growing number of connected nodes has led to massive numbers of devices that provide data and need to be managed. To deal with this complexity we need the ability to manage this heterogenous infrastructure, which includes enabling the efficient provisioning for both cloud and edge infrastructures. While at first glance traditional IT automation and configuration management tools like Chef [93] or Puppet [101] seem like a good fit, these tools are not equipped to incorporate and manage edge resources like the massive sensor arrays and gateways that emerge in the smart city domain. Furthermore, current IoT application development methodologies consider IoT devices as external dependencies that only act as data sources or command receivers. This, however, does not consider the available capabilities of new types of edge devices, even though they provide viable processing power. To incorporate these elements we have to rethink the role of these devices and enable them as first class resources in smart city infrastructures.

Research Question II:
How can IoT applications and respective topologies be optimally deployed while specifically considering available infrastructure resources in smart city ecosystems?

We have to respect that IoT applications are large-scale distributed systems that react and control, as well as analyze and reason about their physical environment by using the underlying infrastructures. The inherently dynamic nature of these applications poses several challenges when deploying, executing, and managing such applications. Applications need to quickly react to changes in requirements and have to deal with unreliable and expensive network links. In addition, applications have to provide stable Quality of Service (QoS) even when experiencing infrastructure outages. Finally, applications need the ability to scale computations across different infrastructures in order to handle the ever increasing load generated by the environment they are operating in. In order to address these challenges we need to separate application-specific topologies and requirements from the actual deployment infrastructure. Based on that, applications and respective topologies can be deployed on and run in different execution environments, reaching from traditional application servers running in the cloud, to novel approaches that also consider other forms of infrastructure resources (e.g., container-based environments provided by the edge infrastructure).

Research Question III:
How can running IoT applications and utilized infrastructure resources be generically analyzed in order to optimize the overall IoT application deployment topology?

Once deployed and running, IoT applications operating in smart city ecosystems produce an ever-growing amount of data that needs to be handled. These large sets of diverse data, commonly referred to as big data, have to be efficiently collected, stored, and analyzed. Therefore, supporting the constant monitoring and collection of information from facilitated resources, using available monitoring capabilities of respective infrastructure or commonly applied monitoring tools is of utmost importance. In addition, a mechanism for managing produced logs, events, and faults is essential to conduct performance analyses that can be used for optimizing resource utilization or evolve the overall infrastructure deployment. Based on that, fine-grained analysis information can be used to adapt application topologies in order to react to defined requirements like SLAs.

1.3 Scientific Contributions

The work conducted during the course of this thesis, guided by the research questions introduced in Section 1.2, has led to the following contributions.

Contribution I:
An approach for seamless provisioning of large-scale IoT deployments on heterogeneous infrastructure resources

In the smart city domain, more and more IoT devices with embedded execution environments emerge that support offloading parts of the application logic towards the edge of the infrastructure. Leveraging these currently untapped processing capabilities of IoT devices allows for improving dependability, resilience, and performance of IoT applications. However, the heterogeneity of available IoT devices poses challenges for application delivery due to significant differences in provided capabilities (e.g., available storage and processing resources), as well as deployed and deployable software components. To accommodate this diversity, a structured approach is needed to uniformly and transparently deploy application components onto a large number of heterogeneous devices, which is especially important in the context of current large-scale IoT applications, such as in the smart city domain. Therefore, we introduce an approach that provides elastic provisioning of application components on resource-constrained and heterogeneous edge devices in large-scale IoT deployments. Details are presented in Chapter 4. Contribution I was originally presented in [129, 131].

Contribution II:
An approach for generating and maintaining optimal IoT application deployment topologies

Utilizing the previously untapped processing power of IoT devices to offload custom logic directly to these edge resources will not only increase the overall robustness of the application, but can also reduce communication overhead. However, to allow the flexible provisioning of applications whose deployment topology evolves over time, a clear separation of independently executable application components is needed. Therefore, we introduce an approach for the dynamic generation of optimized deployment topologies for IoT applications that are tailored to the currently available physical infrastructure based on a declarative, constraint-based model of the desired application deployment. In addition, to also providing an optimal deployment of IoT applications that follow traditional application design paradigms, we introduce a TOSCA-based approach allowing to formally describe components and deployment topologies of such applications. Furthermore based on this formal description the approach enables automating the overall IoT application deployment process. Details are presented in Chapter 5 and Chapter 6 respectively. Contribution II was originally presented in [68, 128].

Contribution III:
An approach for non-intrusive monitoring of IoT applications

Among predominant IoT applications, stream processing applications have emerged as a popular way for implementing high-volume data processing tasks. To cope with the intrinsic request load, components of such applications are usually distributed across multiple infrastructure resources. In this context, performance monitoring and testing are naturally important for understanding as well as analyzing the runtime characteristics of deployed applications in order to identify issues and inform decisions. However, existing approaches for monitoring the performance of distributed systems do not provide sufficient support for targeted monitoring of data-intensive IoT applications. Therefore, we present an approach that allows for in-depth analysis of stream processing applications by non-intrusively adding functionality to acquire and publish performance measurements at runtime, to the application. Details are presented in Chapter 7. Contribution III was originally presented in [133].

Contribution IV:
An approach for analyzing and optimizing large-scale IoT application deployments at runtime

IoT applications that are operated in smart city environments generate large amounts of operational data during their execution. This data contains information from infrastructure monitoring, performance and health events from used toolsets, and application execution logs. These produced data streams contain vital information about the application execution environment that can be used to fine-tune or optimize different layers of the utilized deployment infrastructure. Current approaches do not sufficiently address the efficient collection, processing, and storage of this information in the smart city domain. Therefore, we introduce a generic, scalable, and fault-tolerant data processing approach that allows to perform online and offline analyses on gathered data to better understand the behavior of the available infrastructure in order to optimize the overall IoT application deployment. Details are presented in Chapter 8. Contribution IV was originally presented in [130].

1.4 Organization of this Thesis

The remainder of this thesis is structured as follows.

- Chapter 2 provides background information on basic concepts used in this thesis. Specifically, the topics smart city, cloud computing, and IoT, as well as their relation to each other are introduced.
- Chapter 3 introduces a scenario that will be used throughout this thesis to motivate and evaluate our contributions.
- Chapter 4 presents an approach for provisioning resource-constrained, heterogeneous edge devices in large-scale IoT deployments.
- Chapter 5 discusses an approach for dynamically generating optimized deployment topologies for IoT applications that are tailored to the available physical infrastructure.
- Chapter 6 introduces a TOSCA-based approach for formally describing the internal topology as well as the deployment process of traditional IoT applications.
- Chapter 7 presents a monitoring approach that provides a flexible mechanism to add functionality for acquiring and publishing of performance measurements, at runtime to data-intensive IoT applications.
- Chapter 8 introduces an approach that allows performing online and offline analyses on gathered operational data produced by running IoT applications and utilized infrastructure resources.
- Chapter 9 presents related work categorized according to the contributions presented in Section 1.3.
- Chapter 10 concludes this thesis, discusses the presented contributions in light of the identified research questions, and offers an outlook for ongoing and future research.

CHAPTER 2

Background

In this chapter, we introduce several basic concepts that are used in the remainder of this thesis. First, we illustrate the fundamental properties of the smart city concept, followed by an introduction of cloud computing and the Internet of Things, and discuss how they relate to each other as these topics represent the context of the work conducted in this thesis.

2.1 Smart City

Modern cities are evolving towards smart cities [46, 84] by integrating multiple communication technologies to deliver services to their citizens. In addition, using available information technologies allows cities to become smarter and more efficient in terms of resource utilization. Initially these resources were mainly limited to energy and mobility systems. However, the rapid change of information technology now allows cities to address additional resources and areas like smart buildings, smart traffic systems and roads, energy management, water/waste and pollution management, and emerging concepts such as urban farming.

In essence, the ultimate goal of a smart city is to increase the quality of life of its citizens by providing services that meet their needs more efficiently. Utilizing information and communication technologies (ICT) enables city officials to closely interact with citizens, allows them to analyze and plan the overall evolution of the city, and helps them to decide how to improve the quality of provided services. Supporting city officials in addressing these vital aspects directly improves citizens' quality of life. In order to enable the smart city concept, modern cities and their officials use a plethora of installed devices to collect data from citizens and various areas of everyday life in order to process and analyze gathered data to unveil possible shortcomings in terms of resource utilization or citizen well-being. ICT, in the context of the smart city concept, is used to improve the quality and responsiveness of urban services, provide efficient resource utilization to reduce costs, and encourage close collaboration and interaction of citizens and governments.

Furthermore, the smart city concept wants to overcome the traditional relationship among citizens and service providers by empowering citizens to become vital stakeholders of the city that provide feedback for services, or report problems. Additionally, a smart city has to enable citizens to utilize and engage with provided services easily and conveniently to incorporate various infrastructure types [12], such as hard infrastructures, social networks, as well as ICT in order to empower sustainable economic evolution and represent an inviting environment for citizens.

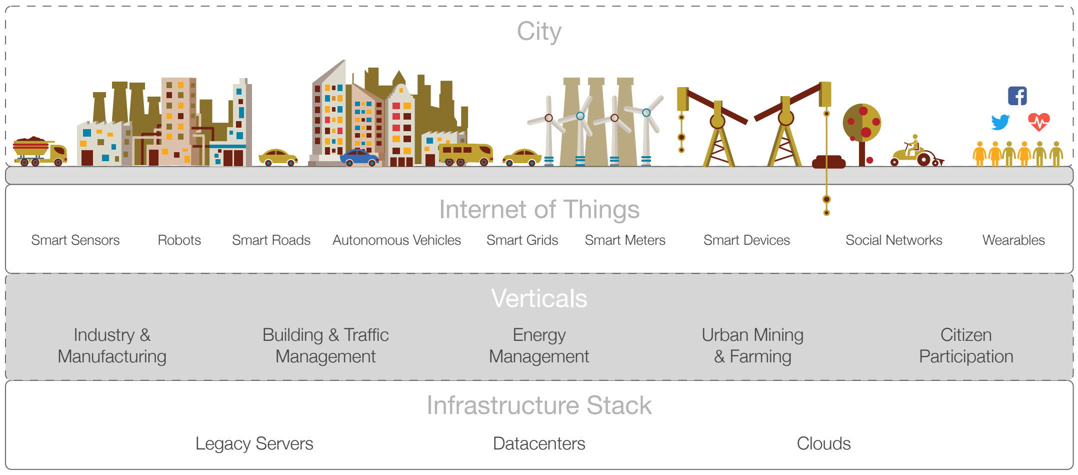


Figure 2.1: Smart City – Overview

Figure 2.1 provides a conceptual overview of a smart city and illustrates the different layers that are required for evolving a modern city towards a smart city. In this figure we see that in order to provide added value for citizens that live in a city, services are provided that incorporate physical devices provided by the Internet of Things (IoT) like smart meters, sensor networks, or other smart objects. Since these services are often designed for one specific area of a smart city (e.g., energy management) and therefore only focus on devices and data of this specific area, they are referred to as verticals respectively vertical solutions. Finally, to allow citizens to use the provided services, these vertical solutions are deployed and operated on heterogenous infrastructure stacks consisting of legacy servers, datacenters, or clouds.

In order to build smart city services, solution providers have to deal with significant challenges due to the lack of available real-world infrastructure that can be used for creating and testing these applications that allow for respecting functional, non-functional, and regulatory requirements. Predominantly, smart city applications are built for and tested on small-scale testbeds [95] or tailored simulations that emulate real-world environments (e.g., Anylogic[1]). These solutions, however, are not ideal, since environmental changes in the real world need to be mirrored in simulations or testbeds, which is usually

[1]http://www.anylogic.com

a tedious manual task. In addition, this approach requires additional planning and efforts for migrating an application from a test infrastructure to a real-world deployment, which has to be repeated for each application release.

2.2 Cloud Computing

The cloud computing paradigm [8, 17, 19] emerged in recent years, which according the US National Institute of Standards and Technology (NIST) is defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services)" [78]. Compared to provisioned resources in traditional datacenter infrastructures, cloud computing has the following essential characteristics:

1. *On-demand self-service*, allowing costumers to request an arbitrary number of computing resources automatically.
2. *Broad network access*, providing computing resources over the network in order to access these resources by using standard mechanisms.
3. *Resource pooling*, allowing providers to serve multiple customers by using a multi-tenant model that dynamically assigns different resources (physical or virtual) based on customer demand.
4. *Rapid elasticity*, allowing elastic provisioning and releasing of resources in order to enable scaling in/out to accommodate load fluctuations.
5. *Measured service*, enabling monitoring, controlling, and reporting of resource usage for both provider and customer of provided services.

Leveraging these properties allows building elastic applications that are able to dynamically adjust resources based on current demand. Predominantly, cloud providers offer services in varying granularity starting with basic infrastructure resources such as Virtual Machines (VMs), platforms that are partially managed by providers, and services that are completely managed by providers and only consumed by customers. According to NIST, these different levels can be categorized into the following three service models (illustrated in Figure 2.2):

- Infrastructure as a Service (IaaS), allows customers to provision computing resources that can be used to deploy and run arbitrary software including operating systems and applications. In this model providers control and manage the underlying cloud infrastructure, while customers have full control over operating systems, storage, and deployed applications.
- Platform as a Service (PaaS), allows customers to develop and deploy applications on managed cloud infrastructure by using pre-configured software environments

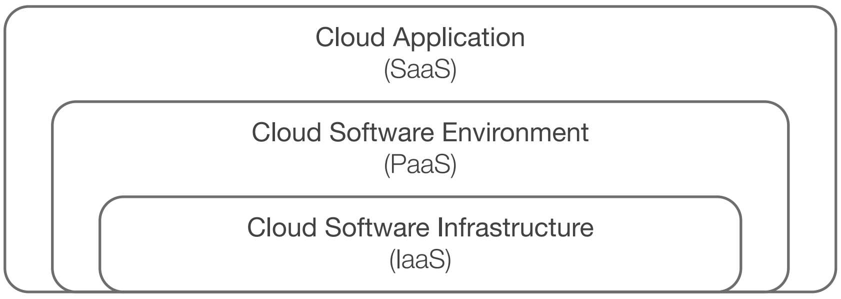


Figure 2.2: Cloud Service Models (adapted from [139])

supported by (possibly different) providers. In this model providers manage the underlying cloud infrastructure and installed software, while customers are controlling the deployed applications and are able to configure the provided application runtime environment.

- Software as a Service (SaaS), allows customers to use applications, which are offered by (possibly different) providers and are running on their cloud infrastructure. In this model the provider manages and controls the underlying cloud infrastructure, software, and applications, while customers are able to configure user-specific application settings.

2.3 Internet of Things

The Internet of Things (IoT) is an emerging paradigm that in combination with the smart city concept allows extending network connectivity and computing capabilities to physical objects, sensors, devices, and other everyday items that are usually not considered as computers, to allow them to generate, collect, consume, and exchange data. Based on this concept, IoT enables these smart objects to be connected, sensed, and controlled over existing network infrastructure, which allows the physical world to be tighter integrated into computer-based systems to improve the efficiency and accuracy of provided services.

The overall vision of IoT is to extend today's Internet to an Internet of Things by incorporating devices from various application domains (Figure 2.3). Shelby and Bormann [116] claim that today's Internet is composed of the core and fringe Internet, as well as the IoT. The core Internet includes servers, backbone routers, and millions of network-related nodes, whose deployment topology rarely changes. On top of the core elements, the vast majority of today's Internet consists of personal computers, laptops, and smart phones that are connected to the Internet, which are referred to as the fringe Internet. Compared to the core elements, the fringe consists of billions of devices that change rapidly according to the number of Internet users. IoT, also sometimes referred to as the embedded fringe, is represented by interconnected, IP-enabled embedded devices like sensors, machines, RFID readers, and building automation equipment. Since the

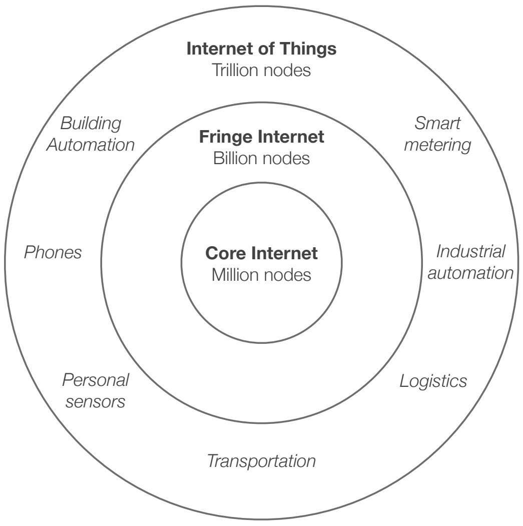


Figure 2.3: Internet of Things – Vision (adapted from [116])

number of nodes in IoT is not dependent on the number of users, an exact estimation on its size is not possible, but it is assumed that IoT will soon exceed the rest of the Internet and will continue to grow rapidly.

IoT can be separated into the following three main perspectives [11] as depicted in Figure 2.4. First, the “Things oriented” perspective focuses on equipping objects with intelligence, typically based on embedded micro-controllers, sensors, and actuators. Second, the “Internet oriented” perspective mainly focuses on the communication-related aspects of smart objects such as communication protocols and providing integrations as well as interoperability by using Web technologies. Third, the “Semantic oriented” perspective of IoT deals with semantic concepts in order to represent, store, interconnect, search, and organize information generated by the IoT.

In order to transform the IoT concept to the real world, IoT applications are built to integrate objects with the help of enabling mechanisms such as sensing and communication features, and a middleware that hides the underlying diverse technology stacks. The huge potential offered by IoT enables the development of a plethora of applications that are usually classified in the following domains [11]: (i) Transportation and logistics, (ii) Healthcare, (iii) Smart environment (home, office, plant), and (iv) Personal and social domain.

2.4 Server Provisioning and Deployment Automation

In essence, server provisioning [56, 125] includes a set of steps that are necessary for preparing a server with an operating system and other system software, in order to provide it via the network. Typically provisioning comprises the following actions: (1) select an

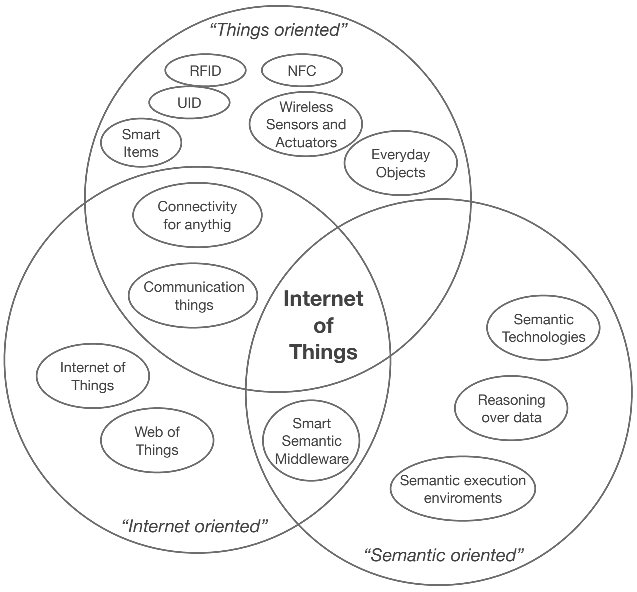


Figure 2.4: Internet of Things – Perspectives (adapted from [11])

available server from a pool of servers, (2) select an image that has to be installed or an image that can be executed, depending if a physical or virtual server is provisioned, (3) install and boot the selected image, (4) assign and configure system resources like network or storage, (5) install and configure appropriate middleware, (6) install and configure necessary applications. Typically, service providers perform these tasks with the help of configuration management (CM) tools like Chef [93] or Puppet [101] that allow for automating this process.

When taking a closer look at the necessary list of actions, we see that according to the last item the installation and configuration of applications is already part of provisioning and therefore one can argue that deployment automation [47] is a subset of server provisioning. However, several aspects are specific to deployment automation, which allow for distinguishing deployment from provisioning. First, usually server provisioning is solely done by the operations team, while deployment automation shares responsibilities between application developers and operators. This means that developers create the application that needs to be deployed, whereas the operators manage the execution environment the application has to be run on top of. Therefore, tools that support deployment automation have to respect these different ways of application delivery. Second, the overall lifecycle of servers and applications is inherently different. Generally, multiple applications can be deployed on one provisioned server, where it is likely that each application can be redeployed multiple times. In comparison, the provisioning of servers is usually only done once throughout multiple application lifecycles. This means that deployment automation tools need to support developers, testers, and operators in performing this magnitude of deployment requests.

However, emerging trends like DevOps [50] do not promote such a clear separation of provisioning and deployment, since they employ a closely integrated team of developers and operators that are not only responsible for application development and deployment, but also deal with installing and configuring the required infrastructure. In addition, the rapid adoption of cloud infrastructure in the overall application delivery process means that virtual servers are provisioned more often and therefore server provisioning and deployment automation tools are getting more integrated.

Nevertheless, in the smart city and IoT domain application delivery still follows heterogenous and often manual processes, where the provisioning of infrastructure resources is still completely separated from the actual application deployment. Therefore, in this thesis we distinguish between the concepts of provisioning and deployment.

CHAPTER 3

Motivating Scenario

In this chapter, we present the scenario as well as high-level requirements that we will use throughout this thesis to motivate and evaluate the contributions of this work.

3.1 Building Management System

Building management systems (BMS) are one of the prominent usage domains of IoT applications in a smart city. A BMS generally aims at efficiently managing building facilities (e.g., heating, ventilation, and air conditioning – HVAC, light controls, power systems, security monitoring, life safety systems and so on) in order to conserve energy, save operation costs, and improve safety and security. Figure 3.1 provides a physical view on such a BMS application. Generally, a BMS project starts with a solution provider collecting information about the target building, which can either be a building already in use or a new building in design. The former would often require retrofitting devices and building facilities, where as the latter is synchronized with the development progress of the building. The scale of a project can range from a single building to a large compound of various building types, such as airports, business districts, and university campuses. Therefore, the number of devices, the volume of data to be processed and the complexity of applications can vary significantly. After surveying and design for the specific building, the solution provider will acquire suitable hardware devices from OEMs, integrate them into an infrastructure solution, develop analytical and control applications, and deploy the applications on dedicated server resources.

This process, however, produces many vertically isolated solutions [27] (often referred to as “silos”), which lead to the following problems for solution providers. First, the more silos are provisioned, the more system instances the solution provider needs to maintain. System maintenance becomes a particularly painful process, since hardware devices are monitored in separate systems, and software instances need to be updated separately and tested on-site with the specific hardware configurations. Second, many

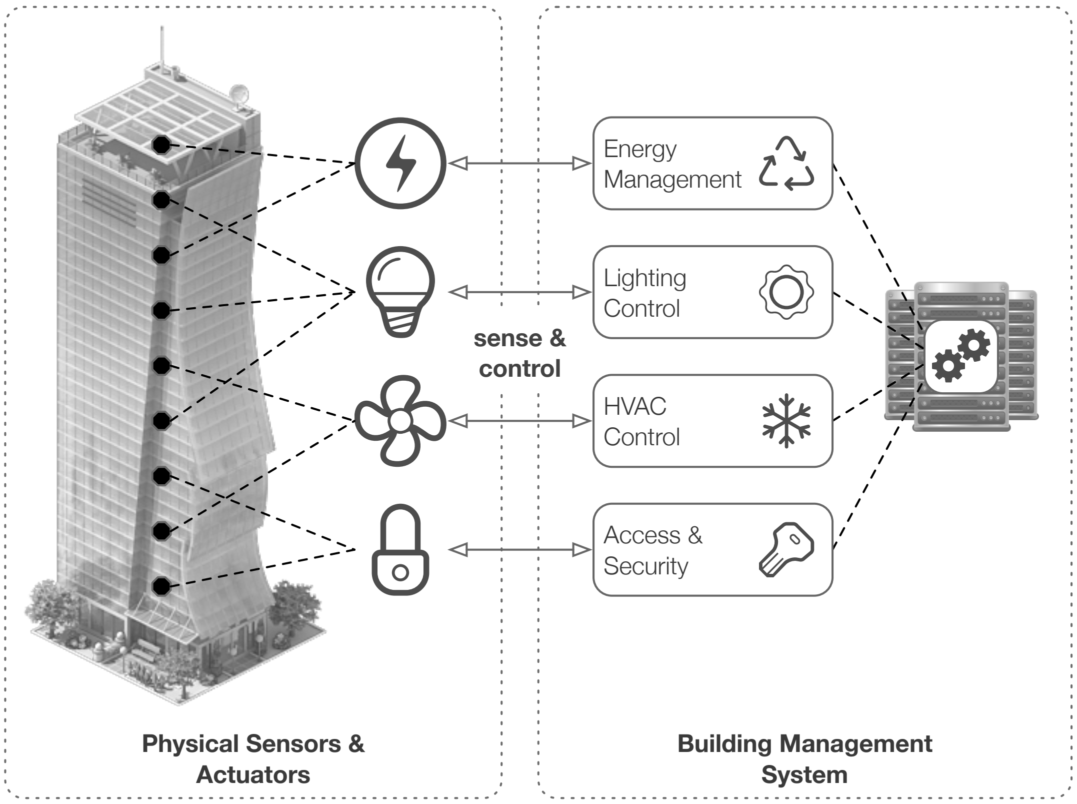


Figure 3.1: BMS Application – Physical View

campuses and building compounds expand continuously to accommodate new users. This is particularly common in large-scale projects, which are usually planned for multiple progressive phases. In this silo-based service delivery model, such expansion may require a bottom-up re-configuration of the whole system and provisioning of new computing resources because of the tight coupling of devices, middleware, and applications. Figure 3.2 illustrates the architecture of a typical isolated IoT application in this context. Currently, these IoT applications are designed and implemented as layered architectures [2], where the bottom layer consists of deployed IoT devices (e.g., sensors, actuators, and gateways), an abstraction layer to uniformly collect data and send operations to the underlying IoT infrastructure, and an application that implements the business logic. According to this layered approach, the application is operated in on-premise infrastructures or in the cloud, whereas the IoT devices reside at the edge of the infrastructure to send data and react to the environment [28].

However, in practice, more and more IoT devices emerge that provide constrained execution environments that can be used for offloading parts of the business logic. Therefore, in this thesis we want to enable these new forms of IoT devices as computing resources that can be utilized in order to build more flexible and resilient IoT applications. Fig-

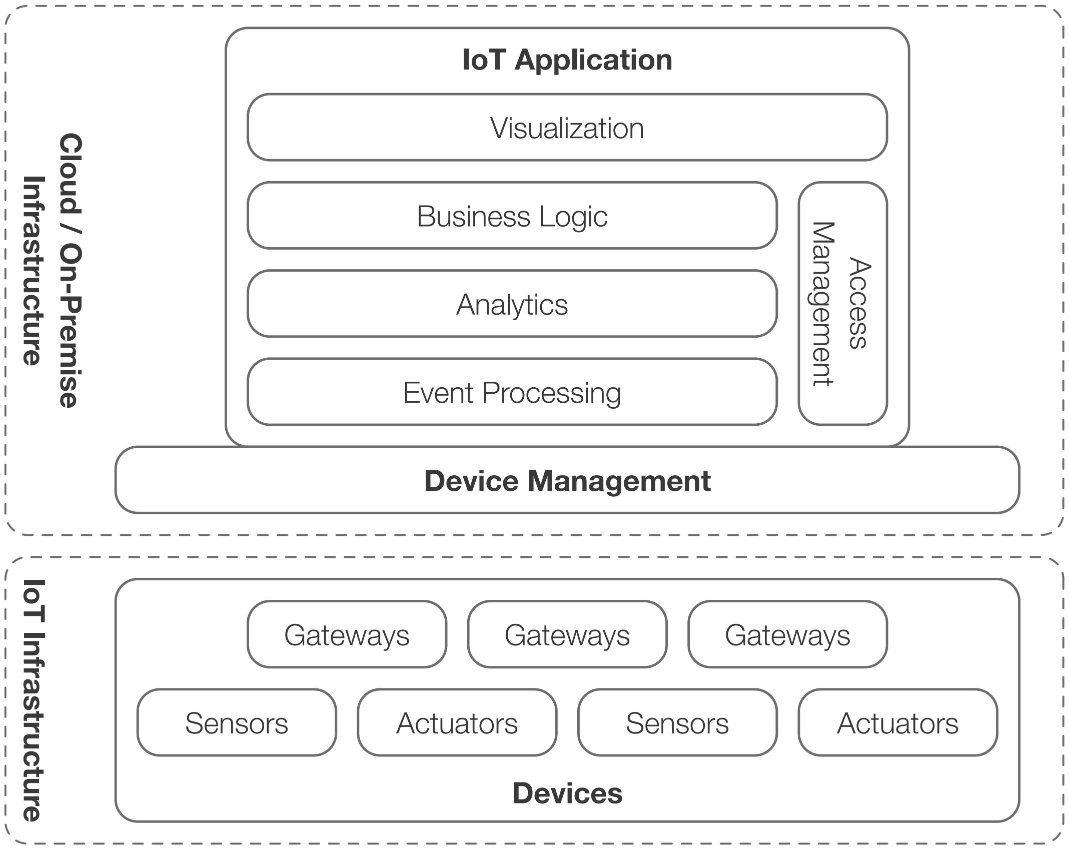


Figure 3.2: Traditional IoT Application – Architectural View

ure 3.3 depicts the architecture of the envisioned IoT applications, which we will use to motivate the contributions in this thesis.

It is worth noting that the limitations demonstrated in delivering BMS solutions are commonly observed in many other IoT application domains, such as smart homes, healthcare, fleet management, and so on. Taking fleet management as an example, multiple fleet management solutions are used to manage different fleets. Each fleet may have a different number of vehicles, which are of different types and serve different purpose (e.g., transportation of goods, emergency services). The silo-based delivery model is commonly applied in state-of-the-art fleet management solutions.

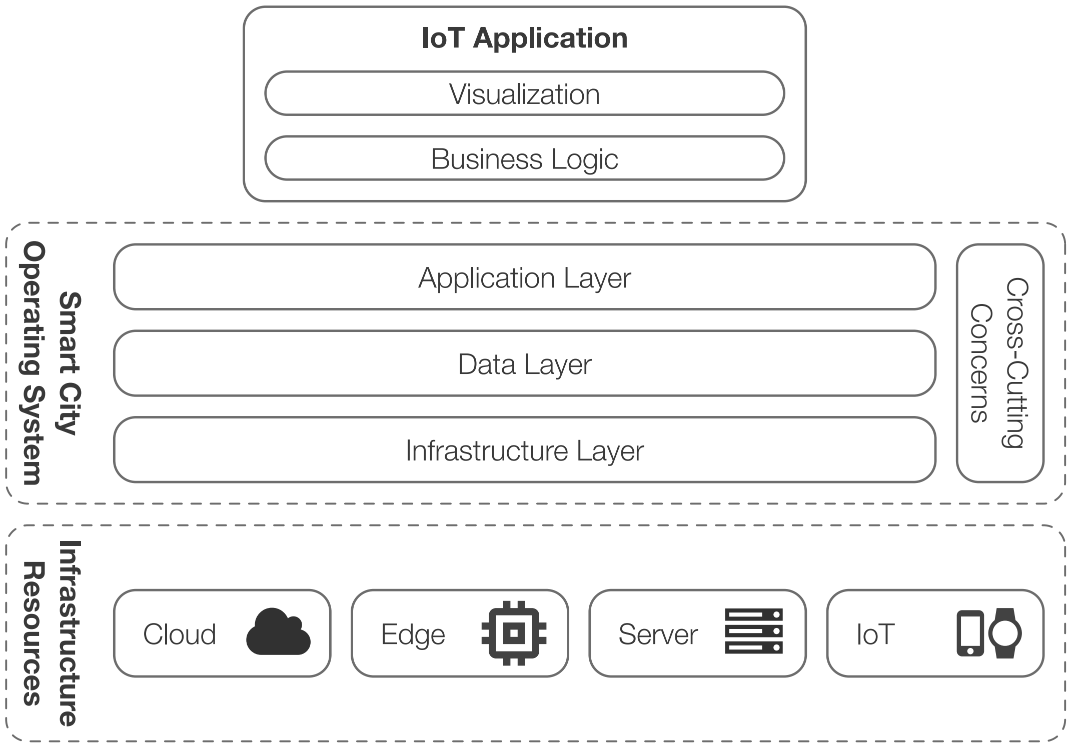


Figure 3.3: Envisioned IoT Application – Architectural View

3.2 Identified Requirements

In order to allow for building and operating IoT applications that are able to utilize various infrastructure resources, specifically incorporating new types of IoT devices as computing resources, which are omnipresent in the smart city domain, we identify the following high-level requirements that will serve as guidelines for our contributions throughout this thesis:

1. We need the ability to manage and operate the massive amounts of IoT devices, which includes provisioning these devices in order to deploy and execute application logic on the devices.
2. We need a model that demands that IoT applications are designed and developed as clearly separated components that can be independently deployed.
3. We need the ability to describe and deploy dynamic IoT application topologies on multiple computing resources including resource-constrained IoT devices.
4. We need the ability to generically monitor deployed and running IoT applications, which are distributed among various computing resources and possibly do not follow traditional data processing models.
5. Finally, we need the ability to analyze running IoT applications and utilized computing resources based on available monitoring data in order to optimize the overall infrastructure deployments.

CHAPTER 4

Provisioning large-scale IoT Deployments

In this chapter, we present an approach that enables elastic provisioning of application components on resource-constrained and heterogeneous edge devices in large-scale IoT deployments. Our approach supports push-based as well as pull-based deployments. In addition, to improve scalability and reduce generated network traffic between cloud and edge infrastructure, we present a distributed provisioning approach that is able to operate within the deployment infrastructure close to the actual edge devices. We show that our solution is able to elastically provision large numbers of devices using a testbed based on a real-world industry scenario.

4.1 Overview

Traditional approaches for developing and designing IoT applications, such as AWS IoT[1] and Bluemix IoT Solutions[2], are based on rigid layered architectures [2]. The bottom layer, consisting of deployed IoT devices and their communication facilities, is managed using a middleware layer that exposes the underlying hardware in a unified manner for consumption by a top-level application layer, which executes relevant business logic and visualizes processed sensor data [73]. Such a layered architecture implies that business logic is only executed in the application layer, and IoT devices are assumed to be deployed with appropriate software and readily available [28]. However, in practice this is not the case. Currently, configuration and provisioning of IoT devices must largely be performed manually, making it difficult to quickly react to changes in application or infrastructure requirements. Moreover, we see the emergence of IoT devices (e.g., Intel IoT gateway[3],

[1]http://aws.amazon.com/iot/

[2]https://www.ibm.com/cloud-computing/bluemix/solutions/iot/

[3]https://www-ssl.intel.com/content/www/us/en/embedded/solutions/iot-gateway/overview.html

SmartThings Hub[4], and Raspberry Pi[5]) that offer functionality beyond basic connected sensors and provide constrained execution environments with limited processing, storage, and memory resources to execute device firmware. These currently unused execution environments can be incorporated in IoT systems to offload parts of the business logic onto devices. In the context of our work, we refer to these devices as *IoT gateways*.

In large-scale IoT systems, such as in the smart city domain, leveraging the processing capabilities of gateways is especially promising, as their currently untapped processing capabilities can be used to improve dependability, resilience, and performance of IoT applications by moving parts of the business logic towards the edge of the infrastructure. Incorporating edge devices as first-class execution environments in the design of IoT applications allows them to dynamically adapt to inevitable changes, such as new requirements or adjustments in regulations, by modifying their component deployment topology and edge processing logic. System integrators can avoid infrastructure silos and vendor lock-in by implementing custom business logic to be executed on gateways, and even purchase and sell such application components in IoT application markets [127]. However, the heterogeneity of currently available IoT gateways poses challenges for application delivery due to significant differences in device capabilities (e.g., available storage and processing resources), as well as deployed and deployable software components. Furthermore, the large number of devices in typical IoT systems calls for a scalable and elastic provisioning solution that is specifically tailored to the resource-constrained nature of IoT devices.

In this chapter, we present LEONORE, an infrastructure and toolset for provisioning application components on edge devices in large-scale IoT deployments. To accommodate the resource constraints of IoT gateways, installable application packages are fully prepared on the provisioning server and specifically catered to the device platform to be provisioned. Our solution allows for both, push- and pull-based provisioning of devices. Pull-based provisioning, a common approach in contemporary configuration management systems, allows devices to independently schedule provisioning runs at off-peak times, whereas push-based provisioning allows for greater control over the deployed application landscape by immediately initiating critical software updates or security fixes. Furthermore, our framework provides a distributed provisioning approach that allows deploying framework components within the deployment infrastructure to reduce network overhead. We illustrate the feasibility of our solution using a testbed based on a real-world IoT deployment from one of our industry partners. We show that LEONORE is able to elastically provision large numbers of IoT gateways in reasonable time. By deploying application packages with significantly different sizes, we furthermore show that our distributed provisioning mechanism can successfully scale with the size of IoT deployments and can substantially reduce required network bandwidth between edge devices and the central provisioning component.

The remainder of this chapter is structured as follows: In Section 4.2 we motivate our work and outline the specific requirements to be tackled. In Section 4.3 we introduce the

[4]http://www.smartthings.com/

[5]https://www.raspberrypi.org/

LEONORE infrastructure and toolset to address the identified requirements in deploying large-scale IoT systems, and present our distributed provisioning approach in Section 4.4. We provide detailed evaluations in Section 4.5, followed by a conclusion in Section 4.6.

4.2 Requirements

One of the most demanding aspects in the smart city domain is the ability to connect and manage millions of heterogeneous devices, which are emerging from the IoT. The extremely fast-paced evolution within the IoT and the changing requirements in the smart city domain itself make this not only a matter of handling large-scale deployments, but more importantly about supporting the ability to manage this change. A specifically demanding area facing these specific challenges is large-scale Building Management and Operations (BMO). BMO providers not only need to be able to manage and stage large numbers of new devices, they also need to be able to react on rapidly changing requirements to their existing infrastructure. However, current solutions are mostly manual and only deal with fragments of a BMO providers's infrastructure, which leads to the incapability of dealing with the vast amount of devices and changing requirements in an efficient, reliable, and cost-effective way. BMOs dealing with large-scale IoT systems need to be able to handle two distinct stages. The first is the initial deployment and staging of devices, the second is the management of updates of varying frequency and priorities. To illustrate this, we consider the case of a BMO that manages several hundreds of buildings with a broad variety of tenants in a large city. These buildings are equipped with a huge amount of heterogeneous IoT devices including simple sensors to detect smoke and heat, elevator and door controls, as well as complex cooling and heating systems. To reliably operate this infrastructure, the BMO relies on physical gateways [143], which provide constrained execution environments with limited processing, storage, and memory resources to execute the device firmware and simple routines. These gateways are usually installed once in a specific location in a building and then connected and integrated into an infrastructure solution to enable the basic bundling and management of a wide variety of connected devices. The current lack of standardization in this novel field combined with the current market situation leads to a significant heterogeneity in terms of software environments when it comes to these gateways. Initially, the gateways need to be staged with the necessary capabilities to ensure their basic functionality. They need to support the connected sensors, must run the latest firmware and have to be integrated into a specific deployment structure. This is followed by long term evolution requirements like changing deployments, shifting capabilities, as well as updating the software environment or firmware. A special case of updates are security updates and hot fixes that need to be deployed quickly to ensure that the infrastructure stays operational. Delays in these updates can expose severe security risks, which make them time critical. The increasing number of connected devices leads to an increased vulnerability to hacks and exploits, and in the IoT domain, where these devices are connected to the real world, this poses a major threat.

We therefore identify the following requirements in the context of this scenario:

- A provisioning framework must consider that participating gateways are resource-constrained in terms of their processing, memory, and storage capabilities.
- Scenarios dealing with large-scale deployments comprising thousands of gateways with a wide variety of different execution environments must be supported.
- Requirements of deployed applications change over time, which makes updates necessary. These updates can either be non-time-critical or time-critical.
- In order to sustain operations, updates need to be efficient and fast, and therefore have to be performed at runtime.

