A Scalable Microservice-based Open Source Platform for Smart Cities

Arthur de Moura Del Esposte

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
for the Master degree in
Computer Science

Program: Computer Science
Advisor: Prof. Fabio Kon

During this work, the author was supported by CNPq. This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq, proc. 465446/2014-0, CAPES proc. 88887.136422/2017-00, and FAPESP, proc. 2014/50937-1 and 2015/24485-9.

São Paulo
April 1st, 2018

A Scalable Microservice-based Open Source Platform for Smart Cities

Arthur de Moura Del Esposte

For all those who are always present at my side, in my mind, or in my heart. Your lights have guided me in this work and support me in everything else.

Acknowledgements

"Oh great ocean
Oh great sea
Run to the ocean
Run to the sea"
—
One Tree Hill - U2

Many people have influenced this work in multiple ways. My journey in the accomplishment of this work began way before, when I left Brasilia and came to live far from those who I love the most. There I was, making the hardest and wisest choice I have ever made. Although the pain of distance was unremitting, it was accompanied by a great excitement for new knowledge, new challenges, and new horizons. Now, at the end of that journey, I have a great feeling of gratitude for all these people I have been away from all this time, especially because they have faithfully followed me all the way as the good family and friends they are.

First, I would like to thank my parents, Antônio César Del Esposte and Lucimar de Moura Del Esposte, for all their understanding, support, dedication, trust, and inspiration during this period. More than that, their immeasurable love are the foundation and inspiration that guide me towards my goals. Indeed their lifelong teachings were the most important ones I applied during this period of work. Thank you for always calling, telling me how your days were, what the new house looks like, asking me to be careful, and always hanging up saying that soon we would be together again. No matter how far I go, all my steps stem from your teachings on how to walk this world. For these reasons and for all the other possible reasons that I did not mention, I dedicate this work to my parents.

To my brother, Heitor de Moura Del Esposte, I am deeply grateful for his tireless loyalty, companionship, faith, respect, trust, and love. Our friendship is my inexhaustible source of inspiration to be always better. Ever since I can remember, you have always stood by my side in all the achievements and failures, and this time it could not be different. In fact, I felt your presence stronger than before, becoming my driving force in my daily life. I also dedicate this work to my best friend, Heitor.

I can not fail to thank my relatives who cheered for me and supported me, as they have done in all the stages since I was born, including uncles, aunts, my cousins, those who are no longer here, and, of course, my grandmother. Still talking about family, I have great gratitude for my closest friends, to whom I had to say goodbye when I came. They are my main link with other memorable moments of my life, like high school, bands, and college. Despite the physical distance, my relationship with most of them has evolved significantly. They always made a point of keeping the flame of our friendship burning, regardless of where we are, which brought me a lot of strength in those times when I have been away. Not only through the various comings and goings between Brasília - São Paulo connection, but mainly through the details is that I can realize that the link I have with these friends exceeds specific moments in the past. They are my link with life in the present and in the future.

Finally, I deeply thank Ana Paula Vieira Araujo for being the one who lives next to me; for being incredibly inspiring; for being my lifemate; for saving me; and for giving me wonderful days in her presence. Your love has led me through incredible places. Your dedication helps me make great strides, such as this one now. Living with you has been a charming experience. Dreaming and building the life with you is invaluable. And all the good that your presence provides has a significant impact on the results I have achieved with this work. I love you.

During this Master's degree, I had the opportunity to socialize with many people that I would like to thank. Special thanks to those who have contributed in some way to my stay in São Paulo and USP, either through their services or simply being part of everyday life. In this context, I acknowledge the neighbors who helped me in my earlier days in São Paulo city (especially Dona Malú); to the doormen of the IME's buildings (especially the CCSL); the coffee ladies; Marcia and Nelson for their work on the CCSL; Vicente for the moments of relaxation; and all the people with whom I shared the lab during those years.

More especially, I would like to thank my fellows from Lappis Society with whom I had the honor of working and living during this period, both inside and outside the university. It has been amazing to be able to share so many ideas, projects, jokes, and stories with such talented and competent people. I am very grateful to those Lappis Society colleagues who stood side by side with me in the struggles of everyday life in the university. You are my battle brothers without whom it would have been much harder to win. I believe we have created a remarkably strong link that will echo for the rest of our professional and personal lives.

I would like to thank Professor Paulo Meirelles for the trust and opportunity provided. Also, I am grateful to him for all the ideological, technical, and methodological teachings which I used extensively during this thesis. I acknowledge my advisor, Professor Fabio Kon, for the opportunity to do this Master's degree, and for his support and guidance along this thesis. He has given me many opportunities of growth, learning, living, and working, that impacted significantly on my academic education.

I acknowledge Professor Fabio Costa, Professor Kelly Braghetto, and Nelson Lago for their excellent contributions to the experiments carried out during this thesis and for co-authoring the papers that were published for the dissemination of the results obtained. We acknowledge Lucas Kanashiro and Eduardo Santana for their contributions in the final experiments of the platform, mainly in the integration of the platform with the simulator and in the definition of the used scenario.

We also acknowledge the following developers for their contributions to the InterSCity platform source code: Alander Marques, Alexandre T. K., Ariel Palmeira, Athos Ribeiro, Cadu Elmadjian, Caio Salgado, Caroline Satye, Danilo Caetano, Debora Setton, Dylan Guedes, Fernanda de Camargo, Fernando Freire, Frederico Lage, Henrique Potter, Igor Lima, João Brito, João Henrique Almeida, Leonardo Pereira, Lucas Brilhante, Lucas Kanashiro, Macartur Sousa, Marisol Solis, Rodolfo Scotolo, Rodrigo Faria, Rodrigo Siqueira, Rogerio Cardoso, Tallys Martins, Thiago Petrone, and Wilson Kazuo.

This research received financial support from CAPES, CNPq, and FAPESP.

Resumo

Arthur de Moura Del Esposte. **A Scalable Microservice-based Open Source Platform for Smart Cities**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

As tecnologias de Cidades Inteligentes surgem como uma potencial solução para lidar com problemas comuns em grandes centros urbanos, utilizando os recursos da cidade de maneira eficiente e fornecendo serviços de qualidade para os cidadãos. Apesar dos vários avanços nas tecnologias de middleware para suporte às cidades inteligentes do futuro, ainda não existem plataformas amplamente aceitas. A maioria das soluções existentes não oferece a flexibilidade necessária para ser compartilhada entre as cidades. Além disso, o vasto uso e desenvolvimento de software proprietário levam a problemas de interoperabilidade e limitam a colaboração entre grupos de P&D. Nesta dissertação, exploramos uso de uma arquitetura de microsserviços para abordar os principais desafios práticos em plataformas de cidades inteligentes. Mais especificamente, estamos preocupados com o impacto dos microsserviços sobre requisitos não-funcionais para permitir o desenvolvimento de cidades inteligentes, tais como o suporte a diferentes demandas de escalabilidade e o fornecimento de uma arquitetura flexível que pode evoluir facilmente. Para esse fim, criamos a InterSCity, uma plataforma para cidades inteligentes de código aberto baseada em microsserviços que visa apoiar o desenvolvimento de aplicativos e serviços sofisticados em múltiplos domínios. Nossa experiência inicial mostra que os microsserviços podem ser usados adequadamente como blocos de construção para obter uma arquitetura flexível e fracamente acoplada. Resultados experimentais apontam para a aplicabilidade de nossa abordagem no contexto de cidades inteligentes, já que a plataforma pode suportar diferentes demandas de escalabilidade. Esperamos permitir pesquisas colaborativas e inovadoras em cidades inteligentes, assim como o desenvolvimento e iniciativas de implantações reais através da plataforma InterSCity. A validação completa da plataforma será realizada usando diferentes cenários de cidades inteligentes e cargas de trabalho. Os trabalhos futuros compreendem o esforço contínuo de projetar e desenvolver novos serviços de processamento de dados, bem como a realização de avaliações mais abrangentes da plataforma proposta por meio de experimentos de escalabilidade.

Palavras-chave: Cidades Inteligentes. Platforma de Software. Microsserviços. Escalabilidade. Software Livre

Abstract

Arthur de Moura Del Esposte. **A Scalable Microservice-based Open Source Platform for Smart Cities**. Thesis (Masters). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2018.

Smart City technologies emerge as a potential solution to tackle common problems in large urban centers by using city resources efficiently and providing quality services for citizens. Despite the various advances in middleware technologies to support future smart cities, there are yet no widely accepted platforms. Most of the existing solutions do not provide the required flexibility to be shared across cities. Moreover, the extensive use and development of non-open-source software leads to interoperability issues and limits the collaboration among R&D groups. Our research explores the use of a microservices architecture to address key practical challenges in smart city platforms. More specifically, we are concerned with the impact of microservices on addressing the key non-functional requirements to enable the development of smart cities such as supporting different scalability demands and providing a flexible architecture which can easily evolve over time. To this end, we are developing InterSCity, a microservice-based open source smart city platform that aims at supporting the development of sophisticated, cross-domain applications and services. Our early experience shows that microservices can be properly used as building blocks to achieve a loosely coupled, flexible architecture. Experimental results point towards the applicability of our approach in the context of smart cities since the platform can support multiple scalability demands. We expect to enable collaborative, novel smart city research, development, and deployment initiatives through the InterSCity platform. The full validation of the platform will be conducted using different smart city scenarios and workloads. Future work comprises the ongoing design and development effort on data processing services as well as more comprehensive evaluation of the proposed platform through scalability experiments.

Keywords: Smart Cities. Software Platform. Microservices. Scalability. Open Source.

List of Figures

2.1 The scalability cube from The Art of Scalability book (Abbott and Fisher, 2009). 13
2.2 Architecture of the CiDAP platform (extracted from (Cheng et al., 2015)) 17
2.3 FIWARE architecture overview 18
2.4 DIMMER architecture overview 23

3.1 The InterSCity Platform Architecture. 28
3.2 InterSCity API gateway 32
3.3 InterSCity load balancing HTTP requests 32
3.4 InterSCity Asynchronous messaging 34
3.5 Competing Consumers design pattern 34
3.6 Example of Input JSON to Register a City Resource. 39
3.7 Example of Response for the Request to Register a Resource. 40
3.8 Example of Input JSON to Publish Context Data. 41
3.9 Example of Input JSON for Actuation Subscription. 42
3.10 Example of Webhook JSON Body. 42
3.11 Example Response of the Search Endpoint. 44
3.12 Example Input JSON with Parameters to Filter Context Data. 46
3.13 Example Input JSON for Request Actuation Commands. 47
3.14 Example Response for Request Actuation Commands. 47
3.15 Screenshot of the Smart Parking application. 50
3.16 Parking spot details in the Smart Parking application. 50
3.17 Smart Parking application life cycle. 52
3.18 Screenshot of the São Paulo Health Dashboard. 53
3.19 São Paulo Health Dashboard application lifecycle. 54
3.20 Screenshot of the SancaLights project. 55
3.21 Screenshot of the SPoiler Twitter bot. 56

4.1 Scalability-seeking Experimental Method 58

4.2 Improvement Cycles of the First Round of the Scalability-seeking Experimental Method 60
4.3 First improvement cycle - InterSCity design. 61
4.4 Improvement Cycles of the Second Round of the Scalability-seeking Experimental Method 63

5.1 Response time degradation. 67
5.2 Speedup - performance improvement varying the number of Resource Adaptors. 68
5.3 Throughput improvement varying the number of Resource Adaptors. . 69
5.4 Heat maps of the distributions of parking spots and car trip destinations in the City of São Paulo 72
5.5 Cluster Node Pools for the experiments 73
5.6 Average workload generated by the InterSCSimulator 75
5.7 InterSCity services autoscaling 76
5.8 Average InterSCity throughput 76
5.9 InterSCity Average Response Time 77

List of Tables

2.1 Smart City Platforms' Features 24

3.1 Technology Stack of InterSCity Microservices 35
3.2 Scalability strategies supported by InterSCity microservices 37

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Objectives and Contributions 3

2 Background and Related Work **7**
2.1 Smart City Platforms 7
2.2 Microservices Architecture 11
2.3 Smart City Projects 15

3 The InterSCity Platform **25**
3.1 Design Principles 25
3.2 Platform Architecture 27
3.3 Design Details to Address Scalability 29
3.3.1 Functional Decomposition 29
3.3.2 Communication Style 31
3.3.3 Design and Technology Heterogeneity 35
3.3.4 Independent Deployment 37
3.4 Microservices 39
3.4.1 Resource Adaptor 39
3.4.2 Resource Catalog 41
3.4.3 Data Collector 43
3.4.4 Actuator Controller 45
3.4.5 Resource Discovery 46
3.5 Implementation Principles 48
3.6 Application Life-cycle 49
3.6.1 Smart Parking App 49
3.6.2 São Paulo Health Dashboard 51
3.6.3 SancaLights 53
3.6.4 Recomenda SP 54

4 Scalability-seeking Experimental Method **57**
4.1 Improvement Cycles of the First Round 58
4.2 Improvement Cycles of the Second Round 62

5 Scalability Evaluation **65**
5.1 Evaluating the Preliminary InterSCity Version 65
5.1.1 Degradation Analysis 66
5.1.2 Scalability Analysis 67
5.1.3 Threats to Validity and Limitations 68
5.2 Evaluating the Latest InterSCity Version 69
5.2.1 Smart City Scenario 70
5.2.2 Experiment Configuration 72
5.2.3 Scalability Analysis 74
5.2.4 Threats to Validity and Limitations 77

6 Conclusion **79**
6.1 Future Work 80

Bibliografia **83**

Chapter 1

Introduction

The rapid growth of cities around the world has created large, densely populated urban centers characterized by complex interconnected structural, social, and economic organizations. According to the Department of Economic and Social Affairs of the United Nations (2014), the slice of the world's population living in urban areas increased from 30% in 1950 to 54% in 2014, and by 2050, it is expected to reach 66%. The urbanization phenomenon around the world imposes several challenges for sustainable development and quality of life in cities, as well as generated several problems related to health care, education, public safety, transportation, pollution, to cite only a few examples.

Since traditional management approaches cannot overcome such challenges, the government, academy, and industry need to join efforts to develop and apply new solutions. Thus, *smart cities* emerge as a new paradigm aimed at addressing the aforementioned problems by using city resources efficiently and providing quality services for its citizens. Smart cities are characterized by the adoption of Information and Communication Technologies (ICT) as an integral part of the city's infrastructure to support multiple solutions for urban challenges (Neirotti et al., 2014). The Internet of Things (IoT), Big Data, Cyber-physical Systems, and Cloud Computing are key enabling technologies of smart cities. Although these areas have evolved considerably recently, their integration is not straightforward, offering a wide range of opportunities and challenges, both in the academy and industry spheres.

Several efforts have been devoted to the study and development of this new paradigm, leading to a proliferation of smart city initiatives around the world, targeting different domains such as urban mobility, energy management, and healthcare. However, most of these initiatives have been developed using ad-hoc approaches. They neither follow a common set of practices and standards nor consider the need for data and resource sharing among the different systems in the city. This hinders the development of smart city solutions that are sustainable in the long term, creating market islands and raising extensibility, adaptability and interoperability issues (Villanueva et al., 2013). To fully exploit the potential of these enablers, future smart cities will demand a unified ICT infrastructure to share their resources properly rather than relying on non-integrated solutions.

Many authors advocate that integrated middleware platforms can provide the required infrastructure to support the construction of sophisticated, cross-domain smart city applications (Villanueva et al., 2013; Hernández-Muñoz et al., 2011; Fazio et al., 2012). The use of smart city platforms facilitates fast development of integrated, high-quality smart-city services and applications (Santana et al., 2017). Comprehensively, a smart city platform is a software infrastructure capable of enabling interoperability between a city's multiple systems, devices, and other services, abstracting their inherent communication complexities with the main goal of offering facilities to support the development of smart city solutions such as mobile apps, Software as a Service (SaaS), and management dashboards. Such platforms must implement a set of common functional requirements in the form of reusable services for application developers. Although each project contains particularities, we can list a set of characteristics common to several of these systems, as observed in previous surveys (Santana et al., 2017):

- Integration and Management of IoT Devices
- Data Management and Processing
- External Data Access
- Context-Awareness
- City Resource Discovery
- Geolocation-based Services

1.1 Motivation

Despite the various advances in middleware technologies, protocols, and standards, many aspects related to the design, development, deployment, and management of smart city platforms still challenge the research community. Consequently, there are no widely accepted platforms yet, and existing solutions do not provide the required flexibility to be shared across cities (PCAST, 2016). In the following, we highlight three key factors that contribute to the lack of practical and reusable solutions in the field.