4.3 The LEONORE Framework

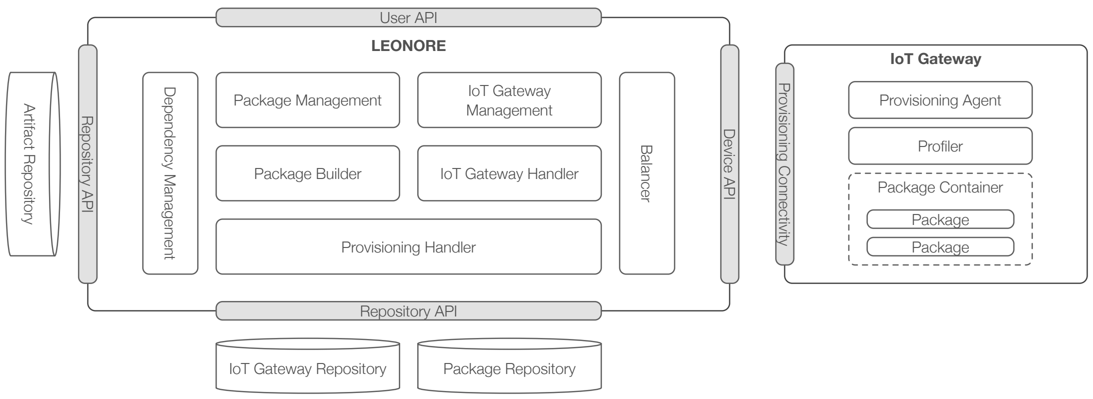


Figure 4.1: LEONORE – Overview

In order to address the previously defined requirements, we present LEONORE, an infrastructure to provision application components on gateways in large-scale IoT deployments. The overall architecture of our approach is depicted in Figure 4.1 and consists of the following components: (i) Application Packages, (ii) IoT gateways, and (iii) LEONORE, the provisioning framework. In the following, we discuss these components in more detail.

4.3.1 Application Packages

Usually an application in the IoT domain consists of different application components and supporting files (e.g., libraries and binaries), which we refer to as artifacts. To enable automatic provisioning of these artifacts, LEONORE builds gateway-specific application packages, which are a compound of various artifacts and have the following structure. First, each package has an `id`, which uniquely identifies the package. Second, each package contains a `binary` folder, to store required artifacts. Furthermore, it also contains the resolved application dependencies to avoid expensive dependency resolution on the gateway. Finally, in the `control` folder all instructions for installing, uninstalling, starting and stopping this package are included. Additionally, a `path` file defines the installation paths and the order of installing/uninstalling artifacts. With this approach the heavy lifting is done by the framework, and gateways only have to unpack the package and execute the provided installation instructions, which usually just copy artifacts in place without any additional processing.

4.3.2 IoT Gateway

To efficiently provision edge devices, we first need a general and generic representation of such devices. We analyzed the capabilities of several gateways that are commonly applied by our industry partners in the domain of Building Management Systems. Our findings show that in general such gateways have limited hardware components and use some rudimentary, tailored operating system (e.g., a BusyBox[6] user land on a stripped down Linux distribution). Installing or updating software components is a tedious manual task, since there are no supporting packaging or updating tools in place, as known from full-featured operating system distributions[7]. Furthermore, due to limited resources in terms of disk space, adding new capabilities often requires the removal of already installed components. Taking all these limitations into account, we derived the final representation of a gateway for our approach as depicted on the right-hand side in Figure 4.1. The IoT gateway has the following components: (i) a *container*, hosting application packages, (ii) a *profiler*, monitoring the current status of the gateway, (iii) an *agent*, communicating with the provisioning framework, and (iv) a *connectivity layer*, supporting different communication protocols and provisioning strategies.

Profiler

As mentioned above, gateways are usually resource-constrained, which means that they only provide limited disk space, memory and processing power. Therefore, keeping track of these resources is of utmost importance. In order to do that the profiler uses pre-defined interfaces to constantly monitor the underlying system (e.g., static information like ID, MAC-address, and instruction set, or dynamic information like disk- and memory-consumption). The profiler sends the collected information either periodically or on request to the provisioning framework. Based on this heartbeat information the provisioning framework can detect failures (e.g., gateway has a malfunction), which allows notifying the operator. Once the framework receives the heartbeat again, the gateway is considered back and running.

Application Package and Container

All packages that are not pre-installed on the IoT gateway have to be provisioned by the framework at runtime. Therefore, the IoT gateway uses a runtime container to store

[6]http://www.busybox.net
[7]e.g., apt (https://packages.qa.debian.org/a/apt.html) or rpm (http://www.rpm.org/)

and run application packages. By using a separate container we ensure that installing or removing packages does not interfere with the underlying system, and avoids expensive reboots or configuration procedures.

Provisioning Agent

An essential part of the overall provisioning framework is the provisioning agent. The pre-installed agent runs on each IoT gateway and manages application packages that are locally hosted and stored. The management tasks of the agent comprise installing, uninstalling, starting, and stopping packages. Furthermore, the agent is responsible for handling requests from the framework and triggers the respective actions on the IoT gateways (e.g., gather latest information via the profiler or trigger the provisioning of an application package).

Connectivity Layer

Since gateways usually use different software communication protocols in large real world deployments (e.g., oBIX[8] or CoAP[9]), our approach provides a pluggable connectivity layer. This layer can either reuse the deployed software communication protocols or extend services provided by the underlying operating system. Additionally, this layer provides extensible strategies to provision the gateway. In the current implementation, we provide two strategies: (i) a *pull-based* approach where the provisioning agent queries the framework for provisioning tasks, and (ii) a *push-based* approach where the framework pushes new updates to the gateway and the agent triggers the local provisioning.

4.3.3 The LEONORE Provisioning Framework

The enabling framework to provision edge devices in large-scale deployments is depicted on the left-hand side in Figure 4.1. LEONORE is a cloud-based framework and the overall design follows the microservice architecture [87]. This approach enables building scalable, flexible, and evolvable applications. Especially the flexible management and scaling of components is important for LEONORE when dealing with large-scale deployments. In the following, we introduce the main components of LEONORE and discuss the balancer-based scaling approach.

Repositories

To manage all relevant information for LEONORE, the framework relies on a number of repositories:

Artifact repository Usually, an application consists of multiple artifacts that are linked together to fulfill specific requirements. To handle these artifacts and make them

[8]http://www.obix.org
[9]http://coap.technology

reusable, a repository is used. The repository manages artifacts by storing source code, prebuilt binaries, dependencies, possible configurations, and further necessary information that is required for the application building process. Furthermore, the repository provides a mechanism to store different versions of an artifact.

IoT gateway repository This repository stores relevant gateway specific information that is needed for creating the deployable application package. This information includes: hardware configuration (e.g., disk space, memory, processor), software (e.g., kernel version, installed components/tools), as well as supported provisioning strategies and communication protocols. Additionally, for each IoT gateway the repository stores the provisioned application packages, which is important in case a different version of an installed package needs to be provisioned, since this might require the removal of an already installed version.

Package repository Application packages specifically built for a set of IoT gateways are stored in the package repository. This approach guarantees that packages are only built once, and all compatible gateways are provisioned with the same package. Furthermore, by storing the packages in a repository it is easier to scale the framework, since no data is stored in memory and therefore components can be easily replicated. After IoT gateways are successfully provisioned, the package is removed after a configurable amount of time to avoid storing unnecessary data.

Package Management

To provision application packages with LEONORE, users have to add artifacts via the package management component. This component is responsible for retrieving all necessary information (e.g., name and version), required binaries, available source files, configurations, policies, and dependencies on other artifacts, from the user. After the user has provided this information along with the artifacts, the package management stores them in the respective repository. The structure of the repository follows the layout of conventional software package management systems (e.g., Maven[10]).

Dependency Management

Since many applications depend on libraries or other applications, LEONORE utilizes the following mechanism to resolve these application dependencies. The data model for the dependency management consists of artifacts, releases, and dependencies between these releases. Each artifact has a set of releases, and each release has a set of dependencies to other artifacts. Thus, the releases and dependencies create a well-structured directed graph where releases are nodes and dependencies are directed edges. This model allows us to reuse well-known graph algorithms (i.e., depth-first search) to find all dependencies for a specific release. Therefore, according to the desired artifact, the dependency management

[10]http://maven.apache.org

finds a list of suitable artifacts and provides a plan that can be used to build the actual application package. The plan includes a dependency tree and all needed artifacts. The dependencies are represented as a directed graph, with nodes representing artifacts like applications, libraries, operating system tools, and hardware components, whereas edges represent dependencies between nodes. As an example, let us consider a Java application, where the application code is packaged as a jar file. In order to execute this application, it has a dependency on the JVM 1.8 for ARM runtime.

Package Builder

To create the actual application package that can be provisioned, the package builder is used. In order to build an application package, the builder performs the following steps: (i) retrieve gateway-specific information from the IoT gateway management, (ii) gather a list of suitable plans using the dependency management, (iii) build an application package based on the plan, (iv) notify the provisioning handler to trigger the actual provisioning if the build was successful, (v) try next plan in list if the build failed, (vi) store application package in package repository.

IoT Gateway Management and IoT Gateway Handler

In order to deal with the bootstrapping problem, i.e., to know which IoT gateways are available for provisioning, LEONORE follows the following approach. When an IoT gateway starts for the first time, the local provisioning agent registers the gateway with the framework by providing its unique identifier (e.g., derived from name, ID and mac-address) and the gathered profile data. Based on this information the IoT gateway management creates an entry in the IoT gateway repository and stores the provided information. The registration process is finalized by negotiating the supported provisioning strategy and communication protocol. This is possible, since each IoT gateway is pre-configured and provides some already installed communication protocols and provisioning strategies. Next, a suitable IoT gateway handler is assigned to this gateway. The handler is responsible for handling any further communication with the gateways, by providing the required communication protocols and provisioning strategies. IoT gateways that use the same protocols and strategies are grouped together and managed by a designated IoT gateway handler. This assures more flexibility and avoids mediating between protocols. Once the registration process is successful, the IoT gateway can be provisioned via the framework.

Provisioning Handler

To provision application packages, the provisioning handler first chooses a suitable provisioning strategy according to the information provided by the IoT gateway management. Then the handler checks if the respective package is already present in the package repository. If it is available it will be used, if not the handler triggers the building of gateway-specific application packages by invoking the package builder. Then, the provisioning handler executes the provisioning strategy. This means that the IoT gateway can either query the framework for application packages or the handler delegates the provisioning request to the respective IoT gateway handler, which pushes the update to the gateway and triggers the provisioning.

Balancer

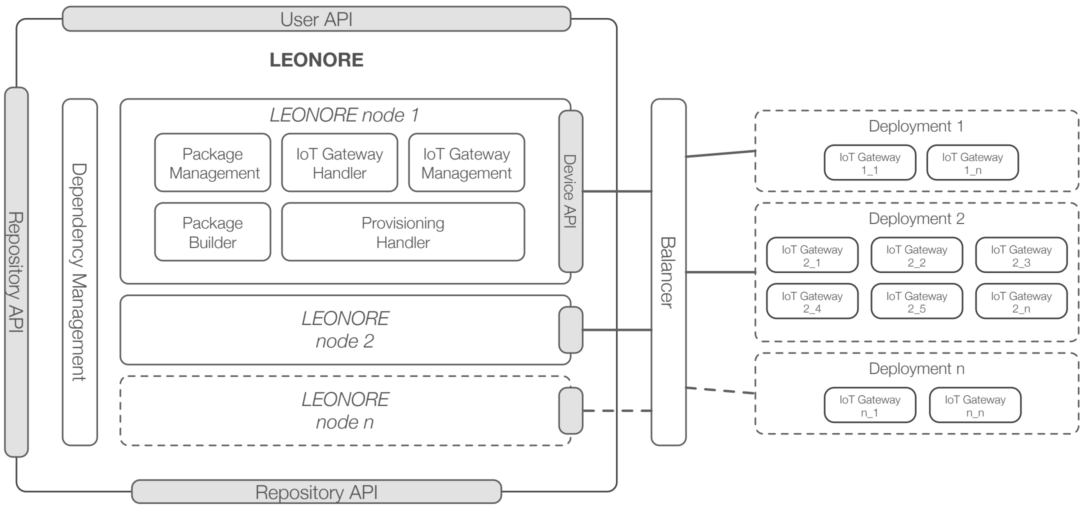


Figure 4.2: LEONORE – Balancer

Since LEONORE needs to provision large-scale deployments of IoT gateways, scalability is essential. Therefore, we provide several strategies to deal with the immense workload. First, the framework's design follows the microservice architecture principle. Thus, optimizing single components is relatively easy by moving them from one host to a more powerful host. Additionally, it is possible to scale components by replicating them and therefore distributing the workload across multiple computing resources. Following this approach, components of LEONORE are classified in scalable and not scalable. Components that should be scaleable are grouped together in so-called LEONORE nodes. These nodes comprise all components that are required to handle and provision IoT gateways. The classification in scalable and not scalable is flexible and can be adapted depending on the requirements. Now that LEONORE provides the ability to replicate components via the notion of nodes, we further need a component that is responsible for creating and destroying these nodes, as well as distributing incoming requests to them. To this end, we introduce a balancer. In general, a balancer aims to optimize resource usage, to minimize response time and to maximize throughput. Figure 4.2 depicts how LEONORE scales up with a growing number of deployments by using the balancer. In Figure 4.2 we see that the balancer receives incoming requests from IoT gateways deployed in different areas. Based on a pluggable strategy the balancer gathers a suitable node from the pool of available LEONORE nodes and assigns the gateway to this node. The node is then responsible for handling any further interaction with the respective IoT gateway. LEONORE nodes are deployed using a $N+1$ strategy with one active node and one hot standby initially. As load increases above the capacity of one node, the framework will immediately start to use the standby node for handling device provisioning requests, and furthermore start another LEONORE node to again maintain a hot standby node. Currently, LEONORE scales nodes based on the number of gateways to be provisioned. In the future, we will provide additional strategies, such as a location-aware strategy that aims at deploying nodes close to affected IoT gateways to reduce network overhead.

4.3.4 Provisioning of Application Packages

Whenever an artifact is requested for a certain deployment, LEONORE performs the following steps: (1) check if the requested artifact is available; (2) resolve the given deployment to retrieve the set of IoT gateways that need to provisioned; (3) find the responsible LEONORE nodes, group the gateways according to their node assignment and delegate the provisioning task to the node; (4) on each node: analyze if the requested artifact is compatible with every IoT gateway, and group gateways that require the same application package (e.g., equal hardware or installed packages); (5) on each node: for each group of IoT gateways resolve dependencies and create application package; (6) on each node: execute required provisioning strategy for each IoT gateway; (7) on each node: wait until IoT gateways successfully provisioned the package to complete the provisioning task; (8) check if all nodes have completed their provisioning task to finalize the overall provisioning.

4.4 LEONORE Optimization

After presenting the overall approach and the realization in the previous section, we now want to discuss certain limitations of our approach and propose an optimization addressing these shortcomings. In the approach presented so far, we assumed that the communication between the cloud and edge infrastructure is always available, reliable, and cheap. However, real world deployments use wireless communication links like 3G or GPRS that are not only slow and unreliable [117], but also expensive as they are usually charged based on transferred data. Additionally, the current approach puts the server-side framework under heavy load, which we already partly addressed by introducing a scalable LEONORE node concept. Nevertheless, by scaling LEONORE across several nodes and therefore provisioning more resources in the cloud, operating expenses increase along with the additional overhead of managing the provisioning and releasing of these nodes. In order to tackle these limitations, we apply the core notion of offloading business logic to the infrastructure edge to LEONORE itself, moving parts of the provisioning logic to suitable gateways in the field.

4.4.1 Server-side Extensions

To allow for the new concept of LEONORE local nodes, we extend LEONORE by adding several new components, which we describe in the following and are depicted on the

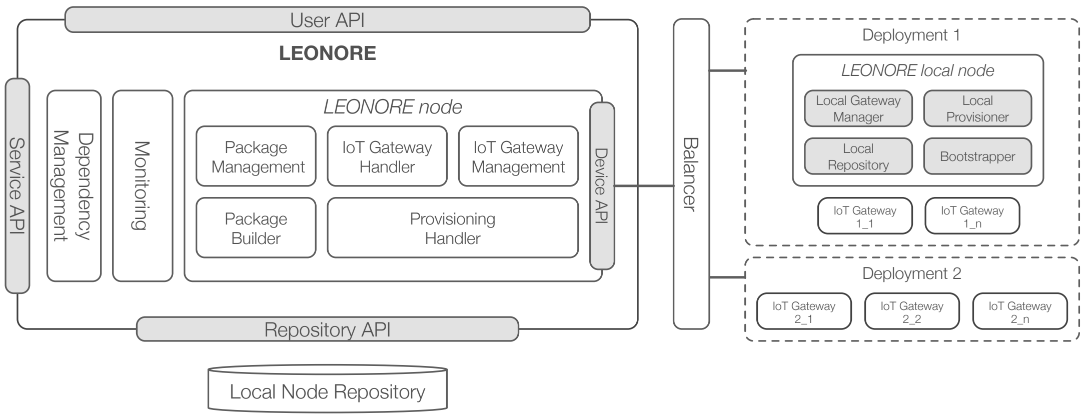


Figure 4.3: LEONORE – Local Provisioner

left-hand side in Figure 4.3.

Monitoring

We introduce a central (i.e., not replicated) monitoring component that collects the following information of the overall framework: (i) The number of provisioned packages. (ii) The consumed bandwidth based on the number of provisioned packages and incoming pulling requests. (iii) The overall time that is needed for provisioning the edge infrastructure. (iv) Information about the provisioned gateways, e.g., used disk space, memory and processing power. (v) For each deployed LEONORE node, relevant metrics of the node (e.g., average response time, uptime, and load average). Based on this monitoring data, collected by LEONORE, it is possible to decide that a LEONORE local node needs to be provisioned and where in the edge infrastructure it is feasible to do so.

Service API

Since in the current version LEONORE does not automatically decide when and where to provision a LEONORE local node, we created a service API that allows operators to retrieve the collected data from the monitoring component. Additionally, operators query for gateways that are suitable for hosting a LEONORE local node. Finally, the service API allows operators to trigger the provisioning and releasing of LEONORE local nodes in the edge infrastructure.

Local Node Repository

To keep track of provisioned LEONORE local nodes, we added a separate repository. For each LEONORE local node deployment, we store in this repository the ID of the node and the gateway in the edge infrastructure that is provisioned with the respective node. The combination of node ID, gateway ID, and gateway IP uniquely identifies the node deployment. Furthermore, the repository stores the current status of a LEONORE local node using the monitoring component.

4.4.2 LEONORE Local Node

In essence, the LEONORE local node provides the same capabilities as the server-side LEONORE node, but is specifically catered to be more lightweight in terms of memory consumption and CPU usage. This approach allows to execute LEONORE local nodes on gateways residing in the edge infrastructure, which only provide a fraction of the processing power of cloud resources. The architecture of the LEONORE local node is depicted on the right-hand side in Figure 4.3. In the following, we outline the basic components of a LEONORE local node.

Local Gateway Handler

Like the gateway handler on the server-side LEONORE node, this component manages a local cluster of gateways. In order to know which gateways need to be handled, how to communicate with them, and what kind of provisioning strategy needs to be used, LEONORE provides this information when provisioning a LEONORE local node. Compared to the server-side manager, this approach is not as flexible, but since the local gateway handler is specifically built for this local cluster of gateways, we keep the overall footprint of the node small by avoiding resource expensive protocol mediation and bootstrapping of gateways.

Local Repository

In order to save bandwidth, the application package that needs to provisioned is not transferred to each gateway, but only sent to the LEONORE local node, which then takes care of provisioning the respective gateways. The node stores the package in a local cache repository using available RAM and/or disk resources if available. This allows for fast read and write access, while explicitly considering the underlying resources of the gateway. After successful provisioning, the cache is cleared to save memory on the gateway.

Local Provisioner

In contrast to the provisioner on the server-side LEONORE node, the local provisioner is more lightweight, since it does not have to deal with the process of building application packages, but only uses the already transferred application package to provision the gateways. Furthermore, the local provisioner provides an optimized local provisioning strategy, which solely supports the push mechanism, since it consumes less resources and puts the node under less load compared to polling in short intervals.

Bootstrapper

Once a LEONORE local node gets provisioned, the bootstrapper component of the local node takes care of the following two tasks. First, it registers the respective local node at LEONORE, which guarantees that the framework is aware of all deployed local nodes. Second, once the registration was successful and the framework accepted the registration request, the bootstrapper gathers health and status information about the local node. This information is then periodically published at a configurable interval, which is then collected by LEONORE.

4.4.3 LEONORE Local Node Deployment

In order to deploy a LEONORE local node in the edge infrastructure, the operator uses the previously described LEONORE service API to retrieve a list of suitable gateways that are capable to run a local node. Next, the operator chooses how to distribute the local nodes across the edge infrastructure. Since the distribution of nodes can depend on various factors, such as available connectivity or logical location, LEONORE provides a pluggable distribution mechanism that can be easily extended. Following our microservice architecture, this can be done by adding an additional microservice to the framework that specifically implements a new distribution approach. In the current implementation, we form clusters of gateways based on physical proximity (e.g., all gateways that are residing on the same floor). Based on these clusters, the distribution mechanism elects a suitable gateway to host the LEONORE local node. Next, after selecting the gateways that will host the local nodes, LEONORE provisions them using the same approach we use for ordinary artifacts. This means that the local node artifacts, which are already residing in the artifact repository, get bundled to an application package and then transferred to the gateway. On the gateway the application package is installed and started by the provisioning agent. Finally, after the startup of the local node the bootstrapper registers the local node at LEONORE. After registration, the LEONORE local node is ready for serving provisioning requests.

4.4.4 Application Provisioning with LEONORE Local Nodes

As described in Section 4.3.4, when an artifact is requested for a specific deployment, LEONORE checks if the artifact is available, finds the responsible LEONORE node according to the retrieved set of gateways and delegates the provisioning to the respective server-side node. On the server side, the gateways are grouped based on available capabilities and application packages are created. Additionally, each server-side node now clusters gateways based on their physical proximity. For each cluster, the server-side node queries the local node repository for an available LEONORE local node. If no local node is present, the server-side node follows the original approach described in Section 4.3.4 and executes the required provisioning strategy for each gateway. However, if a local node is available, it transfers the application package and the directive to provision the cluster of gateways to this local node. The local node then takes care of provisioning these gateways by executing the optimized push-based approach.

4.5 Evaluation

To evaluate our provisioning framework we created a test setup in the cloud using CoreOS to virtualize devices as Docker containers. IoT gateways in our experiments use two types of provisioning strategies – a pull and a push based approach.

When an IoT gateway uses the pull approach, the gateway's agent polls the provisioning framework for new tasks in a configurable interval (e.g., every second). The framework only provides new provisioning tasks for the IoT gateway, which collects and executes these tasks. With short polling intervals, this approach generates increased load on the framework, consumes more bandwidth, and uses more resources on the IoT gateways, but is more fault-tolerant in case of connectivity problems due to inherently frequent retries. For the push-based approach, the IoT gateway's agent only registers the gateway once at the framework and then remains idle until the framework pushes an update. When the agent gets pushed by the framework, it collects the provisioning task, executes it and returns to the idle state. In general, the push-based approach generates less load on both the IoT gateway and framework, but is more vulnerable to connectivity problems and operators need to take care to not inadvertently disrupt gateway operations by placing additional load on it.

To simulate real-world provisioning requests, we use the following two application packages. The first package uses the Sedona Virtual Machine[11] (SVM). SVM is written in ANSI C and is highly portable by design. It allows to execute applications written in the Sedona programming language and is optimized to run on platforms with less than 100KB of memory. For our experiments we developed a small sample application and used SVM Version 1.2.28. The final application package created by LEONORE has approximately 120KB – including the application code (.sab, .sax, .scode and Kits-file) and the required SVM binary.

As second package we use Java 8 for ARM[12] (JVM). In general, using Java on an embedded device is a challenging task, since the JVM binary is quite big and often does not fit due to limited disk space. However, for our experiments we created a compact[13] Java package specifically for our gateway. Additionally, we developed a small sample application that pushes temperature readings to a web server. In total, the JVM application package created by LEONORE has approximately 12MB – including the application code (compiled .class files), JVM binary, and libraries.

In the remainder of this section we give an overview of the used cloud setup, present four scenarios, and analyze the gathered results.

[11]http://www.sedonadev.org

[12]http://www.oracle.com/technetwork/java/javase/downloads/jdk8-arm-downloads-2187472.html

[13]http://docs.oracle.com/javase/8/embedded/develop-apps-platforms/jrecreate.htm

4.5.1 Setup

To see how LEONORE deals with large-scale deployments, we created an IoT testbed (Figure 4.4) in our private OpenStack[14] cloud. In order to simulate large-scale deployments, we first created a snapshot of a real-world gateway that is used by our industry partner. Based on this snapshot, we created an image that can be run in Docker[15]. The running image (Docker container) is then used to virtualize and mimic the physical gateway in our cloud.

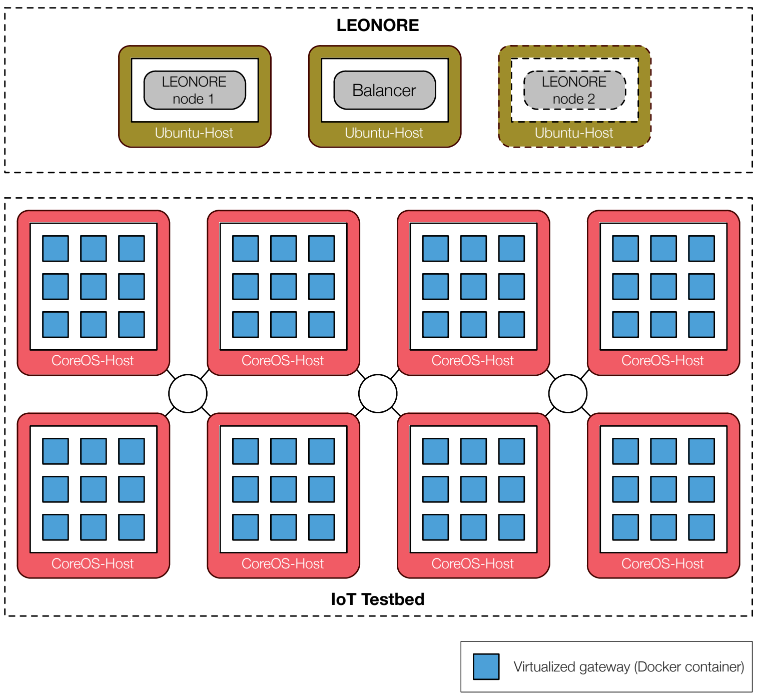
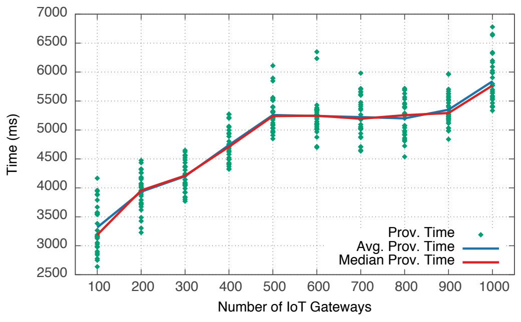


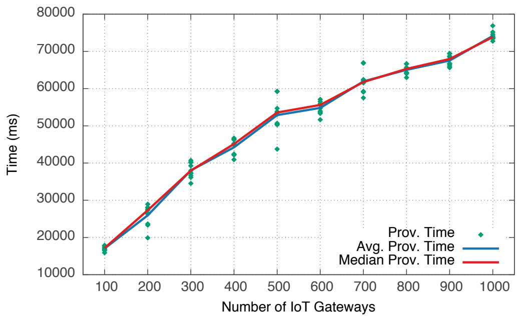
Figure 4.4: Evaluation – Setup

Since for our evaluation we want to use several thousand virtualized gateways, we employed CoreOS[16] clusters. In general, CoreOS is a light-weight Linux distribution designed for security, consistency, and reliability. Instead of installing packages via a package management system like apt, CoreOS uses Docker to manage services at a higher level of abstraction. The service code and all dependencies are packaged within a container that can be run on one or many CoreOS machines. Containers provide benefits similar to full-blown virtual machines, but focus on applications instead of entire virtualized hosts. Since containers use the Linux kernel of the host, they have very little performance overhead, reducing the amount of required compute resources compared

[14]http://www.openstack.org
[15]https://www.docker.com
[16]https://coreos.com



(a) Provisioning Time for SVM



(b) Provisioning Time for JVM

Figure 4.5: Provisioning Time – Pull Strategy

to VMs. CoreOS also provides fleet[17], a distributed init system that allows to treat a CoreOS cluster as if it is a single shared init system. We used fleet's notion of service units to dynamically generate according fleet unit files and use fleet for the automated deployment of virtualized gateways.

For our experiments we used the following setup: a CoreOS cluster of 8-16 virtual machines (depending on the scenario), where each VM is based on CoreOS 647.0.0 and uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB Disk space). Our gateway-specific framework components are pre-installed in the containers.

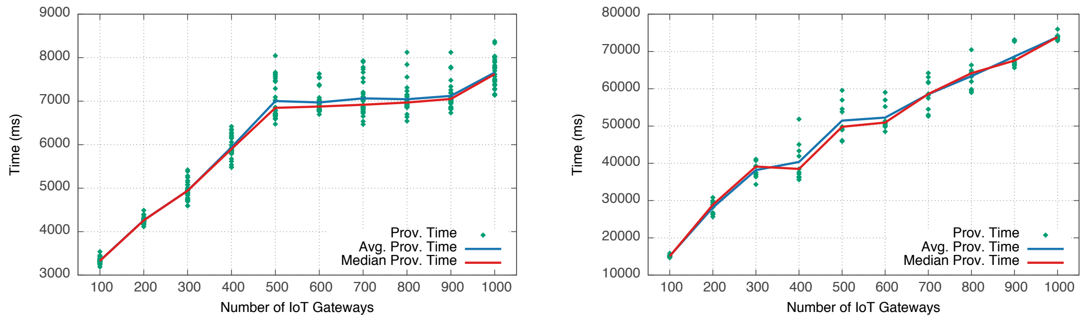
The LEONORE framework is initially distributed over 2 VMs using Ubuntu 14.04. The first VM hosts the balancer and uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB Disk space). In order to represent a LEONORE node we created a reusable snapshot of a VM hosting all necessary LEONORE framework components and repositories. For the initial deployment of LEONORE two instances of this snapshot are started at the beginning of the experiment. However, only of of them is initially used by the framework, whereas the other acts as standby node. During the experiments LEONORE, more precisely the balancer, spins up this additional standby node to distribute the load created by the gateways. The VMs hosting the LEONORE nodes use the m2.medium flavor (5760MB Ram, 3 VCPUs and 40GB Disk space).

In the following scenarios we measured the overall execution time needed for provisioning an increasing number of devices. The provisioning time includes analyzing desired gateways, building gateway-specific application packages, transferring the packages to the gateways, installing the packages on the gateway, and executing them.

4.5.2 Scenario 1: 100 - 1000 IoT Gateways

For the first experiments we picked a scenario with 1000 virtual gateways. The scale of this scenario corresponds to a medium building management system, containing several big buildings (each with more than 10 floors). The 1000 virtual gateways are distributed

[17]https://github.com/coreos/fleet



(a) Provisioning Time for SVM (b) Provisioning Time for JVM

Figure 4.6: Provisioning Time – Push Strategy

among a CoreOS cluster consisting of 8 machines, where each machine hosts 125 containers. To demonstrate the scalability of our framework we show how our approach behaves with increasing load (number of gateways). For this scenario the balancer uses a scaling strategy that spins up another standby node when reaching 500 IoT gateways.

Figure 4.5 shows the overall execution time of the provisioning process for different deployments using the pull-based (gateways poll the framework every second) approach. In Figure 4.5a we show the execution time for provisioning the SVM application package. We see that the execution time increases almost linear until reaching 300 IoT gateways and then shows a sharper increase up to 500. When reaching 500 gateways, the balancer spins up another standby node and evenly schedules requests to the two active nodes. Therefore, provisioning time slightly decreases and at approximately 600 becomes constant. When reaching 900 IoT gateways, the provisioning time starts to rise again, which means that at this point both LEONORE nodes are fully loaded. In order to investigate possible outliers during the evaluation, we created a scatter plot, which is also depicted in Figure 4.5a. Since the SVM application is quite small and the polling interval of one second has a strong impact on the overall execution time, we executed each experiment 30 times. In the scatter plot we notice that at 600 IoT gateways we have some executions that finished more slowly, which is caused by the high network load and small polling interval. In general, the deviation of provisioning times is small. This shows that provisioning using the polling strategy is stable and provides reliable results.

Figure 4.5b shows the execution time when provisioning the JVM application package. We clearly see that due to the increased size of the package the provisioning takes noticeably longer than for the SVM package. When the deployment reaches 500 IoT gateways, the balancer kicks in, which leads to a slight increase. In general, we notice that the provisioning of the JVM package scales linearly and produces almost no outliers, as one can see in the scatter plot in Figure 4.5b. Since this application package is quite big and therefore the provisioning time also increases significantly, the overhead of the polling approach is not noticeable.

Figure 4.6 shows the overall execution time of the provisioning process for different deployments using the push-based (framework pushes provisioning tasks to IoT gateways) approach. In Figure 4.6a we see the overall execution time for provisioning the SVM application package. We notice a sharp increase up to 500 IoT gateways, which is due to the framework pushing requests to all gateways at once and therefore leads to a high load on both the IoT gateways and the framework. Once the balancer spins up another standby node, the execution time is almost constant, because the load is evenly distributed. When the deployment size reaches 900 IoT gateways, the execution time starts to rise again, which indicates that at this scale both nodes are fully loaded. The corresponding scatter plot is also depicted in Figure 4.6a, which reveals that there is only a very small deviation among the data points.

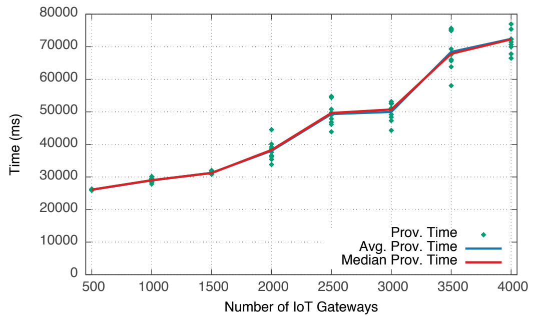
Figure 4.6b depicts the provisioning time when using the JVM application package. Taking the results of the polling approach into account, we notice that the initial execution times are identical. However, at 300 IoT gateways we see that the initial overhead of the pushing approach is compensated and therefore the execution time decreases a little bit. From 400 to 500 IoT gateways, the node reaches maximal load. After the deployment size reaches 500, the balancer schedules the load evenly on two LEONORE nodes. The corresponding scatter plot, depicted in Figure 4.6b, shows that the deviation of data points is very small and the execution time increases linearly.

After comparing both approaches, we see that our framework scales almost linearly and that for smaller application packages the pull-based approach is faster. For bigger packages both approaches put the framework under heavy load, but produced similar results.

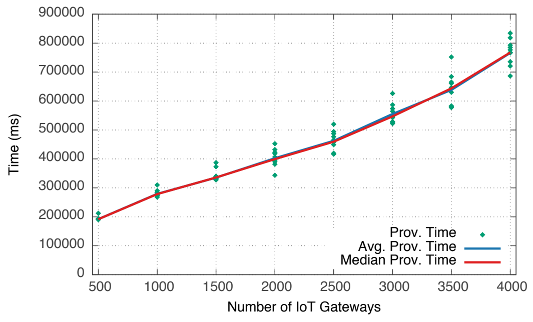
4.5.3 Scenario 2: 500 - 4000 IoT Gateways

For the second experiment we used a scenario with 4000 virtual gateways, which corresponds to a large building management system containing dozens of big buildings (each with more than 10 floors). The 4000 virtual gateways are distributed among two CoreOS clusters, each consisting of 8 machines, where each machine hosts 250 containers. With this scenario we investigate how our framework scales when dealing with a large-scale deployment by using a scaling strategy that spins up another standby node when reaching 2500 IoT gateways.

Figure 4.7 shows the overall execution time of the provisioning process for different numbers of gateways using the push-based approach. In Figure 4.7a we notice that due to the deployment scale the overall execution time for provisioning the SVM application package got slower compared to the first scenario. This is expected since for this scenario we doubled the amount of CoreOS hosts and deployed twice as many containers on each CoreOS machine. This increase, in both the hosts and containers, generates a lot of traffic for the underlying network infrastructure of our cloud, which causes slower response times and therefore the overall provisioning takes longer. Furthermore, for this scenario we configured the balancer to handle 2500 IoT gateways per LEONORE node. We clearly see that up to 2500 IoT gateways, the execution time increases almost linearly. At 2500 the balancer schedules the requests evenly to two nodes, which causes a constant execution time. When reaching 3000 deployments, the execution time rises again, but once more starts to flatten at 4000. When looking at the scatter plot depicted in Figure 4.7a



(a) Provisioning Time for SVM



(b) Provisioning Time for JVM

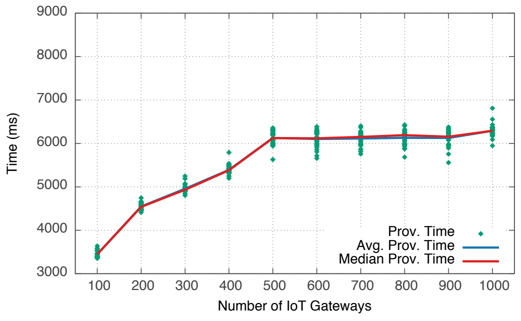
Figure 4.7: Provisioning Time – Push Strategy, Large Deployment

we see that at the beginning of the experiments the deviation among data points is very small and gets bigger with increasing number of IoT gateways. Figure 4.7b depicts the provisioning time when using the JVM application package. Compared to the first scenario, we also notice that the overall execution time got slower. However, now we see that the execution time increases linearly throughout the complete experiment, which provides stable provisioning results. This fact is also supported by the scatter plot that shows only small deviations among the results. Taking both experiments into account, we clearly see that our framework deals well with this rather large scenario and provides almost linear scale.

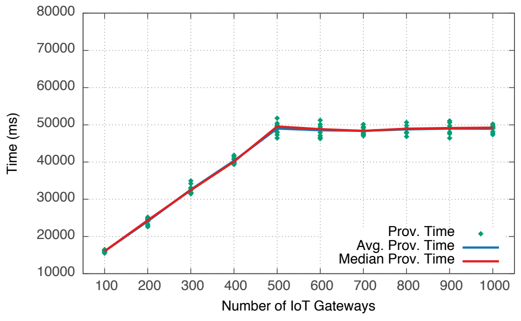
Distributed Provisioning using LEONORE Local Nodes. In order to evaluate the LEONORE local node optimization, we reused the setup discussed in Section 4.5.1. However, for this evaluation we spawn additional IoT gateways for hosting the LEONORE local nodes. For the first scenario, where we evaluate the provisioning time by using up to 1000 virtual gateways, we start 8 additional gateways. These 8 additional gateways, are distributed across the IoT testbed, so that on each CoreOS-Host one of these additional gateways is running. For the second scenario, where we provision up to 4000 virtual gateways, we use 16 additional gateways (i.e., one on each CoreOS Host). After distributing these additional gateways, we then provisioned them to run the LEONORE local node. Once these gateways got provisioned, their sole purpose is to host the local nodes and they are not changed throughout the evaluation. We decided to pre-provision the local nodes to conduct experiments that follow the same provisioning process as used in the evaluation above. Additionally, we argue that the provisioning and distribution of LEONORE local nodes will only happen sporadically and therefore this additional time should not contribute to the overall provisioning time.

4.5.4 Scenario 3: 100 - 1000 IoT Gateways

For the first experiments we picked a scenario with 1000 virtual gateways, which corresponds to a medium building management system. The 1000 virtual gateways are distributed among a CoreOS cluster consisting of 8 machines, where each machine hosts



(a) Provisioning Time for SVM



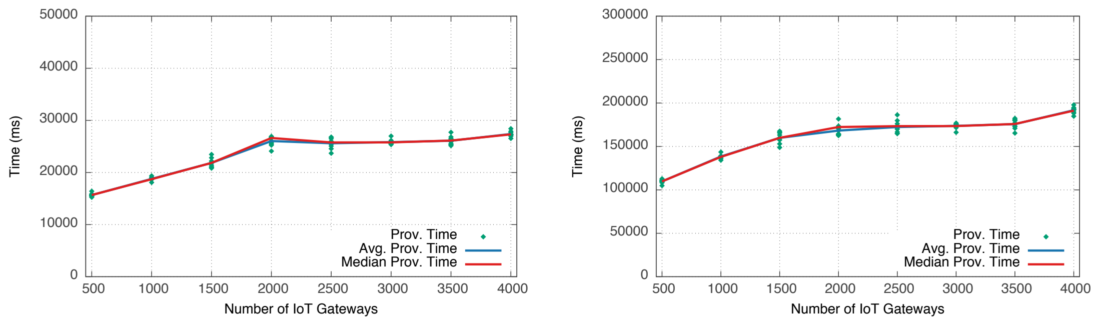
(b) Provisioning Time for JVM

Figure 4.8: Provisioning Time – LEONORE local node

125 containers and an additional container hosting the LEONORE local node. To demonstrate the scalability of our framework we show how our approach behaves with increasing load (number of gateways).

Figure 4.8 shows the overall execution times of the provisioning process for different deployments by using the push-based approach (framework pushes provisioning tasks to IoT gateways), with LEONORE local nodes in place. In Figure 4.8a we see the overall execution time for provisioning the SVM application package. Compared to the basic provisioning approach (Figure 4.6a), we notice an initial steeper increase of the provisioning time for the LEONORE local node approach. This initial overhead is expected, since the local nodes deployed on the gateway are not as powerful as the server-side nodes and therefore need considerably longer for serving the gateways. However, after 500 the initial overhead is compensated and the local node provisioning provides faster results. We see that the provisioning time for the local node stays stable after 500 gateways and results in better overall provisioning times, compared to the original approach. This fact can also be seen in the included scatter plot. The overall improvement can be explained by the fact that the overall load on the framework is more evenly distributed among the local nodes and therefore provides faster provisioning times. We see that the provisioning time improves on average by 10% using the SVM package with the LEONORE local node approach.

Figure 4.8b depicts the provisioning time when using the JVM application package. We notice that initially the execution times of both the original (Figure 4.6b) and local node provisioning approach are identical. However, after reaching 200 gateways we see that the optimized approach starts to outperform the server-based provisioning. The improvement becomes more significant after reaching 500 gateways, since from that scale onwards the provisioning using LEONORE local nodes provides a constant and good execution time, compared to the ever increasing server-side provisioning. This effect is expected, since each local provisioning node only has to handle a fraction of the total gateway deployment. Additionally, considering the fact that the JVM package is quite big, the local node provisioning approach generates significantly less cloud to edge communication, since the application package gets only sent once to each local node



(a) Provisioning Time for SVM   (b) Provisioning Time for JVM

Figure 4.9: Provisioning Time – Large Deployment, LEONORE local node

and further provisioning happens only within the edge infrastructure. These facts lead to an average improvement of 17% when provisioning the JVM application package. Furthermore, in general the local node approach creates more stable results, which can be seen in the scatter plot depicted in Figure 4.8b.

4.5.5 Scenario 4: 500 - 4000 IoT Gateways

For the second experiment we used a scenario with 4000 virtual gateways, which corresponds to a large building management system containing dozens of big buildings (each with more than 10 floors). The 4000 virtual gateways are distributed among two CoreOS clusters, each consisting of 8 machines, where each machine hosts 250 containers and an additional container hosting the LEONORE local node. With this scenario we want to see how our framework scales when dealing with a large-scale deployment.

Figure 4.9 shows the overall execution times of the local node provisioning process for different numbers of deployments by using the push-based approach and the SVM respectively the JVM application package. Once again we compare the provisioning times with (Figure 4.9) and without (Figure 4.7) local nodes in place. In Figure 4.9a we notice that the execution time increases almost linearly when using the SVM application package. However, due to the scale of the scenario we see that the server-side nodes need to handle a lot of load, compared to the distributed local node setup, which explains why the local node setup provides better provisioning times from the beginning. When reaching 2000 IoT gateways, the execution time using the local node provisioning stays constant and does not increase anymore. Here, the LEONORE local node approach generates only a fraction of the load on the server-side framework, compared to the original approach, which causes the drastic improvement in execution time. On average, the distributed work model and reduced bandwidth consumption of LEONORE local nodes, improves the provisioning time by 49%. In Figure 4.9b we notice that the local node provisioning approach for the JVM package also provides significantly better results than the server side approach. Since the JVM package is quite big, the distributed work model performs even better and therefore improves the overall provisioning time on average by 64%. Furthermore, by looking at the included scatter plots in both figures we see that the optimized provisioning approach provides stable results, since the deviation among data points is very small. We clearly see that even though our framework already dealt well with this rather large scenario, by introducing LEONORE local nodes we were able to further improve the overall provisioning time.

4.5.6 Final Remarks

After finishing our experiments and evaluating the results, we see that the introduction of LEONORE local nodes leads to a significant improvement in terms of provisioning time. Additionally, next to the measurement of the provisioning time, we also monitored the amount of data that gets transferred from the cloud to the edge infrastructure during the provisioning of gateway deployments. Our findings show that by using local nodes deployed in the edge, which are managing a cluster of gateways, we reduce the bandwidth usage drastically, since by using local nodes we avoid sending the provisioning package to each gateway, but only send this package once to a local node managing this cluster. Even by taking into account that the LEONORE local nodes need to be provisioned initially, does not diminish the savings. In order to illustrate the bandwidth savings, we will use some numbers from the evaluation scenario above. Let us consider the scenario where we need to provision 1000 gateways with the SVM application package, which has a size of 120KB. With the original provisioning approach, two server-side nodes of LEONORE would transfer the package to each gateway, resulting in 120MB of data that gets sent from the cloud to the edge for each provisioning request. Compared to that, by using LEONORE local nodes we cluster the gateway deployment to 8 clusters and therefore deploy 8 LEONORE local nodes. The local node application package has a size of 14MB and therefore the transferred data sums up to 112MB. Additionally, when provisioning the 1000 gateways we now transfer the SVM application package only to these 8 local nodes and therefore produce in total 112.96MB. For a the relatively small SVM application package, we already save 6% for the initial provisioning cycle. After that, since the LEONORE local nodes are already deployed, we save 99% of the bandwidth for every additional provisioning request.

4.6 Summary

In this chapter we presented LEONORE, a novel infrastructure and toolset to elastically provision application packages on resource-constrained, heterogeneous edge devices in large-scale IoT deployments. LEONORE enables push-based as well as pull-based deployments supporting a vast array of different IoT topology and infrastructure requirements. By introducing the concept of LEONORE local nodes we further enabled efficient distributed deployments in these constrained environments in order to further improve scalability and reduce generated network traffic between cloud and edge infrastructure. For evaluation purposes we utilized a large scale testbed based on a real-world industry scenario. Our evaluation clearly demonstrated that LEONORE is able to elastically provision large numbers of devices in an efficient manner. We further showed that our local node extension significantly improved provisioning time while drastically reducing bandwidth consumption, a factor that is crucial in such constrained environments.

CHAPTER 5

Deploying elastic IoT Applications

In this chapter, we present an approach for the dynamic generation of optimized deployment topologies for IoT applications that are tailored to the currently available physical infrastructure. Based on a declarative, constraint-based model of the desired application deployment, our approach enables flexible provisioning of application components on edge devices deployed in the field. Using our approach, applications can furthermore evolve their deployment topologies at runtime in order to react on environmental changes, such as changing request loads. Our approach supports different IoT application topologies and we show that our solution elastically provisions application deployment topologies using a cloud-based testbed.

5.1 Overview

IoT applications are expected to manage and integrate an ever-increasing number of heterogeneous devices to sense and manipulate their environment. Increasingly, such devices do not only serve as simple sensors or actors, but also provide constrained execution environments with limited processing, memory, and storage capabilities. In the context of our work, we refer to such devices as IoT gateways. By exploiting this accrued execution capabilities offered by IoT gateways, applications can offload parts of their business logic to the edge of the infrastructure to reduce communication overhead and increase application robustness [129]. This explicit consideration of edge devices in IoT application design is especially important for applications deployed on cloud computing [8] infrastructure. The cloud provides access to virtually unlimited resources that can be programmatically provisioned with a pay-as-you-go pricing model, enabling applications to elastically adjust their deployment topology to match their current resource usage and according cost to the actual request load.

In addition to the traditional design considerations for cloud applications, IoT cloud applications must be designed to cope with issues arising from geographic distribution of edge devices, network latency and outages, as well as regulatory requirements. We argue that edge devices must be treated as first-class citizens when designing IoT cloud applications and the traditional notion of cloud resource elasticity [35] needs to be extended to include such heterogeneous IoT gateways deployed at the infrastructure edge, enabling interaction with the physical world. To allow for the flexible provisioning of applications whose deployment topology changes over time due to components being offloaded to IoT gateways, applications need to be composed of clearly separated components that can be independently deployed. The microservices architecture [87] recently emerged as a pragmatic implementation of the service-oriented architecture paradigm and provides a natural fit for creating such IoT cloud applications. We argue that future large-scale IoT systems will use this architectural style to cope with their inherent complexities and allow for seamless adaptation of their deployment topologies. Uptake of the microservice architecture will furthermore allow for the creation of IoT application markets (e.g., [127]) for practitioners to purchase and sell domain-specific application components.

IoT gateways can be considered an extension of the available cloud infrastructure, but their constrained execution environments and the fact that they are deployed at customer premises to integrate and connect to local sensors and actors requires special consideration when provisioning components on IoT gateways. By carefully deciding when to deploy certain components on gateways or cloud infrastructure, IoT cloud applications can effectively manage the inherent cost-benefit trade-off of using edge infrastructure, leveraging cheap communication at the infrastructure edge while minimizing expensive (and possibly slow or unreliable) communication to the cloud, while also considering processing, memory, and storage capabilities of available IoT gateways. It is important to note that changes in application deployment topologies will not only be necessary whenever a new application needs to be deployed, but can also be caused by environmental changes, such as changing request patterns, changes in the physical edge infrastructure (e.g., adding/removing sensors or IoT gateways), evolutionary changes in application business logic throughout its lifecycle, or evolving non-functional requirements.

In this chapter, we present DIANE, a framework for dynamically generating optimized deployment topologies for IoT cloud applications tailored to the available physical infrastructure. Using a declarative, constraint-based model of the desired application deployment, our approach enables flexible provisioning of application components on both, cloud infrastructure, as well as deployed IoT gateways. DIANE is furthermore continuously monitoring the available edge infrastructure and can autonomously optimize application deployment topologies in reaction to changes in the application environment, such as significant changes in request load, network partitions, or device failures. In addition, we introduce a two-fold optimization mechanism that enables the evolution of application deployment topologies at runtime in reaction to changes in their execution environment.

The remainder of this chapter is structured as follows: In Section 5.2 we outline specific requirements that need to be addressed. In Section 5.3 we introduce the DI-

ANE framework to dynamically create application deployment topologies for large-scale IoT cloud systems, and present our approach for optimizing deployments at runtime in Section 5.4. We provide detailed evaluations in Section 5.5, followed by a conclusion in Section 5.6.

5.2 Requirements

The emergence of the IoT in combination with the advent and rapid adoption of the smart city paradigm give rise to a domain of edge devices that are pervasively deployed in large numbers around the globe. As outlined previously, the convergence of cloud computing and IoT paradigms, and especially the evolution of IoT gateways to include constrained execution environments, allows for systems with ever changing deployment topologies due to various evolving factors. Specifically, vital aspects of the smart city domain, like BMSs that need to deal with billions of devices, or Traffic Control Systems (TCS) that depend on optimal resource utilization in order to handle large amounts of sensor data, need to be able to optimize their deployment topologies both during deployment and at runtime in order to enable optimal resource utilization.

To allow for dynamic generation of optimal deployment topologies for such applications, a solution must meet the following requirements:

1. It needs to enable *optimal utilization of edge devices* with
2. the ability to *dynamically move application logic to these devices.*
3. Furthermore, it shall allow for *deployment topologies to evolve during runtime* and
4. needs to respect *non-functional requirements* that arise in this context.

5.3 The DIANE Framework

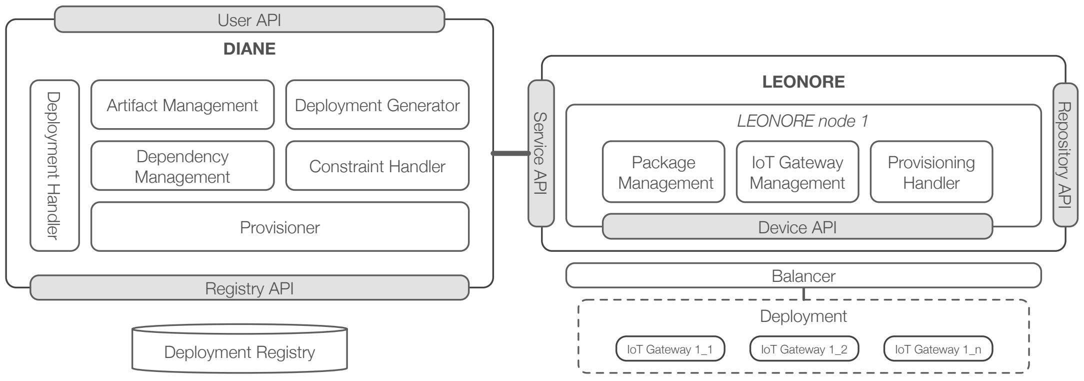


Figure 5.1: DIANE – Overview

In order to address the previously identified requirements, we present DIANE, a framework for the dynamic generation of deployment topologies for IoT applications and application components, and the respective provisioning of these deployment topologies on edge devices in large-scale IoT deployments. The overall architecture of our approach is depicted in Figure 5.1 and consists of the following top-level components: (i) DIANE, and (ii) LEONORE. In the following, we describe these components in more detail and discuss the design and implementation of IoT applications.

5.3.1 IoT Application Design and Implementation

To dynamically generate deployment topologies for IoT applications, the design and implementation of such applications have to follow the microservices architecture approach [87], which enables developers to build flexible applications whose components can be independently evolved and managed. Therefore, each component of an application has to be self-contained, able to run separately, and facilitate loosely coupled communication for interacting with other components. In addition to this application design approach, we are using MADCAT [54] for describing the overall application and its components. MADCAT allows for the creation of applications by addressing the complete application lifecycle, from architectural design to concrete deployment topologies provisioned and executed on actual infrastructure. For our approach, we focus on Technical Units (TUs) and Deployment Units (DUs) to describe applications and their components.

`Technical Units` are used to describe application components by considering abstract architectural concerns and concrete deployment artifacts to capture technology decisions that depend on the actual implementation. To manage multiple possible TUs to realize a specific application component, MADCAT employs decision trees that assist developers of such applications in creating TUs. An example of a TU can be seen in

Listing 5.1. We are using the JSON-LD[1] format to store and transfer MADCAT units.

Listing 5.1: Technical Unit

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "TechnicalUnit",
  "name": "BMS/Unit",
  "artifact-uri": "...",
  "language": "java",
  "build": {
    "assembly": {"file": "unit.jar"},
    "steps": [{"step": 1, "tool": "maven", "cmd": "mvn clean install"}]
  },
  "execute": [{"step": 1, "tool": "java", "cmd": "java -jar @build.assembly.
     file"}],
  "configuration": [{"key": "broker.url", "value": "@MGT.broker.url"}],
  "dependencies": [{"name": "MGT", "technicalUnit": {"name": "BMS/Management"
     }}],
  "constraints": {"type": "...","framework": "Spring Boot","runtime": "JRE 1.
     7","memory": "..."}
}
```

A TU starts with a `context` to specify the structure of the information and a specific `type`. The `name` uniquely identifies the TU and should refer to the application name and the specific component that is described by the TU. The `artifact-uri` defines the repository that stores the application sources and artifacts. The `language` field describes the used programming language and an optional version. In order to create an executable, `build` specifies an `assembly` that describes the location within a repository and the name of the executable. Furthermore, `build` defines `steps` that need to be executed to create the executable. Next, `execute` defines the necessary steps for running the executable. In addition to the execution steps, `configuration` stores a possible runtime configuration (e.g., environment variables) that is needed for execution. To allow configuration items to map to other application components, `dependencies` reference TUs of other application components. Finally, the TU enables developers to provide relevant `constraints` that help users of the application to decide on a suitable deployment infrastructure.

For each TU an operations manager can create one or more `Deployment Units` (DUs). In essence, a DU describes how an associated TU can be deployed on concrete infrastructure. To create a specific DU the provider uses the information contained in the TU and its knowledge about the owned infrastructure. Listing 5.2 shows an example DU created for the TU above.

Listing 5.2: Deployment Unit

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "DeploymentUnit",
```

[1]http://json-ld.org

```
  "name": "BMS/Unit",
  "technicalUnits": [{"name": "BMS/Unit"}],
  "constraints": [{
      "hardware": [{"type": "...", "os": "...", "capabilities": [{"name": "
          JRE", "version": "1.7"}], "memory": "..."}],
      "software": [{"replication": [{"min": "all"}]}]
  }],
  "steps": [...]
}
```

Like a TU, a DU also has a `context`, `type`, and `name`. Next, `technicalUnits` allow referencing TUs that are deployed using this specific DU. Based on the information provided in the TU (e.g., constraints) the infrastructure provider defines `constraints` for `hardware` and `software` that are used to decide on suitable infrastructure resources for executing an application component. Finally, `steps` list the necessary deployment steps.

By using TUs and corresponding DUs it is possible to completely describe an IoT application. To finally provision an application deployment, DIANE uses TUs, DUs and concrete infrastructure knowledge to generate `Deployment Instances` (DIs). DIs represent concrete deployments on actual machines of the infrastructure, by considering defined software and hardware constraints. An example of a DI using the DU and TU from above can be seen in Listing 5.3.

Listing 5.3: Deployment Instance

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "DeploymentInstance",
  "name": "...",
  "machine": {"id": "...", "ip": "..."},
  "application": {"name": "BMS/Unit", "version": "1.0.0", "environment": [{"
      key": "broker.url", "value": "failover:tcp://10.99.0.40:61616"}]}
}
```

Again, a DI has a `context`, `type`, and `name`. The `machine` field stores data about the concrete machine that is provisioned with an application component. Runtime information, needed for executing the application component, is represented by the `application` attribute. It contains the `name` and the `version` of the application component. Finally, runtime configurations required by the component are resolved by the framework and represented in `environment`.

5.3.2 DIANE Framework

The framework that allows generating IoT application deployment topologies and deals with the provisioning of these deployments on edge devices in large-scale IoT deployments is depicted on the left hand side of Figure 5.1. DIANE is a scalable and flexible cloud-based framework and its overall design follows the microservices architecture principle. In the following, we introduce the main components of DIANE and discuss the integration with LEONORE, which we introduced in Chapter 4, for provisioning edge devices. Finally, we describe the concrete process of generating and provisioning application deployment topologies.

To keep track of deployments and their relation to TUs and DUs, DIANE provides a `Deployment Registry`. The registry stores units and deployments using a tree structure that represents the relations among them. By managing TUs and corresponding DUs, the framework can provide application deployment provisioning at a finer granularity. This means that it is possible with DIANE to provision an application deployment topology in one batch, but also provision each component separately.

In order to provision an IoT application deployment topology with DIANE, the user of the framework has to invoke the `User API` by providing the following required information: (i) TUs, (ii) corresponding DUs, and (iii) optional artifacts that are needed by the deployment (e.g., executables) but cannot be resolved automatically by the framework, such as private repositories that are not publicly accessible. Since the focus of our work is on generating and provisioning DIs, a user of the framework is responsible for creating the required MADCAT units and necessary application artifacts. The `Deployment Handler` is responsible for handling user interaction and finally triggers the provisioning of application deployments.

In addition to the discussed units, the framework also requires corresponding application artifacts. Therefore, the `Artifact Management` component receives artifacts, resolves all references, and creates an artifact package that is transferred to LEONORE. Each created artifact package contains an executable, a version, and the commands to start and stop the artifact.

To generate DIs, the `Deployment Generator` resolves the dependencies among the provided TUs and DUs by using the `Dependency Management`. The management component returns a tree structure that represents dependencies among units. In addition, the generator handles possible deployment constraints that are specified in the DUs by invoking the `Constraint Handler`. The invoked handler returns a list of infrastructure resources that comply with the specified constraints. Before generating DIs, the generator needs to resolve application runtime configurations (e.g., application properties) in the TUs. This is done by delegating the configuration resolving process to the constraint handler, which provides a temporary configuration. Finally, the generator creates the actual DIs by mapping DUs to concrete machines and updating possible links in the temporary configuration that correspond to infrastructure properties (e.g., IP address of a machine).

Since units in our approach reference each other, the `Dependency Management` is responsible for resolving these dependencies. For representing the dependencies among the units the management component creates a tree structure. The process of dependency resolution first creates for each TU a new root node. After creating the root nodes it checks if a TU has a reference to another TU and if so creates a new leaf node linking to the respective root node. Next, it checks the provided DUs and appends them to the respective TU node as a leaf. In case a reference cannot be resolved based on the provided units, it queries the `Deployment Registry`. The final product of this process is a tree topology, where each root node represents a TU and the leaves are the corresponding

DUs or a reference to another TU.

To find suitable machines for the deployment of application components, DUs allow defining deployment constraints. In our approach we distinguish hardware and software constraints. Hardware constraints deal with actual infrastructure constraints (e.g., operating system or the installed capabilities of a machine). Whereas, software constraints define requirements that correspond to the application component respectively its deployment (e.g., should this component be replicated and if so on how many machines). In order to provide a list of suitable machines the `Constraint Handler` retrieves a list of all known machines and their corresponding metadata from LEONORE. Then, based on the defined constraints in the DU, it filters out the ones that do not fit or are not needed in case software constraints only demand for a certain number of machines.

For actually provisioning the final DIs the `Provisioner` component is used. The component receives generated DIs and the respective topology of TUs, DUs, and their dependencies. The provisioner then traverses the topology and for each TU and DU combination, it deploys the corresponding DIs by invoking LEONORE, adds the DIs to the respective DU as leaf node and updates the deployment registry.

5.3.3 Provisioning of IoT Application Deployment Topologies

The provisioning of IoT application deployment topologies is started when DIANE receives a request to deploy a specific IoT application or application component. The overall process comprises the following steps: (1) In order to generate the deployment topology of an application or application component with DIANE, the user provides an optional list of artifacts and a mandatory list of MADCAT units (i.e., TUs and DUs). Next, the deployment manager is responsible for handling deployment requests and forwarding them to the artifact manager. (2) The artifact manager resolves artifacts according to the provided information in the TUs by either loading them from a specified repository or using the provided artifacts. (3) After resolving the artifacts, the artifact manager invokes the service API to transfer the artifacts to LEONORE. (4) LEONORE receives the artifacts to subsequently pack and store them in its internal repository. (5) For each TU and DU the deployment handler does the following: (6) Forward the list of TUs and DUs to the dependency management component to resolve dependencies and relations among the units. (7) Resolve possible infrastructure constraints that are defined in the DUs by using the constraint handler. (8) The constraint handler gathers all managed machines and their corresponding context (e.g., IP, name, runtime) from LEONORE. (9) According to specified constraints the handler returns a set of machines that are suitable for deploying a specific DU. (10) Invoke the constraint handler again to generate runtime configurations that are specified in the TU, and generate DIs using the gathered suitable machines and runtime configurations. (11) Finally, for each DI the handler invokes the provisioner that stores the DI and corresponding DUs and TU in the deployment registry, deploys the DI by invoking the service API of LEONORE, which then takes care of provisioning the application deployment on the actual infrastructure.

5.4 DIANE Optimization

After presenting the overall approach and the respective realization in the previous section, we now discuss an extension for optimizing the application deployment topology at runtime. In the approach presented so far, we only consider the initial deployment of application topologies and its respective components. However, since IoT applications have to deal with varying loads during operation, we need a mechanism that allows for adapting application topologies at runtime in order to provide the necessary performance and flexibility. Furthermore, this would also enable applications to fully utilize the available processing power of the edge infrastructure. To address these requirements, we extend DIANE to add a two-fold optimization approach, and apply the introduced notion of offloading business logic to the infrastructure edge to DIANE itself.

5.4.1 Elastic Application Deployment

To allow for the optimization of application topologies at runtime, we introduce the notion of an `Elastic Application Deployment`. In contrast to our initial approach that only deploys application components on a set of pre-defined edge devices, we now extend the provisioning mechanism to allow operators to define a hot pool of devices. On theses additional devices, application components are provisioned, but remain idle until they get started. Therefore, this hot pool will be used for optimizing application components, e.g., by scaling application components up or down depending on the application load. In essence, the elastic application deployment consists of a set of devices that host deployed and running application components, and an additional pool of devices that are provisioned with redundant application components that are initially idle. To manage this new form of deployment, we introduce DIANE Optimizers that get provisioned by DIANE and are running on actual edge devices.

5.4.2 MADCAT Unit Extensions

In order to enable DIANE to start adapting the topology of a running application, we need an approach that allows the acquisition of runtime information of this application. This information should comprise both, details about the facilitated deployment on the infrastructure (e.g., currently used number of edge devices), as well as application-specific performance metrics like current request load. Based on this information, DIANE can then decide on the best optimization strategy and how to apply the strategy appropriately.

Therefore, we extend our application description approach, which is based on the MADCAT methodology. First, we introduce so called `endpoint` attributes in a DU. An endpoint represents a URL where application-specific performance metrics can be acquired. Since we want to provide an extensible approach, the defined endpoint can either be provided by the application itself or by an external monitoring tool. Furthermore, to support multiple performance metrics, an application can have a list of endpoints that can be used by DIANE for gathering runtime information. To identify endpoints, each provided endpoint has a unique name within a DU.

Next, based on monitoring information, we need a mechanism to define certain criteria that allow for deciding if an application topology needs to be adapted. Consequently, we extend the overall MADCAT methodology to introduce Optimization Units. An `Optimization Unit` (OU) is used to describe two types of rules that can be used for optimizing an application deployment. First, `application-rules` define criteria for application-specific performance metrics. Second, `infrastructure-rules` define criteria that are targeted towards the used deployment infrastructure. An example of an OU can be seen in Listing 5.4.

Listing 5.4: Optimization Unit

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "OptimizationUnit",
  "name": "BMS",
  "technicalUnits": [{"name": "BMS/Control"}],
  "application-rules": [
    {"name": "response", "endpoint": "@BMS/Control.endpoints.response", "
        contract": "UNDER", "value": "3"}],
  "infrastructure-rules": [
    {"name": "cpu", "contract": "MIN"}],
  "action-policies": [{"name": "ScalingPolicy"}]
}
```

Listing 5.4 describes an application rule that defines that the response time that can be measured from the given endpoint should be under 3 seconds. Next, an infrastructure rule is defined that demands that the application deployment running on the infrastructure should keep the consumed processing power minimal. The difference between these two types of rules is that the former requires monitoring the application itself by using the defined endpoints, whereas the latter requires in-depth knowledge about the used infrastructure resources.

Next, an OU provides an `action-policies` attribute that references either predefined or custom-built action policies. These policies define a set of actions that have to be used for optimizing the application in case any application-rules are violated. For example, an action policy can define that in order to react to increased load, the application deployment needs to be scaled up by using more available machines, or scale down if performance metrics indicate that the current load can be managed with a smaller deployment.

By using the described unit extensions, operators can now define how a deployed application can be monitored and under which circumstances its deployment should be optimized.

5.4.3 Server-side Extension

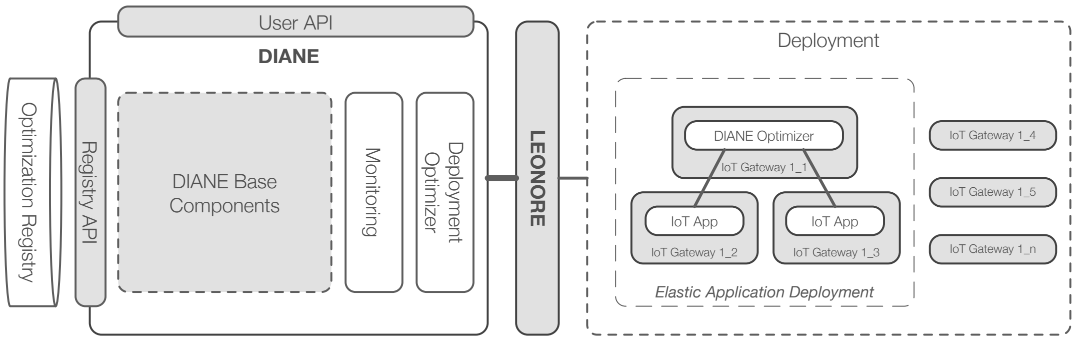


Figure 5.2: DIANE extended

To enable the optimization of deployed application topologies, we extend DIANE by adding several new components, which are depicted in Figure 5.2. In the following, we describe them in more detail.

We extend the `User API` to allow operators to upload OUs that define criteria for triggering the optimization of an application's deployment. Next, operators can use the user API to define custom action policies for describing how an application can be optimized. Since we demand that applications deployed with DIANE follow the microservice architecture approach, optimizing the deployment of an application is relatively easy by evolving the deployment topology. For example, a simple approach to deal with increased load that demands more processing power, is to scale up the application deployment by using additional resources. Uploaded OUs and defined action policies are stored in the optimization registry.

We introduce a `Monitoring` component to collect runtime measurements from deployed and running applications. Based on the details defined in an OU, the monitoring component creates application-specific listeners for the given endpoints to acquire performance measurements from the application in a configurable interval. The collected data is then forwarded to the deployment optimizer, which takes care of further processing.

To optimize the deployment of an application based on defined rules, we introduce a separate `Deployment Optimizer` component. The optimizer receives collected data from the monitoring component and then analyzes the data based on the defined rules and thresholds in the corresponding OU. When the optimizer detects that the application no longer meets the defined criteria it provides the following two optimization modes:

1. *Blackbox Mode*: In blackbox mode, DIANE optimizes the application deployment by treating the deployment infrastructure as black box, which means that the deployment optimizer has no specific knowledge about the used edge devices and their respective resources. In this mode, the deployment optimizer can only optimize for application-rules.

2. *Whitebox Mode*: In whitebox mode, the deployment optimizer has full knowledge of the used deployment infrastructure and can therefore also optimize for infrastructure-rules.

In order to enable these optimization modes, we present the DIANE Optimizer, which can be deployed in the edge infrastructure. The DIANE Optimizer monitors and controls an elastic application deployment that allows for optimizing the deployment topology of an application by either starting currently idle application components, or stopping unnecessary components.

To allow DIANE to facilitate the DIANE Optimizer, the optimizer needs to be associated with an application and then deployed in the edge infrastructure. This is done using the following approach: (i) When an OU is uploaded by an operator via the service API, DIANE extracts which application and respective components are affected. (ii) Next, the respective DIs are analyzed to gather the used deployment in the infrastructure. (iii) To form an elastic application deployment based on the defined action policies, the deployment generator is used to generate a fresh set of DIs that is provisioned, but not yet started to form a pool of idle components to allow for the evolution of the application topology. (iv) Then, the constraint handler is used for finding a suitable machine for running the DIANE Optimizer, and the provisioner is used for deploying the optimizer on the selected machine. (v) Finally, once the optimizer registers itself with DIANE, it is provided with the deployment topology of the application, as well as the provisioned but not yet started DIs that can be used for optimizing the application deployment.

To keep track of uploaded OUs, corresponding action policies, and deployed DIANE Optimizers, we add an `Optimization Registry`. In this repository, for each application that is handled by DIANE, we store defined OUs and additional action polices. In addition, for each DIANE Optimizer deployment, we store the ID of the optimizer as well as the machine in the infrastructure that is hosting the optimizer. The combination of optimizer ID, and machine IP and ID allows DIANE to uniquely identify the optimizer deployment.

5.4.4 DIANE Optimizer

The DIANE Optimizer enables the optimization of an application topology by monitoring the actual deployment infrastructure, which provides valuable insights on the infrastructure performance. The DIANE Optimizer is specifically catered to be lightweight in terms of memory consumption and CPU usage, so that it can be executed on machines residing in the edge infrastructure that only provide a fraction of the processing power of cloud resources. The architecture of the DIANE Optimizer is depicted in Figure 5.3. In the following, we outline the basic components of a DIANE Optimizer.

Once a DIANE Optimizer is deployed in the edge infrastructure, the `Bootstrapper` component of the optimizer is responsible for registering the deployment with DIANE. Based on this information, the server-side framework can keep track of deployed optimizers. Furthermore, during the registration process the optimizer receives the list of machines representing the current deployment of the application, as well as a hot pool of machines

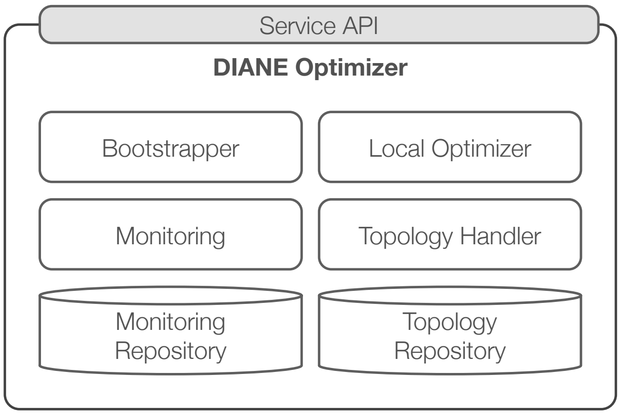


Figure 5.3: DIANE Optimizer

where application components are already provisioned, but not yet started. These lists are then forwarded to the topology handler for further processing.

To form an elastic application deployment the `Topology Handler` first extracts the devices that represent the current application deployment based on the provided information from the bootstrapper. This topology representation is then enriched with the current hot pool of application components and then updated in the `Topology Repository`. Based on this stored topology, the DIANE Optimizer knows which devices are currently used by the application and is also able to optimize the overall application topology by starting idle or stopping running components.

To gather valuable insights from the used deployment infrastructure, the DIANE Optimizer uses a dedicated `Monitoring` component. According to the stored application topology, the monitoring extracts the respective machines. In order to acquire performance measurements from these machines, the DIANE Optimizer facilitates the LEONORE profiler that is pre-installed on the machines to extract performance data like used CPU and consumed memory. Therefore, whenever the application topology is updated (e.g., new machines are added) the monitoring component contacts each machine of the deployment to register an endpoint where the machines, respectively their LEONORE profilers, publish the profiled monitoring information in a configurable interval. The published performance profiles of the machines are then grouped by machine and stored for later analysis in the local `Monitoring Repository`. The repository is implemented as a local cache using available RAM and/or disk resources if available, which allows fast read and write access, while still considering the resource-constraint nature of the underlying infrastructure. To save memory, the cache only keeps the most recent profiles.

In the current version, a DIANE Optimizer does not automatically decide when to optimize its corresponding elastic application topology. Therefore, it provides a `Service API` that allows DIANE to trigger a deployment evolution. Whenever DIANE decides that based on a defined application rule the application deployment has to be optimized, it finds the responsible DIANE Optimizer and invokes the service API by providing infrastructure rules and action policies that need to be respected. Next, the request is forwarded to the local optimizer, which is then responsible for choosing suitable optimization actions and executing them accordingly.

Once DIANE triggers an optimization by invoking the DIANE Optimizer, the `Local Optimizer` performs the following steps in order to process the request: (i) Analyze the given application policy to identify a set of possible deployments that need to be updated for optimizing the application topology. (ii) If infrastructure rules are defined, the set of possible deployments is filtered by using gathered monitoring information. For example, if an application rule describes that the used CPU of the deployment has to be kept minimal, the optimizer will use the performance profiles stored in the monitoring repository to choose a small deployment that can deal with the load by only consuming a fraction of the provided total resources. (iii) If no application rules are defined, the set is reduced by picking deployments naïvely. (iv) After the set of deployments that need to be updated is finalized, the application policy is executed. This means that the application deployment topology is optimized by either starting idle or stopping running application components. (v) Finally, the topology handler is notified to store the evolved application deployment in the topology repository.

5.4.5 Optimizing an Elastic Application Deployment

The process of optimizing an elastic application deployment is initiated by an operator that defines an OU and corresponding action policies. To describe the overall process let us consider that we want to scale up an application deployment to a maximum of 20 machines (action policy) whenever the response time of the application is over a defined threshold (application rule). Furthermore, during scale up the deployment should be kept minimal in terms of used CPU (infrastructure rule). After describing these requirements, the operator uploads the OU and the action policy to DIANE. Based on the input, DIANE creates an elastic application deployment and deploys a DIANE Optimizer. Next, the monitoring component starts collecting response time measurements from the defined endpoints of the application. Once DIANE detects that the response time of the application violates the defined threshold in the OU, it invokes the respective DIANE Optimizer by providing the defined scale up action policy and infrastructure rule. Then, the DIANE Optimizer decides that based on the provided input and gathered performance profiles of the machines, it is sufficient to scale up the application deployment by only using 2 additional devices and queues further scale up requests from DIANE until these devices are fully utilized. In case no infrastructure rules are defined by the operator, the overall approach follows the same steps as above, except that no infrastructure information is used by the DIANE Optimizer and the deployment is scaled up by using a naïve approach (e.g., 5 devices for each scale up request).

Using explicit infrastructure knowledge (whitebox mode) allows the DIANE Optimizer to optimize the application deployment topology more efficiently compared to an approach that only uses pre-defined or naïve adaptation steps (blackbox mode).

5.5 Evaluation

5.5.1 IoT Application Deployment and Execution

To evaluate our approach we implemented a demo IoT application based on a case study conducted in our lab in cooperation with a business partner in the building management domain. In this case study we identified the requirements and basic components of commonly applied applications in this domain. Based on this knowledge we developed an IoT application for managing and controlling air handling units in buildings, where the design and implementation follows the microservices architecture approach. Next, we created a test setup in the cloud using CoreOS[2] to virtualize edge devices as Docker[3] containers. We reuse LEONORE's notion of IoT gateways as representation of edge device in our experiments.

In the remainder of this section we give an overview of the developed demo application and the created evaluation setup, present different evaluation scenarios, and analyze the gathered results.

BMS Demo Application

Currently, IoT applications are designed and implemented as layered architectures [2]. This means that the bottom layer consists of deployed IoT devices, a middleware that provides a unified view of the deployed IoT infrastructure, and an application layer that executes business logic [73]. According to this layered approach, business logic only runs in the application layer and the IoT infrastructure is provisioned with appropriate software, sends data, and reacts on its environment [28]. However, in practice more and more IoT devices provide constrained execution environments that can be used for offloading parts of the business logic. To compare these two deployment approaches we develop an application for a building management system that consists of the following components: (1) An `Air Handling Unit` (unit) is deployed on an IoT device, reads data (e.g., temperature) from a sensor, transmits the data to and reacts on control commands received from the upper layer. (2) A `Temperature Management` (management) represents the processing component of the application and gathers the status information of the units. It receives high level directives from the upper layer and based on the processed unit data and the received directives, forwards appropriate control commands to the unit. (3) Finally, the `Building Controller` (control) is the top level component and decides for each handled management component the directive it has to execute. In the traditional deployment topology that follows the common IoT application deployment model, the unit component is deployed on devices in the IoT infrastructure, and both the processing and control components are executed on a platform in the cloud. We refer to this deployment as *traditional application topology*. In contrast, in a contemporary deployment topology, some of the processing logic is offloaded onto devices in the IoT infrastructure, which we refer to as *evolved application topology*.