First, in addition to traditional functional capabilities, smart city platforms must meet a number of non-functional requirements to enable their use in different environments and applications. Security and privacy policies may change according to the laws and regulations of the city, impacting design decisions regarding storage and availability of the data on the platform. Different contexts may expose a great diversity of requirements, which may dynamically evolve. Thus, smart city platforms must provide a flexible architecture to adopt new technologies and support new functional and non-functional requirements to suit the diversity of the multiple and constantly evolving city environments where they are deployed.

Similarly, a smart city platform must integrate a large number of users, devices, and services as well as their associated data. In particular, these platforms must offer different scalability strategies by design to meet the diverse scalability demands. A smart city platform must handle a large number of devices integrated with the city infrastructure. It needs to store and process large volumes of data related to the city, which are continuously produced and consumed by the devices and client applications. At the same time, the platform must be able to support thousands of requests from the users and services that use its functionalities. The scalability demands for the platform, thus, vary according to the characteristics of the city, as well as those of the deployed applications and services. For instance, a city may start with a pilot project in one of its neighborhoods and then expand to other regions as the required infrastructure becomes available. Therefore, the dynamism and continuous evolution of urban environments require the use of new approaches to develop flexible, evolvable, maintainable, and scalable architectures for smart city platforms (Krylovskiy et al., 2015).

The second point is related to the extensive use and development of non-open source software in the core of smart city platforms which jeopardize their widespread adoption as it leads to interoperability difficulties. Also, this approach limits the collaboration among R&D groups, often forcing them to “reinvent the wheel", which is a recurrent problem in Computer Science research (Peng, 2011; Freire et al., 2012). The use of open technologies is crucial to the sustainability and development of future smart cities, since they prevent vendor lock-in, enable collaborative development, market opportunities, and sharing of solutions. More specifically, systems that control lives in cities, such as smart city platforms, should not be black boxes. The open source approach enables citizens to understood and audit such systems.

Finally, the research community lacks practical and scientific validation to evaluate the different aspects of smart city solutions, such as social and economic impact, internal and external quality, performance, scalability, and feasibility, to cite a few. Among the papers surveyed by Santana et al. (Santana et al., 2017), most of the projects that supposedly meet scalability requirements only present superficial discussions of design and implementation decisions that can lead to a scalable architecture. In (L. Sanchez et al., 2011), the authors observed that although many IoT projects present concrete solutions, validation of the developed technologies and architectural models are limited to proofs-of-concept, not allowing conclusive results.

However, demonstrating the actual scalability of smart city platforms presents significant challenges due to the lack of available infrastructure, real experimental setups, and comprehensive datasets. Therefore, we still need more comprehensive studies with experiments and tests that allow the comparison of different smart city technologies. For this purpose, significant effort should be devoted to (1) deploying existing solutions in real production scenarios, (2) developing methods and tools for more sophisticated simulation-based evaluations, and (3) developing well-defined benchmark strategies for cross-platform assessment.

1.2 Objectives and Contributions

The goal of this Masters research is to advance the state-of-the-art by exploring the impact of a microservices architecture in the design, development, deployment, and performance of scalable smart city platforms. To this end, we propose InterSCity, an open source microservices-based cloud-native platform for smart cities. We expect to provide a high-quality, modular, highly scalable middleware infrastructure to support smart city solutions that can be reused across cities and R&D groups, as well as governments and companies.

InterSCity leverages the microservice approach to implement the fundamental modules described by the reference architecture proposed in (Santana et al., 2017), conceived from the analysis of 23 smart city projects. This reference architecture describes the building blocks needed to meet the main functional and non-functional requirements to guide the development of next-generation platforms for smart cities. Thus, the platform aims at providing high-level services to manage heterogeneous IoT resources, data storage and processing, and context-aware resource discovery.

In this work, we present the preliminary design and implementation details of the InterSCity platform to address the design challenges in smart city systems. Moreover, we present a set of reproducible experiments to evaluate the InterSCity scalability properties, detailing the experimental method and providing a comprehensive analysis of the results. In particular, we expect to achieve the following technical and scientific contributions to smart cities platforms research, denoted by **TC** and **SC**, respectively.

- **TC1 - InterSCity design and implementation** - the initial design and implementation of the InterSCity platform as an open source project to enable novel smart city research, development, and deployment initiatives. Although it comprises fundamental requirements of the InterSCity project, the presented architecture is not definitive. However, it implements the main ideas, design patterns, design principles, and communication protocols which will serve as a basis for future developments. Moreover, its loosely-coupled archictecture based on single-purpose services and our open source approach contribute to its extensibility and adoption.

- **TC2 - Deployment of an online instance** - a production-like environment of the InterSCity platform to integrate with existing available data and services of the São Paulo city. This instance is currently available online[1] being used in an exploratory way by other research groups and students during programming courses, mainly for conducting research and for developing applications. Also, we used this online instance for the 2017 USP Hackathon on Smart Cities[2]

- **SC1 - Advances in smart city platforms evaluation** - advances on the performance evaluation of smart city platforms with the use of a simulation-based approach. More precisely, we discuss and address the challenges related to two aspects of smart city platforms evaluation: (I) workload generation that genuinely represents the dynamics of smart cities; (II) improve the reproducibility of experiments by automating the deployment and configuration tasks through DevOps techniques.

- **SC2 - Analysis of the adoption of microservices in the context of smart cities** - a comprehensive analysis of the impact of InterSCity's microservice architecture to address key research challenges regarding *scalability* in smart city platforms. For this purpose, we deepen the main design decisions to

[1] http://playground.interscity.org/

[2] http://interscity.org/events/hackactona-usp-smart-cities/

meet scalability requirements and designed reproducible experiments to evaluate the InterSCity performance in scenarios with workload variations. In addition to scalability, we explore our early experience in adopting microservices architecture and its impact on other key pragmatic aspects, such as the complexity and evolvability of the system.

- **SC3 - Scalability-seeking experimental method** - an iterative method we have adopted in the InterSCity development life-cycle to develop and evaluate scalable software architectures. This method is based on performance testing with the objective of continuously improving the proposed solution and identifying possible bottlenecks to solve them beforehand.

We developed this research in the context of the InterSCity consortium[3] - INCT of the Future Internet for Smart Cities. The goal of the InterSCity project is to develop multidisciplinary research on software infrastructure for smart cities, producing both scientific and technical contributions to the community (Batista et al., 2016). Therefore, the development of this masters thesis followed InterSCity projects' guidelines aiming at producing high impact, reproducible scientific results as well as open source software that can be maintained later by the project community.

The remainder of this text is organized into four major parts: Chapter 2 sets the background of our work and presents a discussion of the main related work and their differences with our approach. Chapter 3 describes in detail the proposal and implementation of the InterSCity microservices architecture. Chapter 4 introduces our method to iteratively develop and evaluate the InterSCity platform towards a scalable architecture. Chapter 5 presents the experiments we conducted to evaluate the platform and a comprehensive analysis of the obtained results. Lastly, we present the final remarks, open challenges, and opportunities for future work in Chapter 6.

[3]http://interscity.org

Chapter 2

Background and Related Work

This chapter covers the essential background knowledge regarding the subject of this research and discusses some important related work. First, it presents an overview of the concepts about smart city platforms, defining their main functional and non-functional requirements. Then, we conceptualize the microservices architecture paradigm, which is the fundamental basis of the InterSCity platform and the study target of this research. Finally, we present some notable smart city platforms proposed by existing smart city initiatives, driving into their architectures to elucidate how these platforms meet the requirements and identify the key open challenges.

2.1 Smart City Platforms

Discussions about the need for changes in traditional approaches to manage the development of cities have grown significantly driven by the increase in urban problems such as pollution, traffic jams, precarious health infrastructure, and social inequality. In this context, the concept of Smart Cities appears as a new paradigm to address the challenges mentioned above characterized by the extensive use of Information and Communication Technologies (ICT) as means of helping cities use their resources efficiently and improving the living conditions of the population (Neirotti et al., 2014). However, several authors advocate that ICT investments alone do not imply that a city is smart because such investments must be aligned with the development of human capital, improvements in the physical infrastructure of urban environments, and the right policies (Caragliu et al., 2009; González and Rossi, 2011; Partridge, 2004).

The adoption of ICT must meet the same principles of smart cities, being sustainable, promoting the reuse of resources, and reflecting the real needs of urban environments. Therefore, the application of techniques and technologies to enable the development of smart cities must be carefully studied so that they do not jeopardize the growth of cities, rather than helping them. Among other important steps, future smart cities must integrate and leverage key enabling technologies such as the Internet of Things (IoT), Cloud Computing, and Big Data. Such integration is not straightforward as these areas advance rapidly in both the academic and industrial spheres, consequently creating new solutions and standards to address the constantly evolving challenges.

Although several smart city initiatives have produced results and applications to solve a wide range of problems, they are mostly focused on a specific domain, targeting a particular problem, with little software reuse (Santana et al., 2017; Villanueva et al., 2013). The proliferation of vertical solutions and isolated initiatives in cities leads to non-interoperable services, forcing the development of ICT infrastructure from scratch with little resource reuse, and fragmentation of information. The emergence of isolated solutions in different domains, either by government agencies or commercial initiatives, creates unsustainable systems and market islands that rely on their own ICT infrastructures which comprise similar services, such as integrating cyber-physical systems, storing sensor data, processing data streams from various sources, and providing visualization tools (Hernández-Muñoz et al., 2011).

Many authors advocate that services with great potential of reuse could be brought together in an integrated platform to support cross-domain applications rather than vertical silos (Villanueva et al., 2013; Santana et al., 2017; Hernández-Muñoz et al., 2011; Fazio et al., 2012). Such platforms must provide a horizontal middleware infrastructure with facilities for application development, management, and deployment. Henceforth, in this text, we will adopt the term smart city platform as defined by Santana et al., 2017: *"an integrated middleware environment that supports software developers in designing, implementing, deploying, and managing applications for Smart Cities"*.

Identifying the fundamental requirements for smart city platforms is an important task for designing a new solution, as these platforms must integrate heterogeneous software services and serve as a basis for the development of various applications. The identification of these requirements is also important to design appropriate benchmarks for comparing existing solutions. Most of the requirements can be found scattered through papers that present new architectures for smart cities. However, these works fail to properly organize the requirements as they mainly focus on the solution they are proposing.

To the best of our knowledge, only a few studies on the literature aim at organizing these requirements through surveys on existing projects or literature reviews. Silva et al., 2013 described some important requirements that smart city architectures must meet, regardless of how they are implemented. Although the authors point out important requirements, such as objects interoperability, privacy policies support, and real-time monitoring, they do not cover several important aspects, such as development support and scalability demands. Furthermore, the use of vague and high level abstraction to identify some of the requirements as well as the lack of more detailed information impair the understanding of the most relevant needs for the development of smart city solutions.

Santana et al., 2017 conducted a comprehensive survey on smart city platforms to analyze the most used enabling technologies while also identifying functional and non-functional requirements for smart city platforms. Functional requirements encompass the features necessary to support the development of smart city applications and services, while the non-functional requirements are related to the quality with which these services are offered. Below, we list the smart city platforms requirements identified in (Santana et al., 2017), which guided the development of our work.

Functional Requirements

- **IoT Integration and Management** - smart cities platforms must integrate a number of heterogeneous objects that communicate through different protocols to enable collecting data about various aspects of the city (via sensors) and to receive commands to change physical features (via actuators) dynamically. While there are initiatives that aim to provide standardized cloud services for the management and integration of IoT devices, others also address lower-level requirements, such as deploying and management of Wireless Sensor Networks. To properly support the development of applications, smart city platforms must encapsulate IoT devices in logical abstractions that hide the underlying technological particularities, such as network protocols and data representation.

- **City Models and Abstractions** - smart city platforms must provide facilities to intermediate and abstract all the communication among client applications and the underlying devices. In particular, a platform should provide standardized means of access to the city's resources through well-defined models and abstractions, offering services to facilitate the discovery of these resources and the access to the data produced. Abstractions and concepts provided by pure IoT ontologies and standards, such as proposed in (Bauer et al., 2013; Serrano et al., 2015), are not sufficient to fully enable smart city applications. Conversely, application developers require abstractions that correspond to the city's concepts, such as buildings, cars, buses, and light poles, rather than dealing with technical specificities related to the cyber-physical infrastructure that enables the interaction of these objects with the city, such as which sensors those entities couple.

- **Data Management and Processing** - the integration of a large number of devices and data sources require smart city platforms to manage data from the city, providing storage, processing, analysis, and visualization services. Data processing facilities may include inference services, workflow processing, historical data processing, and real-time analysis of data streams. In this sense, smart city platforms prevent each of the client applications from having to implement the infrastructure required for such tasks while promoting the reuse of services.

- **Context-awareness** - the behavior of smart city applications may change according to the user preferences, location, and context. Also, to properly support the development of smart city applications, middleware platforms needs to enable the characterization of city entities, such as their situation and location. In general, instead of providing the raw sensor data which is directly retrieved from an IoT device, the contextual information adds meaning to those data, check for consistency, and may comprise additional metadata (Perera et al., 2014). For instance, the data collected by GPS sensors coupled to public buses could be considered raw data, while the bus' geographical location and speed inferred from the raw data define its context. It is also relevant to understand the context of the application user to properly offer service and features meaningful for his/her context, such as providing traffic information on his/her route instead of roads that are not part of his/her path. Therefore, smart city platforms may leverage context data from city resources to provide more relevant information and services to the supported applications and users. Such services can be used to discover the most important city resources and data to properly support user tasks and to define which information needs to be presented.

- **External Data Access and Application Run-time** - platforms must provide either an API to be remotely accessed through communication protocols (e.g., HTTP and MQTT) or offer a run-time environment to manage the deployment and execution of client applications.
- **Software Engineering Tools** - ideally, smart city platforms should provide a set of software engineering tools to improve the development support, such as native libraries, a Software Development Kit (SDK), and even visual interfaces to help non-programmer users to build dashboard-based applications

Non-Functional Requirements

The adoption and feasibility of a smart city platform will depend not only on their features but also on how it addresses important non-functional requirements. Santana et al., 2017 identified the key non-functional requirements for smart city platforms, which are objective of several studies on the area:

- **Interoperability** - smart city platforms must handle a large number of heterogeneous devices, systems, services, and applications that comprise a smart city environment in an integrated fashion. For this purpose, smart city platforms must enable both underlying IoT infrastructure and applications to get the city information, exchange data, and be able to understand and process it. More specifically, interoperability requirements impact on several levels: (I) the communication protocols and the infrastructure needed for those protocols to operate; (II) the data formats that will be exchange among software components; (III) the semantic associated with the meaning of the content for human interpretation (Serrano et al., 2015)
- **Security and Privacy** - the design of a platform should handle several **security** and **privacy** requirements. The former is related to the robustness of the architecture to handle threats to city infrastructure and information. The latter is associated with the manipulation of data from several sources and from citizens without exposing sensitive data or identifiable information.
- **Adaptation** - smart city platforms must adapt to changes in city environments with low effort and external intervention. The adaptability of a software ecosystem concerns various aspects of the system, such as adapting their behavior based on context data, managing resources elastically to meet diverse scalability demands, and dynamically supporting changes in regulations and policies. This requirement is fundamental to platforms that aim to be shared across several cities.
- **Cloud-native** - although some smart city platforms implement components to comprise underlying IoT devices, protocols, and infrastructure, they have

mostly cloud-native software components. By doing so, these platforms levarages the highly available, elastic infrastructure of Cloud Computing to meet the complexity of smart city solutions.