[2]https://coreos.com
[3]https://www.docker.com

Setup

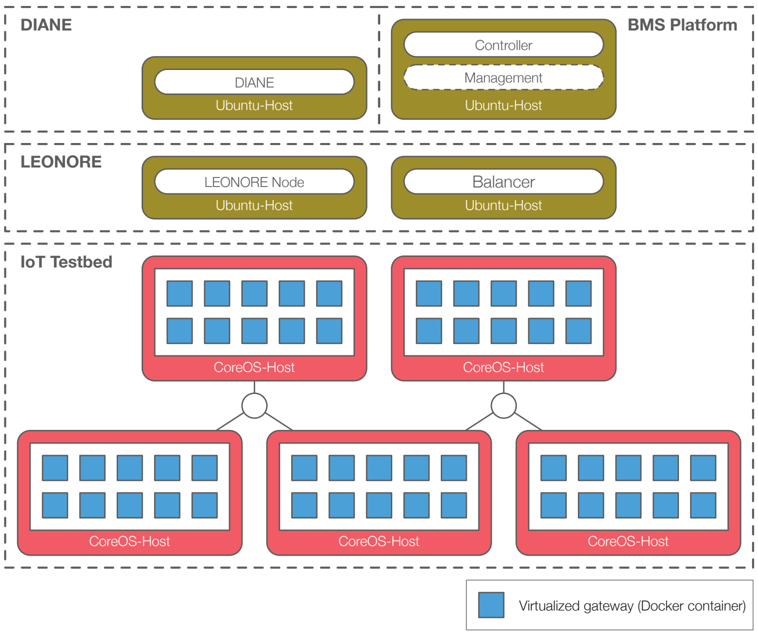
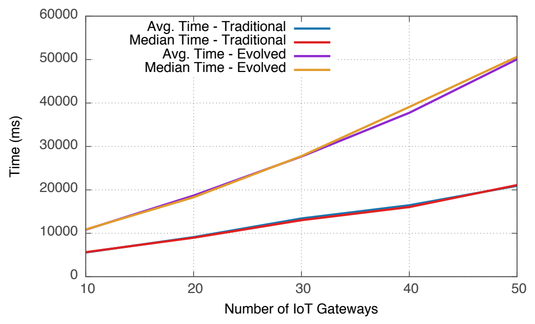


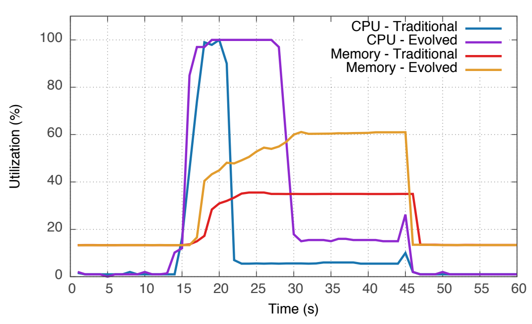
Figure 5.4: Evaluation – Setup

For the evaluation of our framework we create an IoT testbed in our private OpenStack[4] cloud. We reuse a Docker image that was created for LEONORE to virtualize and mimic a physical gateway in our cloud. To run several of these virtualized gateways, we use CoreOS clusters and fleet[5], a distributed init system, for handling these clusters. Based on fleet's service unit files, we dynamically generate according fleet unit files and use them to automatically create, run, and stop virtualized gateways. Figure 5.4 depicts the overall setup that we use for our experiments. As foundation of our setup, an *IoT Testbed* consists of a CoreOS cluster of 5 virtual machines, where each VM is based on CoreOS 607.0.0 and uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB Disk space). The IoT gateway-specific framework components of LEONORE are pre-installed in the containers. On top of the testbed, the LEONORE framework is distributed over 2 VMs using Ubuntu 14.04. The first VM hosts the balancer and uses the m1.medium flavor, whereas the second VM uses the m2.medium flavor (5760MB Ram, 3 VCPUs and 40GB Disk space) and is deployed with a LEONORE node. On top, DIANE is hosted in one VM using Ubuntu 14.04 with the m1.medium flavor. Finally, the platform components of the BMS demo application are deployed on a separate VM using Ubuntu 14.04 and the m1.small flavor (1920MB Ram, 1 VCPUs and 40GB Disk space).

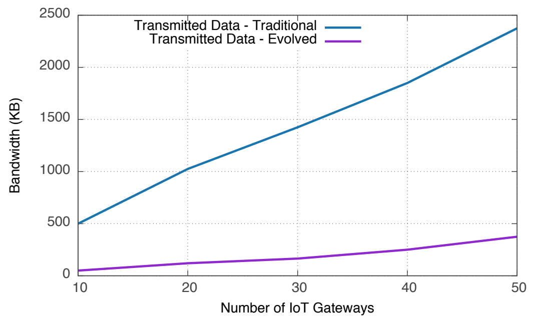
[4]http://www.openstack.org
[5]https://github.com/coreos/fleet



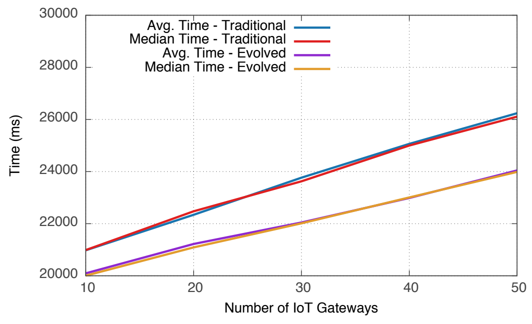
(a) Deployment Time



(b) Device Utilization



(c) Application Execution - Bandwidth



(d) Application Execution - Time

Figure 5.5: Evaluation Results – IoT Application Deployment & Execution

In order to evaluate and compare the two discussed deployment topologies of the application, the BMS platform initially comprises controller and management (traditional application topology), and is then reduced to only host the controller, since the management component is deployed on the devices (evolved application topology). In both scenarios the unit component is deployed and running on the devices in the IoT infrastructure.

IoT Application Deployment

In the first experiment we measure the time that is needed for dynamically creating application deployments for the two BMS IoT application deployment topologies and provisioning of these deployments on IoT devices. In the second experiment we compare the device resource utilization when executing the provisioned application deployments.

Deployment Time Figure 5.5a shows the overall time that is needed for creating and provisioning of application deployments on an increasing number of devices. The time measurement begins when DIANE is invoked and ends when DIANE reports the successful deployment. To deal with possible outliers and provide more accurate information we executed each measurement 10 times and calculated both the average and median time. In Figure 5.5a we see that for the traditional application topology the framework provides a stable and acceptable overall deployment time. In comparison, the deployment of the evolved application topology takes in total almost twice as long, but also provides a stable deployment time. Taking into account that the evolved application topology requires deploying twice as many application components and corresponding artifacts, however, we argue that this increase is reasonable, since the limiting factor is the actual provisioning of devices as we create application packages that have more than doubled in size.

Gateway Resource Utilization Figure 5.5b depicts the CPU and memory utilization of one device when provisioning and executing the two IoT application deployment topologies. The figure shows that initially there is no application component running on the device. After 15 seconds we initiate the deployment via our framework, which provisions the application deployments and starts the execution. Then, the deployments run for 30 seconds. Afterwards, the framework stops the execution. When provisioning the traditional application topology, we clearly see that the CPU utilization has a short high peak due to the startup of the deployment. However, after this high peak the overall utilization of the device is low and leaves room for using this untapped processing power to offload business logic components on the device. To illustrate the feasibility of this claim, we also provision and execute the evolved application topology on the device. We see that in comparison to the traditional application topology, the load on the device is almost twice as high, but except for the high initial CPU load peak, the overall utilization of the device is still acceptable and reasonable.

IoT Application Execution

In the second experiment we collect runtime information from the BMS application to compare both deployment topologies. In order to do that, we deploy both topologies with our framework on an increasing number of devices. However, now we measure bandwidth consumption and execution time when invoking the application's business logic. The measurement begins by invoking the control component of the application to specify a virtual set-point temperature on each device, where each unit component on the device has the same initial temperature reading. To provide reliable results, we execute each measurement 10 times and freshly provision the devices after each measurement with DIANE. Depending on the BMS application deployment topology, the management component is either executed in the platform (i.e., the cloud) or on each device.

Bandwidth Consumption Figure 5.5c shows the average bandwidth consumption that results from invoking the business logic of the two application deployment topologies. We see that the traditional application topology causes a significant amount of data transmission between platform and IoT infrastructure. As a result the transmitted data produces a high load on the network and consumes a lot of bandwidth. This behavior is obvious, since the complete business logic is executed on the platform and devices are only sending measurements and reacting to control messages. In contrast, the evolved application topology produces less traffic and therefore consumes on average only 13% of the bandwidth. This is due to the offloading of the processing (management) component to each device, which therefore drastically reduces the transmitted data between platform and IoT infrastructure.

Execution Time Figure 5.5d shows the time that is needed for executing the previously described business operation of the BMS application for the two application deployment topologies. We see that for both topologies the application scales well and provides reasonably fast results. However, we notice that the offloading of the processing components on the devices reduces the execution time by 7%, since application component interaction within a device is faster than the interaction between device and platform.

After presenting and evaluating the gathered experiment results, we can deduce the following: DIANE is capable of dealing with different application topologies and changes in the IoT infrastructure. The framework scales well with increasing size of application deployment topologies and does not add additional overhead to the overall time that is needed for provisioning the IoT infrastructure. Note that for very large deployments the use of multiple coordinated LEONORE nodes is required. Furthermore, depending on the scenario, it is feasible to offload application components from a cloud platform to devices in the IoT infrastructure. Examples of such scenarios are applications that generate a significant amount of traffic between the platform and the IoT infrastructure and therefore justify the additional deployment overhead.

5.5.2 Elastic Application Deployment

To evaluate our application deployment optimization mechanism we implemented a smart city demo application and reused the test setup presented in Section 5.5.1. In the remainder of this section we give an overview of the developed smart city demo application, discuss the concrete evaluation setup, present different evaluation scenarios, and analyze the gathered results.

Smart City Demo Application

For this experiments we use a demo application that implements the concept of Autonomous Intersection Management[6], which enables autonomous cars in a smart city environment. In our scenario we want to handle large numbers of cars, which requires smart city operators to optimize the deployment topology of such intelligent control systems by using any kind of available processing power.

To analyze this approach, we develop a simple traffic control application that manages incoming requests sent from autonomous cars. The incoming requests need to be processed by the application to calculate if a car's intended path is valid (e.g., safe to use). Since the autonomous cars generate a huge load, the application allows scaling the computation logic across infrastructure boundaries. Therefore, the application is separated into two

[6]http://www.cs.utexas.edu/~aim/

components. A possibly replicated processing component that provides the calculation logic. On top, a central platform component that receives requests by autonomous cars and forwards them to the underlying processing components. Furthermore, to analyze application performance, it provides endpoints to acquire metrics like request load and response time.

Setup

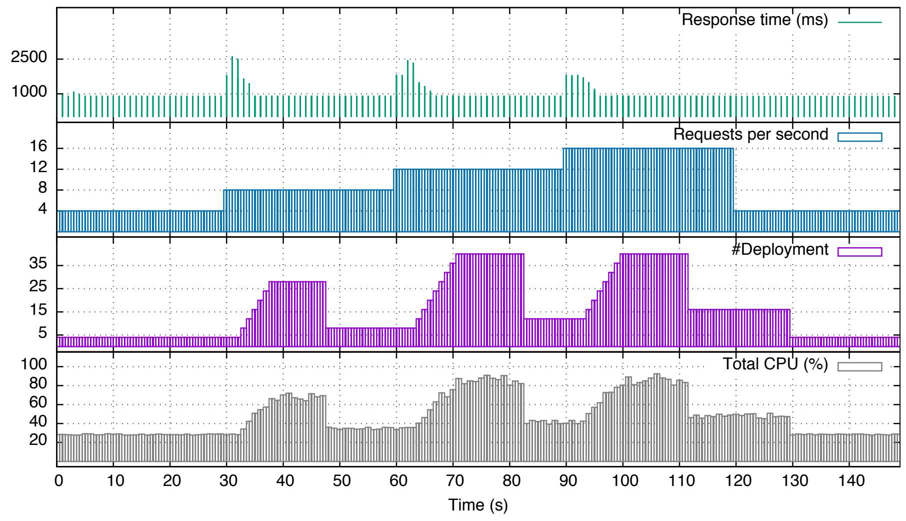
In order to evaluate the introduced application deployment optimization using the DIANE Optimizer, we reuse the setup presented in Section 5.5.1. However, for this evaluation, we exchange the VM hosting the components of the BMS IoT application, with a new VM using Ubuntu 14.04 and the m1.small flavor to host the platform component of the smart city demo application. In order to evaluate and compare the different optimization modes, the processing component of the smart city demo application is deployed and executed on the devices in the IoT infrastructure.

IoT Application Topology Optimization

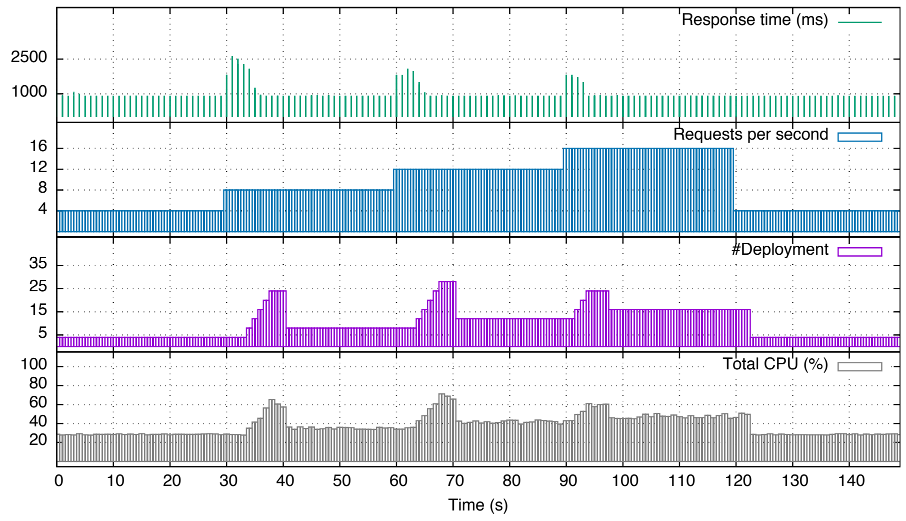
In the following experiments we use DIANE to optimize the deployment topology of the smart city demo application by scaling it across the available IoT infrastructure. We create an OU that defines the allowed threshold for the response time of the application and that the used application deployment should keep the CPU usage across the infrastructure to a minimum. Furthermore, we also define a policy for scaling up the deployment when the response time is over the defined threshold, as well as a policy for scaling down the application by stopping unused infrastructure devices. Additionally, for the experiments we assume that an elastic application deployment was already formed by using a total of 40 machines, plus one additional machine for hosting the DIANE Optimizer.

Next, for comparing the two different optimization modes (blackbox and whitebox) we use different patterns for generating load on the application. In the first scenario, we use a load pattern that simulates a stepwise increase and decrease in requests. In the second scenario, we use a pyramid-like load pattern for sending requests to the application. For the blackbox optimization mode, the deployment topology of the application is scaled without using the provided infrastructure rule, whereas for the whitebox mode we facilitate gathered knowledge about the infrastructure to provide an optimized scaling approach according to the infrastructure rule.

Scenario 1: Step Load Pattern Figure 5.6 illustrates the evaluation results for the first scenario. The x-axis shows the temporal course of the evaluation in seconds. In the 'requests per second' section we see that we begin the evaluation by sending 4 concurrent requests per second to the application and increase the load stepwise every 30 seconds to see if DIANE is able to scale up the application. Finally, at 120 seconds we reduce the load to 4 requests per second to see if DIANE is also able to scale down the application. In the 'response time' section we see the response time for each incoming request. The 'deployment' section illustrates the number of facilitated edge devices by the deployment.



(a) Blackbox



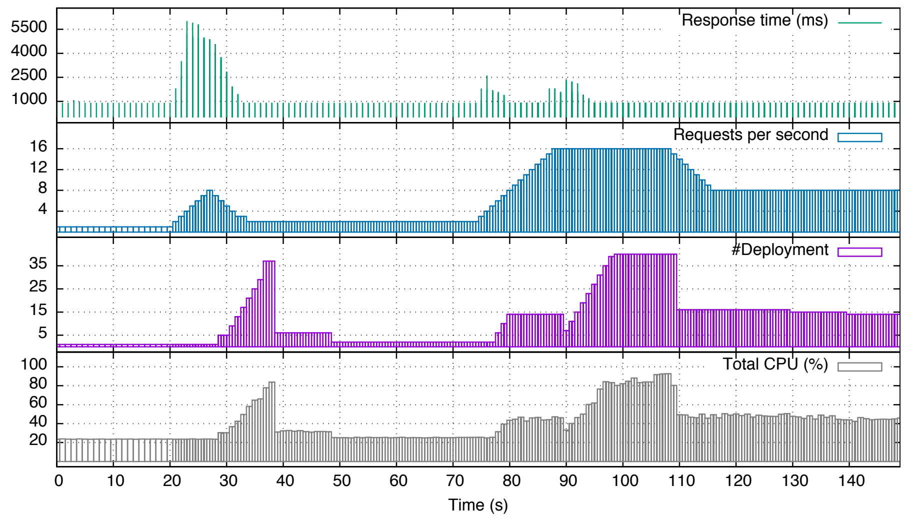
(b) Whitebox

Figure 5.6: Evaluation Results – IoT Application Topology Optimization (Step Load Pattern)

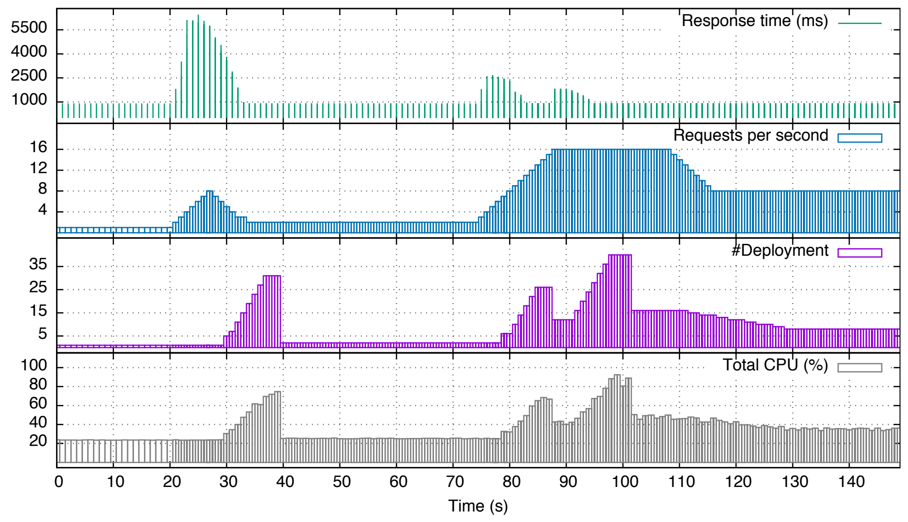
Finally, the 'total CPU' section represents how much of the total available CPU is used by the application deployment at the given time.

By comparing Figure 5.6a that represents the blackbox optimization mode, and Figure 5.6b, which shows the result for using the whitebox optimization mode, we notice that for the first interval of requests the response time of the application is almost constant for both approaches. At 30 seconds, when the request load doubles we notice that in both cases the response time rises. For both modes, at approximately 34 seconds DIANE starts scaling up the application by invoking the DIANE Optimizer, since the response time of the application violates the provided threshold. However, by looking at the results, we notice several differences during the deployment optimization process. The blackbox mode uses a naïve approach that scales up the deployment until the response time is no longer violated. This, in combination with a lot of queued up requests, leads to the fact that the blackbox mode uses a lot of infrastructure resources for a relatively long time before they are released again. In comparison, in the whitebox mode the DIANE Optimizer uses gathered monitoring information from the deployment infrastructure and only scales up the application when the currently used resources are fully utilized. This allows the application to handle the queued up requests with a smaller deployment in shorter time. For the following two increases in requests per second at 60 and 90 seconds, we see that the framework is also able to detect and analogously handle them. Finally, at 120 seconds, we see that the load drops, which is detected by the whitebox mode almost immediately, due to the fact that DIANE Optimizer constantly receives information about the used resources. The blackbox mode needs significantly more time to detect the changed load by monitoring the application and therefore uses resources for a longer period. After comparing both modes using the stepwise load pattern, we can conclude that both approaches allow for optimizing the application deployment according to the provided OU. However, by using gathered knowledge of the infrastructure deployment, the whitebox mode is able to evolve the application topology by using less resources and therefore reduces the total overall CPU utilization by approximately 15%. In addition, we also notice that in total the whitebox mode produces approximately 25% less response time violations compared to the blackbox approach.

Scenario 2: Pyramid Load Pattern Figure 5.7 illustrates the evaluation results for the second scenario. We compare the blackbox optimization mode (Figure 5.7a) and the whitebox approach (Figure 5.7b) using a pyramid-like load pattern. We notice that for the first 20 seconds the response time of the application for both modes is stable. At 20 seconds the first pyramid load pattern starts increasing the load on the application. We see that it takes a considerable amount of time until DIANE triggers the scale up of the application deployment. Compared to the first scenario, we see that both optimization modes are struggling with this type of load pattern and provide almost identical results. However, by comparing both results, we notice that for the first pyramid-like increase and drop in load, the blackbox mode performs better in terms of violated response times compared to the whitebox approach. This can be explained by the fact that the extremely fast load change does not allow the whitebox mode to utilize gathered infrastructure information. In addition, by looking at the deployment size we see that the whitebox mode uses a smaller deployment for a longer time, compared to the blackbox mode. For the next load increase at 75 seconds we see that the blackbox mode uses one small and one big scale up, in terms of deployment size, to compensate for the response time violations, which leads to a high deployment utilization. In contrast, the whitebox mode is able to use the infrastructure resources more efficiently by using more machines for the first scale up, and an additional scale up for a shorter period of time. Therefore, for



(a) Blackbox



(b) Whitebox

Figure 5.7: Evaluation Results – IoT Application Topology Optimization (Pyramid Load Pattern)

the second pyramid-like load change, the whitebox mode uses in total less resources, but again generates more response time violations. After comparing both modes when using the pyramid-like load pattern, we can conclude that on the one hand the whitebox mode in total uses approximately 5% less resources in terms of utilized total CPU. However, on the other hand the blackbox mode produces approximately 30% less response time violations.

To summarize the results, we see that both proposed optimization approaches allow for evolving the application deployment topology at runtime. However, by comparing the results of both scenarios we see that choosing an optimal optimization approach depends on various factors, such as the expected load on the application, and the tradeoff between application performance violations (i.e., response time) and cost benefit by using less infrastructure resources.

5.6 Summary

In order to sense and manipulate their environment, IoT applications are required to integrate and manage a large number of heterogenous devices, which traditionally serve as simple sensors and actuators. Recently, however, devices emerged that in addition to basic sensing and actuating features, also provide constrained execution environments with limited processing, memory, and storage capabilities. To exploit this untapped processing power, applications can offload parts of their business logic onto edge devices. This offloading of application components not only increases the robustness of the overall application deployment, but also allows for cutting down costs by reducing expensive cloud to edge communication overhead. The consideration of edge devices is especially important for IoT applications that are deployed in the cloud, as the cloud allows applications to react to changing requirements by elastically adapting their overall deployment topology. Therefore, in addition to the traditional design considerations for cloud applications, specific issues like the geographical distribution of edge devices and the resulting network latencies need to be explicitly considered in the design of IoT cloud applications. Furthermore, applications need to be designed as clearly separated components that can be deployed independently. This application design approach enables the flexible provisioning of applications whose deployment topology evolves by dynamically offloading components to edge devices.

To support this, in this chapter we introduced DIANE, an approach that dynamically generates optimized deployment topologies for IoT cloud applications, which are tailored to the currently available physical infrastructure. DIANE uses a declarative, constraint-based model of the desired application deployment to allow for flexible provisioning of application components on both, cloud infrastructure, as well as edge devices deployed in the IoT infrastructure. In addition, DIANE provides an optimization approach that allows for evolving application deployment topologies at runtime to enable applications to autonomously react to environmental changes (e.g., changing request patterns).

CHAPTER 6

Deploying IoT Applications with TOSCA

In this chapter, we discuss a complementary deployment approach that employs TOSCA, a standard for cloud service management, to systematically specify the components and configurations of traditional IoT applications. We demonstrate that by using TOSCA, application models can be reused and deployment processes can be automated in heterogeneous IoT system environments.

6.1 Overview

IoT solutions [80] are typically domain-specific, relying on heterogeneous hardware (e.g. sensors, actuators and gateways), communication protocols and data models. To deal with such complexity, a lot of industrial and academic efforts are put into developing gateway frameworks that facilitate device integration and application development. However, such efforts have also led to many proprietary application runtime environments [57, 143] with non-standardized service management processes.

In our previous work, we have developed the IoT PaaS [67] architecture to improve the efficiency and scalability of IoT service delivery. It allows IoT solutions to be delivered on a PaaS cloud as *virtual verticals*, which are composite, configurable, and able to share the underlying cloud platform services and computing resources with other IoT solutions. Although the architecture allows service providers to efficiently deliver and scale up IoT services, the service management tasks, such as application deployment, driver installation and gateway configuration are still handled manually in a case-by-case manner, due to the underlying heterogeneity of IoT infrastructures. The problem is that the state-of-the-art IoT service frameworks, either gateway-based or cloud-based, lack a systematic methodology to specify and maintain the intricate software and hardware dependencies in IoT applications.

This chapter is motivated by the challenges we have experienced in deploying traditional IoT solutions at various scales and in multiple application domains[1] (mostly building automation and vehicle tracking). In order to improve the reusability of service management processes and automate IoT application deployment in heterogeneous environments, we propose to employ the Topology and Orchestration Specification for Cloud Applications (TOSCA) [14, 90] for IoT service management. TOSCA is a new standard aiming at describing the topology of cloud applications by using a common set of vocabulary and syntax. In this chapter, we will demonstrate the feasibility of using TOSCA to specify a typical IoT application in building automation, the Air Handling Unit (AHU). The common IoT components such as gateways and drivers will be modeled, and the gateway-specific artifacts that are necessary for application deployment will also be specified. Based on the case-driven modeling, we will discuss our early experience gained from applying TOSCA for IoT applications. This work is in line with our ongoing effort of enabling the convergence of IoT and cloud [69]. To the best of our knowledge, this is also the first attempt of explicitly addressing the IoT application deployment problem using a cloud-based approach.

The rest of the chapter is organized as follows: In Section 6.2 we provide a short introduction on the preliminaries of this work, and start to model the TOSCA nodes and relationships of a traditional IoT application. The gateway specific application artifacts are specified in Section 6.3. Finally, we discuss the experiences of using TOSCA in Section 6.4. The chapter concludes in Section 6.5.

6.2 Modeling traditional IoT Applications

In this section we provide some preliminaries, before modeling a traditional IoT application in building automation systems to demonstrate the complexity of predominant applications in IoT as well as the feasibility of facilitating TOSCA for the deployment of these applications.

6.2.1 Preliminaries

IoT PaaS

IoT services are often delivered in physically isolated verticals (often referred to as "silos"), in which hardware, middleware and application logics are tightly coupled to fulfill domain or even project-specific requirements. IoT PaaS [67] is a novel IoT service delivery platform that leverages the service delivery model of PaaS cloud. On this architecture, we offer the possibility of providing end-to-end IoT solutions as *virtual verticals* on cloud, opposed to the traditional delivery model of physically-isolated and tightly-coupled vertical solutions. IoT PaaS is a generic, domain-independent architecture that relies on *domain mediators* to integrate domain-specific control protocols and data models. We have demonstrated the domain mediation mechanism with an oBIX (Open Building

[1]http://www.pacificcontrols.net/projects/ict-project.html

Information Exchange) [91] mediator for building automation applications. Our approach further leverages this cloud platform with TOSCA to address the challenges in IoT service delivery.

Gateways

To handle the multitude of field devices in IoT solutions, gateways [124, 143][122] are designed to connect heterogeneous, resource-constraint devices. Gateways support various device drivers and protocol stacks to communicate with devices, for example 6LoWPAN (IPv6 over Lower power Wireless Personal Area Networks). Depending on their applications, they may also support domain-specific, device-oriented data exchange protocols such as BACNet (Building Automation and Control Networks). Gateways can also provide service interfaces, such as the RESTful interfaces of oBIX and CoAP [51](Constrained Application Protocol) to ease the integration of lower-level IoT infrastructures with enterprise applications. In brief, the basic function of gateways is to provide an abstraction of IoT infrastructure by effectively translating device/network interfaces into software interfaces. The process is generally known as device virtualization [41]. On top of this core function, most modern gateways are also built with application runtime environments, which are usually proprietary or non-standardized.

TOSCA

TOSCA is a new OASIS standard for improving portability of cloud applications in face of growingly heterogeneous cloud application environments. In the following we briefly describe the core concepts of TOSCA.

TOSCA specifies a meta-model for describing both the structure and management of IT services. The structure of a service is defined by the *Topology Template*, which consists of *Node Templates* and *Relationship Templates.* Together they represent a service by a directed graph. In this graph, every component is represented by a *Node Template* that instantiates a *Node Type*, which defines the properties and operations of a component. To support reusability, Node Types are defined separately and just referenced in Node Templates. Furthermore, in addition to the reference, usage constraints of components, e.g. number of occurrences, can be specified. In the topology of a service, nodes are connected by relations. *Relationship Templates* specify the relationship among nodes in the Topology Template, where each Relationship Template refers to a separately defined Relationship Type, which in turn defines the semantics and any properties that can be used to represent a relationship, such as "dependOn" or "connectTo". The actual scripts, configuration files and application archives required by an application are called *Artifacts*, which are explicitly specified in *Artifact Types* and *Artifact Templates.* Artifacts are specific to each runtime environment and configuration of an application.

The management process of creating, deploying and terminating a service can be defined by *Plans.* Plans are process models that can be implemented as complex workflows. The specification of these process models relies on existing standards, such as BPMN or BPEL, to automate management processes in different application environments. TOSCA provides two ways of using plans—a container to use a reference of a process model (via *Plan Model Reference*) and to include an actual model in the plan (via *Plan Model*). The process model contains tasks that refer to operations of Interfaces of either Node Templates, Relationship Templates or any other available interface. This guarantees that a plan can directly manipulate nodes of the service topology or interact with external systems.

The topology templates, plans and artifacts of an application are packaged in a Cloud Service Archive (.csar file) and deployed in a TOSCA environment, which is able to interpret the models and perform specified management operations.

It is worth noting that plans are not always required in using TOSCA. The TOSCA environment is able to infer the correct topology and management procedure just by interpreting the topology template. This is known as a "declarative" approach. Plans realize an "imperative" approach that explicitly specifies how each management process should be done. As the first attempt of employing TOSCA for IoT applications, this work uses the declarative approach, i.e. only applying the concepts in Topology Template.

6.2.2 Application description

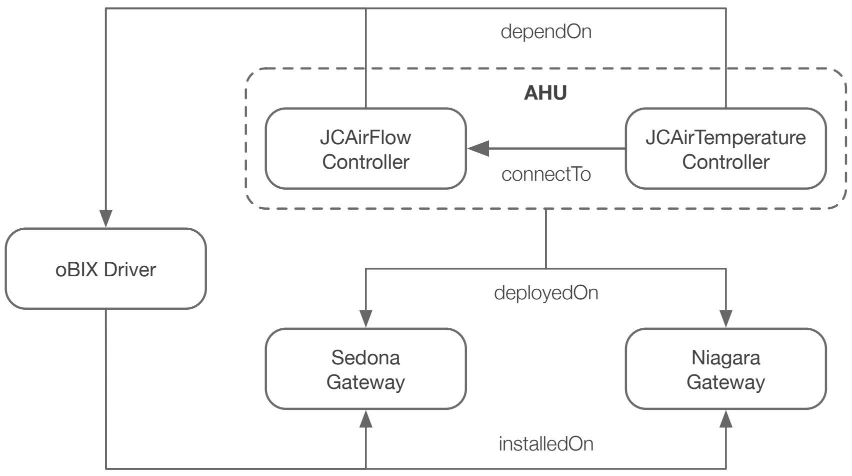


Figure 6.1: Air Handling Unit Usecase

The Air Handling Unit (AHU) is a common facility in modern buildings. Its basic function is to condition and circulate air. In building automation solutions, sensors and actuators are applied to AHUs in order to remotely monitor and control them. Figure 6.1 illustrates a simplified deployment view of an AHU that is commonly found in commercial solutions. Two core components produced by Johnson Controls (JC)—Air Temperature Controller and Air Flow Rate Controller—are connected to output fresh air at a temperature point set by an operator. Other than the control interface defined by Johnson Controls, the AHU relies on the oBIX protocol for applications to access it.

Such AHU applications will be deployed in various gateway models due to the technical requirements of other facilities (e.g., lighting) and legacy building automation systems. We demonstrate the deployment with two gateway frameworks—Niagara and Sedona.

6.2.3 Modeling the nodes

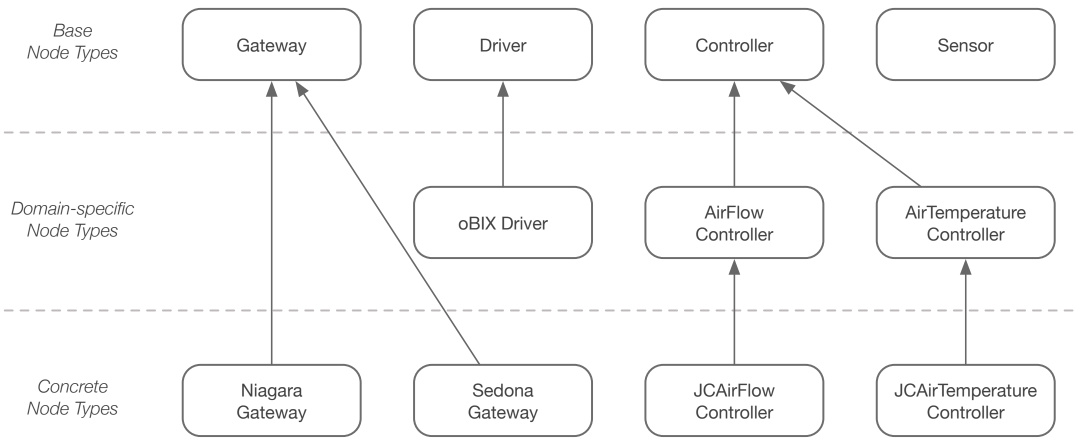


Figure 6.2: Node Types

Modeling nodes is the first step in using TOSCA to model IoT applications. Figure 6.2 illustrates the hierarchical node model we developed for the AHU application.

Base Node Types

The *Base Node Types* are directly derived from a generic TOSCA root node type. This puts them at the same level as other common cloud application nodes, including server, database and so on. The nodes at this level present the most fundamental concepts in IoT applications. Listing 6.1 presents the type definition of three basic node types, namely *Controller*, *Gateway* and *Driver*, in pseudo-XML[2]. *Sensor* is not used in our application, thus not listed.

The most important element of the node types is the *Interface*. Interfaces define the operations that application providers can apply to the class of components. The operations presented in our example belong to a *Lifecycle interface*, which is able to instruct the TOSCA environment to change the status of these nodes. The concrete implementations of these node types need to provide corresponding interface implementations and define required parameters. The properties of these three node types are listed in Listing 6.2.

[2]For emphasizing the core concepts and saving space, we do not use name spaces. When the embedded structure of XML elements are too redundant, we also remove the end tags.

Listing 6.1: Base Node Types

```
<NodeType name="Controller">
  <DerivedFrom typeRef="RootNodeType" />
  <NodeTypeProperties element=
    "ControllerProperties" />
  <Interfaces>
    <Interface name="lifecycle">
      <Operation name="deploy" />
      <Operation name="configure" />
      <Operation name="start" />
      <Operation name="stop" />
      <Operation name="undeploy" />
  ...
</NodeType>

<NodeType name="Gateway">
  <DerivedFrom typeRef="RootNodeType" />
  <NodeTypeProperties element=
    "GatewayProperties" />
  <Interfaces>
    <Interface name="lifecycle">
      <Operation name="poweron" />
      <Operation name="poweroff" />
      <Operation name="reboot" />
  ...
</NodeType>

<NodeType name="Driver">
  <DerivedFrom typeRef="RootNodeType" />
  <NodeTypeProperties element=
    "DriverProperties" />
  <Interfaces>
    <Interface name="lifecycle">
      <Operation name="install" />
      <Operation name="uninstall" />
  ...
</NodeType>
```

Domain-specific Node Types

The *Domain-specific Node Types* are related to IoT applications in a certain industrial domain, which is building automation in our case. For example, oBIX is a protocol widely used in building automation projects. It is based on web standards including XML, HTTP and URI to access building information and control facilities. *AirFlowController* and *AirTemperatureController* are common node types in AHU applications. Listing 6.3 demonstrates their description based on TOSCA.

Derived from the base controller properties, these two specific controllers add the controller-specific operations–*ChangeSetPoint* and *ChangeAirFlowRate*, respectively with

Listing 6.2: Base Node Types Properties

```
<element name="ControllerProperties">
  <complexType>
    <sequence>
      <element name="Driver" type="string" />
  ...
</element>

<element name="GatewayProperties">
  <complexType>
    <sequence>
      <element name="User" type="string" />
      <element name="Password" type="string" />
  ...
</element>

<element name="DriverProperties">
  <complexType>
    <sequence>
      <element name="Version" type="string" />
  ...
</element>
```

Listing 6.3: Domain-specific Node Types

```
<NodeType name="AirTempController">
  <DerivedFrom typeRef="Controller"/>
  <NodeTypeProperties element="AirTempProperties"/>
  <Interfaces>
    <Interface name="AirTempInterface">
      <Operation name="ChangeSetPoint">
        <InputParameters>
          <InputParameter name="SetPoint"
            type="xs:double"/>
  ...
</NodeType>

<NodeType name="AirFlowController">
  <DerivedFrom typeRef="Controller"/>
  <NodeTypeProperties element="AirFlowProperties"/>
  <Interfaces>
    <Interface name="AirFlowInterface">
      <Operation name="ChangeAirFlowRate">
        <InputParameters>
          <InputParameter name="FlowRate"
            type="xs:double"/>
  ...
</NodeType>
```

SetPoint and *FlowRate* parameters. In our case, the controller properties are the same as the input parameters, thus not listed.

Concrete Node Types

The *Concrete Node Types* define the node types to be used in a specific application, with information about specific hardware and software vendors, models and versions. We only present the type definition of a Niagara Gateway in Listing 6.4 as an example, because other concrete node types follow the similar inherited relationship with their parent node as illustrated in Figure 6.2. The properties include information about the hardware model and software version.

Listing 6.4: Concrete Node Types

```
<NodeType name="NiagaraGateway">
  <DerivedFrom typeRef="Gateway" />
  <NodeTypeProperties element=
    "NiagaraGatewayProperties"/>
</NodeType>
```

Node Templates

According to the TOSCA specification, *Node Templates* describe the specific instances of node types. The properties of a certain node type should be set in node templates. Essentially, Node Types describe the model of nodes, whereas Node Templates describe the actual nodes to be used in a certain application deployment. Listing 6.5 presents a template of Air Temperature Controller by Johnson Controls. Since there is an interface to set the output temperature, the *SetPoint* property can be changed at runtime.