- **Evolvability and Extensibility** - urban environments are very dynamic and tend to change regularly regarding organization, regulations, problems, opportunities, and challenges. Therefore, smart city platforms must expand to meet changing requirements in a cost-effectively way. Evolvability is the system's capability to evolve by supporting rapid modification and enhancement with low cost and small architectural impact, and is a fundamental element for the success and economic value of long-lived software (Breivold et al., 2012).

- **Scalability** - a platform must scale well in multiple dimensions to properly support a smart city. Among others, smart city platforms must handle: (I) a large number of devices that compose the city IoT infrastructure; (II) millions of users and components that use the platform services; (III) a very large volume of city-related data that must be stored and processed; (IV) a potentiality large set of new services that can interact with the platform to offer complementary capabilities. The scalability requirements may vary depending on the context and should be addressed since the beginning of any smart city project.

Lastly, as we highlighted in Chapter 1, it is highly desirable for smart city solutions to be built as open source software and support open standards so that they can be shared across different initiatives avoiding cities to *reinvent the wheel.*

2.2 Microservices Architecture

The main objective of this masters research is to propose the initial architecture of the InterSCity platform. In particular, we aim at providing a scalable middleware infrastructure that can evolve over time to meet the constantly varying requirements of smart cities. Our strategy to address scalability and evolvability issues, while providing the required services to support smart cities is to adopt a microservices architecture. Microservices is an architectural style to develop a system as a set of distributed fine-grained, independent services that collaborate through lightweight communication mechanisms (Lewis and Fowler, 2014). The microservices model emerged from the software industry efforts to build large-scale distributed systems gathering the guidelines of traditional Service Oriented Architecture, domain-driven design, continuous delivery, on-demand virtualization, infrastructure automation, and small autonomous teams (Newman, 2015).

This architectural style promotes the modularity via services, a set of independent processes interacting via network protocols such as web services or remote procedure calls (Dragoni et al., 2016). Each microservice must implement a highly cohesive, low-scoped functionality built around business capabilities with explicit boundaries (Lewis and Fowler, 2014). The boundary of a microservice is usually defined by its technology-agnostic application programming interface (API) that can be used by other microservices regardless of the underlying technologies (i.e., programming languages and frameworks). As a consequence, this isolation promotes a low coupled architecture, allowing independent microservices deployment and version management leveraging the use of DevOps techniques (Balalaie et al., 2016), as well as Agile principles, such as continuous delivery and embracing changes. Conversely, the main drawbacks of this approach lie on the increased overall complexity of the system and the extensive use of remote calls, which are more expensive than in-memory calls.

The implementation of a limited amount of functionalities makes microservices' codebase small, facilitating maintenance, fast isolated testing, and limiting the scope of a bug. It's worth noticing that the microservices approach drives to decentralized data management letting each service manage its database, enabling Polyglot Persistence[1] toward different storage technologies for different kinds of data, but also imposing trade-off decisions regarding data consistency (Lewis and Fowler, 2014). At the same pace, microservices-based architectures promote technology heterogeneity, since each service can be implemented using the most appropriate stack of technologies to properly meet the business requirements.

We advocate that the adoption of microservices guidelines to design smart city platforms can address several important challenges towards the development of practical solutions. In addition to the benefits mentioned above, the functional decomposition of large systems into smaller collaborating services is key to meet two non-functional requirements: **scalability** and **evolvability**. The microservices approach supports an evolutionary design without slowing down change in three aspects. First, except for changes on communication interfaces, modifying, adding features, or fixing errors on a single microservice codebase will not affect others, neither on code level nor deployment. Second, the loosely coupled architecture is optimized for replaceability (Newman, 2015). The barriers to replace a small service with an entirely new better, implementation are very low, both regarding writing the new code and regarding deploying the new service. Last, the independence of microservices and their well-defined APIs encourage the addition of new services to meet changing requirements.

Regarding scalability, the functional decomposition of a microservices architecture supports different levels of scalability by splitting workloads among all services. This scaling corresponds to the Y-axis scaling of Scale Cube (Figure 2.1), the 3D model of scalability from the book *The Art of Scalability* (Abbott and Fisher, 2009). The entire workload of the system will be handled by separate services, probably non-uniformly. It is also possible to apply the X-axis scaling strategy by balancing the load among several instances of the same service (e.g., behind a load balancer). However, in a microservice architecture, we only need to scale out the stressed services, rather than scaling the entire system. Finally, microservices may also support scalability by data partitioning, represented by the Z-axis on the Scale Cube. By applying this strategy, different copies of the same service are responsible for a subset of the data. In this scenario, a component of the system must use additional information, such as a primary key of the requested data or user identity, for routing the requests to the appropriate server. Z-axis scaling are commonly used in the database layer by NoSQL technologies, such as MongoDB[2], and can be implemented for the microservices with higher data demand.

[1] https://martinfowler.com/bliki/PolyglotPersistence.html
[2] http://mongodb.org

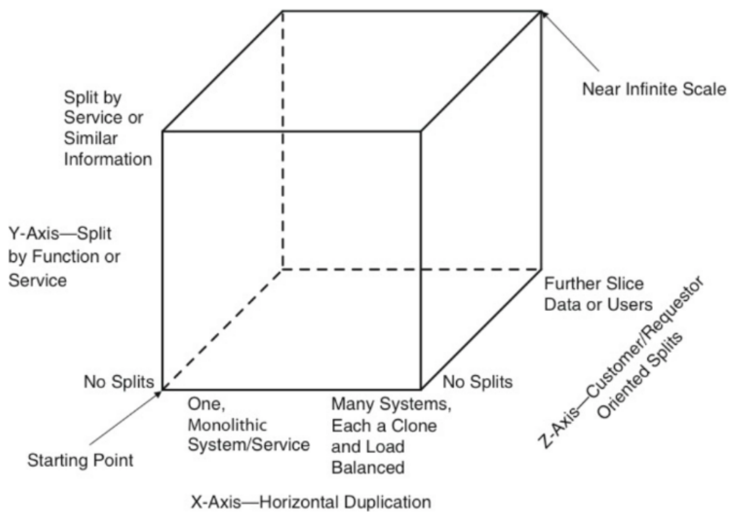

Near Infinite Scale
Split by Service or Similar Information
Y-Axis—Split by Function or Service
Further Slice Data or Users
Z-Axis—Customer/Requestor Oriented Splits
No Splits
No Splits
Starting Point
One, Monolithic System/Service
Many Systems, Each a Clone and Load Balanced
X-Axis—Horizontal Duplication

Figure 2.1: *The scalability cube from The Art of Scalability book (Abbott and Fisher, 2009).*

For comparative purposes, in the following, we describe the differences among the studied approach and two of the most adopted system architecture styles: monoliths and traditional SOA.

Microservices vs. Monoliths

Monolith is a software built on a single codebase whose modules cannot be deployed and executed independently, since they rely on the sharing of resources on the same machine, such as memory, databases, and files. Dragoni et al., 2016 describe in detail some of the major problems of monolithic software such as those listed below:

- The larger is the size of a monolith, the harder is to maintain and evolve due to its complexity. Microservices-based architecture relies on smaller services that can be managed in separate codebases that can evolve independently.

- Changing one small module of a monolith requires rebooting the whole application, which may hinder the development, maintenance, and execution of the system. In a microservices architecture, only the modified services need to be updated and rebooted.

- Monoliths limit scalability since the increased load usually only stresses a few specific modules (e.g., classes), making the allocation of new resources for the other components inconvenient (e.g., for load balancing purposes). As previously discussed, microservices natively supports Y-axis scaling of the Scale Cube and encourages the application of other scalability strategies as well.

- A monolith codebase imposes a technological lock-in for developers as changing in programming languages or frameworks may cause a system-wide impact. Conversely, due to loosely coupled, technology-agnostic communication interfaces, developers may choose the most appropriate technologies for implementing a specific microservice, both in the logic and database tiers.

Microservices vs. Traditional SOA

Microservices leverage the well-accepted service-oriented computing model in several aspects by refining traditional SOA guidelines for building large-scale systems. Although SOA is a comprehensive umbrella term, in this text, we will refer to the conventional SOA approach considering the architectures that aim at integrating monolithic applications through an Enterprise Service Bus (ESB) and the ecosystem of Web service specifications (WS-*)[3]. Loosely defined, microservices intend to remove unnecessary levels of complexity of traditional SOA to focus on the development of simple, cohesive services with a single responsibility (Dragoni et al., 2016). In the following, we detail the fundamental differences between microservices-based architectures and traditional SOA:

- **Service Size** - SOA aims to integrate services of different granularities, which includes general purpose services or a huge monolithic system that supports an entire business process. Conversely, microservices lead to systems composed of smaller services suggesting that if a service is too large, it should be split into two or more services that provide single business capabilities.
- **Integration Mechanisms** - Usually, SOA systems rely on a centralized communication mechanism, such as an ESB, that includes sophisticated facilities for message routing, choreography, transformation, and application of business policies (Lewis and Fowler, 2014). The ESB is a potential single point of failure for the entire application. Conversely, the microservices approach favours *smart endpoints and dump pipes*. Therefore, microservices implement the necessary business capabilities collaborating through public APIs and lightweight messaging brokers which support simple, reliable asynchronous communication.
- **Data Management** - SOA usually relies on integration through shared databases and models. In this context, several services could read and write on the same database. The microservices community encourages decentralized databases, where each microservice manages its data and uses the most appropriate database technology.
- **Communication Protocols** - As microservice architectures tend to eliminate unnecessary communication complexity, their services do not implement the heavy WS-* protocols stack, which is extensively adopted by SOA systems. Thus, microservices commonly expose their API through the REpresentational State Transfer (REST) architectural style. REST offers a set of principles and constraints to expose operations related to resources encapsulated by the services over lightweight protocols, mainly HTTP.

Although there are efforts to understand the impact of microservices architecture on other research areas (Le et al., 2015; Gopu et al., 2016), very few works explore the potential of microservices in the context of smart cities. By refining the main

[3]The term WS-* encompasses SOAP Web services and their specifications, such as WSDL, WS-Policy, and UDDI.

SOA guidelines to achieve a more flexible, evolvable, and scalable architecture, the microservices approach has an excellent potential to compose the architecture of next-generation smart city platforms to address the key research and practical challenges in the area. In this sense, this masters research advances the state-of-the-art by exploring microservices in smart city solutions as well as providing a novel open-source smart city platform as a practical outcome from this effort. The next section presents the main related works, highlighting their differences regarding our proposal as well as the open challenges.

2.3 Smart City Projects

Several efforts have been devoted to the study and development of platforms that address the key challenges of smart cities. In particular, projects that aim at addressing the practical problems related to the development, deployment, and maintenance of smart city services and applications are the most relevant in the context of this work. Thus, in this section, we present some notable smart city platforms proposed in related work, deepening on their major architectural decisions to meet the above-listed requirements.

Many middleware platforms were developed in recent years to address multiple requirements towards the construction of cross-domain smart city solutions, as opposed to traditional approaches based on vertical silos. The Civitas middleware (Villanueva et al., 2013) fulfills the main functional requirements by proposing a set of essential standards and tools to enable smart city ecosystems. Civitas treats everything as a software object to achieve interoperability and adaptability requirements. The Gambas project (Apolinarski et al., 2014) offers tools to facilitate the development and deployment of smart city applications, including a runtime environment and an SDK. On the other hand, Gambas use a traditional Service-Oriented Architecture to provide its services but lacks more sophisticated data processing facilities. Although both Civitas and Gambas address challenges concerning functional requirements, these platforms do not meet some of the key non-functional requirements, such as scalability, extensibility, and evolvability.

OpenIoT[4] is one of the most relevant projects handling the main requirements of smart city platforms. It is an open source layered middleware platform that aims at enabling semantic interoperability across IoT applications, including smart cities (Soldatos et al., 2015). The platform provides visual tools to facilitate the administration and implementation of applications directly on top of it. Although OpenIoT offers several facilities to support IoT applications, it is mainly focused on address functional requirements and interoperability issues. On the other hand, its architecture does not deal with important aspects related to scalability, adaptability, and extensibility.

Almanac[5] is an open source federated smart city platform which provides cloud-based services to support application development based on a SOA architecture (Bonino et al., 2015). Its features include semantic interoperability, data processing

[4]https://github.com/OpenIotOrg/openiot

[5]http://www.almanac-project.eu/news.php

and management, city models, and integration of IoT devices. However, the most notable advances in this project are two-fold: (I) the support for federated platforms offering cross-city and cross-entity services through the combination of multiple Almanac instances, and (II) privacy management on different aspects by supporting policy definitions to address role management, task control, and data access. While the Almanac project brings important contributions by exploring different strategies to deliver an adaptable architecture that can be shared across cities, we intend to achieve adaptability and still keep a flexible architecture that can adequately evolve through a microservice approach to properly meet the continually changing demands of different urban environments.

In the following, we delved into the architectures of the most relevant related projects in the context of this research: SmartSantander, FIWARE, and DIMMER.

SmartSantander and CiDAP

Perhaps the most notable project that targets the realistic deployment and validation of smart city solutions is SmartSantander, which is one of the projects of the Future Internet Research and Experimentation initiative of the European Commission. The SmartSantander initiative aims at advancing the IoT research by creating an experimental test facility for experimentation in architectures, technologies, and solutions for the IoT, especially in the context of smart cities. For this purpose, the project provides a smart city testbed with research facilities composed of more than 20,000 IoT devices deployed in urban environments around the city of Santander in Spain (Luis Sanchez et al., 2014).

The SmartSantander testbed aims at supporting experimentation with smart city services in a realistic setting at a large scale, comprising a heterogeneous number of devices, protocols, and services as well as covering different domains of the city, such as environmental monitoring, parking space monitoring, gardens precision irrigation, augmented reality for Points of Interest in the city through NFC tags, and participatory sensing. To this end, the project implements an architecture composed of three tiers: *IoT node tier* for operation and deployment of IoT devices around the city; *IoT Gateway tier* to link IoT devices at the edges of the network to a core network infrastructure; *Server tier* based on cloud computing to host data and high-level IoT services.

The main contributions of SmartSantander project are regarding the proposed architectural reference model for real IoT experimentation and the lessons learned in deploying a city-scale IoT infrastructure, primarily concerning practical challenges in managing, maintaining, and handling such large number of heterogeneous devices. Although the SmartSantander's *server tier* provides access to the underlying IoT infrastructure of the city, the project does not provide middleware facilities for application development, such as storing, processing, and analyzing generated data.

Despite the relevance of the SmartSantander project to enable experiments in real scenarios, it is not clear whether it meets important requirements that would allow its applicability in other contexts (e.g., in cities with different characteristics), such as adaptability, flexibility, and extensibility. The city of Santander has less than 200,000 inhabitants; we did not find any discussion on the use of this platform in more complex contexts, such as in large metropolises. Also, to the best of our knowledge, no evaluation of the scalability of the SmartSantander platform has been published yet.

In this sense, Cheng et al., 2015 proposed the CiDAP, a big data analytics platform for smart cities deployed in the SmartSantander testbed. The main objective of this platform is to use the data collected from the testbed and analyze it to understand the city's behavior. Figure 2.2 presents the overview of CiDAP's architecture as the middle layer between underlying data providers and smart city applications.

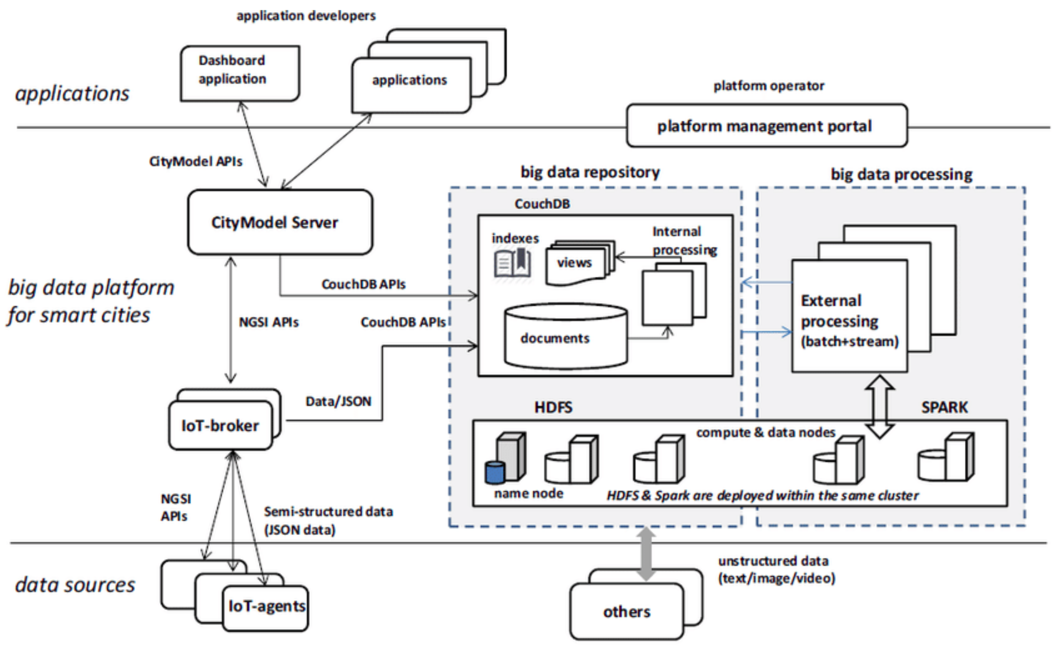


Figure 2.2: *Architecture of the CiDAP platform (extracted from (Cheng et al., 2015))*

The *IoT Broker* is the component responsible for collect data from different formats and forwards to the *Big Data Repository*, which may perform simple processing tasks to transform data before store it. However, the *Big Data Processing* is the component responsible for performing more complex and intensive data processing, such as data mining and aggregations. Finally, CiDAP offers a REST API through the *CityModel* component to enable external applications to fetch generated results from data processing. In general, CiDAP combines a set of Big Data tools, such as Apache Spark and NoSQL databases, to improve the scalability of the platform and to be able to deal with historical data, near-time data, and also real-time data.

To evaluate the performance of the proposed architecture, the CiDAP authors performed a microbenchmark in a small setup composed of few machines in the same local network, which does not truly represents the production environment in which such system should operate. They focused on evaluating the performance of their *Big Data Repository* component in handling simultaneous updates and queries on sensors data. The experiments showed that CiDAP could handle 470 simple queries/s from external applications when launching 8 *CityModel* parallel instances and considering a static database with predefined views (without live updates from Santander testbed). Their experiments also showed that the *Big Data Repository* could handle almost 300 sensor data updates per second, which was measured with an empty NoSQL database. Therefore, the authors failed to evaluate their platform scalability considering a city-scale realistic scenario and combining the interaction of both applications and underlying IoT layer with the platform. Such interactions could produce different performance results in both queries and updates operations, evidencing possible points of failure and bottlenecks in their architecture.

As a middleware layer, the CiDAP project aims at sharing a common Big Data infrastructure among many smart city applications to enable the processing of data from sensors around the city. However, the project fails to handle actuators and to provide more sophisticated context-awareness features. Besides, it is unclear what components they have developed to allow integration of the Big Data tools used and what impact these components have on the system architecture in terms of performance, adaptability, and evolvability.

FIWARE

The FIWARE[6] project is an initiative funded by the European Commission that received large investments (more than EUR 100 million) through a public-private partnership. It aims to build an open source software platform to ease the development of new smart applications, including in the context of smart cities, by bringing together several isolated European projects to develop solutions for the Future Internet. To this end, the project specified a reference architecture composed of Generic Enablers (GE), which are software components that implement general-purpose functions that can be combined through open APIs to provide middleware facilities for application development. Figure 2.3 shows FIWARE architecture overview presenting its GEs.

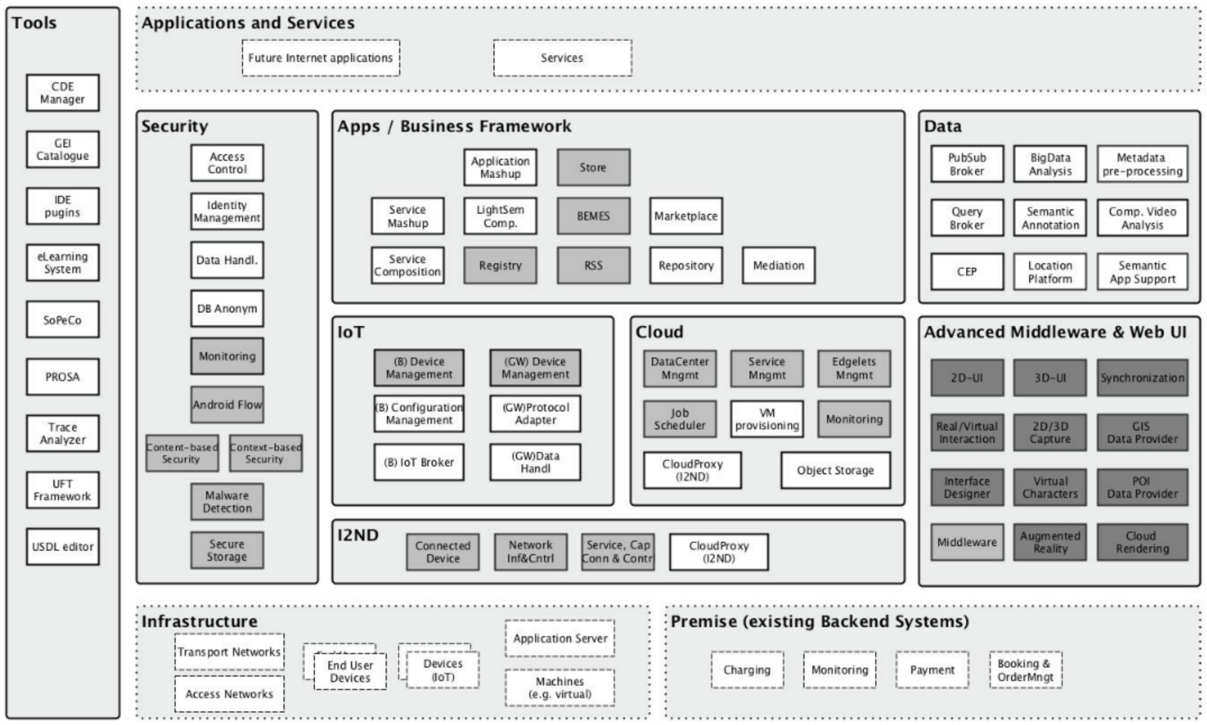


Figure 2.3: *FIWARE architecture overview*

[6]https://www.fiware.org/

FIWARE offers generic open specification for each of its GEs with essential functionalities, interfaces, and APIs along with an open-source solution that implements such specifications. One of the main advantages of FIWARE approach is to use open standards, avoiding vendor lock-in and allowing the alternative implementations of its GEs. Most of its components have a REST API compatible with the NGSI[7], a RESTful specification to exchange context information. FIWARE categorizes its enablers as follows:

- **Data/Context Management:** components that aim at storing, gathering, processing, and analyzing data at large scale. Currently, the project offers 10 different GEs of this category. The most notable implementations of this category are:
  - Orion[8]: implements the Publish/Subscriber Context Broker GE based on NGSI specification.
  - Cosmos[9]: implements the Big Data Analysis GE
  - Proactive Technology Online[10]: implements the Complex Event Processing (CEP) GE
- **Internet of Things Services:** components that provide the technical means to enable sensor networks and routing sensor data to other GEs by adapting IoT protocols and management functionalities for the devices. Thus, these GEs are usually hosted on edge devices or IoT gateways. Currently, the project offers 6 GEs of this category. The most notable implementations of this category are:
  - IoT Broker[11]: implements the IoT Broker GE that aims at separating IoT applications from the underlying device installations
  - IDAS[12]: implements the Backend Device Management GE that aims to be a bridge between different IoT and Web protocols
- **Advanced Web-based User Interface:** components to design and develop user interfaces including geographical information and 3D charts. This category is composed of 13 enablers which are software engineering tools and libraries to support the development of front-end applications, such as the following implementation:
  - Geoserver 3D[13]: implements the GIS Data Provider GE which is able to host geographical data and serve it in 3D form to both mobile and web clients

[7]Next Generation Services Interface
[8]https://catalogue.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker
[9]https://catalogue.fiware.org/enablers/bigdata-analysis-cosmos
[10]https://catalogue.fiware.org/enablers/complex-event-processing-cep-proactive-technology-online
[11]https://catalogue.fiware.org/enablers/iot-broker
[12]https://catalogue.fiware.org/enablers/backend-device-management-idas
[13]https://catalogue.fiware.org/enablers/gis-data-provider-geoserver3d

- **Security:** components to define and enforce declarative security and privacy. This category has 6 GEs available. The most notable implementations of this category are:
  - Keyrock[14]: implements the Identity Management GE covering a large number of aspects involving users' access, including authentication from users to device, authorization and trust management, user profile management, and Single Sign-On (SSO)
  - AuthZForce[15]: implements the Access Control GE that provides an API to get authorization decisions based on authorization policies, playing a role of a Police Decision Point (PDP)
- **Applications/Services and Data Delivery:** components and tools for data visualization, easy generation of mashups and app-store-like distribution of services and data. This category is composed of 8 GEs. The most notable implementation of this category is:
  - Wirecloud[16]: implements the Application Mashup GE that aims at allowing end users without programming skills to easily create web applications and dashboards/cockpits to visualize their data
- **Cloud Hosting:** components that aim at providing FIWARE services via cloud infrastructure. This category has 9 GEs. The most notable implementations of this category are:
  - OpenStack[17]: implements the Cloud IaaS GE that provides the facilities to provision virtual machines, as well as to associated compute, storage and network resources
  - Murano[18]: implements the Application Management GE that provides a ready-to-use catalogue of applications that can be deployed in a OpenStack[19] cloud infrastructure

While some components have been developed within the context of the project, other consolidated open source projects have been adopted as the reference implementation for some GEs, such as Open Stack, CKAN[20], and Docker[21]. There are some previously work in the literature that exploited FIWARE GEs components in various domains(Cola et al., 2015; Salhofer, 2018; Steinmetz et al., 2017; Bellabas et al., 2013). These works usually combine a small set of GEs to provide the basis for the development of different prototype applications for end users. Such projects demonstrate that smart applications can benefit from FIWARE components as reusable

[14]https://catalogue.fiware.org/enablers/identity-management-keyrock
[15]https://catalogue.fiware.org/enablers/authorization-pdp-authzforce
[16]https://catalogue.fiware.org/enablers/application-mashup-wirecloud
[17]https://catalogue.fiware.org/enablers/iaas-ge-fiware-reference-implementation
[18]https://catalogue.fiware.org/enablers/application-management-murano
[19]https://www.openstack.org
[20]https://github.com/ckan/ckan
[21]https://www.docker.com

building blocks. However, these works also evidence the technical disparity between the different GEs in terms of technical quality, ease of integration, documentation, and provided functionalities. For instance, the implementation of the FIWARE Context Broker GE, Orion[22], is the most used component of FIWARE architecture, while several other components are rarely used.

To the best of our knowledge, no papers are reporting the use of FIWARE in its entirety, which makes it difficult to understand its functional capabilities and project scope as a platform for smart cities. Even though on the one hand the proposed reference architecture can meet several requirements as a middleware platform, on the other hand, there is little evidence to show that its implementation comes with the claimed flexibility and interoperability. We have found only one previous work that evaluated FIWARE performance and scalability properties through experimental evaluation (Pereira et al., 2018). The authors focused on evaluating the publish-subscribe communication model provided by FIWARE considering a large dataset and a simple setup, which is basically a running instance of the Orion component. Accordingly to their results, FIWARE performed significantly well in comparison to another IoT middleware (OneM2M/ETSI M2M). However, as observed in other studies, the authors only considered a small part of the FIWARE platform (one reference implementation of its GE), which considerably limits conclusions regarding the platform performance and scalability as a whole.

The modularization of FIWARE architecture into systems that cooperate through web standards indicates the adoption of a Service Oriented Architecture (SOA). Although the division of categories helps in understanding the architecture proposed by FIWARE, a large number of options to compose an instance of the platform and the overlapping of existing functionalities among several components make its architecture extraordinarily complex. In addition, although the community provides Docker images for each of the components to facilitate their deployment and running environment, there is a considerable manual effort required to compose them (Cola et al., 2015). Besides, several implementations of FIWARE GE are deprecated, and there are no alternatives provided by the project community. Therefore, FIWARE can be considered more as a reference architecture that indicates the main building blocks for smart city platforms, proposing open source implementations for each of these blocks, than a functional, integrated platform for smart cities that can properly support the development of applications in various contexts.

DIMMER

Krylovskiy et al., 2015 presented the DIMMER[23] project whose goal is to build a service platform and applications to increase energy efficiency of a city at the district level. Within the context of DIMMER project, they proposed the DIMMER platform, a microservice-based IoT platform to support applications that aim at improving the energy efficiency and management in cities. DIMMER is a key related work in the context of this masters research since this is the only previous work we have found that explores the use of a microservices architecture to build smart city platforms.

[22]https://github.com/telefonicaid/fiware-orion

[23]District Information Model and Management system for Energy Reduction

Among other needs and challenges inherent to the design of smart city platforms, the DIMMER's authors highlight the following points as the main motivators for their adoption of the microservice architecture:

- The IoT infrastructure required to realize the vision of future smart cities comprises a large variety of services with varying requirements, imposing the need for smart city platform architectures able to support new services and standards in the future
- Earlier successful IoT commercial and academic projects have successfully adopted modern Web and cloud standards to provide scalable distributed systems, demonstrating several befits of the IoT and cloud integration
- Microservices architecture has already been used in other contexts than smart cities, becoming the industry standard for building large-scale systems such as Netflix[24], Uber[25], and Spotify[26]

DIMMER platform aims at supporting a variety of Web, mobile, and desktop applications for different stakeholders and users. Figure 2.4 shows a high-level representation of the DIMMER platform's architecture, highlighting the set of its most important microservices and encompassing underlying sensor technologies and district information models. DIMMER microservices are divided into two categories: Middleware Services and Smart City Services. The former include services to provide modeling abstractions of underlying IoT devices, services to discovery city resources , and services to access historical and (near) real-time sensor data. The latter comprise services that implement the core functionalities of the platform based on GIS[27] and BIM[28] models, such as Energy Efficiency Engine, Energy Data Simulator, and Context Awareness Framework.

DIMMER architecture implements a set of microservices design guidelines to achieve a loosely-coupled, evolvable architecture. To provide convenient high-level APIs for different customers, DIMMER used the API Gateway[29] design pattern to hide the communication details of background microservices and to offer a single entry point for its facilities. They also explored the benefits of decentralized governance of microservices by adopting different standards, policies, and technologies that are more suitable for their job. Consequently, each microservice manages its data models backed by distinct storage database systems and may also communicate through different protocols (i.e., HTTP or MQTT).

Among other contributions regarding their early experience in adopting microservices, Krylovskiy et al., 2015 also covered the impact of microservices architecture on practical aspects. They highlighted that defining the service interfaces is the fundamental step to enable decentralized development and governance of microservices. Also, they observed a clear need for applying DevOps techniques to achieve a sustainable, standardized operational process to deploy the entire DIMMER platform.