Listing 6.5: Node Templates

```
<NodeTemplate id="JCAirTempControllerTemp"
    name="Johnson Controls Air Temperature Controller"
    type="JCAirTempController">
  <Properties>
    <ControllerProperties>
      <Driver>oBIX</Driver>
    </ControllerProperties>
    <AirTempProperties>
      <SetPoint>21</SetPoint>
    </AirTempProperties>
  </Properties>
</NodeTemplate>
```

6.2.4 Modeling the relationships

The relationships required in our AHU application are common to many IoT applications. Listing 6.6 presents the four basic relationships illustrated in Figure 6.1. The names of the relationship types are self-explanatory. Each of the relationships are characterized by a source type and a target type. The relationship definitions can be used to specify the semantics of links between nodes and the methods of connections. Similar to node types, relationship types can also be inherited with more concrete properties, and instantiated into *Relationship Templates*.

Listing 6.6: Relationship Types

```
<RelationshipType name="dependOn">
  <DerivedFrom typeRef="RootRelationshipType" />
  <ValidSource typeRef="Controller" />
  <ValidTarget typeRef="Driver" />
</RelationshipType>

<RelationshipType name="connectedTo">
  <DerivedFrom typeRef="RootRelationshipType" />
  <ValidSource typeRef="Controller" />
  <ValidTarget typeRef="Controller" />
</RelationshipType>

<RelationshipType name="installedOn">
  <DerivedFrom typeRef="RootRelationshipType" />
  <ValidSource typeRef="Driver" />
  <ValidTarget typeRef="Gateway" />
</RelationshipType>

<RelationshipType name="deployedOn">
  <DerivedFrom typeRef="RootRelationshipType" />
  <ValidSource typeRef="Controller" />
  <ValidTarget typeRef="Gateway" />
</RelationshipType>
```

6.3 Deploying IoT Application Artifacts

Artifacts are the actual scripts, files, packages, executables and all other necessary software pieces to be deployed in order to run an application. Common artifacts in cloud applications may include installation scripts, configuration files, archives and so on. For traditional IoT applications, even though the basic artifact types are similar to cloud applications, the actual artifacts required by each application are highly dependent on the deployment environments. Based on the basic modeling in the previous section, we will demonstrate how TOSCA can help to manage IoT application deployment on heterogeneous environments.

6.3.1 Artifact Types

The two gateways used in our implementation—Niagara and Sedona—feature different runtime environments, programming languages and deployment procedures. However, the basic artifact types can be modeled in the same way as cloud applications[3].

File Artifact Generic artifact type that contains certain information required during an application's lifecycle.

Script Artifact Executable or interpretable artifact that encapsulates instructions in a script language for a certain operation.

Archive Artifact A collection of files that are packaged for deployment.

For the deployment on gateway environments, we extend these basic artifact types to two other common types: *SourceArtifact* and *BinaryArtifact*. They are listed in Listing 6.7, and their properties in Listing 6.8. Sources are to be compiled by a compiler decided by the language of the source, whereas binaries are executed in a specific runtime environment.

Listing 6.7: Artifact Types

```
<ArtifactType name="SourceArtifact">
  <DerivedFrom typeRef="FileArtifact" />
  <PropertiesDefinition element=
    "SourceArtifactProperties" />
</ArtifactType>

<ArtifactType name="BinaryArtifact">
  <DerivedFrom typeRef="FileArtifact" />
  <PropertiesDefinition element=
    "BinaryArtifactProperties" />
</ArtifactType>
```

[3] http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html

Listing 6.8: Artifact Types Properties

```
<element name="SourceArtifactProperties">
  <complexType>
    <sequence>
      <element name="Language" type="string" />
      <element name="Compiler" type="string" />
  ...
</element>
<element name="BinaryArtifactProperties">
  <complexType>
    <sequence>
      <element name="Environment" type="string" />
  ...
</element>
```

6.3.2 Artifact Templates

Application deployment usually constitutes a series of operations specified by the vendor of a gateway. These operations transform artifacts, put them into specified locations, and set their status. We will model the Niagara and Sedona artifacts respectively in the following.

Niagara artifacts

Similar to nodes and relationships, the actual artifacts used in an application are specified in templates. There are four files required in deploying a Niagara application, explained as follows.

Slots In Niagara, a component is defined as a collection of slots, which specify the *properties*, *actions* and *topics* of events that the component is listening to. Although Niagara is essentially a Java runtime environment, the vendor, Tridium, made a customized process based on slot definitions. Slots are included in a comment section at the beginning of a class in java source. Thus the Java source file has to be preprocessed by a tool called *Slot-o-matic*, which translates the slot definition into actual java code that invokes BAJA (Building Automation Java Architecture) API. This has to be reflected in the artifact definition.

module-include.xml This file indicates Niagara environment to register the components to an internal registry.

module.palette This file specifies where to display the components in Niagara Development Environment, which puts the components in a tree-like structure.

The actual contents of these artifacts are out of the scope of this chapter. Due to the similarity of simple file artifact specifications, we only exemplify the first two artifacts in Listing 6.9. Note that the language and compiler property of the slot artifact are respectively *slot* and *Slot-o-matic*. At deployment stage, this will indicate TOSCA environment to invoke the Slot-o-matic tool for the source code.

Listing 6.9: Niagara Artifact Templates

```
<ArtifactTemplate id="uid:iot-niagara-file-1"
  type="SourceArtifact">
  <Properties>
    <FileArtifactProperties>
      <FileType>java</FileType>
    </FileArtifactProperties>
    <SourceArtifactProperties>
      <Language>slot</Language>
      <Compiler>Slot-o-matic</Compiler>
    </SourceArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="src">
      <Include pattern="*.java" />
  ...
</ArtifactTemplate>

<ArtifactTemplate id="uid:iot-niagara-file-2"
  type="FileArtifact">
  <Properties>
    <FileArtifactProperties>
      <FileType>xml</FileType>
    </FileArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="/">
      <Include pattern="module-include.xml" />
  ...
</ArtifactTemplate>
```

Sedona artifacts

Sedona framework is an open source IoT application environment. Compared to Niagara framework, Sedona is designed to keep the framework and application footprints small so that they can be deployed on resource-constraint devices. Applications are portable among devices with Sedona framework thanks to the Sedona Virtual Machine (SVM), which is similar to the concept of JVM. In fact, Sedona language is also similar to Java.

The artifacts required by the Sedona framework to deploy an application are more complicated. Figure 6.3 illustrates the artifact structure using a screenshot from the development environment. The function and type of each artifact is explained as follows.

Sedona source files. These files are indicated by the ".sedona" extension and compiled by the *sedonac* tool.

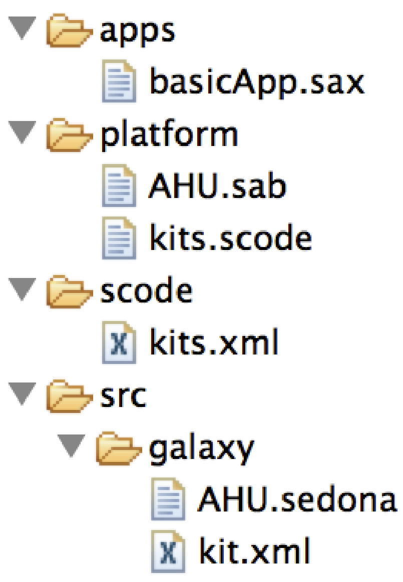


Figure 6.3: Sedona artifacts

kit.xml Each Sedona component is called a kit. The kit.xml file defines the metadata for compiling sources into a kit.

kits.xml specifies the kits that are needed to build a deployable image, or the archive that can be deployed to SVM for running an application. The oBIX driver required to run the AHU application is compiled into the image.

***.scode** The image file built according to the specifications in kits.xml.

***.sax** Sax file constitutes the actual application configuration including for example the communication port and access credential.

***.sab** It is the executable binary file that is compiled according to the specification in the corresponding .sax file. The control logic of the AHU application is realized in this file.

All the required artifacts described above have to be correctly presented in the directory structure defined by the Sedona framework. Listing 6.10 presents examples of using TOSCA to specify the source, binary and deployable image to ensure that the files are correctly deployed.

Listing 6.10: Sedona Artifact Templates

```
<ArtifactTemplate id="uid:iot-sedona-file-1"
  type="SourceArtifact">
  <Properties>
    <FileArtifactProperties>
      <FileType>sedona</FileType>
    </FileArtifactProperties>
    <SourceArtifactProperties>
      <Language>sedona</Language>
      <Compiler>sedonac</Compiler>
    </SourceArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="src">
      <Include pattern="*.sedona" />
  ...
</ArtifactTemplate>

<ArtifactTemplate id="uid:iot-sedona-file-6"
  type="BinaryArtifact">
  <Properties>
    <FileArtifactProperties>
      <FileType>sab</FileType>
    </FileArtifactProperties>
    <BinaryArtifactProperties>
      <Environment>SVM</Environment>
    </SourceArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="platform">
      <Include pattern="*.sab" />
  ...
</ArtifactTemplate>

<ArtifactTemplate id="uid:iot-sedona-file-7"
  type="ArchiveArtifact">
  <Properties>
    <FileArtifactProperties>
      <FileType>scode</FileType>
    </FileArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="platform">
      <Include pattern="*.scode" />
  ...
</ArtifactTemplate>
```

6.4 Discussion

TOSCA, as a newly established standard to counter growing complexity and isolation in cloud application environments, is gaining momentum in industrial adoption as well as academic interests. Following the first edition of TOSCA standard, we have showed that it is capable of specifying the basic constructs of IoT applications. By archiving the previous specifications and corresponding artifacts into a csar file, and deploying it in a TOSCA environment, the deployment of the AHU application onto various gateways can be automated. This section will further discuss our experience gained from attempting to employ TOSCA in the IoT domain that is abundant of proprietary and largely heterogeneous frameworks.

Acknowledge the heterogeneity In our previous work, we have proposed and prototyped the IoT PaaS framework that aims at more efficient and scalable IoT service delivery. The basic assumption is that IoT infrastructure is heterogeneous and will continue to be so. Thus, rather than proposing another "universal" architecture, we try to develop a methodology to easily integrate different domain-specific protocols and data models. That is *domain mediator* in the IoT PaaS architecture. Even worse than the situation in data exchange protocols, the deployment processes of an application can vary among IoT solutions even if the applications are realizing the same service. Following the same principle of avoiding proposing another "universal" management process, we leverage TOSCA to manage such heterogeneity in a coherent way—using a common vocabulary and syntax to describe application configurations and their deployment processes. As demonstrated in our AHU example, the node and relationship models can be shared for the same application, and the artifact models can be reused for gateways using the same software framework.

Other TOSCA features We used several main features of TOSCA, namely *Node types*, *Relationship types*, *Artifact types*, *Properties*, *Interfaces* and corresponding templates. This assumes that TOSCA will process this service template in a declarative manner—the process of deployment is implicitly inferred according to the relationships expressed in the topology template. This will work for relatively simple applications as the simplified AHU. However, for more complicated applications, *Plans*, or the imperative approach, will be needed to explicitly invoke lifecycle operations and automate complex management processes. Furthermore, explicitly expressing the *Requirement* and *Capability* types will help the TOSCA environment to more accurately understand the dependencies between nodes, thus improve the reusability of models.

Tooling and efforts of applying TOSCA As a new standard, the implementation of corresponding TOSCA tools is still in progress. The available tools have not realized all the standardized features. We are in the process of connecting the work-in-progress TOSCA environment with established IoT frameworks. We view this as a crucial effort in the early stage of the new standard. When the tool is matured and user contributions grow, more efforts will fall on collecting and improving models so that they can easily be reused. The tedious efforts of modeling each tiny aspect of IoT applications will eventually be rewarded with greater efficiency and reusability in the application management process.

6.5 Summary

In the face of the growing heterogeneity in IoT infrastructure and the need for more reusable and scalable IoT solutions, we propose to leverage cloud as a horizontal platform for managing the lifecycle of traditional IoT applications. In this chapter, we presented the first efforts of using TOSCA to formally describe the internal topology of application components and the deployment process of IoT applications. The feasibility of TOSCA for this purpose is demonstrated by describing the application components, relationships and artifacts of the AHU application using the first edition of the TOSCA specification. By inputing these descriptions to the TOSCA environment, the deployment process can be interpreted and automated.

CHAPTER 7

Monitoring IoT Applications

In this chapter we present an approach that allows for in-depth analysis of data-intensive IoT applications by non-intrusively adding functionality for acquiring and publishing performance measurements at runtime, to the application. Furthermore, our approach provides a flexible mechanism for integrating different execution environments, which can be used for deploying and monitoring applications independent from a specific operator model. Additionally, we present an extensible approach for gathering and analyzing measurement data. In order to evaluate our solution, we developed a scenario application, which we used for testing and monitoring its performance on different execution environments.

7.1 Overview

Modern information systems need to process an ever-increasing volume of data from various sources while providing timely responses to requests from stakeholders. Traditionally, such systems persist incoming data in a database and then execute queries on the stored data to perform required analyses and produce desired results, but are increasingly incapable of coping with the sheer volume of data to process, often making it infeasible to even store all incoming raw data [49]. The requirement for timely responses to complex queries over continuous streams of high-volume data led to the emergence of stream processing systems that do not rely on traditional data processing models. Stream processing systems are designed to continuously process and analyze incoming data streams to produce results in reaction to events observed in the incoming data, as opposed to separately triggered requests to analyze previously stored data.

Due to their intrinsic requirements, performance testing and monitoring [82] of stream processing applications are inherently important for stakeholders to assess and understand the status and runtime characteristics of deployed applications. Since the capabilities of single machines are insufficient for providing the necessary processing power for handling these huge amounts of data, stream processing applications have to scale computations across multiple machines and face the challenge of becoming inherently distributed [142]. In addition to several design and management challenges, this also leads to increased complexity when dealing with performance testing and monitoring. Especially for applications where monitoring was not considered initially, gathering meaningful measurements is challenging.

While there is a host of existing monitoring systems for gathering performance data about applications [39, 64, 88] and their runtime infrastructure [34, 75], to the best of our knowledge, there is no solution that specifically targets stream processing applications and their structure in a way that allows for detailed examination and comparison of their runtime characteristics, independent of the used stream processing framework. Furthermore, emitting data to monitoring systems usually requires code changes in applications that specifically target the used monitoring system.

In this chapter, we introduce MOSAIC, a service oriented framework for monitoring the runtime performance of distributed stream processing applications, independent from a particular operator model (e.g., query, graph, or API). In addition, we present an approach that allows adding monitoring functionality to existing JVM-based applications, without changing the applications' code or requiring recompilation. MOSAIC allows for the integration of different stream processing engines for executing and monitoring applications, and provides a flexible mechanism for gathering and analyzing performance measurements based on a generic domain model. We illustrate the feasibility of our approach by monitoring and analyzing the performance of a representative stream processing application deployed on two different stream processing engines.

The remainder of this chapter is structured as follows: In Section 7.2 we further motivate our work and outline the specific problem and requirements in the context of our work in the smart city domain. In Section 7.3 we introduce the MOSAIC framework and accompanying toolset to address the identified problems. We provide a detailed evaluation of framework characteristics and capabilities in Section 7.4, followed by a conclusion in Section 7.5.

7.2 Requirements

The rapid adoption of the smart city paradigm combined with the extensive growth of today's metropolises have led to a significant increase of monitoring and control system deployments [59]. These systems penetrate all vital areas of today's cities, including building monitoring and management, traffic control, as well as energy management systems via smart meters. Naturally, these system rely on stream processing applications that allow to rapidly process and react to relevant events that occur in associated, large-volume data sources. Performance, availability, and reliability of these systems has become a critical factor in the operation of modern city infrastructures.

However, the enormous scale combined with the distributed and heterogeneous nature of the deployed stream processing applications poses significant challenges for monitoring mechanisms. Due to the wide variety of available stream processing frameworks and used processing models, finding the right framework for any given task and objectively assessing the resulting application are not trivial. In order to enable a holistic approach to monitor such applications, we argue that the following requirements must be met:

- *Distributed Operation*: A monitoring mechanism must be able to handle the inherent distributed nature of modern stream processing applications. Since single machines cannot provide the necessary processing power for running such applications, computations must be scaled across a distributed infrastructure. Apart from the overhead of managing and provisioning these infrastructures, also monitoring gets more complex as performance measurements of multiple resources have to be considered and appropriately handled.

- *Non-Intrusiveness*: A monitoring mechanism should allow to extend the set of measured metrics beyond traditional performance monitoring, in order to provide an in-depth look into relevant application characteristics without requiring code-level changes, or having to rebuild the stream processing application itself. Furthermore, it should also be possible to acquire measurement data from applications, where monitoring was not considered from the beginning.

- *Model Independence*: A monitoring mechanism should provide means to monitor applications independent from a particular processing model or execution environment, in order to enable an integrated and holistic monitoring concept. Current stream processing frameworks employ various models to define application logic (e.g., queries based on domain-specific languages (DSLs), operator graphs, or programming APIs), which should be supported by the monitoring framework without requiring changes to the application code.

To provide accurate performance monitoring, a solution that respects the requirements defined above should also allow for the analysis of application performance behavior, which comprises the following steps: (i) Acquisition: the process of measuring performance data, (ii) Publication: the process of publishing acquired data, (iii) Management: the process of managing and storing collected data, and (iv) Analysis: the process of extracting information from monitoring data.

7.3 The MOSAIC Framework

In order to address the previously defined requirements, we present MOSAIC, a framework for non-intrusive monitoring of stream processing applications. The overall architecture of our approach is depicted in Figure 7.1 and consists of the following components: (i) a Stream Processing Environment, (ii) MOSAIC Base, and (iii) MOSAIC, a cloud-based middleware framework. In the following, we discuss these components in more detail, and present the overall approach of weaving, deploying, and monitoring stream processing applications.

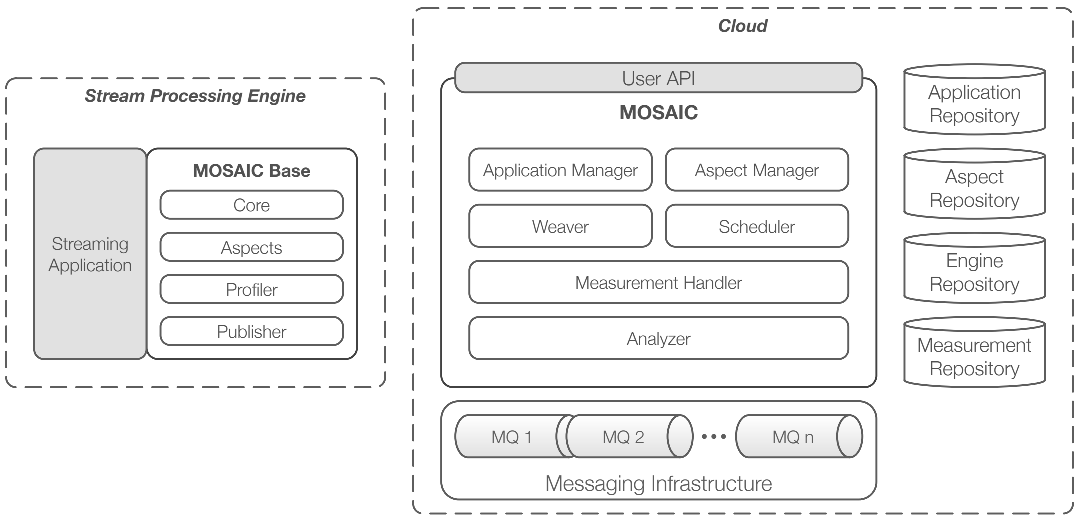


Figure 7.1: MOSAIC – Overview

7.3.1 Stream Processing Environments

Usually stream processing applications consist of various processing steps, such as validation, transformation, aggregation, and analysis [49]. Such applications are used to process data in order to extract or create information. Since stream processing applications have to deal with an ever-growing amount of data, the actual processing work is then distributed across multiple worker resources. In order to manage this distributed processing more efficiently, stream processing applications are deployed and executed on top of stream processing engines that provide seamless provisioning of worker nodes.

To allow our framework to monitor stream processing applications that are executed on top of stream processing engines, we focus on frameworks deployed on the Java Virtual Machine (JVM), such as Apache Spark [4] and Apache Storm [6, 123]. We facilitate aspect-oriented programming (AOP) to weave components of our framework into the actual stream processing application in order to add the necessary monitoring functionality without requiring changes to the underlying application.

Next, to represent a stream processing application and associate respective monitoring data, we introduce a domain model that consists of the following elements. The central element is a `Node`. A node is a representation of a worker performing a specific task in a distributed stream processing application. In order to identify the node, it has a `nodeId`. In addition, a `nodePurpose` attribute describes what the node is actually doing. This approach allows grouping nodes according to their functionality, but also supports distinguishing single nodes. For example, consider an aggregation operation that is intense in computation, and the application requires multiple instances for this specific operation. In such cases, several nodes that share a common purpose are executed, but have different identifiers. Monitoring information that is associated with nodes is commonly referred to as `Measurement`, where we distinguish between `RuntimePerformance` and `JvmProfile`. RuntimePerformance represents runtime measurements and allows monitoring the runtime of one or several operation steps. Furthermore, since an operation can consist of multiple steps, we introduced a `sequence` attribute that connects these records. The JvmProfile is used to monitor resource statistics of the underlying JVM.

7.3.2 MOSAIC Base

In order to allow for detailed and application-specific monitoring, we split our framework in two parts. One part, the actual monitoring and analysis framework (MOSAIC) is deployed in the cloud. The other part (MOSAIC Base) is depicted on the left-hand side in Figure 7.1 and contains the application-specific components that need to be integrated into the application in order to acquire the required monitoring information.

Core

The `core` component is the centerpiece of the framework. It contains the domain model, the basic runtime performance measurement functionality, and an abstraction for transferring as well as storing acquired measurement information. Additionally, the component provides out-of-the-box integrations for several stream processing engines. The core component also offers extension APIs to allow for easy integration of additional stream processing engines.

Aspects

In order to acquire measurement data, we use aspects respectively advised code that is woven into the target application, as provided by the AspectJ [9] AOP framework. The basic approach of obtaining measurement data consist of the following three steps: First, save the start-timestamp before the monitored code block. Second, save the end-timestamp after the monitored code block is finished. Finally, publish the data. By following the aspect oriented programming principle this can be done using advices of type `Around` (an entire method is wrapped around a pointcut), or `Before` and `After` (two advised methods are invoked) advices. We implemented two different abstract aspects for measuring runtime performance. Additionally, to connect runtime measurements of processing steps (i.e., to establish a sequence of measurements), we need a correlation identifier for a sequence, which must be passed on from one process step to the next. Since we cannot assume that data that is used within an application provides such a sequence identifier, we add and pass on sequence identifiers within an application. This approach comprises a static crosscutting advice to add a sequence identifier to data objects, and two dynamic crosscutting aspects that create and pass on sequences.

Profiler

The `profiler` component contains all classes for monitoring the JVM resources. In addition, the component provides functionality required for profiling JVM resources for a particular code block in order to create execution profiles.

Publisher

The `publisher` groups different built-in mechanisms for publishing and storing monitoring information. Currently, we provide functionality to log and persist measurement information for files, JDBC, JMX, and log4j. Data is then distributed via the Java Message Service (JMS).

Since the integration must be as flexible as possible, aspects that acquire performance measurements are woven into the target application. These woven aspects use functionality provided by the core component and publish the measurement data using the publisher component. Once an aspect is woven into the target application, the advised code interacts with the provided functionality of MOSAIC Base when executed.

7.3.3 MOSAIC

The enabling framework for monitoring and analyzing stream processing applications is depicted on the right-hand side in Figure 7.1. MOSAIC is a cloud-based framework and the overall design follows the micro service architecture [87]. This approach enables developing a scalable and evolvable framework. In addition, it allows the flexible management and scaling of components, which is important for MOSAIC when handling stream processing applications deployed at large scale. In the following, we will introduce the main components of MOSAIC.

Messaging Infrastructure

For handling the multitude of monitoring information, MOSAIC uses a distributed messaging fabric that minimizes unnecessary network traffic compared to a centralized message bus. Additionally, it allows components of the system to freely choose the most suitable (e.g., closest) connection point. For transmitting and consuming data, MOSAIC uses a publish/subscribe mechanism, where producing components publish data in the messaging infrastructure to a specific routing key, which represents the application's unique id. Consuming components of MOSAIC, which are mostly analyzer components, consume data by subscribing to the according routing key. This approach provides a flexible and easily extendable mechanism to handle monitoring information.

Repositories

In order to manage registered stream processing applications, developed aspects for acquiring measurement data, and monitoring data, MOSAIC provides several repositories.

- **Application repository.** To allow our framework to monitor stream processing applications, operators of such applications need to register them via the provided `User API`. During the registration process operators upload the application as a jar package, add the required runtime configuration, and finally state the intended execution environment (i.e., stream processing engine). Next, based on the stated execution environment, operators can define which monitoring information they are interested in and state the required granularity. For example, operators can define if they are just interested in runtime performance of the application, or also additional JVM-specific profiles. All this application-specific information is managed and stored in the application repository.

- **Aspect repository.** The framework provides abstract aspects that can be used by defining concrete pointcuts. These abstract aspects can be extended to integrate arbitrary execution environments that are used for running stream processing applications. In addition to abstract aspects, MOSAIC also provides built-in integrations for Apache Spark Streaming [4] and Apache Storm [6]. Any additional or new aspects developed and registered by an operator are stored and managed in the aspect repository.

- **Engine repository.** In order to allow our framework to deploy stream processing applications on their intended stream processing engine, we need engine-specific drivers. As for aspects, we define an abstract driver that can be extended to integrate additional, currently not supported, execution environments. The built-in drivers and additional drivers are stored and managed in the engine repository.

- **Measurement repository.** Since our framework does not only allow monitoring of stream processing applications, but also analyzing gathered measurement data, MOSAIC provides a measurement repository for persisting gathered monitoring information.

Application Manager

The application manager is responsible for handling registered stream processing applications and associated information in the application repository. Furthermore, during the application registration process, the manager takes care of verifying that the provided information and application package are valid, and if a suitable engine-driver is available. If an engine-driver is available, this driver is then associated with the application and will be used for subsequent deployment. If no suitable engine-driver is present, the operator will be notified and is then required to provide a driver via the user API. Furthermore, to keep track of deployed applications and associated monitoring information, the application manager handles the state of applications.

Aspect Manager

Since we want to create and acquire a broad variety of measurement data, MOSAIC needs to provide suitable aspects. To handle these aspects efficiently the aspect manager is used. As already described, MOSAIC provides built-in aspects and allows operators to add their custom-developed aspects by extending the abstract aspects. In addition to aspects, the manager is also responsible for handling and creating configuration files that define pointcuts for concrete aspects. Based on this configuration the `Weaver` component then weaves the aspect code at the defined places in the application to provide the expected monitoring information.

Weaver

To provide a flexible and extensible weaving component that allows using different aspect-oriented programming frameworks, we define a weaver interface and provide a concrete implementation that relies on AspectJ [9]. Based on AspectJ, we decided to use load-time weaving by facilitating the Java agent provided by AspectJ. This approach allows us to include the Java agent as an argument during application startup and add a configuration file to the classpath. The Java agent is then responsible for weaving the aspects at application load-time. This weaving option itself causes comparatively little overhead [118], since an advice, once weaved, behaves like a usual method call. However, this weaving approach increases the time needed for starting the application, which is acceptable since we consider long running stream processing applications.

Scheduler

The scheduler component is responsible for deploying the stream processing application, the required functionality for weaving, and MOSAIC Base on the actual execution environment. In order to deploy an application and start the monitoring process, the scheduler contacts the application manager and receives the application package and the corresponding information, stating the intended execution environment and required monitoring information. Based on this information, the aspect manager is contacted, which provides the tailored MOSAIC Base containing the required aspects and configurations. Next, the weaver provides the specific functionality that is needed for weaving MOSAIC Base in the application. Finally, the correct engine driver is loaded and the complete package is deployed on the stream processing environment, where the application is started. During startup the weaving process is triggered, which integrates the monitoring-specific code into the application. Once the application is successfully deployed, the scheduler notifies the application manager that the application is running.

Measurement Handler

To handle published monitoring information, the measurement handler is used. Based on the currently registered and running applications, the handler starts application-specific data handlers that consume the published data and store them in the measurement repository. This approach allows for flexible and efficient management of measurements.

Analyzer

In addition to collecting application-specific performance measurements, we also want to analyze this data to obtain actionable insights. Therefore, we provide an abstract analyzer and corresponding implementations that, based on our domain model, allows analyzing metrics like runtime per processing step, overall runtime of a sequence of processing steps, and the latency between processing steps.

7.3.4 Weaving and Monitoring Approach

Since we cannot show a complete list of provided monitoring capabilities of our framework due to space constraints, we will focus on one specific example and explain how the overall process, from scheduling an application to receiving monitoring data, works.

For this example, we want to measure the runtime of a processing step in an Apache Storm stream processing application. Listing 7.1 shows a stub for measuring the runtime performance, where an around advice is used. An abstract pointcut, which has to be defined when using this abstract aspect is used for advising the monitoring code. The advice continues the execution at the given join point and measures invocation time.

```
@Aspect
public abstract class AbstractRuntimePerformanceAspect {
    @Pointcut
    public abstract void scope();

    @Around("scope() && this(jpo)")
    public Object around(ProceedingJoinPoint pjp, Object jpo) throws
        Throwable {
        ...
    }
}
```

Listing 7.1: Abstract Runtime Performance Aspect

To use this abstract aspect we now specify that we want to measure the runtime of a processing step (a bolt) in our Apache Storm application. According to this definition, MOSAIC creates a configuration file that defines the pointcut for the aspect, as shown in Listing 7.2. The concrete-aspect element defines an aspect and a new name for the aspect. Furthermore, the extends attribute defines the abstract aspect. The expected pointcuts of the abstract aspect are set using the pointcut element.

```
<aspectj>
  <aspects>
    <concrete-aspect name="BoltRuntimePerformanceAspect" extends="
        AbstractRuntimePerformanceAspect">
      <pointcut name="scope" expression="execution(* backtype.storm.topology.
          IRichBolt.execute(..))" />
    </concrete-aspect>
  </aspects>
</aspectj>
```

Listing 7.2: Configuration (*aop.xml*) Example

After defining what and how we want to monitor an application, the framework has to deploy and execute the application on the intended execution environment. As discussed above, the scheduler component is responsible for deploying the application and the additional tailored MOSAIC Base components, as well as the needed weaving functionality, on the execution environment. Next, the scheduler starts the application on the target environment, which triggers the load-time weaving process using the weaving functionality provided by our framework. Once MOSAIC Base is weaved into the target application, the execution is initiated and the application starts processing data. While processing incoming data, the monitored processing step of our target application will be invoked. Since we weaved our framework into the application, we can monitor the processing step as depicted in Figure 7.2.

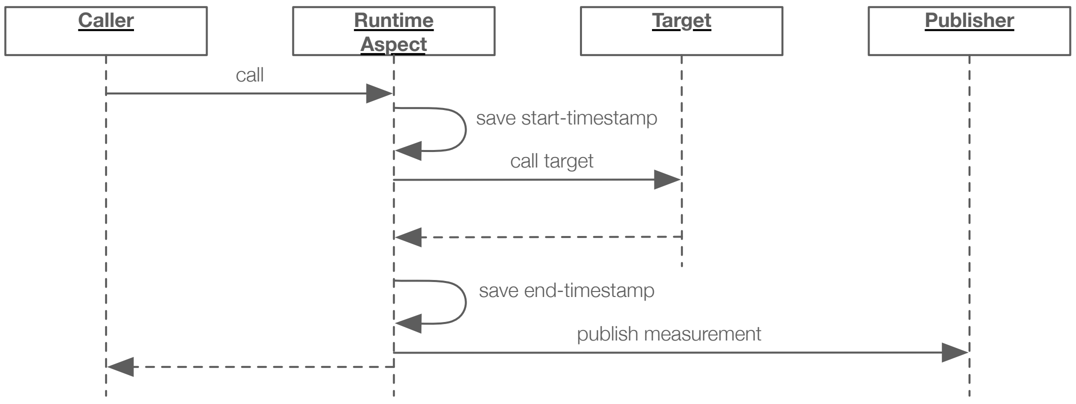


Figure 7.2: Monitoring of a Processing Step with a Runtime Aspect

As illustrated in Figure 7.2 before the actual processing step (target) of our application is called, the runtime aspect is invoked, which saves the start-timestamp and then executes the actual processing step. When the processing step is finished, the runtime aspect saves the stop-timestamp, and publishes the measurement using the publisher provided by MOSAIC Base. The measurement is then consumed on the cloud-based framework, stored in a repository, and can then be further analyzed.

7.4 Evaluation

To evaluate our approach we chose a representative scenario and implemented it on top of two stream processing engines. Next, we created a test setup in the cloud using several VM instances for hosting our framework, as well as the stream processing engines.

In the remainder of this section we give an overview of the chosen scenario and the developed applications, discuss the concrete evaluation setup, present different evaluation scenarios, and analyze the gathered results.

7.4.1 Scenario

To illustrate the feasibility of the MOSAIC approach, we will use a modified version of the well-known Traveling salesman problem (TSP) [106] as a scenario application. A

TSP graph G is a complete weighted undirected graph specified by a pair (N, d), where N is a set of nodes and d is a function that translates the distance between two nodes in a numerical value. d satisfies two conditions: (1.) Symmetry: $d(i, j) = d(j, i), \forall i, j \in N$. (2.) $d(i, j) >= 0, \forall i, j \in N$. A *path* of the TSP graph G is a set of edges that describes a path containing each node exactly once (i.e., a Hamiltonian graph). The path distance is the sum of distances of all edges. The solution for our modified traveling salesman problem is a path with the minimal possible path distance.

7.4.2 Sample Application

Based on the described TSP scenario we implemented a stream processing application for both, Apache Spark and Apache Storm. In order to make the gathered monitoring results comparable, the implementations are designed to share the same basic architecture.

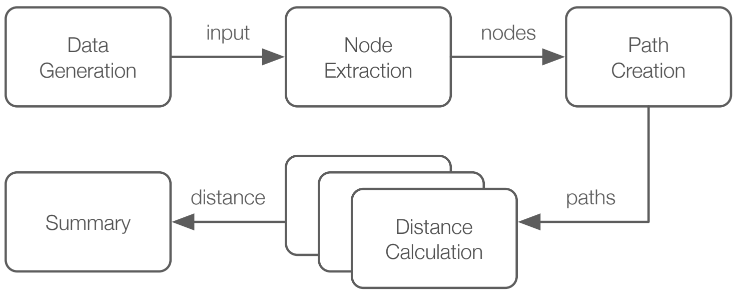
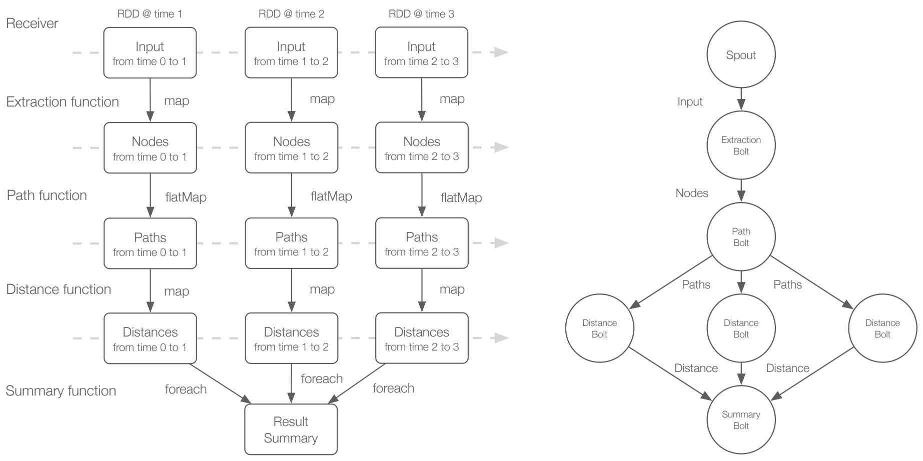


Figure 7.3: Sample application – Overview

The overall architecture of the sample application is split into five processing steps as shown in Figure 7.3. In the first step the input data for the application is created. Processable input data for our scenario is a random string containing vertices in a specified format (i.e., x- and y-coordinates in a two-dimensional space), where before and after each vertex there can be random characters. For the purpose of this scenario, we assume that the coordinate data is embedded in a text document and coordinates must be extracted from the input data in a separate step. The result of this extraction step is a list of Node objects with an id to identify the node, as well as x and y coordinates. In the path creation step, all paths are created. This step also allows splitting paths into subsets to subsequently pass them on to multiple workers for parallel distance calculation. In the last step, the minimal distance of the received set of paths is calculated. Finally, the summary step is responsible for summarizing the partial results of the distance step, which represents the minimal distance of all paths.

7.4.3 Apache Spark Streaming Implementation

Following the aforementioned application architecture, we implemented the described scenario based on Apache Spark Streaming. First, we implemented a stream `Receiver`



(a) Spark (b) Storm

Figure 7.4: Sample application – Implementation

that creates data for the application to consume. For all other steps we implemented `Functions`, which are used by map or flat map transformations and one output operation. Figure 7.4a shows an overview of our Apache Spark Streaming application. The Receiver creates data, which is stored in Spark RDD's over time. We decided to use a time window so that a RDD is created in a defined interval, containing all data records stored in this time interval. The second step, extraction, is done via a map function, where lists of nodes are extracted from the input data in a RDD. Paths are created by using a flat map transformation. Using an identifier, each path list can be associated to its source node list. The path creation step is followed by a map transformation, which calculates distances for each path list and determines the minimal distance. The summary function determines the absolute minimum distance for an input string by aggregating all received minimum distances with the same identifier.

7.4.4 Apache Storm Implementation

As mentioned above, we also implemented the described scenario according to the architecture description using Apache Storm as depicted in Figure 7.4b. For each step described above, we implemented a Storm-based component. The first step, data creation, is implemented as a `Spout` that creates and emits data to the topology. All other steps are implemented as `Bolts`, linked together using Storm's shuffle grouping method. Path bolts are different to other bolts since they might emit multiple tuples, where each tuple contains a set of paths. Path lists can be associated with their source, and thus with each other, by using an identifier.

7.4.5 Evaluation Scenarios

In order to gather sufficient monitoring information from our sample application, we defined two scenarios. In the first scenario we executed 25 test runs with a break of 1 second between each run. In the second run, we changed the break to 10 milliseconds. For each run we generated a random input string with a size of approximately 17MB. With these two scenarios we simulate different load patterns for the application to highlight notable differences.

7.4.6 Setup

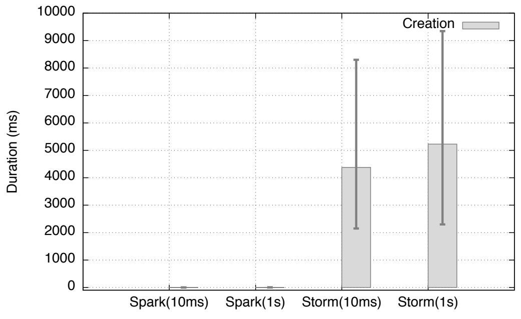
To create analyzable data, we executed both described implementations based on the defined scenarios. Since the main focus of our investigation is on the engineering perspective of how performance can be monitored, and not a performance analysis itself, we used the following setup in our private OpenStack [92] cloud.

MOSAIC is deployed on one instance using the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB disk space). For implementing our messaging infrastructure we use a RabbitMQ [103] cluster consisting of 2 VM nodes using Ubuntu 14.04 and the m1.small flavor (1920MB Ram, 1 VCPUs and 40GB disk space). Next, for hosting the Apache Spark Streaming and Apache Storm engine, we created two separate VM nodes, each using the m2.flavor (5760MB RAM, 3 VCPUs and 40GB disk space). The physical machines hosting the VMs are connected with regular 1000BASE-T ethernet links.