[24]https://netflix.com
[25]https://www.uber.com
[26]http://spotify.com
[27]Geographic Information Systems
[28]Building Information Models
[29]http://microservices.io/patterns/apigateway.html

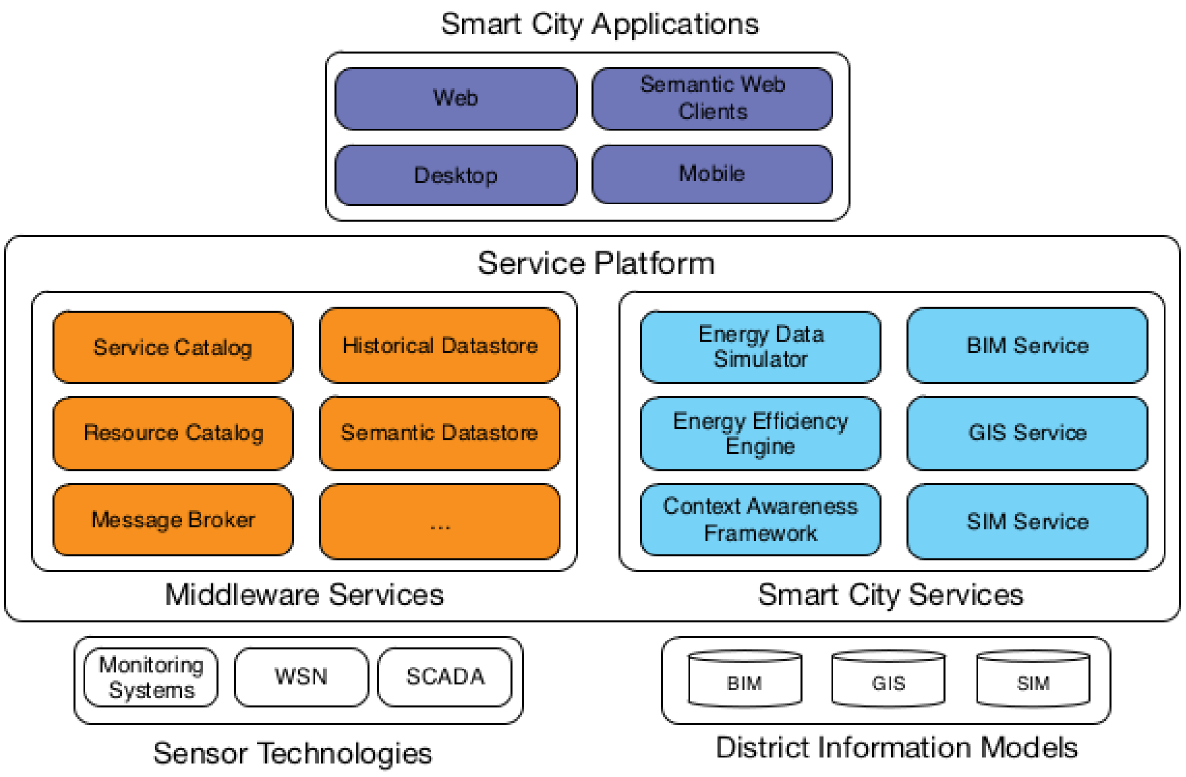


Figure 2.4: *DIMMER architecture overview*

The DIMMER project brings significant contributions for smart city platforms due to its pioneering experience in adopting microservices in the context of smart cities. However, their work leaves several open questions that demand further experimental and empirical studies to obtain more conclusive results on the adoption of microservices in the design of smart city platforms, such as on issues related to scalability and perfomance. Technically, we could not take advantage of their advances since DIMMER is a proprietary software project. Our work has a broader scope, as the InterSCity platform we are proposing in this research is designed to support smart city applications from multiple domains and is fully developed as an open source project, as opposed to the DIMMER platform.

The projects discussed above present tools and architectures to design smart city platforms. However, most of the authors failed to either provide a thorough explanation of how their architectures solve the scalability problems or perform scientific validation of the scalability of the proposed platforms. Moreover, the widespread use and development of closed source software limit the analysis of scalability and exploratory work by the research community to just considering the discussions that were presented in the original papers. To tackle these problems, this research deepens the debate about the use of a microservices architecture on smart cities through the open-source InterSCity platform. Importantly, we also present reproducible scientific experiments to demonstrate the scalability of the platform. Table 2.1 summarizes the features provided by the mentioned platforms in comparison to InterSCity, where **NF** stands for Near Future and **NC** means that it is Not Clear whether the platform supports it.

Table 2.1: *Smart City Platforms' Features*

Feature	Civitas	Gambas	Almanac	Open IoT	Smart Santander	Fiware	**InterSCity**
Data Management	x	x	x	x	x	x	x
Data Processing	x		x	x	x	x	NF
IoT Integration	x			x	x	x	x
Actuation Support			x			NC	x
Context Awareness		x	x	x		x	x
City Abstractions		x					x
SE. Tools		x	x	x			
Specific-domain Services						x	
Cloud-native			x	x	x	x	x
Scalability			x		x		x
Interoperability	x		x	x	x	x	x
Security	x	x	x	x	x	x	
Privacy	x	x	x		x	x	
Adaptation			x				x
Evolvability	x	NC		NC		x	x
Production Deployment					x	NC	NF
Experimental Evaluation							x
Free/Open Source				x		x	x

Chapter 3

The InterSCity Platform

Future smart cities will demand high-quality research in multiple areas. The InterSCity project (Batista et al., 2016) is carried out by a multidisciplinary consortium that aims to develop scientific and technological research to address key challenges related to the software infrastructure of smart cities, focusing on Networking and High-Performance Distributed Computing, Software Engineering, and Analysis and Mathematical Modeling for the Future Internet and Smart Cities. The project started in the end of 2016, bringing together researchers from different areas aiming at enabling the development of reusable open-source technologies and methods to support future smart cities while advancing the state-of-the-art in the field.

This chapter presents the InterSCity platform[1], an open-source, microservices-based middleware to support the development of smart city applications and to enable novel, reproducible research, and experiments in this field. The platform was designed from the outset following the reference architecture for smart city platforms proposed in (Santana et al., 2017). This reference architecture aims at guiding the development of next-generation smart city platforms by describing and organizing the major building blocks required to meet the wide range of functional and non-functional requirements described in Section 2.1. By implementing key building-blocks of this architecture, the InterSCity Platform covers the major features required to support integrated smart city applications in different domains, such as public transportation, public safety, and environmental monitoring. Currently, the InterSCity Platform provides a set of high-level cloud-native services to manage heterogeneous IoT resources, data storage and management, and context-aware resource discovery.

3.1 Design Principles

The integration of the smart city enablers, such as the Internet of Things, Big Data, and Cloud Computing, is not straightforward, as these areas evolve rapidly and their combination raises complex issues. This brings new challenges, approaches, and dynamics that result in the constant emergence of novel technologies, standards, and services. Smart city applications further enhance the dynamism of the involved technologies, since they encompass complex environments composed of

[1]The source code is available online on https://gitlab.com/smart-city-software-platform under the MPL 2.0 license

several interconnected subsystems which are constantly evolving and presenting new challenges. Such dynamism impacts several design decisions in smart city platform architectures. In particular, we intend to address scalability and evolvability issues in smart city platforms research and development.

Here, we present the design principles adopted in the InterSCity Platform which are aligned with microservices patterns, which are critical for the wide adoption of the platform in different smart city projects.

- **Modularity via Services.** Modularity is a key concept used in software architecture to divide systems into smaller functional units. Microservice architectures achieve modularity through single-purpose, small services that communicate through lightweight mechanisms to achieve a common goal.

- **Distributed Models and Data.** In monolithic architectures and even in traditional service-oriented systems, it is fairly common to create a unified domain model and a centralized storage backend. With microservices, each service has its own database and models, which may evolve independently of external services. Decentralized data management and the possibility to use different technologies that best fit each context are relevant advantages. On the other hand, increased operational complexity is the main drawback.

- **Decentralized Evolution.** Microservices must be autonomous, providing well-defined boundaries and communication APIs so that they can evolve and be maintained independently. Moreover, this principle ensures that each service may implement its functionalities using the most appropriate technology, provoking positive *technology heterogeneity*. Similarly, each microservice may *scale independently* using different strategies, since scalability requirements vary across services. Finally, this design principle should reflect on the configuration and deployment procedures, which may be performed independently as well.

- **Reuse of Open Source Projects.** Reusing software components is a fundamental practice of software engineering to achieve productivity, cost effectiveness and software reliability. We always give preference to the use of existing robust open-source tools, libraries, and frameworks instead of implementing components from scratch, since the quality of popular open-source packages is admittedly good as already empirically observed (Taibi, 2013). Moreover, we adopt a rigorous technology selection criteria and only incorporate open source components that have an active developer community, stable release support, and appropriate documentation to guide usage, development, and deployment.

- **Adoption of Open Standards.** As important as the reuse of open-source projects is the adoption of open, well-accepted standards that are designed to provide interoperability at different levels. This prevents technology and vendor lock-in. The use of open Internet and web standards is essential to enable the *true* Internet of Things, being widely used in related projects found in the literature (Fazio et al., 2012; Amaral et al., 2015; Hernández-Muñoz et al., 2011).

- **Asynchronous versus Synchronous.** Although most services provide RESTful APIs, they must implement asynchronous messaging as much as possible to avoid blocking in synchronous request-reply interactions. Asynchronicity should be achieved by using notifications, the publish/subscribe design pattern, and event-based communication strategies to support low latency and scalability. Besides, the platform must rely on a lightweight message bus with the single purpose of providing a reliable messaging service rather than traditional SOA approaches that use sophisticated, heavy middleware such as an Enterprise Service Bus (ESB).
- **Stateless Services.** This design principle supports scalability by advocating that services should be stateless to enable any service instance to respond to any request, facilitating load distribution and elasticity. Thus, the design of microservices should separate state data, such as context and session data, to be managed by an external component whenever possible.

During this masters research, we applied the above design principles aiming at meeting a scalable architecture. Moreover, these principles were fundamental to achieve an evolvable architecture that can be further improved.

3.2 Platform Architecture

The InterSCity microservices architecture is shown in Figure 3.1, addressing important aspects of IoT and Data management, providing high-level RESTful services to support the development of smart city applications, services, and tools for different purposes. The underlying IoT Gateways can register new devices to the platform and send sensor data through a REST API. InterSCity abstracts the complexity involved in the communication between smart city applications and IoT devices, as well as the complexity of city-scale data management.

InterSCity provides well-defined boundaries to communicate with both IoT devices and smart city applications. Currently, the platform is composed of six different microservices that provide features for the integration of IoT devices (Resource Adaptor), data and resource management (Resource Catalog, Data Collector, and Actuator Controller), resource discovery through context data (Resource Discovery) and visualization (Resource Viewer). Although all microservices expose REST APIs for synchronous messaging over HTTP, most of the communication for the composition of services is done through asynchronous calls, relying on the lightweight message bus RabbitMQ[2] for asynchronous messaging through the Advanced Message Queuing Protocol[3] (AMQP).

Interactions between the platform and its clients involve the manipulation of *city resources*. A city resource is a logical concept that encapsulates a physical entity that makes up the city, such as cars, buses, traffic lights, and lampposts. Resources comprise attributes (e.g., location and description) and functional capabilities to provide data and receive commands, which are respectively supported by sensors and actuators

[2]www.rabbitmq.com
[3]www.amqp.org

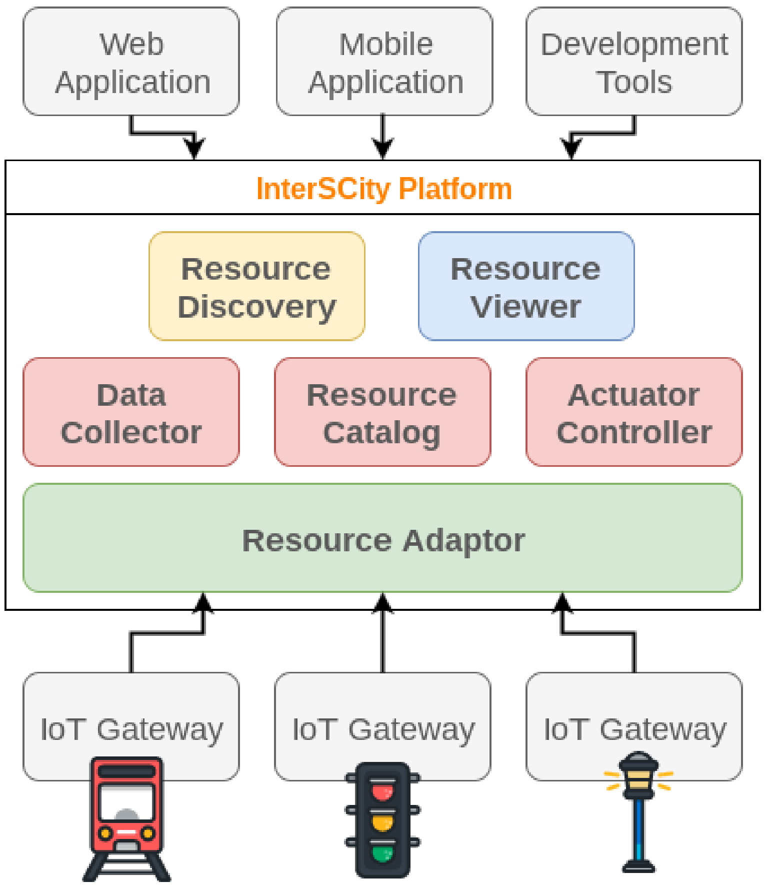


Figure 3.1: *The InterSCity Platform Architecture.*

coupled to the resource. This approach facilitates the interaction of client applications with a real city environment since it grants an abstraction composed of city concepts rather than the cyber-physical particulars, that comprise the underlying IoT layers of Smart City ecosystems. As a consequence, for instance, two buses registered in the platform are accessed through the same standardized API regardless of their devices and communication technologies. Likewise, two physical sensor or actuator devices with similar purposes are encapsulated as a common capability, abstracting all the specific details related to data representation and deployment of these devices. Such strategy could be compared to the duck typing strategy used by modern programming languages, such as Python and Ruby[4].

Similarly, the underlying IoT Gateways must use the *city resource* and *capability* abstractions. Thus, define how to model a resource and its capabilities is one of the responsabilities of IoT Gateway developers. For example, in some cities, there are buses equipped with GPS and other sensors that can monitor their speed, current occupation, and internal temperature. If we model each bus as a city resource, we have two different possibilities to model its capabilities. First, we could define four different capabilities without any relation between this information, except for belonging to the same resource. This strategy is suitable especially if the data is collected and sent at different time intervals. The second strategy could be register a single generic

[4]https://en.wikipedia.org/wiki/Duck_typing

capability (i.e., *bus_monitoring*) that encompasses the data from the four sensors, creating a temporal link between them. This approach enables client applications to retrieve all the state of the bus at any given time easily since it does not require further efforts to define temporal correlation. Also, it boosts the publishing and using of resource context data rather than raw sensor data. Both strategies could be used for data that is collected from the occurrence of events rather than periodically, depending on the amount of information that is read in each event. For instance, a bus could trigger an event when it breaks, informing its location and occupation through a single capability (i.e., *bus_failure*) since this two information is useful for city managers to take contingency actions to supply the demand of the broken bus.

Although the city resources and *capabilities* abstractions provided by the InterSCity platform are simple, they introduce a flexible semantics that can encompass a large number of services, sensors, and actuators. However, such flexibility also may lead to the proliferation of ambiguous and redundant capabilities. In this sense, we have prepared a guide on how to model the city's resources and define the names of available capabilities. This guide is handy online in the InterSCity's documentation repository[5].

3.3 Design Details to Address Scalability

Scalability is a key non-functional requirement for smart city platforms. InterSCity was primarily designed to support smart city with a large number of users, data, and services. InterSCity leverages the microservices architecture as the main strategy to achieve scalability due to four fundamental reasons, which we will explore in more detail later:

- **Functional Decomposition** - the microservices architecture promotes modularity via single-purpose, small services that communicate through lightweight mechanisms to achieve a common goal. Also, functional decomposition leads to scalability as the system workload is split among distributed microservices;
- **Communication Style** - microservices cooperate through well-defined communication mechanisms and provide cohesive APIs for external access;
- **Design and Technology Heterogeneity** - different design and technology decisions can be made for distinct microservices, although they do not exclude general choices that consider the whole system; and
- **Independent Deployment** - each microservice can be deployed, replicated, and replaced independently.

3.3.1 Functional Decomposition

By implementing a microservices architecture, the InterSCity platform decomposes its functionalities across a set of small, interconnected, collaborative services. The InterSCity architecture supports decentralized management of city resources dividing

[5] https://gitlab.com/smart-city-software-platform/docs

functional responsibilities and data persistence across the microservices. The Resource Adaptor is a proxy microservice that simplifies the communication of external IoT systems with the rest of the platform. It provides a single entry point for the underlying IoT gateways to register and update resources on the platform, post sensed data from those resources, and subscribe to events that indicate actuator commands. As a proxy, Resource Adaptor is responsible for validating the requests, augmenting them with additional metadata and required adaptations. The adaptor either broadcasts the requests to the entire system or calls a specific microservice to handle the request synchronously.

The Resource Catalog is a vital microservice of the InterSCity architecture since it manages the static data of city resources, working as a catalog for these resources. After registering a new resource, the Resource Catalog assigns it a new Universally Unique IDentifier (UUID) (Leach et al., 2005) and asynchronously notifies the event of resource creation to other microservices. Both client applications and IoT gateways must use the UUID in later interactions that target a specific resource, such as to get and publish sensor data.

City resources may have functional capabilities to provide data and receive commands, which are respectively supported by sensors and actuators coupled to the resource. InteSCity splits the management responsibilities of sensors and actuators between two microservices: Data Collector and Actuator Controller. The Data Collector microservice stores sensor data collected by city resources. Sensor data consist of context information or an event linked to a resource capability which is observed at a particular time. Data Collector provides an API to allow access to both current and historical context data of city resources using a rich set o filters that can be accessed in search endpoints. The Actuator Controller microservice, in turn, provides standardized services to intermediate all actuation requests to city resources with actuator capabilities. It is responsible for receiving and validating actuator requests from clients and asynchronously notifying the underlying IoT gateway through Webhooks. Also, this microservice records the history and tracks the status of actuation requests that are available through its API, e.g., for auditing purposes.

Both the Resource Discovery and the Resource Viewer microservices provide more sophisticated services by orchestrating Data Collector and Resource Catalog. Resource Viewer is a web visualization microservice for presenting city resources information graphically based on Resource Catalog and Data Collector back-end services. The purpose of Resource Viewer is to present general and administrative visualizations of city resources, including location, real-time context data, and representative charts of historical data. The Resource Discovery microservice provides a context-aware search API that may be used by client applications to discover available city resources. This API provides filters that can be combined to discover resources. For instance, filters can combine information such as location data, interval rules for current context data, and other types of meta-data. More details of each microservice are presented later in this chapter.

As a result of its functional decomposition, the workload of InterSCity is handled by separate distributed services, cooperating for its scalability. The partitioning of the system workload is also reflected on the database tier since each microservice has its database and manages a small set of the system's data. Different smart city contexts may lead to different ways of splitting the workload of the platform among its microservices in a non-uniform way. For instance, in a city with a large number of sensors, two microservices of the InterSCity platform may be used more often: Resource Adaptor and Data Collector. In another type of scenario, a city with actuator resources might have management systems that send actuation commands to the underlying IoT devices, thus placing a higher demand on Actuator Controller.

3.3.2 Communication Style

The InterSCity microservices need to cooperate constantly to provide the services for client applications and enable the integration of underlying IoT devices and services. Design decisions regarding the communication mechanisms of the system is a major concern to the system scalability since they may impact on the overall performance. Decisions on how microservices interacts with the whole ecosystem of the platform may comprise the choice of communication protocols, the discovery of other services, and the handling of asynchronous messages, to name a few.

InterSCity implements two fundamental communication mechanisms to support the scalable orchestration of its microservices: (1) Synchronous communication over HTTP and (2) Asynchronous messaging through the AMQP[6].

For HTTP communication, we applied the *API Gateway* design pattern[7] by using Kong[8], a distributed, scalable open-source gateway that aims at facilitating the orchestration of microservice APIs. Kong is backed by NGINX[9] and extends its HTTP Web server features to offer a dynamic management layer for the microservices HTTP APIs. In summary, the API Gateway receives all incoming HTTP requests, determines which InterSCity microservice should respond to a specific request, and forwards the request to the identified microservice. For this purpose, we use URI-based rules, where the system uses the path part of HTTP requests to identify the target microservice of a request. Figure 3.2 illustrates this strategy, highlighting the interaction of a client application and an IoT Gateway with the platform through the HTTP APIs of its microservices, which in turn are accessed via the API Gateway.

A clear advantage of implementing the above-mentioned API Gateway strategy is that clients only need to interact with a single entry point (host address and port) to access all InterSCity facilities, instead of keeping references to multiple microservices.

The API Gateway also enables scalability as it leverages the load balancing feature of NGINX so that it is possible to deploy multiple instances of the same service easily. Figure 3.3 illustrates this by presenting the main interactions among clients, the API Gateway, and the available microservice instances. Whenever we run a new instance of a microservice, the microservice uses Kong's REST API, which is an implementation of the *Self-registration* design pattern[10], to register itself with Kong as a target for an existing URI rule. In the example, distinct Resource Catalog instances register themselves as targets for requests that match the */catalog* URI rule, indicated by the

[6] https://www.amqp.org
[7] http://microservices.io/patterns/apigateway.html
[8] https://getkong.org
[9] https://www.nginx.com
[10] http://microservices.io/patterns/self-registration.html

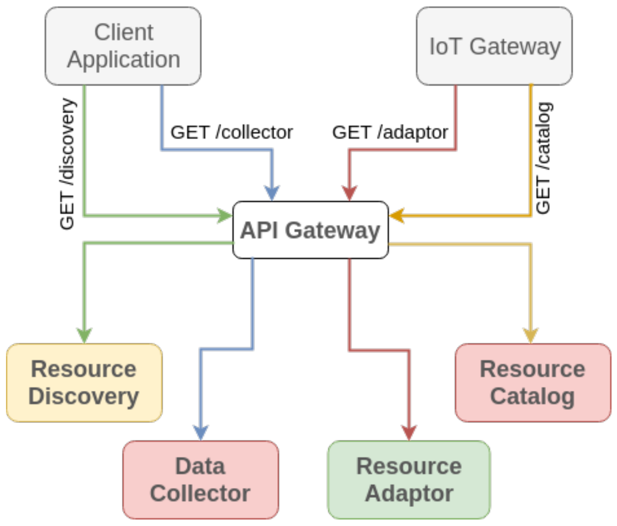


Figure 3.2: *InterSCity API gateway*

black arrows. This approach enables the dynamic creation of new instances of a service to split and distribute the increasing workload using a round-robin strategy, without affecting other parts of the system or requiring further reconfiguration. Round-robin load balancing is represented by the blue and yellow arrows in Figure 3.3.

The API Gateway constantly monitors the availability of target instances by health checking the endpoints in order to adjust its load balancing accordingly by not sending new requests to unhealthy nodes. It identifies microservice instances as healthy or unhealthy based on the status code in HTTP responses. A success code indicates a healthy endpoint, whereas non-success status codes, timeouts and TCP errors denote an unhealthy endpoint.

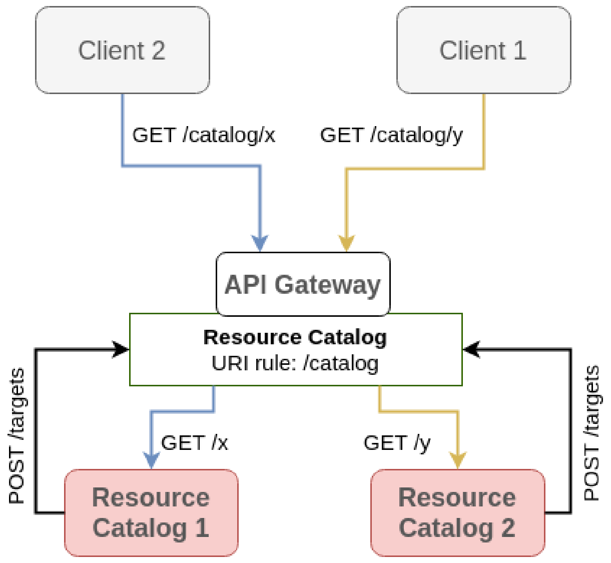


Figure 3.3: *InterSCity load balancing HTTP requests*

As Kong allows the registration of service instances, the API Gateway also supports the *Server Side Discovery* design pattern[11]. This pattern facilitates HTTP-based communication among microservices in the platform since instead of having to manage references to all running instances, clients need to know only the address of the API Gateway, using URI-based rules to communicate with the target microservice, as shown in Figure 3.2. A direct advantage of using the `API Gateway` design pattern is the guarantee that requests for a given microservice will be handled by any of its instances in an independent way, allowing for efficient load-balancing. This is an important feature since the number of service instances and their locations may change dynamically.

Although services provide well-defined RESTful APIs, we adopt asynchronous communication whenever possible to avoid the additional latency of blocking synchronous request-reply interactions. In this context, a microservice may use the publish/subscribe design pattern rather than directly exchanging messages with other services. InterSCity carries out asynchronous messaging by using RabbitMQ[12], a widely used lightweight, open-source messaging middleware that implements the AMQP protocol.

Events of different types may generate data that need to be broadcast to other modules in the platform. Examples are the registration of a new IoT resource, the reception of sensor data, and requests to actuators. Each type of event is mapped to a *topic*, to which interested services can subscribe to receive new messages through queues maintained by RabbitMQ. By default, each subscription will create a new queue. Messages sent to a specific topic are pushed to all of its subscribed queues.

InterSCity thoroughly exploits RabbitMQ messaging features by using routing keys that enable the use of more sophisticated criteria to route messages to subscribers. A routing key is a list of dot-separated words related to metadata about InterSCity's internal abstractions. They are added by publishers when sending messages to RabbitMQ. When a city resource publishes sensor data on the platform, a new message is published on topic **data_stream** with a routing key of the form **uuid.capability.others**, where *uuid* is the unique identifier of the resource, the *capability* is the type of the sensor data, and *others* represent any additional information regarding the posted sensor data that could also be used to build the routing key. In this sense, subscribers must inform a binding key when defining a queue to receive messages from a topic.

Binding keys may use wildcard characters based on two special characters, as explained in RabbitMQ's documentation[13]:

- *** (star)**: can substitute for exactly one word.
- **# (hash)**: can substitute for zero or more words.

Figure 3.4 shows an example of an InterSCity topic-based message exchange using routing keys. When the Resource Adaptor receives sensor data from the underlying IoT infrastructure, it publishes the data on topic *data_stream*, which has consumer queues for Data Collector, Resource Catalog, and other services interested on the topic.

[11] http://microservices.io/patterns/server-side-discovery.html
[12] https://www.rabbitmq.com
[13] https://www.rabbitmq.com/tutorials/tutorial-five-python.html

As Data Collector is responsible for saving the history of sensor data, it is interested on any data on this topic (binding key: #). On the other hand, Resource Catalog is only interested in sensor data that contain geolocation information so that it can update the location of moving city resources on its database (binding key: **#.location.#**). Other microservices could be interested in any data from a specific type of sensor, such as temperature data (binding key: ***.temperature.#**), regardless of which city resource provided the data.

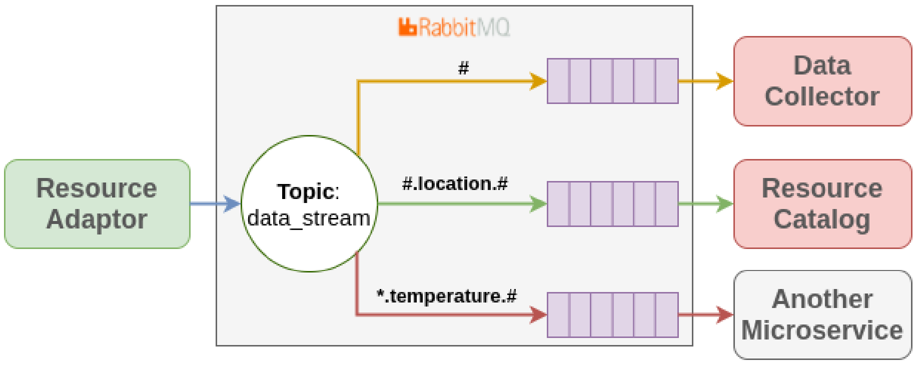


Figure 3.4: *InterSCity Asynchronous messaging*

Additionally, to decrease the latency of communication among microservices, InterSCity leverages asynchronous messaging to support scalability by adopting a worker-based strategy. InterSCity services that need to receive asynchronous messages from the platform must implement a background worker for each subscribed topic on RabbitMQ. By design, these services support the addition of more workers to improve the processing rate of the queued jobs by implementing the *Competing Consumers* design pattern (Hohpe and Woolf, 2003). Figure 3.5 presents this design pattern, where additional workers read messages from the same queue concurrently and as fast as possible, enabling parallel processing of background tasks. As workers read from the same queue, messages are not replicated for each of them. Consequently, in the example, messages **X** and **W** will be processed in this order by the first two workers that finish their current jobs, represented by messages **Y**, **Z**, and **A**.

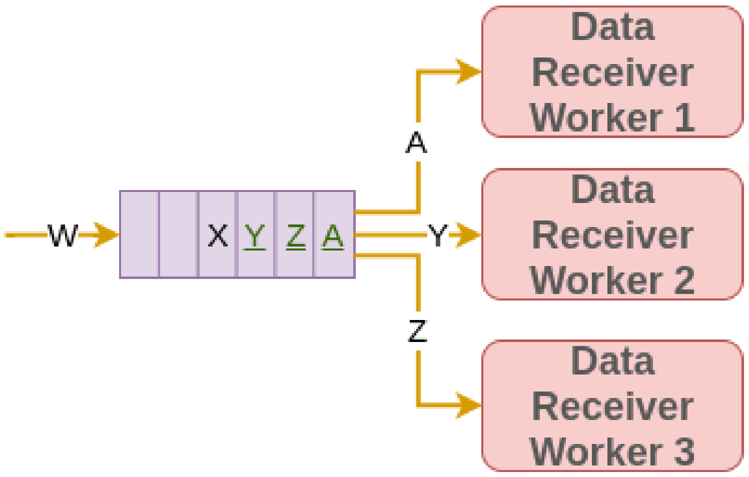


Figure 3.5: *Competing Consumers design pattern*

In addition to scalability, the above design decisions regarding InterSCity communication style led to the implementation of a flexible and evolvable architecture. The loosely coupled message-oriented communication approach favors the continuous development of the platform, enabling the extension of existing features through service composition and the addition of new microservices to meet the constantly evolving smart city requirements. Moreover, decoupled communication interfaces allow us to maintain microservices in separate code repositories enabling: decentralized version and dependency management, independent, faster tests, safe refactoring, the evolution of existing features, and the adoption of the most appropriate technologies in each context. The lower boundary provided by Resource Adaptor enables InterSCity to integrate heterogeneous IoT technologies without affecting other services continuously. It can also be used to incorporate the existing legacy ICT infrastructure of cities, such as open data initiatives. It is worth noting that InterSCity' upper API isolates client applications from the modification or addition of new technologies in the smart city infrastructure.

3.3.3 Design and Technology Heterogeneity

Since microservices communicate via standardized protocols, each microservice can be built with the most appropriate stack of technologies for its purpose. They can also be maintained in separate repositories, enabling polyglot persistence and technology diversity. With this in mind, Table 3.1 shows the technologies used by InterSCity services.