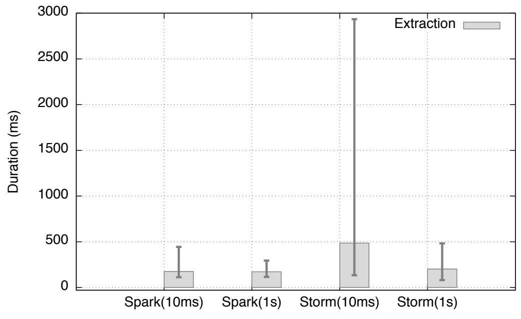
7.4.7 Results

In this section, we analyze the measurement data gathered from the test runs. By comparing the results of both implementations, we illustrate the benefit of our framework. The common data model for monitoring different stream processing applications, implemented using different frameworks, allows for a direct comparison of monitored processing steps. Figure 7.5 shows an overview of runtime measurements for each processing step of our application. The plotted durations (end time − start time) are aggregated by node purpose (i.e., processing step) and node identifier. The node identifier allows to associate a record with the particular run or implementation (10 ms or 1 second break, and Spark or Storm implementation). The different sub-figures show the different processing steps. In the following, we discuss notable results of the comparison between the Spark and Storm application.

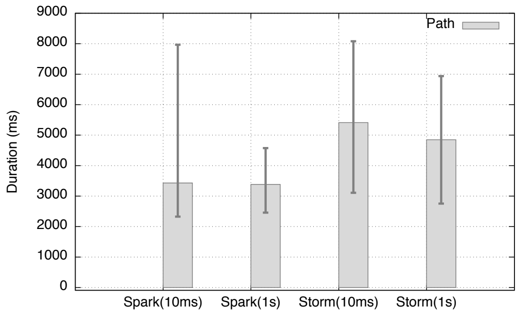
In Figure 7.5a we can see the runtime of the first processing step of our application. By looking at the figure we notice that measuring the invocation time of the store method of a Spark Receiver, does not reflect the time consumed for actually creating, reading or receiving the data. Only the time used for transferring the data to Spark is measured. This explains the large gap when comparing Spark and Storm results of the first step (Creation). Figure 7.5b, Figure 7.5c and Figure 7.5d show the gathered measurement data for the extraction, duration, and distance processing steps. We see that the Spark implementation performs better regarding runtime measurements at these processing



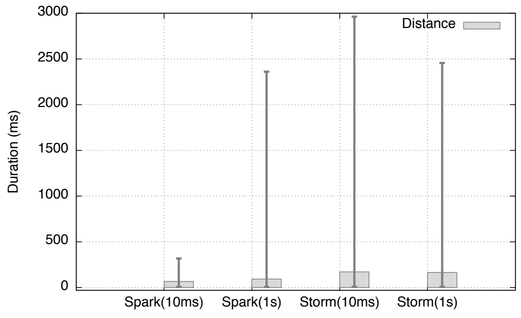
(a) Creation



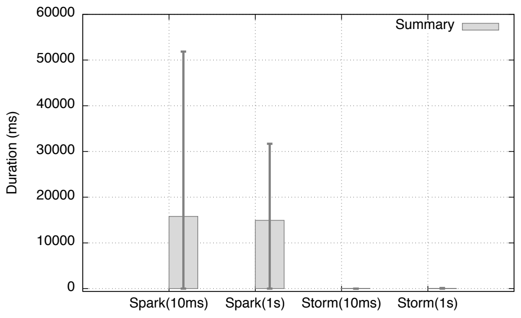
(b) Extraction



(c) Path



(d) Distance



(e) Summary

Figure 7.5: Evaluation Results – Duration per process step

steps, compared to the Storm implementation. Figure 7.5e shows the runtime of the last processing step, namely summary. When comparing the results, we notice a large difference between Spark and Storm. The explanation for this gap is as follows: Spark immediately starts the invocation of output operations for each time slot, even when no data was received within the period of a time slot. However, when no data is received, Spark blocks the output operation's function invocation and waits for a considerable amount of time. This behavior distorts the results for the last process step, since for the first few time slots no data has been passed to the output operation as the path calculations have not been finished. These records have a considerable impact on the

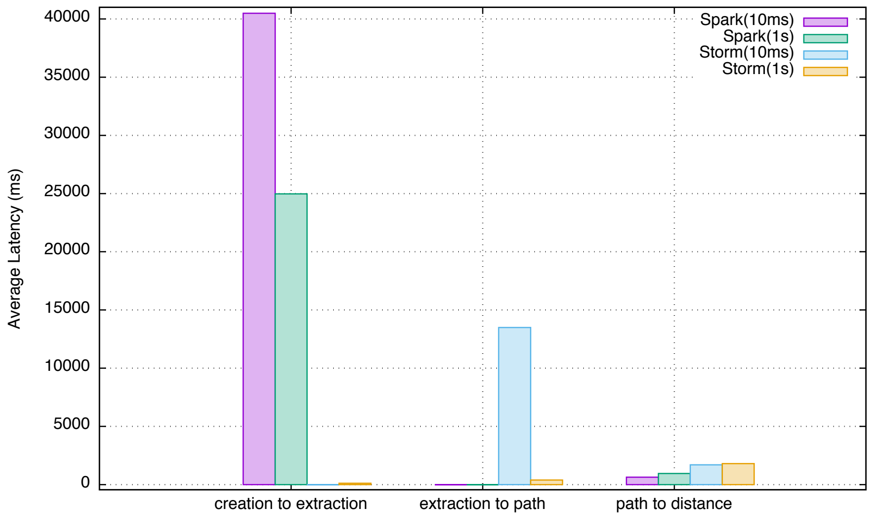


Figure 7.6: Evaluation Results – Latency between process steps

aggregated data.

Considering these results, it might appear that Spark performs significantly better. However, simply looking at runtimes of single process steps does not include the time that has been consumed by the framework or for transferring data from one processing step to another. Using the injected correlation identifiers, runtime measurement records of different process steps can be connected to each other and the time between the end time of a step and start time of a following step can be determined. Figure 7.6 depicts the latency between the processing steps.

We notice that Spark exhibits significantly higher inter-step latencies. However, these differences can be explained by the looking at the different processing models used by Spark and Storm. First, Spark creates micro batches over time, which in combination with a window operation means that after the store method of the Receiver is invoked, it can take some time until Spark passes the created data record to the next processing step. Second, since Spark combines data records in RDDs according to the window operation, the number of records that are transferred from the receiver to the first mapping function (processing step) can be considerable in size. This especially applies to the scenario with a data creation delay of only 10 milliseconds. In comparison, Storm emits a tuple as soon as it arrives at its topology. Finally, Spark and Storm employ different task scheduling models. Both engines have a fixed number of task executors. However, Spark reserves task executors for Receivers, which means that a Receiver is running continuously, whereas functions are scheduled and executed when a task executor becomes available. In Storm, Spouts are treated equally to Bolts, which means that their executions are also paused when there are no task executors available. For Spark this means that the extraction step is only executed when a task executor is available, thus processing steps may be paused. In contrast, a Receiver is running all the time, which adds an additional delay between

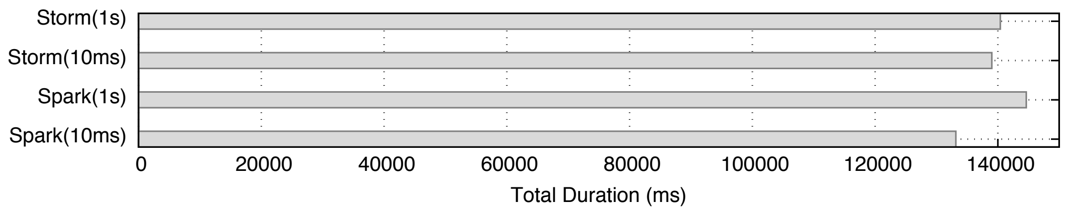


Figure 7.7: Evaluation Results – Absolute total duration

these two processing steps. In the processing model of Storm, where Spouts are scheduled and paused as well, the longest running processing step is the bottleneck, which is the path step in our application. After an extraction task is executed, it might take some time until a task executor becomes available for running the path step of the preceded extraction result, as the task executors might be busy with running other queued path tasks. When all task executors are busy with executing path tasks, no more tuples are emitted by Spouts, thus there is no increased latency between the Spout and the Bolt used for extraction.

To also discuss the total runtime of our application, Figure 7.7 shows the time between the absolute minimum start time that has been recorded for the creation processing step and the absolute maximum end time of the summary processing step. We notice that the Spark implementation performed better for the test run with the 10 millisecond delay between data creation, whereas the Storm implementation was faster for the test run with the 1 second delay between data creation.

Based on the gathered results, we showed that our approach enables acquiring performance measurements that reflect the differences between the discretized stream processing model of Spark and the continuous operation processing model of Storm. Whereas the advantages and disadvantages of these processing models are not the subject of our approach, the analysis of the gathered results proves the applicability and purpose of our framework.

7.5 Summary

In general, the runtime performance of applications is a crucial aspect, since applications that can not fulfill their performance requirements do not provide their intended purpose. Especially in the era of big data with the ever-growing amounts of data, this is particularly demanding for stream processing applications. In order to allow this type of applications to deal with the immense load, they must be scaled to multiple machines, as single machines can not provide the necessary processing power. Scaling applications appropriately requires actionable performance measurements that need to be acquired by monitoring the application. Monitoring stream processing applications, however, is a challenging task, due to their distributed nature. In addition, stream processing applications often do not provide built-in monitoring functionality that allows gathering and analyzing their runtime performance. Furthermore, traditional runtime environments such as stream processing engines do not allow fine-grained monitoring of deployed applications, but are only capable of providing engine-specific runtime data, which is not sufficient for analyzing the performance of an application appropriately. This calls for a structured approach that allows non-intrusive monitoring of stream processing applications in oder to acquire application-specific runtime performance data.

In this chapter, we introduced MOSAIC a cloud-based framework that provides a flexible approach for adding functionality for acquiring and publishing of performance measurements, at runtime to stream processing applications. The framework allows integrating different stream processing engines for deploying and executing applications, a generic domain model for storing and publishing measurements, and a mechanism for gathering and analyzing these measurements. To evaluate our approach, we developed a representative stream processing application, which we used for testing and monitoring its performance by using Apache Spark Streaming respectively Apache Storm as the underlying stream processing engines. Finally, we discussed the gathered results and showed that our approach provides actionable insights on the performance behavior of an application.

CHAPTER 8

Analyzing large-scale IoT Deployments

In this chapter, we present a generic, scalable, and fault-tolerant data processing approach based on the cloud that allows operators to perform online and offline analyses on gathered data to better understand and optimize the behavior of the available smart city infrastructure. Our approach is designed for easy integration of new data sources, provides an extensible API to perform custom analysis tasks, and a DSL to define adaptation rules based on analysis results. We demonstrate the feasibility of the proposed approach using a scenario application for autonomous intersection management in smart city environments. Our framework is able to autonomously optimize application deployment topologies by distributing processing load over available infrastructure resources when necessary based on both, online analysis of the current state of the environment, as well as patterns learned from historical data.

8.1 Overview

Smart city applications are large-scale distributed systems that react to and manipulate their physical environment using an underlying IoT infrastructure. The growing number of connected devices in current IoT infrastructures poses challenges not only for smart city applications that need to process and react to data produced by these devices, but especially for operators of the underlying smart city infrastructure who must ensure that deployed applications can optimally fulfill their requirements at all times. The inherently dynamic environment in which smart city applications are executed, creates a number of challenges. Applications must be able to quickly react to changes in business requirements and regulations, efficiently manage unreliable and expensive network links, and aim of maintaining optimal QoS in the face of infrastructure outages. While IoT infrastructures provide large amounts of performance and health data about the execution environment of smart city applications, this data is currently not effectively used to improve application execution. Moreover, application management policies for smart city applications that incorporate infrastructure data must be replicated for each new application. We argue that IoT application engineering and management can be significantly simplified by providing a dedicated component for processing, analyzing, and reacting to IoT infrastructure data.

In this chapter, we introduce Ahab, a distributed, cloud-based stream processing framework that offers a unified way for operators to better understand and optimize the managed infrastructure in reaction to data from the underlying infrastructure resources, i.e., connected IoT devices, runtime edge infrastructure, as well as the overall application execution environment. The resulting management policies can be reused for managing various infrastructure components, and the gathered data can be used to guide infrastructure evolution.

The remainder of this chapter is structured as follows: In Section 8.2 we motivate our work using a real-world scenario and outline specific requirements. Section 8.3 introduces Ahab, an approach to address the aforementioned requirements, along with a detailed evaluation in Section 8.4. We conclude the chapter in Section 8.5 with a summary.

8.2 Requirements

The success of the smart city paradigm and the advent of the IoT has led to significant convergence of traditional infrastructure and software systems. Todays cities are evolving into complex cyber-physical systems of systems integrating billions of connected and highly distributed devices. These devices are deployed in all vital areas of a city, from its infrastructure to the citizens, forming a complex network of sensors as well as computational power. Two core aspects in the evolution of smart cities are transportation and traffic systems, most notably, the advent of self-driving cars and the evolution of the city's infrastructure towards a cyber-physical adaptive system. Traffic systems already start to react to specific demands, depending on the time of day, weather conditions, and seasonal changes. Traffic lights as well as speed limits adapt to traffic conditions, road blocks, or accidents. However, this is only the first step in a natural evolution towards high density autonomous systems. Future cities will rely on adaptive roads that change according to their environment, as well as high volume autonomous car networks combined with multimodal public transport systems that efficiently transport citizens and goods in and between cities.

In order to enable these systems, it is crucial to provide sustainable computational capabilities. These systems not only need to react in an on-demand manner, they also must be able to adapt their processing capabilities accordingly in a fast and highly efficient manner. Given their complexity, paired with the inherent high availability requirements, it is not sufficient to rely only on the cloud for computational capacity. To ensure they work under all circumstances, it is necessary to utilize the computational network of the IoT infrastructure itself. This has a number of advantages ranging from economical and ecological benefits to the consideration of utilizing localities like saving bandwidth and the ability to sustain operations in disaster situations that could affect network uplinks.

For demonstration purposes, we consider the case of a multimodal traffic management system that incorporates autonomous cars as well as means of public transport. The system itself needs to react depending on daily commuter patterns, weather conditions, and accidents. It is responsible for coordinating autonomous cars, traffic lights, and one-way streets, as well as to adapt public traffic interval times accordingly. We specifically take a look at one of the most demanding elements of such a system, the intersection control. In order to enable high volume autonomous car traffic and to utilize the benefits that come with the ability of cars communicating with each other, the system needs to be able to coordinate hundreds of cars per second, per intersection. This enables the high density traffic flow that is one of the major benefits of self driving cars in the smart city domain. To enable this element, it is vital to utilize the available edge infrastructure for processing to handle the highly varying demand that comes along with it.

8.3 The Ahab Framework

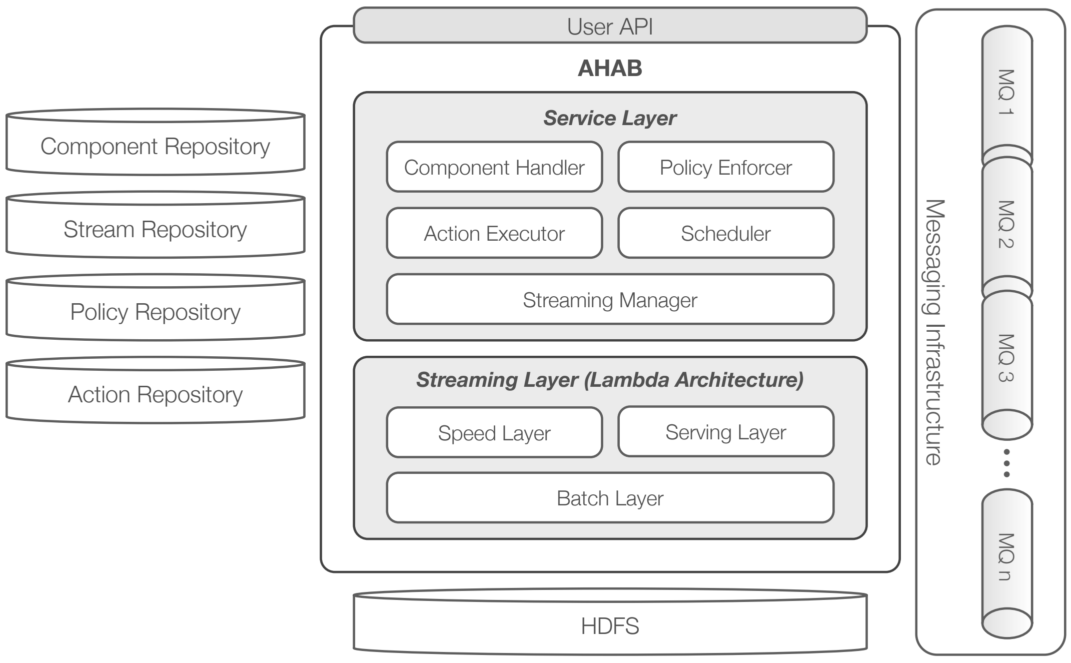


Figure 8.1: Ahab – Overview

To address the requirements discussed above, we present Ahab, a distributed, cloud-based stream processing framework allowing operators to manage and adapt IoT infrastructure components and running applications based on information extracted from data streams published by smart city infrastructure resources. The overall architecture of our approach is depicted in Figure 8.1 and consists of the following components: (i) the

User API, (ii) Repositories, (iii) a Messaging Infrastructure, and (iv) Ahab, the stream processing framework. In the following, we discuss these components in more detail.

8.3.1 User API

Since our approach should be able to handle different data streams published from various resources, Ahab provides a `User API` allowing operators managing smart city infrastructures, to define which resources of the infrastructure provide data and how this data can be processed. The API allows operators to register IoT components and respective management policies, which allows Ahab to adapt registered components based on information extracted by processing and analyzing incoming data streams. To provide an extensible and easy way to describe system components in our approach, we integrate and extend MONINA [53], a language for specifying monitoring and adaptation policies. Listing 8.1 illustrates a simplified example of a typical IoT infrastructure definition.

Listing 8.1: Sample IoT Infrastructure Definition

```
stream Response {
  processing_time_ms : Integer}

action Scale {
  amount : Long}

component Application {
  name : String
  endpoint {
    at "/application"
    stream Response
    action Scale}}

stream AverageResponse {
  processing_time_ms : Integer}

stream ProcessedResponse {
  from Application
  stream Response as r
  create AverageResponse(
    avg(r.processing_time_ms))
  window 10 seconds}

policy ScaleUp {
  from AverageResponse as a
  when a.processing_time_ms > 2000
  execute Application
          .Scale(5)}
```

In Ahab we introduce the `stream` concept to refer to both, incoming data streams provided by infrastructure components, as well as processing streams created by Ahab. To avoid limiting the structure of incoming data streams, our approach allows registering custom processing streams, which take care of transformation and further processing.

Next, a `component` defines infrastructure resources that either just produce data streams (e.g., connected edge devices) or need to react on information extracted from streams. To allow components to react on extracted information, components can define `action` attributes. Actions, together with `policy` directives, which define adaptation criteria that need to be met, allow Ahab to autonomously manage and adapt system components.

8.3.2 Repositories

To store registered infrastructure components, used streams, management policies, and actions, Ahab uses several repositories.

Component Repository Every component that represents a resource in the IoT infrastructure (e.g., connected IoT devices) and provides data that can be used to analyze the overall behavior of the system, gets first registered via the User API and is stored in the `Component Repository`. Based on this information Ahab can correlate incoming data streams with registered components and further process them accordingly. In addition to components that produce data, operators can also register components (e.g., IoT applications) that need to be managed and adapted by Ahab in order to react to incoming or processed data streams.

Stream Repository To keep track of the various incoming data and processing streams used by Ahab, our approach uses a `Stream Repository`. According to the information stored in the repository, operators can easily add additional streams or management policies that facilitate and further analyze or process data produced from underlying streams.

Policy Repository Since Ahab not only processes and analyzes data streams, but also adapts registered components based on policies, operators can register these policies in the `Policy Repository`. This approach allows Ahab to keep track of all registered policies and furthermore allows operators to create generic policies that are applicable for more than one component.

Action Repository In order to allow Ahab to manage and adapt registered components, operators need to define and register adaptation actions, which are then stored in the `Action Repository`. Once Ahab detects that a certain policy is violated or can not be met anymore, it tries to find and execute corresponding adaptation actions.

8.3.3 Messaging Infrastructure

For handling the multitude of data streams, Ahab provides a distributed messaging fabric. This approach minimizes unnecessary network traffic, compared to a centralized message bus, and allows components of the system to freely choose the most suitable (e.g., closest) connection point. For transmitting and consuming data, Ahab uses a publish/subscribe mechanism, where producing components publish data in the messaging infrastructure to a specific routing key. This routing key contains the name of the component and the type of data stream it is publishing. For Ahab we distinguish different types of data streams: incoming data streams, such as access logs or metrics from infrastructure components, and processing data streams, that process and analyze other data streams to extract more actionable information. Consuming components of Ahab, which are mostly processing streams and service layer components, consume data by subscribing to the appropriate routing key. For example, processing streams that are only interested in logs published by a specific component define both the specific component name and the type of stream they are interested in, as routing key. On the other hand, more generic streams that can be used for several components only specify the type of data stream as routing key. This approach provides a flexible mechanism that can be easily extended and allows operators of Ahab to choose the suitable data granularity.

8.3.4 Ahab Architecture

The enabling stream processing approach is a cloud-based framework depicted in the center in Figure 8.1. Ahab is separated in two layers. On the bottom, the `Streaming Layer` handles and processes data streams, and is implemented as a Lambda Architecture[1]. On top, the `Service Layer` is handling the underlying streaming layer and takes care of analyzing, managing, and adapting registered components. In the following, we discuss these two layers in more detail, present the main components of Ahab, and describe how they interact with each other.

Streaming Layer

To handle massive quantities of real-time data and also provide batch processing of, e.g., historical data, the streaming layer of Ahab is implemented as a Lambda Architecture. In general, the Lambda Architecture is a generic, scalable, and robust data processing system, specifically designed to serve massive workloads and a wide range of use cases. To balance latency, fault-tolerance, and throughput, the Lambda Architecture combines both batch processing and stream processing capabilities. Lambda Architectures use batch processing to provide accurate and comprehensive views of batch data, while concurrently running stream processing produces views on real-time data.

The streaming layer of Ahab contains the following sub-layers, as proposed by the Lambda Architecture: a `batch layer`, a `speed layer` and a `serving layer`. First, all data that enters Ahab via the messaging system is dispatched to both the batch and speed layer for further processing. The batch layer stores the data in HDFS[2], grouping data originating from the same source. The batch layer then precomputes views by using all available data in HDFS. These views are subsequently used by the serving layer. The serving layer indexes the views produced by the batch layer in order to provide ad-hoc querying. To compensate for the high latency of updates to the serving layer, the speed layer only deals with recent data. The speed layer provides online processing of

[1]http://lambda-architecture.net
[2]https://hadoop.apache.org

data streams, which means that results are available almost immediately after data is received by Ahab. While these results might not be as accurate or complete as views generated by the batch layer, they can be updated as soon as the results of the batch layer for the same data are available. By combining batch and real-time views in the `serving layer`, Ahab can serve massive quantities of incoming data and also cover a broad area of use cases. This approach allows for optimizing components either based on real-time information (e.g., to handle critical situations), or based on batches of historical information, or using a combination of both.

As underlying stream processing engine we employ Apache Spark[3] respectively Apache Spark Streaming running in a Hadoop cluster. Spark provides a unified processing engine and programming model that natively supports processing both stream and batch workloads. Spark discretizes streaming data into micro-batches [142], packages them into small tasks, and assigns them to resources (workers) for processing. In order to deal with large workloads, Spark provides an optimized load balancing and resource usage mechanism, which allows utilizing workers of the cluster more efficiently by dynamically assigning tasks to workers based on locality of data and available resources. In addition, by using small discretized tasks that can run on any resource without affecting correctness, Spark provides fast failure discovery.

Service Layer

On top of the streaming layer, the `Service Layer` manages the underlying streaming layer and registered components based on actionable information produced by streams, policies, and adaptation actions. The design of the service layer follows the microservice architecture approach [87], which enables building a scalable, flexible, and evolvable framework. Especially the flexible management and scaling of services is important for Ahab in order to enable efficient and fast adaptation of components. In the following we introduce the main components of the service layer.

Component Handler To process incoming data streams produced by infrastructure components or to optimize managed infrastructure components with Ahab, operators have to register them using the User API, by providing the following information: (i) name of the component, (ii) an endpoint that defines where the component is reachable, (iii) data streams that get published, and (iv) actions that can be executed to adapt the component. This information is then transformed by the `Component Handler` using the MONINA language and stored in the component repository.

Streaming Manager Since Ahab is not only handling incoming data streams produced by various infrastructure resources (i.e., components), but also allows operators defining custom streams that process and analyze other data streams managed by Ahab, the `Streaming Manager` is responsible for efficiently managing this large number of streams.

[3]http://spark.apache.org

When new components are registered using the User API, the manager generates a new key for each provided stream in the component description. This key is then registered in the stream repository and can be used by other streams to consume this stream via the messaging infrastructure. To register custom streams, Ahab provides two mechanisms for operators. First, simple streams can be defined using the MONINA language and registered via the User API. This definition is then translated by the streaming manager into an actual streaming application that implements the behavior of the stream. Second, Ahab provides a streaming library that allows operators to develop custom streaming applications that implement more complex streams. The developed streaming application is then registered using the User API.

Next, the streaming manager analyzes the streaming application, extracts which streams are produced by the respective application, generates corresponding keys and registers them in the stream repository. Finally, the manager invokes the scheduler to submit the new streaming application to the streaming layer.

Scheduler Since custom streams need to be executed in Ahab's streaming layer, the `Scheduler` is responsible for submitting custom processing streams represented as streaming applications to this layer. When invoked, the scheduler first analyzes which streams are consumed and calculates the needed processing power (e.g., number of cores) for executing this application. Based on the used streams the scheduler decides whether the application should be executed in either the batch or speed layer of the streaming layer. Next, the application is packaged with the required streaming library and additional execution parameters, and finally submitted to the streaming layer where it is executed in the underlying streaming engine.

Policy Enforcer To manage registered components, Ahab uses a policy based approach, where operators can define management policies using the MONINA language and register them via the User API. Furthermore, operators can register specific adaptation actions that should be executed when a policy violation is detected. These policies are then stored in the policy repository and forwarded to the `Policy Enforcer`. For each policy registered in the repository, the policy enforcer creates a policy stream that specifically implements the defined policy. This policy stream is then executed using the scheduler and triggers a notification once it detects that the policy can no longer be met.

Action Executor In order to handle notifications triggered by the policy streams, the `Action Executor` is used. The action executor continuously listens for incoming notifications. When it receives a notification, it analyzes which policies are violated and checks the repositories to find suitable adaptation actions for each affected component. Then, the list of found actions is executed to adapt the component and therefore guarantee that the management policy is met again.

8.4 Evaluation

To evaluate our approach we implemented a smart city demo application based on the scenario identified in Section 8.2. Next, we created a test setup in the cloud using CoreOS[4] to virtualize edge devices of our IoT infrastructure as Docker[5] containers.

In the remainder of this section we give an overview of the developed smart city demo application, discuss the concrete evaluation setup, present different evaluation scenarios, and analyze the gathered results.

8.4.1 Smart City Demo Application

In order to evaluate our approach we developed a demo application that implements the concept of Autonomous Intersection Management[6] (AIM) [43], which presents an essential element in enabling autonomous cars in a smart city environment. To fully utilize the autonomous capabilities of self driving cars to allow the high volume traffic in our presented scenario it is essential to enable intelligent resource management.

One area where the demand for such an intelligent mechanism is especially demanding, are road intersections. Currently, cities use traffic lights and dozens of signs to assist human drivers to safely pass road intersections. However, with the upcoming advent of autonomous cars the AIM project shows that it is vital to adapt modern-day intersection management in future smart cities allowing autonomous vehicles to interact with intelligent traffic control systems to enable more efficient and effective traffic management. Considering the huge numbers of cars in our scenario, we also need a way that allows smart city operators to scale such intelligent control systems by using any kind of available processing power of a smart city infrastructure (e.g., cloud resources or edge infrastructure).

To demonstrate this aspect, we developed a simple traffic control application that handles incoming requests sent from autonomous cars. These incoming requests get processed by the application to determine if a car's intended path is safe to use or not. To deal with the load generated by the autonomous cars, the application can offload the computation to available resources, which allows scaling across infrastructure boundaries. Furthermore, to analyze the application's performance, the application publishes metrics such as request load and response time.

8.4.2 Setup

For the evaluation of our framework, we created an IoT testbed in our private OpenStack[7] cloud. We reuse a Docker image from our recent work [129] to virtualize and mimic a physical edge device in our cloud. To run several of these virtualized devices, we use a CoreOS cluster of 5 virtual machines, where each VM is based on CoreOS 607.0.0 and

[4]https://coreos.com
[5]https://www.docker.com
[6]http://www.cs.utexas.edu/~aim/
[7]https://www.openstack.org

uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB disk space). To simulate a medium scale IoT infrastructure we use 100 virtual edge devices evenly distributed among the CoreOS cluster.

As our messaging infrastructure we use a RabbitMQ[8] cluster consisting of 3 VM nodes using Ubuntu 14.04 and the m1.small flavor (1920MB Ram, 1 VCPUs and 40GB disk space). Since the streaming layer of Ahab is based on Spark Streaming and Hadoop, we use a Hadoop cluster consisting of one master node (Ubuntu 14.04 VM and m1.medium flavor) and 8 worker nodes (Ubuntu 14.04 VM and m1.small flavor). The service layer is hosted on an additional VM using Ubuntu 14.04 and the m1.medium flavor.

Finally, the smart city demo application is deployed on a separate VM using Ubuntu 14.04 and the m1.small flavor.

8.4.3 Scenario 1

In the first experiment we use Ahab to elastically scale the smart city demo application across the available IoT infrastructure by analyzing the stream of metrics published by the application. More specifically, in this scenario Ahab calculates, for each request sent to the application the response time and the number of concurrent requests per second. Next, Ahab analyzes logs published by the IoT infrastructure to detect how many devices are currently used by the application and how many are still available (i.e., idle). In addition to these processing streams, we define a stream that detects changes in the response time of the application by using a sliding window of 4 seconds that gets updated every 2 seconds. In this window the stream calculates the average of the windowed response times and triggers a notification whenever the average changes. Finally, we define a policy that defines the allowed threshold for the average response time increase and an action for scaling up the application by providing additional idle infrastructure devices that can be used. Furthermore, we also define a stream that detects when the current number of requests drops or infrastructure devices are not used anymore. Based on this stream, we define a policy for scaling down the application by releasing infrastructure devices.

Figure 8.2 illustrates the evaluation results for the first scenario. The x-axis shows the temporal course of the evaluation in seconds. In the 'requests per second' section we see that we started the evaluation by sending 10 concurrent requests per second to the application and increase the load stepwise every 30 seconds to see if Ahab can scale up the application. Finally, at 120 seconds we reduce the load to 10 requests per second to see if Ahab is also able to scale down the application. In the 'response time' section we see the response time for each incoming request. The 'devices' section illustrates the number of used edge devices by the demo application. The 'logs' section represents the processed incoming data stream of logs and metrics. Finally, the 'actions' section illustrates when Ahab detects that a policy was violated and triggers a respective action to compensate the violation. We executed this experiment 10 times, where each run produced almost identical results.

[8] http://www.rabbitmq.com

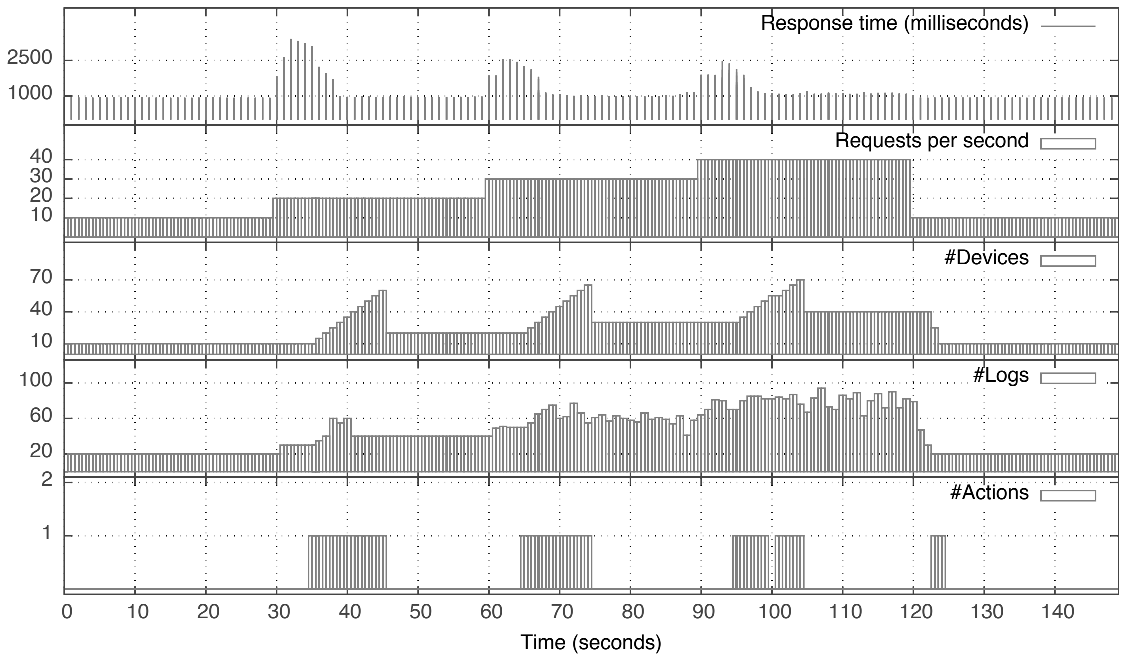


Figure 8.2: Evaluation Results – Scenario 1

For the first interval of requests, we see that the response time of the application is almost constant. At 30 seconds, we doubled the requests per second and see that the response time rises, which indicates that the application is overloaded. At 33 seconds, Ahab detects that the response time increase is too high, initiates a scale up action, allowing the application to use 5 additional idle edge devices for handling the load. Since at this point a lot of requests got queued up by the application, Ahab issues one action per second until 44 seconds, where finally the application can handle the load by using in total 60 devices. After the response time reaches a normal level again and the application is only using a fraction of the devices to handle the requests, Ahab issues another action for scaling down the application and releasing devices. We see that the following 2 increases in requests per second at 60 and 90 seconds were also detected and analogously handled by the framework. Furthermore, for increasing requests and number of used infrastructure devices the incoming stream of logs and metrics that needs to be processed and analyzed by Ahab also increases. Finally, at 120 seconds, we see that the load drops from 40 to 10 requests per second, which is also detected by Ahab, leading to 2 actions for scaling down the application again. In general we see that Ahab is able to elastically scale the application by only using information extracted from published real-time data. Furthermore, considering the amount of data that has to be processed, Ahab is performing well and provides good results.

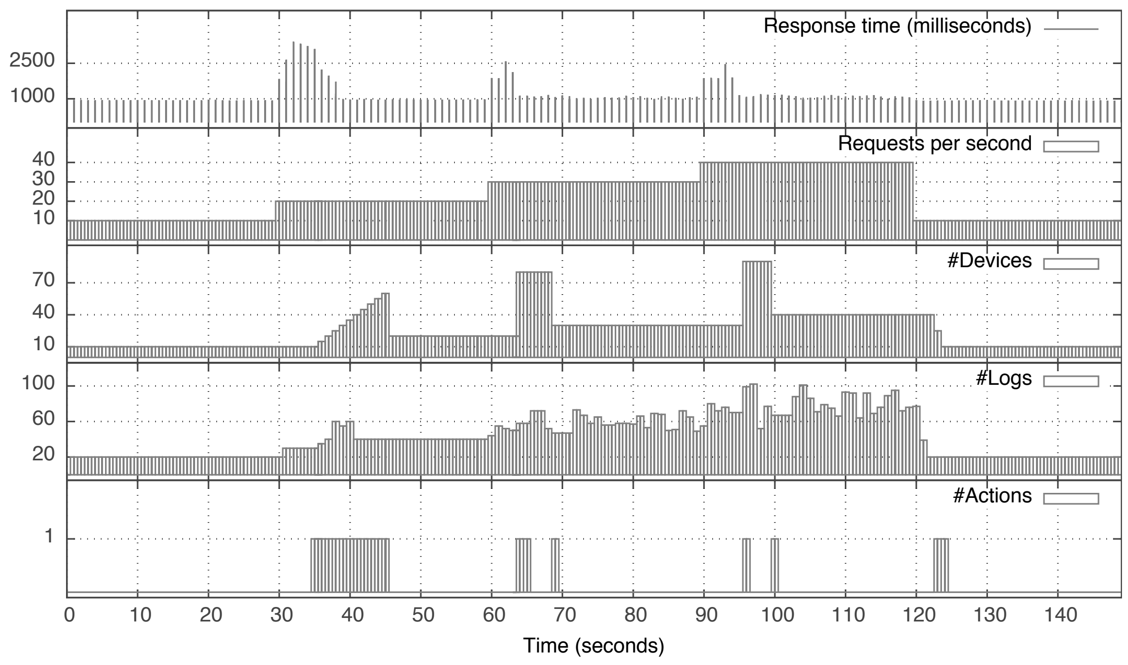


Figure 8.3: Evaluation Results – Scenario 2

8.4.4 Scenario 2

In the second experiment, we once more use Ahab to elastically scale the smart city demo application, but in addition to analyzing and processing real-time data, as done in the first experiment, we now also consider historical data. For that reason we extend the first experiment to not only store incoming data to HDFS, but also save detected changes in the response time and respective actions. We furthermore added a batch processing stream that analyzes how Ahab compensated detected changes in the response time of the demo application in the past by using the historical information stored in HDFS. By also considering extracted information from historical data Ahab should be able to scale the demo application more efficiently when handling policy violations. We also repeated this experiment 10 times and each test run produced almost identical results.

Figure 8.3 illustrates the evaluation results for the second scenario. For the first request increase at 30 seconds, we see the same result as in the first scenario. Ahab scales the application stepwise, using 5 devices per step, until the application is able to handle the load and the response time reaches a normal level. This is done since at this time there is no historical data present that can be used by Ahab. At 44 seconds, Ahab detects that not all assigned devices are used by the application and issues another action to release idle devices. At 60 seconds we see the next request increase and once more the response time of the application rises. However, to compensate this increase, Ahab now uses the data collected from the previous compensation process and scales up the application by using 60 additional idle devices at once. By doing so, we see that the response time increase is compensated faster and that the overall compensation process takes only 5 seconds, compared to the initial one with more than 10 seconds. For the next request increase from 30 to 40 at 90 seconds, we see that Ahab once more uses the historical data to scale up the application. In general we can conclude that by using information from both real-time and historical data, Ahab is able to react faster and more efficiently on changes, which leads to improved performance and less policy violations.

8.5 Summary

Todays smart cities produce an ever growing amount of data. These data streams contain various kinds of information such as monitoring events from underlying infrastructure, metrics published by used toolsets, as well as logs produced by executing smart city applications. This vital information about the smart city environment can be used to adapt and optimize different layers of the infrastructure in order to allow for more efficient execution and management of smart city applications. Therefore, efficient capturing, processing, storage, and analysis of this information is of upmost importance in the smart city domain, but is not sufficiently addressed by current approaches.

In this chapter, we introduced Ahab, a cloud-based, generic, scalable, and fault-tolerant big data analytics framework that allows operators to perform online and offline analyses on the gathered data. Based on this extracted actionable information operators are able to better understand and optimize the behavior of the managed smart city infrastructure. Ahab is designed to easily integrate new data streams and provides extensible APIs to perform tailored analysis tasks. To allow for the flexible definition of adaptation rules, the framework furthermore provides a DSL based on the MONINA language. In order to demonstrate the feasibility of the proposed approach we implemented an example application for autonomous intersection management in smart city environments and showed that our framework is able to autonomously optimize application deployment topologies by distributing processing load over available IoT infrastructure resources when necessary. Our approach allows using both, online analysis of the current state of the environment, as well as patterns learned from historical data, in order to manage and adapt application deployments more efficiently.

CHAPTER 9

Related Work

In this chapter, we discuss the most important related research work in the context of this thesis. This includes work from various areas such as provisioning, deploying, monitoring, and analyzing IoT applications as well as their supporting infrastructure.

9.1 Related Work on Provisioning IoT Deployments

Since currently IoT applications are receiving a lot of attention, we notice that the scale of these applications can vary from embedded services to enterprise applications. In order to address different ways of designing, developing, deploying and managing applications not only in the cloud, but also in the underlying IoT infrastructure, [94] presents initial guiding principles. In addition, [121] presents challenges for building a city-scale IoT framework, where among others the important challenge of fine-grained provisioning and management of resources is discussed. To tackle some of the aforementioned challenges, [13] defines an abstract IoT reference architecture for standardizing the Internet of Things. In addition, [114] addresses general problems of managing resource constrained devices, which are often used for building IoT solutions, by adopting existing network management protocols. Based on general challenges and reference architectures, platforms specifically targeted for deploying and provisioning of IoT applications emerged. INOX [25] is a robust and adaptable Platform for IoT that provides enhanced application deployment capabilities by creating a resource overlay to virtualize the underlying IoT infrastructure. An additional abstraction layer on top of the IoT infrastructure is frequently used in the literature (e.g., [67, 83]), which allows keeping the underlying infrastructure untouched when deploying an IoT solution. In contrast, in this thesis we consider IoT devices as first-class execution environments, which provides more control and better resource utilization. The Smart-M3 platform [60] aims to create a M3 space by deploying agents on IoT devices that interact based on a space-based communication model. Although the authors mention the provisioning of IoT devices, they solely focus on the actual application design. [23] introduces over the air provisioning of IoT devices using Self Certified ECDH authentication technology. Although this approach shares the same general idea, the authors explicitly focus on one specific device and do not provide a general and scalable approach. [96] presents a solution for automatic configuration of IoT devices based on interpretable configurations. Compared to our approach, the authors assume pre-installed application components on IoT devices and only focus on provisioning application-specific configurations.

Configuration management solutions represent another important area of interest, which in general address a similar problem. The most prominent representatives being Chef [93] and Puppet [101]. However, current tools come with the following limitations that make them unsuitable for the IoT domain. First, they are inherently pull based approaches with clients running on the respective machines, making push based hot fixes (e.g. important security updates) impossible. Second, dependency resolution is usually handed off to a distribution package manager, which is not suitable for the strongly resource-constrained environments we are dealing with.

Finally, since our approach provides an optimization for provisioning edge devices in the domain of building management, we also have to consider relevant work in this research topic. Among others, [99] presents an approach that achieves energy related optimizations for buildings by running simulations in the cloud. In addition, [100] proposes a service-oriented platform that allows performing (near) real-time energy optimizations for buildings. While these approaches specifically focus on energy optimizations in buildings at the application level, our approach aims at optimizing deployment topologies on the infrastructure level.

9.2 Related Work on Deploying and Optimizing IoT Applications

In the literature the overall terminology of IoT is well-defined [28, 73]. However, the characterization of IoT applications is not that clear. First, IoT applications can be defined as applications that hide the underlying IoT infrastructure by introducing an abstraction layer [42, 61, 89, 97] and on top of that layer execute business logic in the cloud [69]. Second, there are distributed applications that consist of an enterprise application for managing underlying devices, and simple application parts that reside in components that are deployed in the edge infrastructure and allow for sensing as well as reacting to their environment [26, 143]. Both approaches have in common that devices, which are deployed in the IoT infrastructure, are defined as external dependencies. Hence, these devices are not considered as an integral part when designing and developing an application. In order to address this issue, recent approaches explicitly respect IoT devices as part of the application that require efficient management in order to provide scalable as well as flexible IoT applications [58, 85, 120, 138]. However, none of the approaches discussed so far considers provisioning and deploying parts of the application on resource-constrained devices that provide limited execution environments [114], which would help facilitating this untapped processing power for building robust and adaptable applications.

For the actual deployment of applications, there exists only a limited amount of prior work (e.g., [20, 66, 76, 102, 104]) in the literature that deal with the location-aware placement of cloud application components. In contrast to our approach, these works do not support placing application components on constrained edge infrastructures in order to allow for improving the deployment topology of an application.

Additionally, since our approach also allows for optimizing an application deployment topology, we also have to consider relevant work in this research topic. There is a significant body of work on optimization algorithms for adapting deployments of cloud applications. Among others (e.g., [64, 71, 140]), Emeakaroha et al. [37] present a scheduling heuristic for cloud applications that considers several SLA objectives. The approach provides a mechanism for load balancing the execution of an application across available cloud resources, as well as a feature for automatically leasing additional cloud resources on demand. Frey et al. [38] introduce CDOXplorer, a simulation-based genetic algorithm for optimizing the deployment architecture and corresponding runtime configurations of cloud applications. By applying techniques of the search-based software engineering field, CDOXplorer analyzes the fitness of a simulated set of possible application configurations, which allows for optimizing the overall application. Wada et al. [134] propose an evolutionary deployment optimization for cloud applications. By introducing a multi objective genetic algorithm that provides a set of optimal deployment configurations, the authors are able to optimize the application deployment to satisfy SLAs under conflicting quality of service objectives. In contrast to our work, none of the approaches presented so far, considers application topologies that are deployed on edge devices, and therefore can be seen as supplemental approaches to our notion of IoT deployments.

Next to algorithms, several approaches emerged in the literature that are specifically targeted at adapting application deployments in the cloud. For example, Menasce et al. [79] present Sassy, a framework that enables applications to be self-adaptive and self-optimizing. Based on a self-architecting approach, Sassy provides a near-optimal application deployment by considering both quality of service and functional requirements. Kritz [62] introduces AppScale, a cloud platform that allows deploying and scaling applications over diverse cloud infrastructures. By offering a set of APIs, AppScale provides operators of cloud applications simplified development, easy deployment, and transparent optimization mechanisms. CloudScale [65] is a middleware for building applications that are deployed on and running in the cloud. By using a transparent approach, CloudScale enables the development of cloud applications like regular programs without the need for explicitly dealing with the provisioning of cloud resources. In order to scale applications, CloudScale provides a declarative deployment model that enables operators to define requirements and corresponding policies. Compared to our approach, all these platforms have in common that they transparently adapt the application topology by optimizing the underlying cloud deployment. However, by only focussing on one specific type of infrastructure (i.e., the cloud), these platforms do not provide a generic approach that can also be used for optimizing application deployments on edge infrastructures as proposed in this thesis.

Since in this thesis we also focus on deploying traditional heterogeneous IoT applications with TOSCA, in the following we will present related work in this context. The research and application of TOSCA is still in its infancy. The early works are generally focused on exploring the possibilities of applying TOSCA for various management tasks, thus providing feedbacks to the standardization efforts and gaining experiences for industrial adoption. Wettinger et al. [135] present several concepts that integrate both model-driven cloud management and configuration management. The goal of the overall approach is to combine the advantages of these service management paradigms based on TOSCA. Binz et al. [15] use TOSCA to describe application topologies in a portable and manageable way. Based on this common TOSCA description the authors present an approach that merges two application topologies into one, to save resources by sharing similar components, but preserve the functionality of both applications. Breitenbucher et al. [18] propose an approach that enables the management of composite applications and their deployment on a higher level of abstraction. Furthermore the authors show how high and low level management tasks can be implemented separately and fully automated applied to the respective applications, by facilitating the features of TOSCA. The work proposed in this thesis is well in line with these early academic works on applying TOSCA to various scenarios. We present the first effort of extending the application scope of TOSCA to an even more challenging area—IoT applications.

In this thesis we used Niagara and Sedona for demonstrating our TOSCA-based deployment approach. However, there are also other IoT frameworks that aim at facilitating device integration, protocol normalization and IoT application development. As these frameworks approach IoT infrastructures with different focuses, they are incompatible with each other and more of such frameworks are expected to emerge in the future. IoT-SyS[1] [57] is a gateway concept that integrates various sensor and actuator systems, which can be found in current home and building automation systems. The integration middleware provides a stack of communication protocols for embedded devices based on various standards to support interoperability that gets directly deployed on 6LoWPAN devices. openHAB[2] presents an integration platform that operates on a higher level of abstraction. The architecture is based on an event bus in combination with a publish-subscribe pattern, realized on OSGi. To integrate any kind of device of an IoT infrastructure, abstract items are defined that represent these devices. In addition, bindings are used to bind items to concrete hardware, protocols or interfaces. This concept allows the platform to be vendor-neutral and hardware/protocol-agnostic. Since in IoT Systems most devices use their own proprietary communication stack and interfaces, it is challenging to offer the gathered data in a standardized way. Dawson-Haggerty et al. [29] propose sMAP that tries to overcome this challenge by presenting physical information via RESTful interfaces using a simple JSON schema. This allows consumers to retrieve data, without the need to access the underlying infrastructure and dealing with proprietary formats. Based on sMap, Dawson-Haggerty et al. [30] present BOSS, a distributed system that provides a collection of crucial, common and reusable services that enable the development of portable and robust applications for heterogeneous physical environment.

[1]https://code.google.com/p/iotsys/
[2]http://code.google.com/p/openhab/

All introduced frameworks require considerable efforts to understand their application management process, and tedious manual configurations are a norm. Our work is complementary to the aforementioned frameworks. To be best of our knowledge, this thesis presents the first attempt to address the IoT application deployment problem by using a domain-independent standard to explicitly specify the component topologies and management operations.

9.3 Related Work on Monitoring IoT Applications

With the advent of big data with ever growing data volumes, it is important that applications that deal with this data perform well in order to deliver the intended benefit. Therefore, monitoring of applications is crucial as it enables organizations to analyze and assess the performance of their applications. Ganglia [75] is a monitoring framework for distributed systems. It centrally collects certain metrics, such as CPU usage, memory as well as process information of nodes in a distributed system and allows visualizing collected data. Ganglia is based on a hierarchical design and relies on a multicast-based protocol. In contrast to our approach, Ganglia is strictly bound to a defined list of metrics, which for example does not allow monitoring runtime performance of single process steps of an application. Imamagic et al. [52] present Nagios, an open source solution for monitoring network services in order to detect failures. A service in Nagios can be represented as a host, a network or a service metric (e.g., process runtime). Nagios is built as a distributed system consisting of a server that collects data from sensors by using a plugin. Compared to our approach that allows collecting and analyzing performance measurements, Nagios focuses on states, which means that any performance measurement acquired, must be reduced to a state by defining thresholds for metrics. Additionally, Nagios does not provide any mechanism for monitoring applications that do not measure any performance metrics by default. Di Nitto et al. [32] propose MODAClouds a platform for monitoring that automatically improves quality of service attributes of cloud-based services. The overall approach consists of a monitoring platform, a self-adaption platform, and an execution platform. The monitoring platform gathers and analyzes data, collected by data collectors. Compared to our approach, data collectors do not create measurement data, but collect data that has already been created by some other component or application of the system. Moldovan et al. [81] introduce MELA, an approach for monitoring and analyzing elastic services deployed in the cloud. Based on monitored metrics, the authors focus on determining relationships among performance, cost, and resource usage of a service. In order to do that, the authors propose elasticity relationships of elastic services, which are used for applying analyses techniques. However their approach is also capable of collecting and analyzing monitoring data of services respectively applications, the authors rely on data that is either emitted by the service or the execution environment. In contrast, our approach allows collecting and analyzing measurements from applications that do not provide any built-in monitoring capabilities.

Leitner et al. [64] propose an approach for monitoring high-level performance metrics of cloud applications based on complex event processing. The approach is based on a multi-step event correlation approach, which in combination with a hierarchy of predefined events, allows specifying and monitoring application performance metrics. Although the authors provide a flexible and extensible monitoring approach, they solely focus on measurement data created by applications or infrastructure components. Thus, their approach does not allow to monitor applications that do not provide this feature. Yuen et al. [141] present a scalable network for monitoring distributed applications. The network consists of proxies that collect performance data from applications and then report this data to distributed monitors in order to aggregate the data. To reduce the delay caused by the monitoring approach, the authors introduce a monitoring algorithm called SMon, which continuously adapts the network in real-time. However, this work shares similarities with our approach, it focuses on the actual collection of performance data and does not provide a mechanism to acquire and analyze the gathered results. Frischbier et al. [39] discuss ASIA, an approach for monitoring distributed event-based enterprise systems. ASIA provides a mechanism for effectively monitoring the state of a distributed system by dynamically integrating functionality for monitoring into components of the system at runtime, which is based on aspect-oriented programming. Even though this work is similar to our approach, the authors do not provide a flexible and extensible model for defining measurement data. Funika et al. [40] introduce a system for monitoring performance of distributed applications. By using semantic information about monitored components allows automated guidance in order to fine-tune measurements. This can be used for identifying possible performance flaws faster and enables reacting on certain events more efficiently. In contrast to our approach, this work only considers applications that provide built-in monitoring functionality and additionally does not allow distributing and collection measurement data. Li et al. [72] present Sparkbench, a platform specifically built for evaluating Spark-based stream processing applications. Although Sparkbench shares similarities with our approach, it is targeted for Spark and therefore not applicable for other stream processing platforms.

Next to systems that are specifically built for acquiring monitoring information, most stream processing engines already provide built-in monitoring functionalities. Apache Storm [6] provides special bolts to collect and publish metrics [7]. In essence the predefined metrics are categorized into system metrics (e.g., memory usage) and topology metrics (e.g., topology statistics such as tuples emitted per minute). For adding custom metrics Storm provides an API. However, compared to our approach, the functionality for acquiring custom metrics would require changes in the application code. Apache Spark [4] provides performance data of system components via Metrics [5]. Although Spark provides mechanisms for publishing measured data to various mediums, the monitoring capabilities are limited to engine-specific components of Spark.

9.4 Related Work on Analyzing IoT Deployments

With the advent of the smart city paradigm and its enablers, IoT [11] and the Cloud of Things (CoT) [33], modern cities produce an ever growing amount of data that needs to be handled. To allow cities and governments to facilitate these large data sets, commonly referred to as big data [16, 77], effective management, processing, and analysis is of upmost importance. Recently, big data has received a lot of attention not just from cities and governments, but also from academia and industry. In general, when dealing with big data we can identify the following key challenges [10, 22] relevant for our approach. First, the management of big data, which comprises handling and storing big amounts of data efficiently and effectively. For storing data, Chang et al. propose Bigtable [21], a distributed storage system, which allows for managing structured data. Bigtable is specifically tailored for handling and storing petabytes of data that are distributed across huge clusters of servers. Based on a simple data model that provides clients dynamic control over data layout and format, Bigtable provides a flexible and high-performance solution for managing big data. In addition to Bigtable, there are also various other approaches available for storing big data (e.g., [63] and [31]). Next to storing, also the overall management of data is important. The cloud computing paradigm [3] emerged as a potential candidate that can provide the necessary resources to deal with the immense load needed for handling big data. Sakr et al. [107] and Ranjan et al. [105] present basic goals and challenges for deploying data-intensive applications in the cloud. In addition, Ji et al. [55] provide a comprehensive overview of commonly used approaches for processing big data in the cloud. In addition to general challenges and issues, Xhafa et al. [137] propose an approach for processing big data streams in real time by facilitating the Yahoo!S4 (the simple scalable streaming system). The approach facilitates S4's actor model in order to allow for distributed computing. To demonstrate the feasibility, the authors evaluated their approach by using real time data streams from a global flight monitoring system. Based on the evaluation the authors were able to show that their approach provides reasonably fast results. Compared to our approach, the authors solely concentrate on processing real time data and do not store data for later batch processing to extract additional information. In contrast to classical stream processing systems [1, 136] that use a fixed amount of processing nodes, Heinze et al. [44] present an elastically scalable data stream processing system based on FUGU [45]. By using a model that analyzes and estimates latency spikes generated by scaling the system up and down, the authors propose an elastic latency-aware algorithm for the placement of stream processing operators to minimize SLA violations. Based on an evaluation the authors show that their approach can significantly decrease latency violations by postponing scaling decisions and allows the system to adapt its scaling strategy based on user input. Although this approach shares similarities with our work, our approach adapts managed infrastructure components based on knowledge extracted from processed data streams. Satzger et al. [108] propose ESC, a stream computing platform. In order to adapt to varying computational demands, ESC facilitates the cloud to dynamically attach and release resources. Furthermore, ESC provides a simple programming model based on DAGs that hides underlying aspects such as load distribution from the user. In contrast to our approach, the need for storing data for historical purposes is not addressed in this work.

In addition to processing, analyzing big data is a vital aspect. Hummer et al. [48] present a scalable platform that allows active event-based aggregation of data streams.

The approach provides an active query model and a language to correlate data streams. To handle the immense workload, the platform is designed for distributed query execution and can be deployed in the cloud. In order to provide more accurate system analytics, Chen et al. [24] present a query execution model that can be applied on both static relational data and dynamic streaming data. Based on this execution model the authors propose a system that combines stream processing and database capabilities. Although this approach also considers both real time and historical data, it does not provide a complete and general solution that can be used to adapt infrastructure components. In order to combine processing and storing of big data in the context of IoT, Villari et al. [126] propose AllJoyn Lambda. The authors use AllJoyn, a communication platform for IoT devices, and integrate it with a Lambda Architecture. With this approach the authors provide a scalable solution for processing and storing big data, and furthermore allow real time analytics. While this approach also uses a lambda architecture for managing both real time and batch data, the authors solely focus on processing and storing of data, but do not provide a generic user interface that allows operators of smart city infrastructures to facilitate extracted information.

CHAPTER 10

Conclusions

In this chapter, we summarize the main results of this thesis. In Section 10.1 we discuss the core outcomes of the conducted work and how the state of the art in research was advanced as part of this work. Then, the research questions posited in Section 1.2 are revisited and critically analyzed in Section 10.2. Finally, in Sections 10.3 and 10.4 we present open topics in related research areas for future research and discuss ongoing work that builds on the contributions presented in this work.

10.1 Summary of Contributions

With the emergence of IoT, stakeholders in the smart city domain started to deploy IoT devices that provide capabilities for sensing and manipulating their environment. On top of deployed IoT devices and available smart city infrastructure, IoT applications emerged as powerful tools for stakeholders to provide innovative smart city services for citizens. However, IoT applications have to manage large amounts of data provided by the rapidly growing number of IoT devices in order to fulfill their requirements. Thus, such applications need the ability to fully utilize the complete range of available infrastructure resources in a smart city. Therefore, in this thesis we presented novel approaches that enable efficient operation and management of IoT applications in a smart city ecosystem. By integrating the proposed approaches into a comprehensive middleware toolkit, the contributions of this thesis represent the first steps towards a Smart City Operating System (SCOS) that will serve as a central element in future smart city application ecosystems. SCOS is designed to resemble a modern computer operating system, providing unified abstractions for underlying resources and management tasks, but is specifically tailored to the city scale. This approach enables building IoT applications by only focusing on specific demand, while completely hiding the complexities and problems of operating applications. Figure 10.1 provides an overview of SCOS and highlights its building blocks that can be directly mapped to contributions of this thesis. In the following we will summarize the contributions of this thesis and discuss how they build the foundation of SCOS.

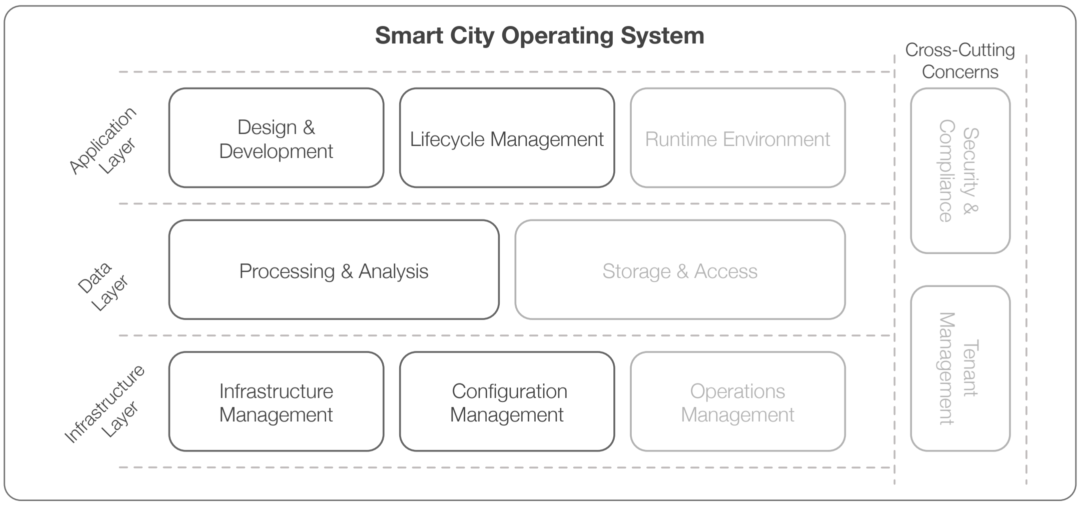


Figure 10.1: Smart City Operating System – Thesis Contributions

In this thesis, we first introduced a methodology that makes IoT devices first class citizens in the design, development, and operation of IoT applications. This allows building more resilient and performant IoT applications by fully leveraging the available capabilities of these resources. Since managing and configuring the underlying infrastructure resources is a vital part of the *Infrastructure Layer* of SCOS, we presented an elastic provisioning approach that allows for addressing the intrinsic heterogeneity of currently available IoT devices by respecting the significant differences in resource capabilities (e.g., available storage and processing resources), as well as deployed and deployable software components. In addition, the presented approach is specifically tailored to the resource-constrained nature of IoT devices in order to enable installing as well as updating software and application capabilities on these devices. Furthermore, to deal with the massive number of connected IoT devices in the smart city, the presented approach is designed to be elastic and scalable in order to deal with these large-scale deployments. Next, the *Application Layer* of SCOS should provide methodologies and tools to support the efficient design, development, and operation of IoT applications. To address this aspect, we introduced a declarative, constrained-based model that describes IoT applications as a set of clearly separated components to enable the independent deployment of application components across infrastructure boundaries. Based on this model, we derived a deployment approach to provision elastic IoT application deployments, whose topology can change over time. By carefully deciding when to deploy certain application components on IoT resources or cloud infrastructure, the approach allows to effectively manage the inherent cost-benefit trade-off of using edge infrastructure, leveraging cheap communication at the infrastructure edge while minimizing expensive (and possibly slow or unreliable) communication to the cloud. Furthermore, an essential part of application operation in the application layer of SCOS is monitoring running IoT applications, which especially in smart cities is challenging since new types of IoT applications emerge frequently that are able to address new and previously untouched areas. Therefore, we introduced a non-intrusive monitoring approach that supports in-depth analysis of data-intensive IoT applications and their inherent distributed structure. The presented monitoring approach allows for detailed examination and comparison of application runtime characteristics, independent of the underlying execution environment. Finally, to provide means for processing and analyzing data in the *Data Layer* of SCOS, we introduced an approach that supports the efficient management of IoT applications by providing a dedicated component for processing, analyzing, and reacting to available infrastructure data. The presented approach allows using the large amounts of performance and health data about the execution environment of IoT applications, which are provided by the underlying infrastructures, to analyze and optimize the overall application execution.

We extensively evaluated the results of our investigations in the context of multiple scenarios and showed that the contributions of this thesis enable the efficient delivery of IoT applications in smart city ecosystems. Furthermore, we showed that enabling IoT applications to fully utilize the full range of available smart city infrastructure resources allows for the creation and operation of more robust, resilient, and performant applications that are able to address the increasingly complex challenges of today's and future smart cities.

10.2 Research Questions Revisited

In Section 1.2 we introduced the research questions that guided the work in this thesis. In this section, we revisit these questions and summarize how they have been addressed within the context of this thesis.

Research Question I:
How can resource-constrained edge devices be seamlessly incorporated in the provisioning process of IoT applications in smart city ecosystems?

We have addressed this question with the contributions in Chapter 4. We introduced LEONORE, an approach to elastically provision application packages on resource-constrained, heterogeneous edge devices in large-scale IoT deployments. LEONORE supports push-based as well as pull-based deployments, and enables the integration of various IoT infrastructure and topology capabilities. In order to reduce produced network traffic between cloud and edge infrastructure and improve the overall scalability of our approach, we introduced the concept of LEONORE local nodes to allow for efficient distributed deployment in these constrained environments.

Research Question II:
How can IoT applications and respective topologies be optimally deployed while specifically considering available infrastructure resources in smart city ecosystems?

We have addressed this question with the contributions in Chapters 5 and 6. First, we introduced DIANE, an approach that allows for dynamic generation of deployment topologies for IoT applications that are specifically optimized to the currently available computational infrastructure. By facilitating a declarative, constraint-based model of the intended application deployment, our approach enables flexible provisioning of application topologies on both, edge devices deployed in the IoT infrastructure and cloud resources. Furthermore, to allow applications to autonomously react to environmental changes such as changing request patterns, we extended DIANE to provide an optimization approach for evolving application deployment topologies. Second, since in the smart city domain we also have to deal with heterogenous IoT applications that follow traditional application design paradigms, we presented an approach that facilitates TOSCA to formally describe both the components and deployment topology of such applications, as well as the required deployment process on the physical devices.

Research Question III:
How can running IoT applications and utilized infrastructure resources be generically analyzed in order to optimize the overall IoT application deployment topology?

We have addressed this question with the contributions in Chapters 7 and 8. First, we introduced MOSAIC, an approach to add functionality for acquiring and publishing of performance measurements to data-intensive IoT applications, with a specific focus on stream processing applications. Our approach supports integrating different underlying execution environments for deploying and executing applications, a generic domain model for storing and publishing measurements, and a mechanism for gathering and analyzing these measurements. Second, to analyze the plethora of available performance data from IoT applications and the used infrastructure, we introduced Ahab, a big data analytics approach that supports performing online and offline analyses. The extracted actionable information enables operators to better understand and optimize the behavior of managed infrastructure. The presented approach is designed to easily integrate new data streams, provides extensible APIs to perform tailored analysis tasks, and provides a DSL based on the MONINA language that allows defining flexible adaptation rules for evolving IoT deployment topologies.

10.3 Future Work

In this thesis we presented different aspects for enabling efficient operation and management of IoT applications in smart city environments. In the following, we outline some remaining challenges and possibilities for future research.

- With respect to the provisioning approach presented in Chapter 4, we see the necessity to improve our IoT device representation to better utilize the underlying device-specific capabilities. Additionally, we want to introduce update priorities, in order to allow for clear distinctions between ordinary and important (e.g., security patches) updates, since delays in these important updates can expose the infrastructure to severe security risks. Next, to allow our provisioning approach to scale across privately managed infrastructure boundaries, we aim to address security aspects such as authentication and authorization, as these were out of scope for this thesis.

- Based on the introduced orthogonal IoT application deployment approaches in Chapter 5 and 6, we plan to further adapt our methodologies to allow for more detailed descriptions of application topologies and enable local coordination of topology changes among edge resources.

- To see possible limitations and investigate how the monitoring approach discussed in Chapter 7 can be further improved, we plan to integrate additional execution environments (e.g., [108]). We further plan to incorporate more sophisticated statistical methods and visualization techniques to provide deeper insights and help drawing better conclusions. Furthermore, based on the analysis of performance measurements, resource planning and stochastic models can be derived to evaluate application behavior under uncertain load. Since many organizations use monitoring tools (e.g., Ganglia [75], Nagios [52], and Splunk [119]) for monitoring applications and IT systems, we plan to provide appropriate interfaces to support these tools and ease integration with our approach. As the performance of IoT applications also depends on the network performance, we see the necessity to also consider measuring network performance. This network-specific measurement data would allow for deeper analysis in general and support root-cause analysis in the case of performance issues.

- Finally, to address additional challenges in the smart city domain in the context of the introduced analysis and optimization approach in Chapter 8, we will investigate whether unsupervised machine learning techniques are suitable for autonomous improvements of IoT application deployments. Additionally, we plan to extend our approach to address privacy and security requirements (e.g., [74, 98]) when dealing with big data.

10.4 Ongoing Work

In Section 10.1 we presented the contributions of this thesis and discussed how they build the foundations of SCOS. However, since the scope of SCOS exceeds the frame of this thesis, in this section we outline our ongoing work towards establishing SCOS as a central element in future smart city application ecosystems.

10.4.1 Towards a Smart City Operating System

In the following, we present for each layer of SCOS the missing building blocks that we did not tackle in this thesis and discuss their functionality. Figure 10.2 shows an overview of SCOS and highlights the elements that we will introduce in the following.

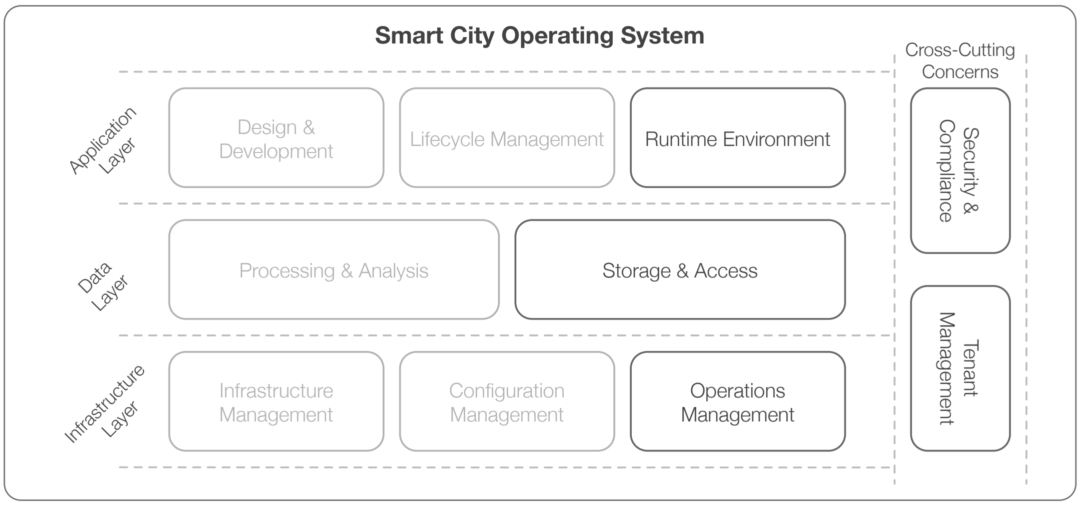


Figure 10.2: Smart City Operating System – Ongoing Work

Infrastructure Layer

The *Infrastructure Layer* manages the underlying infrastructure resources, configures and provisions them, and constantly monitors these resources. In this thesis we already introduced approaches for managing and configuring the underlying infrastructure resources, but did not address the fact that in order to efficiently manage these resources, also mechanisms are required that allow for monitoring and analyzing them. Thus, in the following we introduce this missing building block and discuss its features.

Operations Management Based on the *Infrastructure Management* and *Configuration Management*, SCOS needs a mechanism that enables monitoring and analyzing the performance of the underlying infrastructure resources. Therefore, *Operations Management* supports the constant monitoring and collection of information from connected resources, by using available monitoring capabilities of the respective infrastructure (e.g., cloud monitoring APIs[1], commonly applied monitoring tools like Ganglia [75]), or by provisioning software capabilities that allow gathering performance measurements (e.g., tailored profilers for edge devices [131]). In addition to monitoring, operations management also provides mechanisms to manage gathered logs, events, and faults. Based on collected information from the underlying infrastructure resources, operations management is able to conduct performance analyses that can be used for optimizing resource utilization or evolve the overall infrastructure deployment. Furthermore, fine-grained analysis information can be used by other subsystems of SCOS to adapt application topologies in order to react to defined requirements like SLAs. Finally, operations management provides APIs that allow operators of SCOS to define custom adaptation routines (e.g., scaling algorithms) to guarantee a defined availability for infrastructure resources or deal with network outages [53].

Data Layer

The *Data Layer* is responsible for on the one hand storing and providing access for data that is residing in our ecosystem, and on the other hand processing and analyzing data that can be further used by other layers of SCOS in order to generate valuable insights. In this thesis we already introduced an approach for processing and analyzing available infrastructure data. However, we did not provide mechanisms for storing and accessing data in SCOS. In the following we will introduce and discuss this missing building block.

Storage & Access Since in modern smart cities running applications, infrastructure resources, and citizens produce an ever growing amount of data, which is commonly referred to as big data [16], the data layer provides the *Storage & Access* subsystem for managing and handling data. It allows stakeholders of SCOS to store and consume large sets of diverse data by providing generic and extendable APIs. For efficiently managing data in SCOS, the subsystem allows considering the plethora of available data formats, the intrinsic diversity of data, and also enables respecting potentially noisy data [120] that is produced by the underlying infrastructure with its millions of managed resources. Additionally, the subsystem supports the ability for handling both, static data that is not frequently accessed or processed, as well as dynamic data that is constantly and relentlessly changing. To address the challenges that emerge from handling these different types of data, the storage & access subsystem provides a flexible approach that supports various storage facilities like traditional relational databases, document-oriented, and complex unstructured data stores. Providing data storages in a Data as a Service (DaaS) fashion enables the seamless integration of data facilities into SCOS, eases the integration of new data storages, and also allows for easy and uniform data access as well as storage functionality for components inside and applications on top of SCOS. Furthermore, the storage & access subsystem provides mechanisms for merging and combining different types of data, which can be used as foundation for analysis and planning operations in

[1]e.g., https://cloud.google.com/monitoring/api/

upper layers. Finally, since the ownership of data is an important concern in SCOS, the storage & access subsystem incorporates novel concepts that protect data, but also allow open data exchange where different levels of data owners can share data, by integrating the principle of a Hub of all Things[2]. Following this approach enables the clear concept of ownership and allows addressing emerging data compliance and security requirements.

Application Layer

The *Application Layer* provides a comprehensive set of methodologies and tools for efficient design, development, distribution, and operation of applications. Since in this thesis we already provided the building blocks for design and development, as well as lifecycle management, we will concentrate in the following on the missing building block that provides a transparent runtime environment for applications.

Runtime Environment To allow for seamless execution of applications, the SCOS *Runtime Environment* provides a configurable and adaptive execution environment for cloud-based applications that is independent of the underlying physical infrastructure. The execution environment incorporates a pluggable, unifying infrastructure abstraction [110] to transparently support and manage multiple application deployment mechanisms, such as container-based deployments (e.g., Docker[3]) and virtual machine-based deployments that are provisioned using predominant cloud offerings (e.g., OpenStack[4] or Amazon EC2[5]). The runtime environment furthermore provides a service mobility mechanism [111] that allows for seamless migration of application components between data centers and stakeholder premises. By moving processing logic closer to data sources and/or data sinks, network overhead and associated costs can be reduced. Additionally, component migration allows for the execution of applications that could otherwise not be executed due to compliance constraints.

Cross-Cutting Concerns

An important layer of SCOS that we considered out of scope of this thesis, comprises the cross-cutting concerns. Components or applications of SCOS require common functionality (e.g., authentication) that span across several layers. Since such functionality is affecting the overall system, it is centralized in one place in order to avoid updating components throughout the system in case a certain behavior (e.g., logging) has to be changed.

Tenant Management Applications in a smart city ecosystem operate under complex compliance and security regulations. Furthermore, since these applications have to operate at large scale, are maintained by varying stakeholders, and provided in different possible

[2]http://hubofallthings.com/
[3]https://docker.com
[4]https://openstack.org
[5]https://aws.amazon.com/ec2

facets, a plethora of constraints need to be efficiently managed. Therefore, the *Tenant Management* subsystem supports the magnitude of participating stakeholders and allows them to specify their own security and compliance guidelines. Next, in order to allow large-scale interactions of stakeholders in a smart city environment, tenant management supports a flexible interaction approach that allows expressing specific constraints that need to be respected. For example, considering an interaction among stakeholders in this context, constraints that are valid for two stakeholder having direct interaction can become invalid if another stakeholder joins the interaction, which is triggered by the complex data regulations in such environments. Another important aspect of tenant management is enabling the clean separation and isolation of any type of data, but especially for sensitive data. Thus, tenant management enables each stakeholder of SCOS to clearly define the following constraints regarding its data. First, tenants specify which data they provide and in which quality. Second, tenants can decide which data can be shared or consumed. Third, tenants can describe with whom they want to share data, or who is specifically allowed to consume provided data. Finally, tenants can specify which data and from whom they want to consume data. Based on this specification, the tenant management subsystem derives a constraint matrix that clearly regulates data exchange in SCOS, which avoids undesirable data transfer by respecting various forms of interactions (e.g., direct or transitive). Nevertheless, this approach still empowers novel mechanisms that allow data processing in highly constrained interaction scenarios by using capability migration [111]. In addition to data concerns, tenant management also manages a consolidated view on resources that are consumed by and available to tenants of SCOS. Based on the underlying transparent infrastructure layer, tenant management not only uniformly provides cloud resources like virtual machines, but also offers other forms of infrastructure resources such as IoT devices. This enables tenants and their respective applications to lease and release resources following a consistent, but flexible and extensible model.

Security & Compliance Stakeholders in smart city environments implicitly expect and demand services to be secure, as well as to preserve their privacy. Thus, SCOS provides a *Security & Compliance* subsystem that allows addressing both, basic and complex security aspects. First, since data in SCOS is constantly flowing among different components or applications, which can reside inside or on top of SCOS, the security & compliance subsystem provides mechanisms to protect data in transit by using strong encryption mechanisms. In addition, components of SCOS that are dealing with sensitive data are also provided with approaches for securely storing this data. Second, since SCOS must be able to deal with a broad variety of stakeholders and users, the security & compliance subsystem provides capabilities that facilitate strong authentication mechanisms (e.g., biometric and multi-factor authentication) that can be used by components of SCOS to clearly specify who can access a specific service. Third, next to authentication, SCOS also provides authorization capabilities that allow enforcing permissions before accessing applications or manipulating data. Fourth, in order to allow operators to manage SCOS more efficiently, the security & compliance subsystem provides auditing and logging functionality on component level. Fifth, in order to keep the overall stack of components in SCOS secure, the security & compliance subsystem provides configuration management for automatically delivering software and security updates for different layers of SCOS. Sixth, since applications in the smart city domain need the ability to adapt to users, they come with various configuration alternatives that depend on the user's preferences. Thus, it is important for SCOS to allow applications to collect user-specific data in order to characterize a specific user. However, this form of characterization, which comprises both behavior and preferences, represents a possible threat for users. Therefore, the security & compliance subsystem provides mechanisms that explicitly assure and preserve the user's privacy. In addition to these common security capabilities, the security & compliance subsystem also deals with security requirements that emerge from the underlying infrastructure layer. Given the large number of resources that are available, SCOS provides security management that is able to deal with the intrinsic scalability requirements. Next, since especially IoT resources embody a vital aspect not only in enterprise systems, but also in consumer solutions, the security & compliance subsystem enables flexible security models. Based on these models, SCOS can adapt to and respect emerging complex security requirements from the various domains it is operating in.

10.4.2 Summary

With the rapid adoption of the smart city paradigm in cities around the globe and its respective success, more and more capabilities of modern cities are provided as applications. This fact, in combination with the plethora of supported ecosystems, diversity of stakeholders operating in this smart city ecosystem, and the magnitude of potential users, generates various challenges that need to be respected in order to build and provide truly future-proof smart city applications. Therefore, based on the contributions of this thesis, we presented our ongoing work towards a Smart City Operating System (SCOS). By resembling a modern computer operating system that is specifically tailored to the scale of modern cities, SCOS represents a key element for supporting ongoing smart city application engineering as well as the foundation for enabling the future Internet of Cities.