Table 3.1: *Technology Stack of InterSCity Microservices*

Microservice	Language	Framework	Database	Cache
Actuator Controller	Ruby	Ruby on Rails	MongoDB	
Data Collector	Ruby	Ruby on Rails	MongoDB	In-memory MongoDB (Percona)
Resource Adaptor	Ruby	Ruby on Rails	PostgreSQL	Redis
Resource Catalog	Ruby	Ruby on Rails	PostgreSQL	Redis
Resource Discovery	Ruby	Ruby on Rails		
Resource Viewer	Javascript	EmberJS		

Although we embrace the diversity of technologies in InterSCity, we apply this principle with some caveats to avoid increasing overall system complexity, and the proliferation of practical problems since a radical adoption of technology diversity could lead to an unmaintainable architecture (Balalaie et al., 2016). For this reason, we initially opted to use Ruby-based tools to develop the fundamental services of the platform due to the high productivity and flexibility of this language and the easy learning curve for newcomers who aim to contribute with the InterSCity codebase. Technological and design aspects related to contributions in the form of new microservices are discussed with the project team and may use other technical options according to their purpose. However, if the new microservices are supposed to be maintained within the main platform repository, they must have at least one maintainer that has fluency in the adopted technologies and must conform to the quality requirements of the project, such as having highly automated test coverage.

Initially, we adopted PostgreSQL[14] in all microservices as it is widely adopted in the software industry and it supports georeferenced queries, which are essential in the smart city domain. However, we moved towards the plurality of database systems to better fit the scope of each microservice and to adequately support horizontal scalability so that the database layer does not become a bottleneck. For example, we migrated the database infrastructure of the Data Collector and Actuator Controller microservices to MongoDB[15], a NoSQL database system that is more appropriate for databases with flexible schemas and non-normalized databases.

In addition to technology stack, several other decisions may affect the internal implementation of a service, such as the use of caching mechanisms, database schema and indexing, the choice of algorithms, and API design. Each microservice will be required to scale at different paces depending on both the specificities of the urban and technological context and the continuously increasing demands. Although the loosely coupled architecture allows microservices to scale out independently, different design strategies must be applied to overcome their own traits to achieve horizontal scalability. Table 3.2 summarizes the scalability strategies currently supported by InterSCity microservices, indicated by ✓ and points out new strategies that could be supported in the near future, denoted by **NF**, which are mapped as future work for the next stages of the InterSCity project.

Deployment of the InterSCity platform may have several instances of each of its microservices behind Kong's *load balancer* to handle higher loads transparently for customers. Services that receive asynchronous messaging, such as Data Collector, Actuator Controller, and Resource Adaptor, are designed to support the addition of more background workers to handle highly intensive demands for event-based jobs. Data Collector also uses database caching through an in-memory MongoDB instance to provide low-latency readings of the latest data collected by city resources, while Resource Catalog use Redis[16] to cache static data of most accessed city resources. In the future, the Resource Discovery could cache static resource meta-data provided by the Resource Catalog, since they do not change very often.

Currently, we did not support database sharding for any of our microservices. Even though, we mapped the use of this strategy for future work as it is a pontential approach for scaling the writes in the database. A database shard consists in a horizontal partition of data among several distributed instances of the database (*shards*). To achieve database sharding, each microservice needs to define a mechanism

[14]www.postgresql.org
[15]https://www.mongodb.com
[16]www.redis.io

Table 3.2: *Scalability strategies supported by InterSCity microservices*

Microservice	HTTP Load Balancer	Background Workers	Caching	Database Sharding
Resource Adaptor	✓	✓		
Resource Catalog	✓		✓	NF
Data Collector	✓	✓	✓	NF
Actuator Controller	✓	✓		
Resource Discovery	✓		✓	
Resource Viewer	✓		NF	

to route write and read operations to the proper database instance to achieve a satisfactory load balancing. For instance, if a single InterSCity instance is used for multiple cities, it could have a database shard for each city to properly distribute the database operations.

3.3.4 Independent Deployment

An important aspect of the InterSCity architecture is that its components are designed as single-purpose, independently deployable services. In this sense, two sets of deployment-related design decisions need to be considered: those related to the entire system, and those that concern individual microservices. The former are decisions related to the choice of cloud provider, to the allocation of computing resources for the system, to the packaging of service instances (i.e., virtual machines, containers, and physical hosts), to the handling of common procedures (e.g., update, backup, and monitoring), and to the deployment of complementary services used by the system (e.g., databases, message brokers, and proxies). The latter concerns the distribution of microservice instances across the available hosts (e.g., single service per host, multiple services per host), service configuration, failure recovery, and scaling of a microservice to appropriately respond to an increasing workload.

The above examples of decisions related to the deployment of microservices-based systems highlight the multiple trade-offs and challenges faced by engineers, as also observed in (Balalaie et al., 2016). In this sense, DevOps techniques and tools are essential to achieving a reliable and reproducible deployment process, as well as to improve the operating environment (Newman, 2015; Balalaie et al., 2016). The inherent complexity of a distributed microservices environment requires the use of such techniques to automate the necessary procedures. These include automated tests and the upgrading of microservice instances in the production environment, as well as support for more complex tasks, such as scaling and distributing microservices automatically based on the monitoring of services and computational resources.

Thus, we encapsulated microservices into individual Docker[17] lightweight containers which can be deployed and maintained independently. Continuous integration tools play an important role in the automated execution of both individual and integration tests, and to ensure that container images are built correctly along the development of microservices. To perform automated, consistent, remote deployments in a predefined set of virtual machines, we adopted the Ansible[18] automation engine, which provides a set of configuration management tools and scripts, facilitating the deployment of the InterSCity platform and associated applications. We used these tools to deploy an online instance of the InterSCity which is available in http://playground.interscity.org/. However, InterSCity is designed to leverage cloud-native facilities, such as scalability and elasticity. Thus, alternatively to Ansible, we use Kubernetes[19] to support the deployment of Docker containers throughout a cluster of virtual machines in a cloud infrastructure.

Kubernetes is an open source project that aims at automating the management of container-based applications by offering tools for supporting two critical tasks. First, it provides a declarative structure through YAML files for developers to define the desired state for the containers that comprise the managed system (JOY, 2015) in compliance with the *Infrastructure as Code* (IaC) principle, one of the pillars of DevOps (HUMMER et al., 2013). Second, the Kubernetes engine runs the specified configuration on the cloud provider to manage the deployment of the system by scheduling containers across the available machines.

In dynamic smart city contexts, with varying demands throughout the day and random citizen behavior, InterSCity must be able to individually scale out the stressed services to properly support workload fluctuations, rather than scaling the entire system as a whole. As we designed InterSCity microservices as stateless services, we can place several copies of the same microservice behind a load balancer (i.e., Kong). Likewise, multiple instances of the same microservice leverage the worker-based approach to distribute processing jobs across asynchronous workers. By monitoring computational resources used by the platform microservices, Kubernetes is capable of adjusting the number of instances for each microservice to adequately support a varying workload automatically. Ideally, the microservices should scale automatically allocating resources based on either forecasting models(MOURA et al., 2016) or more reactive actions based on changing demands. However, allocating more resources for the system may also be performed manually. In our experiments to evaluate the InterSCity scalability, described in Chapter 5, we used both approaches.

[17]https://www.docker.com
[18]https://www.ansible.com
[19]https://kubernetes.io/

3.4 Microservices

In this section, we detail the microservices that comprise the initial architecture of the InterSCity platform. It is worth noting that since this is an early implementation of the InterSCity architecture, there may be significant design changes in the future, including the addition of new microservices, API improvements, the creation of new abstractions, to name a few. As discussed earlier, the proposed architecture encourages the realization of these changes due to its flexibility and evolvability properties. As we detail each microservice, we also discuss existing problems due to the misapplication of the design principles presented in Section 3.1.

Although all microservices have received external contributions due to our open source approach, the vast majority of the technical development of the initial InterSCity microservices are technological outputs from this master's research effort.

3.4.1 Resource Adaptor

The Resource Adaptor microservice is responsible for providing a single point of entry for underlying IoT Gateway. Thus, it receives all requests from IoT Gateway, validates their inputs, and either calls another microservice to handle the request if it needs a synchronous response or publishes messages to RabbitMQ. Thus, this microservice provides endpoints to register new city resources, post sensor/context data, and to subscribe for receiving actuator commands through WebHook.

The following are the main HTTP endpoints provided by Resource Adaptor. For more details on all available endpoints for this microservice, check the online API documentation[20].

POST /adaptor/resources

IoT Gateway must use this endpoint to register a new city resource with a JSON body containing the resource's metadata, such as description, capabilities, and location, as shown by Figure 3.6.

```
{
  "data": {
    "description": "A public Hospital",
    "capabilities": [
      "medical_procedure",
    ],
    "lat": -23.559616,
    "lon": -46.731386
  }
}
```

Figure 3.6: *Example of Input JSON to Register a City Resource.*

After validating the input json, the Resource Adaptor redirects the original request to a similar endpoint of Resource Catalog (**POST /catalog/resources**) which is

[20] http://playground.interscity.org

responsible for registering the resource on its database and notify the rest of the platform through RabbitMQ. Also, Resource Catalog generates a UUID for the new resource which must be used for future interactions with the platform. However, an IoT Gateway may also suggest a UUID for the new resource by including it on the input JSON. Figure 3.7 shows the output JSON for a valid resource which contains the fields provided in the original request, the new UUID, and additional location data inferred from latitude and longitude, such as the country, state, city, and neighborhood.

```
{
  "data": {
    "uuid": "45b7d363-86fd-4f81-8681-663140b318d4",
    "country": "Brazil",
    "state": "Sao Paulo",
    "city": "Sao Paulo",
    "neighborhood": "Butanta",
    "description": "A public Hospital",
    "capabilities": [
      "medical_procedure",
    ],
    "status": "active",
    "lat": -23.559616,
    "lon": -46.731386,
    "created_at": "2018-03-21T16:23:00.312Z",
    "updated_at": "2018-03-21T16:23:00.312Z"
  }
}
```

Figure 3.7: *Example of Response for the Request to Register a Resource.*

POST /adaptor/resources/{uuid}/data

IoT Gateway must use this endpoint to send context data from city resources to the platform. For this purpose, they must inform the resource's UUID in the URL. Figure 3.8 shows an example of input JSON for this endpoint. With a single request, it is possible to publish data of multiple capabilities as well as more than one reading at a time. Each unique context data requires a timestamp which denotes the moment the data were collected.

Context data may vary widely in terms of both information collected and data structure. In this sense, InterSCity allows IoT Gateway to publish different types of data allowing each capability to include one or more fields. In the example of Figure 3.8, there are three integer fields inside the *environment_monitoring* capability as well as *bus_monitoring* capability which contains a more complex data: the *location* field.

After validating the input JSON, Resource Adaptor separates each context data, adds some metadata, and publishes them individually in RabbitMQ allowing other interested microservices to use them. For example, Data Collector will save all provided data, while Resource Catalog will only use data tagged for location update since it is responsible for handling resources' location.

```
{
  "data": {
    "environment_monitoring": [
      {
        "temperature": 10,
        "humidity": 45,
        "pressure": 25,
        "timestamp": "2017-06-14T17:52:25.428Z"
      },
      {
        "temperature": 20,
        "humidity": 64,
        "pressure": 25,
        "timestamp": "2017-06-14T17:57:25.428Z"
      }
    ],
    "bus_monitoring": [
      {
        "location": {
          "lat": -10.00032,
          "lon": -23.200223
        },
        "speed": 54,
        "bus_line": "875C-10-1",
        "timestamp": "2017-06-14T17:52:25.428Z"
      }
    ]
  }
}
```

Figure 3.8: *Example of Input JSON to Publish Context Data.*

POST /adaptor/subscriptions

If a managed resource has actuator capabilities, the IoT Gateway must subscribe to receive actuator commands for that resource through WebHooks. Accordingly, the platform will send a HTTP POST request with the actuation command details in a JSON body that subscribed IoT Gateway must handle. Figure 3.9 shows an example of input JSON to this endpoint which is quite simple since it only has the UUID of the resource, the list of actuation capabilities to subscribe, and the URL that will be used by InterSCity to notify actuation commands.

InterSCity will keep background workers to perform the HTTP requests for subscribers on actuation commands, which will send a POST request on the informed URL with a JSON such as shown in Figure 3.10. As the sensor data, actuation commands have dynamic structure.

3.4.2 Resource Catalog

The Resource Catalog microservice handlers resources' static data, available capabilities, and resources location. It has a relational database based on PostgreSQL and relies on Redis for caching. This microservice does not depend on other

```
{
  "subscription": {
    "uuid": "45b7d363-86fd-4f81-8681-663140b318d4",
    "capabilities": [
      "illuminate"
    ],
    "url": "http://myendpoint.com"
  }
}
```

Figure 3.9: *Example of Input JSON for Actuation Subscription.*

```
{
  "action": "actuator_command",
  "uuid": "0dbdae10-4156-4433-9291-5d261eb0d8eb",
  "url": "http://myendpoint.com",
  "capability": "illuminate",
  "created_at": "2017-06-07T20:16:16.348Z",
  "command": {
    "intensity": "high",
    "status": "on"
  }
}
```

Figure 3.10: *Example of Webhook JSON Body.*

microservices, but it publishes new data in RabbitMQ on either resources creation or resource update.

Admittedly, Resource Catalog violates the Single Responsibility principle reflecting in the size of its interface. We mapped out as future work the creation of a new microservice to handle the specificities of geolocation-based features and use the most appropriate technologies for this purpose, such adding support for GeoJSON format, described at RFC 7946 (Butler et al., 2016), as it encodes a variety of geographic a single point of entry for underlying IoT Gateway. In the following, we detail the endpoint to search for resources. For more details on all available endpoints for this microservice, check the online API documentation[21].

GET /catalog/resources/search

This is the main endpoint provided by Resource Catalog as other microservices and client applications use it to search for available city resources. One can combine multiple query parameters to filter the results, which includes:

- **capability** - search for resources with the specified capability
- **lat** - search for resources whose location has the specified latitude
- **lon** - search for resources whose location has the specified longitude

[21] http://playground.interscity.org

- **radius** - search for resources near to the point defined by *lat* and *lon* parameters considering the *radius* distance in meters from that point
- **description** - search for resources with the specified description
- **status** - search for resources based on their status (i.e., active)
- **country** - search for resources located at the informed country
- **state** - search for resources located at the informed state
- **city** - search for resources located at the informed city
- **neighborhood** - search for resources located at the informed neighborhood

The search responds a paginated list of up to 50 resources as shown by Figure 3.11. Clients may use the *page* parameter to set which page the endpoint will return.

3.4.3 Data Collector

Data Collector is a key microservice of the InterSCity platform. It is responsible for managing the resources' context data provided through sensor capabilities. Therefore, this microservice stores all context data published on RabbitMQ in a MonboDB database since these data have a dynamic structure as previously mentioned. However, we cache the latest context data of all resources in an in-memory MongoDB instance to avoid performing complex queries in a large amount of historical data.