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B Zdonik. The Design of the Borealis Stream Processing Engine. In *Cidr*, volume 5, pages 277–289, 2005.

[2] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: new wine or just new bottles? *Proceedings of the VLDB Endowment*, 3(1-2): 1647–1648, September 2010. ISSN 2150-8097. doi:10.14778/1920841.1921063.

[3] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing. In *Proceedings of the 14th International Conference on Extending Database Technology - EDBT/ICDT '11*, pages 530–533. ACM, March 2011. ISBN 9781450305280. doi:10.1145/1951365.1951432.

[4] Apache Spark. http://spark.apache.org.

[5] Apache Spark Monitoring. http://spark.apache.org/docs/latest/monitoring.html.

[6] Apache Storm. http://storm.apache.org/.

[7] Apache Storm Metrics. https://storm.apache.org/documentation/Metrics.html.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010. ISSN 0001-0782. doi:10.1145/1721654.1721672.

[9] AspectJ. https://eclipse.org/aspectj/.

[10] Marcos D. Assunção, Rodrigo N. Calheiros, Silvia Bianchi, Marco A.S. Netto, and Rajkumar Buyya. Big Data Computing and Clouds : Trends and Future Directions. *Journal of Parallel and Distributed Computing*, 79:1–44, August 2014. ISSN 07437315. doi:10.1016/j.jpdc.2014.08.003.

[11] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010. doi:10.1016/j.comnet.2010.05.010.

[12] Michael Batty, Kay W Axhausen, Fosca Giannotti, Alexei Pozdnoukhov, Armando Bazzani, Monica Wachowicz, Georgios Ouzounis, and Yuval Portugali. Smart cities of the future. *The European Physical Journal Special Topics*, 214(1):481–518, 2012.

[13] Martin Bauer, Mathieu Boussard, Nicola Bui, Jourik De Loof, Carsten Magerkurth, Stefan Meissner, Andreas Nettsträter, Julinda Stefa, Matthias Thoma, and JoachimW Walewski. IoT Reference Architecture. In *Enabling Things to Talk*, pages 163–211–211. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40403-0. doi:10.1007/978-3-642-40403-0_8.

[14] Tobias Binz, Gerd Breiter, Frank Leyman, and Thomas Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(3):80–85, March 2012. doi:10.1109/MIC.2012.43.

[15] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, Frank Leymann, and Andreas Weiß. Improve Resource-Sharing through Functionality-Preserving Merge of Cloud Application Topologies. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*, pages 96–103, Aachen, Germany, 2013. SciTePress. ISBN 978-989-8565-52-5. doi:10.5220/0004378000960103.

[16] Christian Bizer, Peter Boncz, Michael L. Brodie, and Orri Erling. The meaningful use of big data. *ACM SIGMOD Record*, 40(4):56–60, January 2012. ISSN 01635808. doi:10.1145/2094114.2094129.

[17] Robert Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. NIST cloud computing reference architecture. In *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, pages 594–596, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/SERVICES.2011.105.

[18] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, and Frank Leymann. Pattern-based Runtime Management of Composite Cloud Applications. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*, pages 475–482, Aachen, Germany, 2013. SciTePress. ISBN 978-989-8565-52-5. doi:10.5220/0004376104750482.

[19] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599—616, 2009. doi:10.1016/j.future.2008.12.001.

[20] Rajkumar Buyya, Rodrigo N Calheiros, and Xiaorong Li. Autonomic Cloud computing: Open challenges and architectural elements. In *Proceedings of the 3rd International Conference on Emerging Applications of Information Technology*, EAIT'12, pages 3–10, 2012. doi:10.1109/EAIT.2012.6407847.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A

Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008. ISSN 07342071. doi:10.1145/1365815.1365816.

[22] Surajit Chaudhuri. What next? A Half-Dozen Data Management Research Goals for Big Data and the Cloud. In *Proceedings of the 31st symposium on Principles of Database Systems, PODS 2012*, pages 1–4, New York, New York, USA, 2012. ACM. ISBN 9781450312486. doi:10.1145/2213556.2213558.

[23] Deji Chen, Mark Nixon, Thomas Lin, Song Han, Xiuming Zhu, Aloysius Mok, Roger Xu, Julia Deng, and An Liu. Over the air provisioning of industrial wireless devices using elliptic curve cryptography. In *Proceedings of the International Conference on Computer Science and Automation Engineering*, CSAE'11, pages 594–600, 2011. doi:10.1109/CSAE.2011.5952541.

[24] Qiming Chen and Meichun Hsu. Cut-and-Rewind: Extending Query Engine for Continuous Stream Analytics. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXI*, volume 9260 of *LNCS*, pages 94–114. Springer, 2015. doi:10.1007/978-3-662-47804-2_5.

[25] Stuart Clayman and Alex Galis. INOX: A Managed Service Platform for Interconnected Smart Objects. In *Proceedings of the Workshop on Internet of Things and Service Platforms*, IoTSP 2011, pages 2:1–2:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1043-7. doi:10.1145/2079353.2079355.

[26] Walter Colitti, Kris Steenhaut, Niccolo De Caro, Bogdan Buta, and Virgil Dobrota. REST Enabled Wireless Sensor Networks for Seamless Integration with Web Applications. In *Proceedings of the 8th IEEE International Conference on Mobile Adhoc and Sensor Systems*, MASS 2011, pages 867–872, Oct 2011. doi:10.1109/MASS.2011.102.

[27] Jonathan Cook, Darrell Smith, and Alan Meier. Coordinating Fault Detection, Alarm Management, and Energy Efficiency in a Large Corporate Campus. In *Proceedings of ACEEE Summer Study on Energy Efficiency in Buildings*, pages 83–93, 2012.

[28] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, 2014. doi:10.1109/TII.2014.2300753.

[29] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. sMAP: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys 2010*, page 197, New York, New York, USA, November 2010. ACM Press. ISBN 9781450303446. doi:10.1145/1869983.1870003.

[30] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. BOSS: building operating system services.

In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 443–458. USENIX Association, April 2013.

[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205, October 2007. ISSN 01635980. doi:10.1145/1323293.1294281.

[32] Elisabetta Di Nitto, Marcos Aurelio Almeida da Silva, Danilo Ardagna, Giuliano Casale, Ciprian Dorin Craciun, Nicolas Ferry, Victor Muntes, and Arnor Solberg. Supporting the Development and Operation of Multi-cloud Applications: The MODAClouds Approach. In *Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC 2013, pages 417–423. IEEE, 2013. ISBN 978-1-4799-3036-4. doi:10.1109/SYNASC.2013.61.

[33] Salvatore Distefano, Giovanni Merlino, and Antonio Puliafito. Enabling the Cloud of Things. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 858–863. IEEE, July 2012. ISBN 978-1-4673-1328-5. doi:10.1109/IMIS.2012.61.

[34] Dropwizard Metrics. http://metrics.dropwizard.io/.

[35] Schahram Dustdar, Yike Guo, Rui Han, Benjamin Satzger, and Hong-Linh Truong. Programming Directives for Elastic Computing. *IEEE Internet Computing*, 16(6): 72–77, 2012. doi:10.1109/MIC.2012.99.

[36] Schahram Dustdar, Fei Li, Hong-Linh Truong, Sanjin Sehic, Stefan Nastic, Soheil Qanbari, Michael Vögler, and Markus Claeßens. Green software services: From requirements to business models. In *Proceedings of the 2nd International Workshop on Green and Sustainable Software*, GREENS 2013, pages 1–7, May 2013. doi:10.1109/GREENS.2013.6606415.

[37] Vincent C Emeakaroha, Ivona Brandic, Michael Maurer, and Ivan Breskovic. SLA-Aware Application Deployment and Resource Allocation in Clouds. In *Proceedings of the 35th Annual IEEE Computer Software and Applications Conference Workshops*, COMPSACW'11, pages 298–303. IEEE, 2011. ISBN 978-1-4577-0980-7. doi:10.1109/COMPSACW.2011.97.

[38] Soren Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 512–521, 2013. ISBN 9781467330763. doi:10.1109/ICSE.2013.6606597.

[39] Sebastian Frischbier, Erman Turan, Michael Gesmann, Allesandro Margara, David Eyers, Patrick Eugster, Peter Pietzuch, and Alejandro Buchmann. Effective runtime monitoring of distributed event-based enterprise systems with ASIA. In *Proceedings*

of the 7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, pages 41–48. IEEE, 2014. ISBN 978-1-4799-6833-6. doi:10.1109/SOCA.2014.25.

[40] Wlodzimierz Funika, Piotr Godowski, Piotr Pegiel, and Dariusz Król. Semantic-Oriented Performance Monitoring of Distributed Applications. *Computing and Informatics*, 31(2):427–446, 2012.

[41] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services. *IEEE Transactions on Services Computing*, 3(3):223–235, July 2010. doi:10.1109/TSC.2010.3.

[42] Dominique Guinard, Iulia Ion, and Simon Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers Perspective. In *Proceedings of the 8th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services, MobiQuitous 2011*, volume 104 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 326–337. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30972-4. doi:10.1007/978-3-642-30973-1_32.

[43] Matthew Hausknecht, Tsz-Chiu Au, and Peter Stone. Autonomous Intersection Management: Multi-intersection optimization. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4581–4586. IEEE, September 2011. ISBN 978-1-61284-456-5. doi:10.1109/IROS.2011.6094668.

[44] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS 2014*, pages 13–22, New York, New York, USA, May 2014. ACM. ISBN 9781450327374. doi:10.1145/2611286.2611294.

[45] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshops, ICDEW 2014*, pages 296–302. IEEE, 2014. doi:10.1109/ICDEW.2014.6818344.

[46] Robert G Hollands. Will the real smart city please stand up? Intelligent, progressive or entrepreneurial? *City*, 12(3):303–320, 2008.

[47] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 1st edition, 2010.

[48] Waldemar Hummer, Benjamin Satzger, Philipp Leitner, Christian Inzinger, and Schahram Dustdar. Distributed continuous queries over Web service event streams. In *Proceedings of the 2011 7th International Conference on Next Generation Web*

Services Practices, NWeSP 2011, pages 176–181. IEEE, 2011. ISBN 9781457711268. doi:10.1109/NWeSP.2011.6088173.

[49] Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013. ISSN 1942-4795. doi:10.1002/widm.1100.

[50] Michael Hüttermann. *DevOps for Developers*. Apress, 2012. ISBN 978-1430245698. doi:10.1007/978-1-4302-4570-4.

[51] IETF. Constrained Application Protocol (CoAP), 2015. http://tools.ietf.org/html/draft-ietf-core-coap-08.

[52] Emir Imamagic and Dobrisa Dobrenic. Grid Infrastructure Monitoring System based on Nagios. In *Proceedings of the 2007 Workshop on Grid monitoring*, pages 23–28. ACM, 2007. ISBN 9781595937162. doi:10.1145/1272680.1272685.

[53] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Generic Event-Based Monitoring and Adaptation Methodology for Heterogeneous Distributed Systems. *Software: Practice and Experience*, 44(7): 805–822, July 2014. doi:10.1002/spe.2254.

[54] Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. MADCAT - A Methodology for Architecture and Deployment of Cloud Application Topologies. In *Proceedings of the 8th International Symposium on Service-Oriented System Engineering*, pages 13–22. IEEE, 2014. doi:10.1109/SOSE.2014.9.

[55] Changqing Ji, Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. Big Data Processing in Cloud Computing Environments. In *Proceedings of the 12th International Symposium on Pervasive Systems, Algorithms and Networks, ISPAN 2012*, pages 17–23. IEEE, 2012. ISBN 978-1-4673-5064-8. doi:10.1109/I-SPAN.2012.9.

[56] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 471–480, New York, NY, USA, 2010. ACM. doi:10.1145/1772690.1772739.

[57] Markus Jung, Jurgen Weidinger, Wolfgang Kastner, and Alex Olivieri. Building Automation and Smart Cities: An Integration Approach Based on a Service-Oriented Architecture. In *Proceedings of the 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 1361–1367. IEEE, March 2013.

[58] Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, and Shahid Khan. Future internet: The internet of things architecture, possible applications and key challenges. In *Proceedings of the 10th International Conference on Frontiers*

of Information Technology, FIT'12, pages 257–260, 2012. ISBN 9780769549279. doi:10.1109/FIT.2012.53.

[59] Rob Kitchin. The real-time city? Big data and smart urbanism. *GeoJournal*, 79 (1):1–14, 2014. doi:10.1007/s10708-013-9516-8.

[60] Dmitry G Korzun, Sergey I Balandin, and Andrei V Gurtov. Deployment of Smart Spaces in Internet of Things: Overview of the Design Challenges. In *Lecture Notes in Computer Science*, pages 48–59–59. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40316-3. doi:10.1007/978-3-642-40316-3_5.

[61] Matthias Kovatsch. Firm firmware and apps for the Internet of Things. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, SESENA '11, pages 61–62, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0583-9. acmid:1988064.

[62] Chandra Krintz. The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment. *IEEE Internet Computing*, 17(2):72–75, 2013. doi:10.1109/MIC.2013.38.

[63] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35, April 2010. ISSN 01635980. doi:10.1145/1773912.1773922.

[64] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *Proceedings of the 5th International Conference on Cloud Computing*, CLOUD'12, pages 213–220. IEEE, 2012. ISBN 978-1-4673-2892-0. doi:10.1109/CLOUD.2012.21.

[65] Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. CloudScale - a Novel Middleware for Building Transparently Scaling Cloud Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC'12, pages 434–440. ACM, 2012. ISBN 9781450308571. doi:10.1145/2245276.2245360.

[66] Fei Li, Schahram Dustdar, Jakob Bardram, Martin Serrano, Manfred Hauswirth, Vasilios Andrikopoulos, and Frank Leymann. Eupaas - elastic ubiquitous platform as a service for large-scale ubiquitous applications. In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science*, CLOSER'13, pages 309–314. SciTePress, 2013.

[67] Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Efficient and Scalable IoT Service Delivery on Cloud. In *Proceedings of the 6th International Conference on Cloud Computing*, CLOUD'13, pages 740–747, 2013. doi:10.1109/CLOUD.2013.64.

[68] Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Towards Automated IoT Application Deployment by a Cloud-Based Approach. In *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications*, SOCA'13, pages 61–68, 2013. doi:10.1109/SOCA.2013.12.

[69] Fei Li, Michael Vögler, Sanjn Sehic, Soheil Qanbari, Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. Web-Scale Service Delivery for Smart Cities. *IEEE Internet Computing*, 17(4):78–83, 2013. doi:10.1109/MIC.2013.79.

[70] Fei Li, Soheil Qanbari, Michael Vögler, and Schahram Dustdar. *Green in Software Engineering*, chapter Constructing Green Software Services: From Service Models to Cloud-Based Architecture, pages 83–104. Springer International Publishing, 2015. doi:10.1007/978-3-319-08581-4_4.

[71] Jim Zw Li, Murray Woodside, John Chinneck, and Marin Litoiu. CloudOpt: Multi-goal Optimization of Application Deployments Across a Cloud. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 162–170, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing. ISBN 978-3-901882-44-9. acmid:2147697.

[72] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 53:1–53:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3358-0. doi:10.1145/2742854.2747283.

[73] Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information Systems Frontiers*, pages 1–17, April 2014. doi:10.1007/s10796-014-9492-7.

[74] Chang Liu, Rajiv Ranjan, Xuyun Zhang, Chi Yang, and Jinjun Chen. A Big Picture of Integrity Verification of Big Data in Cloud Computing. In *Handbook on Data Centers*, pages 631–645. Springer New York, 2015. ISBN 978-1-4939-2091-4. doi:10.1007/978-1-4939-2092-1_21.

[75] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004. ISSN 01678191. doi:10.1016/j.parco.2004.04.001.

[76] Philip Mayer, José Velasco, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Tomáš Bureš. The Autonomic Cloud. In *Software Engineering for Collective Autonomic Systems*, pages 495–512. Springer, 2015. ISBN 978-3-319-16310-9. doi:10.1007/978-3-319-16310-9_16.

[77] Viktor Mayer-Schönberger and Kenneth Cukier. *Big Data: A Revolution that Will Transform how We Live, Work, and Think.* Houghton Mifflin Harcourt, 2013. ISBN 0544002695.

[78] Peter Mell and Timothy Grance. The NIST definition of cloud computing. *NIST Special Publication*, 800-145, 2011.

[79] Daniel a. Menascé, Hassan Gomaa, Sam Malek, and João P. Sousa. Sassy: A framework for self-architecting service-oriented systems. *IEEE Software*, 28(6): 78–85, 2011. ISSN 07407459. doi:10.1109/MS.2011.22.

[80] 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, September 2012. doi:10.1016/j.adhoc.2012.02.016.

[81] Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. MELA: Monitoring and Analyzing Elasticity of Cloud Services. In *Proceedings of the 5th International Conference on Cloud Computing Technology and Science*, CloudCom'13, pages 80–87, 2013. ISBN 978-0-7695-5095-4. doi:10.1109/CloudCom.2013.18.

[82] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., feb 2009. ISBN 0596520662, 9780596520663.

[83] Sean Murphy, Abdelhamid Nafaa, and Jacek Serafinski. Advanced service delivery to the Connected Car. In *Proceedings of the 9th International Conference on Wireless and Mobile Computing, Networking and Communications*, pages 147–153, 2013. doi:10.1109/WiMOB.2013.6673354.

[84] Taewoo Nam and Theresa A. Pardo. Conceptualizing smart city with dimensions of technology, people, and institutions. In *Proceedings of the 12th Annual International Digital Government Research Conference: Digital Government Innovation in Challenging Times*, dg.o '11, pages 282–291, New York, NY, USA, 2011. ACM. doi:10.1145/2037556.2037602.

[85] Stefan Nastic, Sanjin Sehic, Michael Vögler, Hong-Linh Truong, and Schahram Dustdar. PatRICIA – A Novel Programming Model for IoT Applications on Cloud Platforms. In *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications*, pages 53–60, 2013. doi:10.1109/SOCA.2013.48.

[86] Stefan Nastic, Michael Vögler, Christian Inzinger, Hong-Linh Truong, and Schahram Dustdar. rtGovOps: A Runtime Framework for Governance in Large-Scale Software-Defined IoT Cloud Systems. In *Proceedings of the 3rd International Conference on Mobile Cloud Computing, Services, and Engineering*, MobileCloud'15, pages 24–33, March 2015. doi:10.1109/MobileCloud.2015.38.

[87] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015. ISBN 1491950315.

[88] NewRelic RPM. http://www.newrelic.com.

[89] Huansheng Ning and Ziou Wang. Future Internet of Things Architecture: Like Mankind Neural System or Social Organization Framework? *IEEE Communications Letters*, 15(4):461–463, 2011. ISSN 1089-7798. doi:10.1109/LCOMM.2011.022411.110120.

[90] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA), 2015. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[91] OASIS. Open Building Information Exchange (oBIX), 2015. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=obix.

[92] OpenStack. http://www.openstack.org.

[93] OpsCode, Inc. Chef, 2014. http://opscode.com/chef. [Online; accessed January 17, 2014].

[94] Edewede Oriwoh, Paul Sant, and Gregory Epiphaniou. Guidelines for Internet of Things Deployment Approaches – The Thing Commandments. *Procedia Computer Science*, 21:122–131, 2013. doi:10.1016/j.procs.2013.09.018.

[95] Georgios Z Papadopoulos, Julien Beaudaux, Antoine Gallais, Thomas Noel, and Guillaume Schreiner. Adding value to WSN simulation using the IoT-LAB experimental platform. In *Proceedings of the 9th International Conference on Wireless and Mobile Computing, Networking and Communications*, WiMob'13, pages 485–490. IEEE, 2013. doi:10.1109/WiMOB.2013.6673403.

[96] Apostolos Papageorgiou, Manuel Zahn, and Ernö Kovacs. Auto-configuration System and Algorithms for Big Data-Enabled Internet-of-Things Platforms. In *Proceedings of the International Congress on Big Data*, pages 490–497, 2014. doi:10.1109/BigData.Congress.2014.78.

[97] Pankesh Patel, Animesh Pathak, Thiago Teixeira, and Valérie Issarny. Towards application development for the internet of things. In *Proceedings of the 8th Middleware Doctoral Symposium*, MDS '11, pages 5:1–5:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1072-7. doi:10.1145/2093190.2093195.

[98] Charith Perera, R Ranjan, Lizhe Wang, S U Khan, and Albert Y Zomaya. Big Data Privacy in the Internet of Things Era. *IT Professional*, 17(3):32–39, 2015. ISSN 1520-9202. doi:10.1109/MITP.2015.34.

[99] Ioan Petri, Haijiang Li, Yacine Rezgui, Yang Chunfeng, Baris Yuce, and Bejay Jayan. A modular optimisation model for reducing energy consumption in large scale building facilities. *Renewable and Sustainable Energy Reviews*, 38:990–1002, 2014. ISSN 13640321. doi:10.1016/j.rser.2014.07.044. http://linkinghub.elsevier.com/retrieve/pii/S1364032114004961.

[100] Ioan Petri, Yacine Rezgui, Tom Beach, Haijiang Li, Marco Arnesano, and Gian Marco Revel. A semantic service-oriented platform for energy efficient buildings. *Clean Technologies and Environmental Policy*, 17(3):721–734, 2015. ISSN 1618-954X. doi:10.1007/s10098-014-0828-2.

[101] PuppetLabs, Inc. Puppet, 2014. http://puppetlabs.org/. [Online; accessed January 17, 2014].

[102] Hangwei Qian and Michael Rabinovich. Application Placement and Demand Distribution in a Global Elastic Cloud: A Unified Approach. In *Proc. Int. Conf. Autonomic Computing*, ICAC'13, pages 1–12. USENIX Assoc., 2013.

[103] RabbitMQ. http://www.rabbitmq.com.

[104] Sasa Radovanovic, Norbert Nemet, Mica Cetkovic, Milan Z Bjelica, and Nikola Teslic. Cloud-based framework for QoS monitoring and provisioning in consumer devices. In *Proceedings of the Third International Conference on Consumer Electronics*, ICCE-Berlin'13, pages 1–3, 2013. doi:10.1109/ICCE-Berlin.2013.6697979.

[105] Rajiv Ranjan, Lizhe Wang, Albert Zomaya, D Georgakopoulos, X Sun, and G Wang. Recent advances in autonomic provisioning of big data applications on clouds. *IEEE Transactions on Cloud Computing*, 3(2):101–104, 2015. ISSN 2168-7161. doi:10.1109/TCC.2015.2437231.

[106] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis II. An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM Journal on Computing*, 6(3):563–581, 1977. doi:10.1137/0206041.

[107] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys & Tutorials*, 13(3):311–336, 2011. ISSN 1553-877X. doi:10.1109/SURV.2011.032211.00087.

[108] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Proceedings of the 4th International Conference on Cloud Computing*, CLOUD'11, pages 348–355. IEEE, 2011. doi:10.1109/CLOUD.2011.27.

[109] Johannes M Schleicher, Michael Vögler, Christian Inzinger, and Schahram Dustdar. Towards the internet of cities: A research roadmap for next-generation smart cities. In *Proceedings of the ACM First International Workshop on Understanding the City with Urban Informatics*, UCUI'15, pages 3–6. ACM, 2015. doi:10.1145/2811271.2811274.

[110] Johannes M Schleicher, Michael Vögler, Christian Inzinger, and Schahram Dustdar. Smart Fabric – An Infrastructure-Agnostic Artifact Topology Deployment Framework. In *Proceedings of the 4th International Conference on Mobile Services*, MS'15, pages 320–327. IEEE, 2015. doi:10.1109/MobServ.2015.52.

[111] Johannes M Schleicher, Michael Vögler, Christian Inzinger, Waldemar Hummer, and Schahram Dustdar. Nomads - Enabling Distributed Analytical Service Environments for the Smart City Domain. In *Proceedings of the International Conference on Web Services*, ICWS'15, pages 679–685, June 2015. doi:10.1109/ICWS.2015.95.

[112] Johannes M Schleicher, Michael Vögler, Schahram Dustdar, and Christian Inzinger. Enabling a Smart City Application Ecosystem: Requirements and Architectural Aspects. *IEEE Internet Computing*, 20(2):58–65, Mar 2016. doi:10.1109/MIC.2016.39.

[113] Johannes M. Schleicher, Michael Vögler, Christian Inzinger, Sara Fritz, Manuel Ziegler, Thomas Kaufmann, Dominik Bothe, Julia Forster, and Schahram Dustdar. A Holistic, Interdisciplinary Decision Support System for Sustainable Smart City Design. In *Proceedings of the International Conference on Smart Cities*, page to appear, 2016.

[114] Anuj Sehgal, Vladislav Perelman, Siarhei Kuryla, and Jurgen Schonwalder. Management of resource constrained devices in the internet of things. *IEE Communications Magazine*, 50(12):144–149, 2012. doi:10.1109/MCOM.2012.6384464.

[115] Sanjin Sehic, Stefan Nastic, Michael Vögler, Fei Li, and Schahram Dustdar. Entity-adaptation: A programming model for development of context-aware applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 436–443, New York, NY, USA, 2014. ACM. doi:10.1145/2554850.2555015.

[116] Zach Shelby and Carsten Bormann. *6LoWPAN: The wireless embedded Internet*, volume 43. John Wiley & Sons, 2011.

[117] Ganesh Shrestha and Jürgen Jasperneite. Performance Evaluation of Cellular Communication Systems for M2M Communication in Smart Grid Applications. In *Communications in Computer and Information Science*, pages 352–359–359. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31217-5. doi:10.1007/978-3-642-31217-5_37.

[118] Michel Soares, Marcelo Maia, and Rodrigo Silva. Performance Evaluation of Aspect-Oriented Programming Weavers. In *Proceedings of the International Conference on Enterprise Information Systems*, pages 187–203. Springer, 2015. ISBN 978-3-319-22348-3. doi:10.1007/978-3-319-22348-3_11.

[119] Splunk. http://www.splunk.com.

[120] John A Stankovic. Research Directions for the Internet of Things. *IEEE Internet of Things Journal*, 1(1):3–9, 2014. ISSN 2327-4662. doi:10.1109/JIOT.2014.2312291.

[121] Evangelos Theodoridis, Georgios Mylonas, and Ioannis Chatzigiannakis. Developing an IoT Smart City framework. In *Proceedings of the 4th International Conference on Information, Intelligence, Systems and Applications*, pages 1–6, 2013. doi:10.1109/IISA.2013.6623710.

[122] ThereCorporation. ThereGate. http://therecorporation.com.

[123] Ankit Toshniwal, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, and Maosong Fu. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD'14, pages 147–156. ACM, 2014. ISBN 9781450323765. doi:10.1145/2588555.2595641.

[124] Tridium. JACE Controller, 2015jo. http://www.tridiumeurope.com/22-jace.html.

[125] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1:1–1:39, March 2008. doi:10.1145/1342171.1342172.

[126] Massimo Villari, Antonio Celesti, Maria Fazio, and Antonio Puliafito. AllJoyn Lambda: An architecture for the management of smart environments in IoT. In *Proceedings of the International Conference on Smart Computing Workshops*, pages 9–14, November 2014. ISBN 9781479964475. doi:10.1109/SMARTCOMP-W.2014.7046676.

[127] Michael Vögler, Fei Li, Markus Claeßens, Johannes M Schleicher, Sanjin Sehic, Stefan Nastic, and Schahram Dustdar. Colt collaborative delivery of lightweight iot applications. In *Internet of Things. User-Centric IoT*, volume 150 of *Lecture Notes of the Institute for Computer Sciences*, pages 265–272. Springer International Publishing, 2015. doi:10.1007/978-3-319-19656-5_38.

[128] Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. DIANE – Dynamic IoT Application Deployment. In *Proceedings of the 4th International Conference on Mobile Services*, MS'15, pages 298–305, 2015. doi:10.1109/MobServ.2015.49.

[129] Michael Vögler, Johannes M Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments. In *Proceedings of the 9th Symposium on Service-Oriented System Engineering*, SOSE'15, pages 78–87. IEEE, 2015. doi:10.1109/SOSE.2015.23.

[130] Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar. Ahab: A Cloud-based Distributed Big Data Analytics Framework for the Internet of Things. *Software: Practice and Experience*, page to appear, 2016.

[131] Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. A Scalable Framework for Provisioning Large-scale IoT Deployments. *ACM Transactions on Internet Technology*, 16(2):11:1–11:20, March 2016. doi:10.1145/2850416.

[132] Michael Vögler, Johannes M Schleicher, Christian Inzinger, Schahram Dustdar, and Rajiv Ranjan. Migrating Smart City Applications to the Cloud. *IEEE Cloud Computing*, page to appear, Mar-Apr. 2016.

[133] Michael Vögler, Johannes M Schleicher, Christian Inzinger, Bernhard Nickel, and Schahram Dustdar. Non-Intrusive Monitoring of Stream Processing Applications. In *Proceedings of the 10th International Symposium on Service-Oriented System Engineering*, SOSE'16, pages 190–199. IEEE, 2016. doi:10.1109/SOSE.2016.11.

[134] Hiroshi Wada, Junichi Suzuki, Yuji Yamano, and Katsuya Oba. Evolutionary deployment optimization for service-oriented clouds. *Software: Practice and Experience*, 41(5):469–493, 2011. ISSN 1097-024X. doi:10.1002/spe.1032.

[135] Johannes Wettinger, Michael Behrendt, Tobias Binz, Uwe Breitenbücher, Gerd Breiter, Frank Leymann, Simon Moser, Isabell Schwertle, and Thomas Spatzier. Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*, pages 437 – 446, Aachen, Germany, 2013. SciTePress.

[136] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, volume 10 of *SIGMOD'06*, pages 407–418. ACM, 2006. ISBN 1595934340. doi:10.1145/1142473.1142520.

[137] Fatos Xhafa, Victor Naranjo, and Santi Caballe. Processing and Analytics of Big Data Streams with Yahoo!S4. In *Proceedings of the 29th International Conference on Advanced Information Networking and Applications*, AINA'15, pages 263–270. IEEE, 2015. ISBN 978-1-4799-7905-9. doi:10.1109/AINA.2015.194.

[138] Stephen S Yau and Arun Balaji Buduru. Intelligent Planning for Developing Mobile IoT Applications Using Cloud Systems. In *Proceedings of the International Conference on Mobile Services*, MS'14, pages 55–62, 2014. ISBN 9781479950607. doi:10.1109/MobServ.2014.17.

[139] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Proceedings of the Grid Computing Environments Workshop*, GCE '08, pages 1–10, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/GCE.2008.4738443.

[140] Wei Yuan, Hailong Sun, Xu Wang, and Xudong Liu. Towards Efficient Deployment of Cloud Applications through Dynamic Reverse Proxy Optimization. In *Proceedings of the 10th International Conference on High Performance Computing and Communications & International Conference on Embedded and Ubiquitous Computing*, pages 651–658. IEEE, 2013. ISBN 978-0-7695-5088-6. doi:10.1109/HPCC.and.EUC.2013.97.

[141] C. H Philip Yuen and S. H Gary Chan. Scalable real-time monitoring for distributed applications. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2330–2337, 2012. ISSN 10459219. doi:10.1109/TPDS.2012.60.

[142] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, number 1 in SOSP'13, pages 423–438. ACM, 2013. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522737.

[143] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. IOT Gateway: BridgingWireless Sensor Networks into Internet of Things. In *Proceedings of the IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing*, pages 347–352, 2010. doi:10.1109/EUC.2010.58.

APPENDIX A

Curriculum Vitae

Michael Vögler
Macholdastraße 24/4/79
1100 Wien, Austria

Born	February 4, 1986
Email	voegler@dsg.tuwien.ac.at
Web	http://dsg.tuwien.ac.at/staff/mvoegler

Work Experience

University Assistant at the Distributed Systems Group — 02/2013 – present
TU Wien
http://dsg.tuwien.ac.at/

Project Assistant at the Distributed Systems Group — 10/2011 – 01/2013
TU Wien
http://dsg.tuwien.ac.at/

Software Engineer — 08/2010 – 09/2010
Andritz Hydro GmbH
http://www.andritz.com/hydro

Teaching Assistant at the Distributed Systems Group — 2009 – 2011
TU Wien
http://dsg.tuwien.ac.at/

Software Engineer — 08/2009 – 09/2009
Andritz Hydro GmbH
http://www.andritz.com/hydro

Internship 11/2007 – 05/2008
SAP Austria Global Support Center
http://www.sap.at

Education

Ph.D. in Computer Science at the Distributed Systems Group, 2013 – 2016
TU Wien

Dipl.-Ing. (M.Sc.) in Software Engineering & Internet Computing, 2009 – 2011
TU Wien

B.Sc. in Software & Information Engineering, 2006 – 2009
TU Wien

Teaching

Bachelor Level Courses

- Distributed Systems Lab (184.167 - UE 2.0)

Master Level Courses

- Distributed Systems Technologies (184.260 - VU 4.0)
- Software Architectures (184.159 - VU 2.0)
- Distributed Systems Engineering (184.153 - VU 2.0)
- Project in Software Engineering & Internet Computing (184.715 - PR 6.0)

Co-Supervised Master's Theses

- Gregor Schauer: *Predicting Scalability of Standalone Applications in Cloud Environments*
- Bernhard Nickel: *Enhanced Performance Testing and Monitoring of JVM-based Distributed Data-Processing Applications*
- Jakob Korherr: *RESTful web applications with reactive, partial server-side processing in Java EE*
- Markus Claeßens: *Automated Application Deployment in heterogeneous IoT Environments by OpenTOSCA*

Publications

Journal Papers

- Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar. Ahab: A Cloud-based Distributed Big Data Analytics Framework for the Internet of Things. *Software: Practice and Experience*, page to appear, 2016
- Michael Vögler, Johannes M Schleicher, Christian Inzinger, Schahram Dustdar, and Rajiv Ranjan. Migrating Smart City Applications to the Cloud. *IEEE Cloud Computing*, page to appear, Mar-Apr. 2016
- Johannes M Schleicher, Michael Vögler, Schahram Dustdar, and Christian Inzinger. Enabling a Smart City Application Ecosystem: Requirements and Architectural Aspects. *IEEE Internet Computing*, 20(2):58–65, Mar 2016. doi:10.1109/MIC.2016.39
- Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. A Scalable Framework for Provisioning Large-scale IoT Deployments. *ACM Transactions on Internet Technology*, 16(2):11:1–11:20, March 2016. doi:10.1145/2850416
- Fei Li, Michael Vögler, Sanjn Sehic, Soheil Qanbari, Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar. Web-Scale Service Delivery for Smart Cities. *IEEE Internet Computing*, 17(4):78–83, 2013. doi:10.1109/MIC.2013.79

Conference/Workshop Proceedings

2016

- Johannes M. Schleicher, Michael Vögler, Christian Inzinger, Sara Fritz, Manuel Ziegler, Thomas Kaufmann, Dominik Bothe, Julia Forster, and Schahram Dustdar. A Holistic, Interdisciplinary Decision Support System for Sustainable Smart City Design. In *Proceedings of the International Conference on Smart Cities*, page to appear, 2016
- Michael Vögler, Johannes M Schleicher, Christian Inzinger, Bernhard Nickel, and Schahram Dustdar. Non-Intrusive Monitoring of Stream Processing Applications. In *Proceedings of the 10th International Symposium on Service-Oriented System Engineering*, SOSE'16, pages 190–199. IEEE, 2016. doi:10.1109/SOSE.2016.11

2015

- Johannes M Schleicher, Michael Vögler, Christian Inzinger, and Schahram Dustdar. Towards the internet of cities: A research roadmap for next-generation smart cities. In *Proceedings of the ACM First International Workshop on Understanding the City with Urban Informatics*, UCUI'15, pages 3–6. ACM, 2015. doi:10.1145/2811271.2811274

- Johannes M Schleicher, Michael Vögler, Christian Inzinger, and Schahram Dustdar. Smart Fabric – An Infrastructure-Agnostic Artifact Topology Deployment Framework. In *Proceedings of the 4th International Conference on Mobile Services*, MS'15, pages 320–327. IEEE, 2015. doi:10.1109/MobServ.2015.52

- Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. DIANE – Dynamic IoT Application Deployment. In *Proceedings of the 4th International Conference on Mobile Services*, MS'15, pages 298–305, 2015. doi:10.1109/MobServ.2015.49

- Johannes M Schleicher, Michael Vögler, Christian Inzinger, Waldemar Hummer, and Schahram Dustdar. Nomads - Enabling Distributed Analytical Service Environments for the Smart City Domain. In *Proceedings of the International Conference on Web Services*, ICWS'15, pages 679–685, June 2015. doi:10.1109/ICWS.2015.95

- Michael Vögler, Johannes M Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments. In *Proceedings of the 9th Symposium on Service-Oriented System Engineering*, SOSE'15, pages 78–87. IEEE, 2015. doi:10.1109/SOSE.2015.23

- Stefan Nastic, Michael Vögler, Christian Inzinger, Hong-Linh Truong, and Schahram Dustdar. rtGovOps: A Runtime Framework for Governance in Large-Scale Software-Defined IoT Cloud Systems. In *Proceedings of the 3rd International Conference on Mobile Cloud Computing, Services, and Engineering*, MobileCloud'15, pages 24–33, March 2015. doi:10.1109/MobileCloud.2015.38

- Michael Vögler, Fei Li, Markus Claeßens, Johannes M Schleicher, Sanjin Sehic, Stefan Nastic, and Schahram Dustdar. Colt collaborative delivery of lightweight iot applications. In *Internet of Things. User-Centric IoT*, volume 150 of *Lecture Notes of the Institute for Computer Sciences*, pages 265–272. Springer International Publishing, 2015. doi:10.1007/978-3-319-19656-5_38

2014

- Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. MADCAT - A Methodology for Architecture and Deployment of Cloud Application Topologies. In *Proceedings of the 8th International Symposium on Service-Oriented System Engineering*, pages 13–22. IEEE, 2014. doi:10.1109/SOSE.2014.9

- Sanjin Sehic, Stefan Nastic, Michael Vögler, Fei Li, and Schahram Dustdar. Entity-adaptation: A programming model for development of context-aware applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 436–443, New York, NY, USA, 2014. ACM. doi:10.1145/2554850.2555015

2013

- Stefan Nastic, Sanjin Sehic, Michael Vögler, Hong-Linh Truong, and Schahram Dustdar. PatRICIA – A Novel Programming Model for IoT Applications on Cloud Platforms. In *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications*, pages 53–60, 2013. doi:10.1109/SOCA.2013.48
- Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Efficient and Scalable IoT Service Delivery on Cloud. In *Proceedings of the 6th International Conference on Cloud Computing*, CLOUD'13, pages 740–747, 2013. doi:10.1109/CLOUD.2013.64
- Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. Towards Automated IoT Application Deployment by a Cloud-Based Approach. In *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications*, SOCA'13, pages 61–68, 2013. doi:10.1109/SOCA.2013.12
- Schahram Dustdar, Fei Li, Hong-Linh Truong, Sanjin Sehic, Stefan Nastic, Soheil Qanbari, Michael Vögler, and Markus Claeßens. Green software services: From requirements to business models. In *Proceedings of the 2nd International Workshop on Green and Sustainable Software*, GREENS 2013, pages 1–7, May 2013. doi:10.1109/GREENS.2013.6606415

Book Chapters

- Fei Li, Soheil Qanbari, Michael Vögler, and Schahram Dustdar. *Green in Software Engineering*, chapter Constructing Green Software Services: From Service Models to Cloud-Based Architecture, pages 83–104. Springer International Publishing, 2015. doi:10.1007/978-3-319-08581-4_4

http://dsg.tuwien.ac.at/staff/mvoegler