To enable other microservices and client applications to access both the latest and historical context data of city resources, Data Collector offer a set of endpoints. It is worth noticing that all endpoints offers a rich set of combinable filters. Consequently, one may use both GET and POST HTTP verbs for each endpoint. The former exists for requests with fewer and simpler query parameters. The later enables more complex combinations of filters as it accepts a JSON body with the set of filters. Bellow are the four endpoints of Data Collector:

- **[GET, POST] /collector/resources/data** - Get historical context data of several city resources
- **[GET, POST] /collector/resources/{uuid}/data** - Get historical context data of a specific city resource
- **[GET, POST] /collector/resources/data/last** - Get the latest context data of several city resources
- **[GET, POST] /collector/resources/{uuid}/data/last** - Get the latest context data of a specific resource

The available filters for the above endpoints are the following:

- **capabilities** - the search will only include context data records that belongs to one of the capabilities defined in this parameter, which must be an array of strings.

```
{
  "resources": [
    {
      "capabilities": [
        "semaphore"
      ],
      "uuid": "f43d42ab-58ea-4eff-9c24-2e3c52103f00",
      "description": "A traffic semaphore",
      "status": "active",
      "lat": -23.559616,
      "lon": -46.731386,
      "country": "Brazil",
      "state": "Sao Paulo",
      "city": "Sao Paulo",
      "neighborhood": "Butanta",
      "created_at": "2017-08-08T17:44:26.728Z",
      "updated_at": "2017-08-08T17:44:26.728Z",
    },
    {
      "capabilities": [
        "temperature",
        "environment_monitoring"
      ],
      "uuid": "45b7d363-86fd-4f81-8681-663140b318d4",
      "description": "A traffic semaphore",
      "status": "active",
      "lat": -23.559616,
      "lon": -46.731386,
      "country": "Brazil",
      "state": "Sao Paulo",
      "city": "Sao Paulo",
      "neighborhood": "Butanta",
      "created_at": "2017-08-08T17:44:26.728Z",
      "updated_at": "2017-08-08T17:44:26.728Z",
    }
  ]
}
```

Figure 3.11: *Example Response of the Search Endpoint.*

- **start_date** - the lower limit of date/time considered for selecting context data, ignoring older records. The value for this parameter must comply with ISO 8601 (ISO, 1988).
- **end_date** - the upper limit of date/time considered for selecting context data, ignoring newer records. The value for this parameter must comply with ISO 8601 (ISO, 1988).
- **matchers of value** - select context data based on match rules for specific attributes. The following matchers are available:
  - **eq** - specifies equality condition
  - **gt** - select those data where the value of the specified attribute is greater than the specified value
  - **gte** - select those data where the value of the specified attribute is greater than or equal to the specified value
  - **lt** - select those data where the value of the specified attribute is less than the specified value
  - **lte** - select those data where the value of the specified attribute is less than or equal to the specified value
  - **in** - selects those data where the value of the specified attribute is in the specified array
  - **ne** - selects those data where the value of the specified attribute is not equal to a specified value. This filter is usually combined with the *capabilities* to avoid returning data that do not even have this attribute
- **uuids** - search for context data that belong to one of the resources specified in this parameter. It must be an array of valid UUIDs and should only be used for those endpoints that targets context data from several city resources.

Figure 3.12 is an example input json with parameters to the request **POST /collector/resources/data**

3.4.4 Actuator Controller

Client applications may send commands to change the state of city resources through their actuation capabilities. In this sense, the Actuator Controller microservice receives and validates all actuation requests from clients. Actuator Controller publishes valid commands in RabbitMQ to enable InterSCity's background workers to notify subscribers. Also, it stores all actuation requests and their respective status in a MongoDB database for audit purposes and to allow clients to monitor their requests through its API. Actuation commands may have one of the following statuses:

- **pending** - the command was received by the platform, but not by the target resource yet
- **failed** - the command could not be sended to the target resource

```
{
  "uuids": [
    "5ad20589-a3db-4521-b1bc-a21dde00a25c",
    "b5d170b5-aaf3-42bc-9e47-58e3fe2a4846"
  ],
  "capabilities": [
    "environment_monitoring",
  ],
  "matchers": {
    "temperature.gte": 13.498,
    "temperature.lte": 18.091,
    "humidity.eq": 20.02
  },
  "start_range": "2016-06-25T12:21:29",
  "end_range": "2016-06-25T16:21:29"
}
```

Figure 3.12: *Example Input JSON with Parameters to Filter Context Data.*

- **processed** - the command was processed by the target resource
- **rejected** - the command was rejected by the target resource, probrably because it does not konw how to process the input

Actuator Controller provides two endpoints:

- **POST /actuator/commands** - Client applications may use this endpoint to send commands to underlying city resources. This API allows one to send multiple commands at once rather than perform several requests to change multiple resources, as Figure 3.13. Whenever a client application requests a command, the command status is *pending*. As a result, the response of this endpoint is divided into success commands (which passed in all validations) and failed commands (with validation or input issues), as presented in Figure 3.14.
- **GET /actuator/commands** - This endpoint returns the list of actuation commands requested by client applications for resources with actuation capabilities. It is also possible to filter the results by status, by uuid of the target resource, and by the actuation capability.

3.4.5 Resource Discovery

Resource Discovery is another fundamental InterSCity microservice, as it enables client applications to discover the city resources based on several filters. Usually, an application will first discover the available city resources and their UUIDs to later interact with the other services. To this end, Resource Discovery offers a unique endpoint that allows client applications to combine a set of filters regarding resources' contexts.

One may combine static and dynamic filters to discover existing resources through query parameters such resources geoghraphical location and its current state based on the latest context values available on the platform. For example:

```
{
  "data": [
    {
      "uuid": "b0ae6f76-521d-4199-9595-f52c99361052",
      "capabilities": {
        "illuminate": {
          "intensity": "high",
          "status": "on"
        }
      }
    },
    {
      "uuid": "0dbdae10-4156-4433-9291-5d261eb0d8eb",
      "capabilities": {
        "illuminate": null
      }
    }
  ]
}
```

Figure 3.13: *Example Input JSON for Request Actuation Commands.*

```
{
  "success": [
    {
      "subscription": {
        "id": {
          "$oid": "598a183c913ccd0001a8cb65"
        },
        "capability": "illuminate",
        "created_at": "2018-03-22T20:33:49.584Z",
        "updated_at": "2018-03-22T20:33:49.585Z",
        "uuid": "b0ae6f76-521d-4199-9595-f52c99361052",
        "command": {
          "intensity": "high",
          "status": "on"
        },
        "status": "pending"
      }
    }
  ],
  "failure": [
    {
      "error": "Invalid command [\"Value can't be blank\"]",
      "value": null,
      "capability": "illuminate",
      "uuid": "0dbdae10-4156-4433-9291-5d261eb0d8eb",
      "code": 400
    }
  ]
}
```

Figure 3.14: *Example Response for Request Actuation Commands.*

- **GET /discovery/resources?capability=environment_monitoring &temperature.gte=18&temperature.lte=30** - Search for resources with the capability of environment monitoring and whose latest temperature readings was between 18 and 30 degrees Celcius
- **GET /discovery/resources?lat=-23.543750&lon=-46.638736&radius=500** - Search for resources near to the São Paulo's *Galeria do Rock* location, considering a radius of 500 meters

Similarly to the search endpoint of Resource Catalog, the Resource Discovery will return a JSON with the representation of the resources that matched to the client's query.

3.5 Implementation Principles

With pragmatic documentary purpose for future contributions to the InterSCity source code, in this section we enumerate the main implementation principles adopted during the development of the platform. Unlike the listed design principles which aims at addressing general system aspects at a higher level, implementation principles bring a guideline to maintain or improve the code quality.

- **Automated Testing.** Each new feature must be supported by corresponding automated tests to ensure its correctness and support changes. Such principle is fundamental for the evolvability of the platform and must be applied whenever possible, from small methods to more complex features.
- **Database Indexing.** Database indexing is critical for performance of queries. Thus, it is extremely important for the system scalability and performance to provide proper database indexes for implemented queries. Both MongoDB and PostgreSQL provide indexing strategies.
- **Data Pre-fetching.** Avoiding unneeded database access is important to improve the overall code performance. In this sense, it is always worthwhile to retrieve the data that will be used in a feature or algorithm using as few queries as possible. When the code needs to load the children of a parent-child relationship, is quite normal to introduce the N+1 problem[22] due to the lazy evaluation of most database access frameworks, such as the ones provided by the Rails community.
- **Single Responsibility.** Every software module must have at most one reason for changing and to be maintained. At the lowest level, this means that methods and classes should be cohesive and have a single responsibility. Similarly, InterSCity microservices should also implement a small set of functionalities related to a single responsibility in a cohesive manner.

[22]https://www.sitepoint.com/silver-bullet-n1-problem/

- **Code as Documentation.** Even if there are good descriptions and diagrams documenting the system, in the end, for developers, the main form of documentation is the source code itself. Thus, it is quite important to keep the code as clean as possible to enable other developers to browse through the code and read it easily. It is also important to keep the code simple.

- **Code Review.** Following open source communities' practices, InterSCity's repositories should have a set of core commiters with writing access. External contributions must be done through Merge Requests, which should be reviewed and merged by a experienced commiter.

- **Put Everything Under Version Control.** All elements that comprise InterSCity, including code, documentation, tools, scripts, and tests, should be under version control. Currently, we rely on GIT. This is critical to enable collaborative work and to keep the open source community active.

3.6 Application Life-cycle

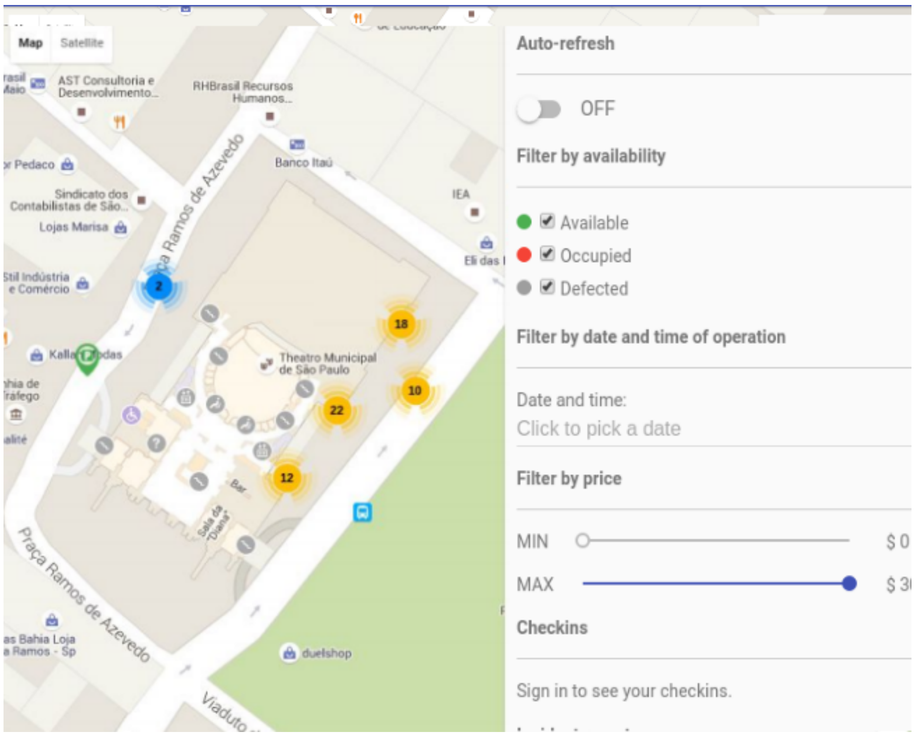
To demonstrate the InterSCity approach, here we illustrate how applications can be built over the InterSCity services to interact with city resources. For this purpose, we exemplify the use case of four experimental smart city application developed on top of the platform. External contributors developed these applications and as such, are credited as *indirect* technical contributions of this masters research.

The first example is a Smart Parking app developed by University of São Paulo (USP) undergraduate and graduate students during a programming course in 2016 which aims at supporting drivers to easier find available parking spots in the city. The second application is the São Paulo Health Dashboard, a management system that provides useful visualization tools to support the analysis of a large amount of medical procedures data. This application was initially developed as an ad-hoc solution during a summer course on Smart Cities at USP, being later migrated to run on the top of the InterSCity. Currently, it has been maintained by a student of scientific initiation as part of the InterSCity consortium.

Finally, the last two solutions were developed during the USP Hackathon on Smart Cities held at IME/USP in 2017 which used the online instance of the InterSCity platform. The first is a full-stack project called SancaLights that aims at providing low-cost sensors and a management system for public light monitoring. The second is Twitter bot that provides recommendations in real time about everyday life at São Paulo.

3.6.1 Smart Parking App

The Smart Parking application aims to help the difficult task of finding available parking spots in a large city, by offering a map with geolocated real-time information of parking spaces. The system is based on both static and sensor data of individual parking spots equipped with embedded sensors to notify the presence of a parked car. As can be seen in Figure 3.15, the Smart Parking App allows drivers to discover close parking spaces by offering visualization filters related to their availability, prices, and operating hours. One can check the details of a parking spot and view the route from the user's current location, as shown in Figure 3.16. The hardware part of this example was simulated via a specific software component that mimicked the behavior of physical sensors. The source code is available in the platform distribution[23].



Auto-refresh

OFF

Filter by availability

● ✓ Available
● ✓ Occupied
● ✓ Defected

Filter by date and time of operation

Date and time:
Click to pick a date

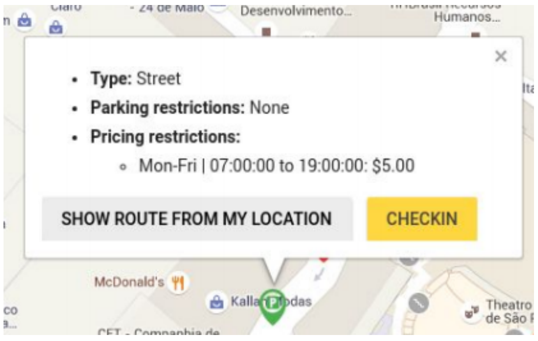
Filter by price

MIN $ 0
MAX $ 3

Checkins

Sign in to see your checkins.

Figure 3.15: *Screenshot of the Smart Parking application.*



- **Type:** Street
- **Parking restrictions:** None
- **Pricing restrictions:**
  - Mon-Fri | 07:00:00 to 19:00:00: $5.00

Figure 3.16: *Parking spot details in the Smart Parking application.*

Figure 3.17 illustrates a message flow that resulted from the use of the Smart Parking application supported by the InterSCity platform hosted on a cloud infrastructure. This example considers a smart parking infrastructure supported by cyber-physical systems to detect the presence of cars in parking spaces based on technologies that are commonly used in smart parking solutions, such as Wireless Sensor Networks (WSN), Light Dependable Resistor (LDR) sensors, Infra-Red (IR)

[23]www.gitlab.com/smart-city-platform/smart_parking_maps

sensors, and magnetic sensors (Bachani et al., 2016). These sensors send data continuously to a remote IoT Gateway via wireless protocols, such as ZigBee or Bluetooth, as illustrated in Step 1. The responsibilities of the IoT Gateway are two-fold: (I) registering each connected parking spot as a city resource with the *"availability"* sensor capability (Step 2) and (II) notifying the platform when a parking space becomes available or unavailable (Step 3). For these purposes, the IoT Gateway must track resource UUIDs provided by InterSCity to be able to send context data through the Resource Adaptor API upon state change events. We highlight that the concepts of city resource and capability abstract the implementation details of the underlying WSN infrastructure.

To use The Smart Parking application, one must define a target location or automatically use his/her current GPS data in addition to setting custom parameters to filter parking spaces according to the desired characteristics (Step 4). As an example, the application may query the platform for all the parking space resources that match the selected parameters within a 500 meters radius of the target location through the Resource Discovery API (Step 5). As a result, the Smart Parking application renders the current state of the returned parking resources on the map, as shown in Figure 3.15. Users can set the update time interval to get the current state of the returned resources as well as to modify the parameters, which will result in new requests such as the one performed in Step 5. Additional requests may be performed to get detailed information about a specific parking spot from the Resource Catalog API, such as presented in Figure 3.16, or even to obtain its availability history through the Data Collector microservice (Step 6).

Acknowledgements

We acknowledge the student members of the Smart Parking App group who developed the application on top of the InterSCity platform: Debora Setton, Hans ´ Harley, Jefferson Silva, Nury Arosquipa, and Thiago Petrone.

3.6.2 São Paulo Health Dashboard

The São Paulo Health Dashboard[24] was developed during a summer course by graduate students as an ad-hoc solution to provide visualization mechanisms for medical procedures data from São Paulo health systems. Later, this project was migrated to run on the top of the InterSCity platform for two major reasons: (I) make its data available so that other applications could reuse them; (II) decrease the backend size of the application by reusing InterSCity facilities. Currently, this project has been evolved within the InterSCity consortium together with members of the city's health department and its source code is available online[25].

Figure 3.19 shows the main page of the São Paulo Health Dashboard. The application uses medical procedures data from Sistema Único de Saúde (SUS) of São Paulo city, which include location and specialties of health facilities as well as anonymised data of patients, such as census sector, genre, and age. Among other

[24]http://healthdashboard.interscity.org/
[25]https://gitlab.com/eduardopinheiro482/health-smart-city

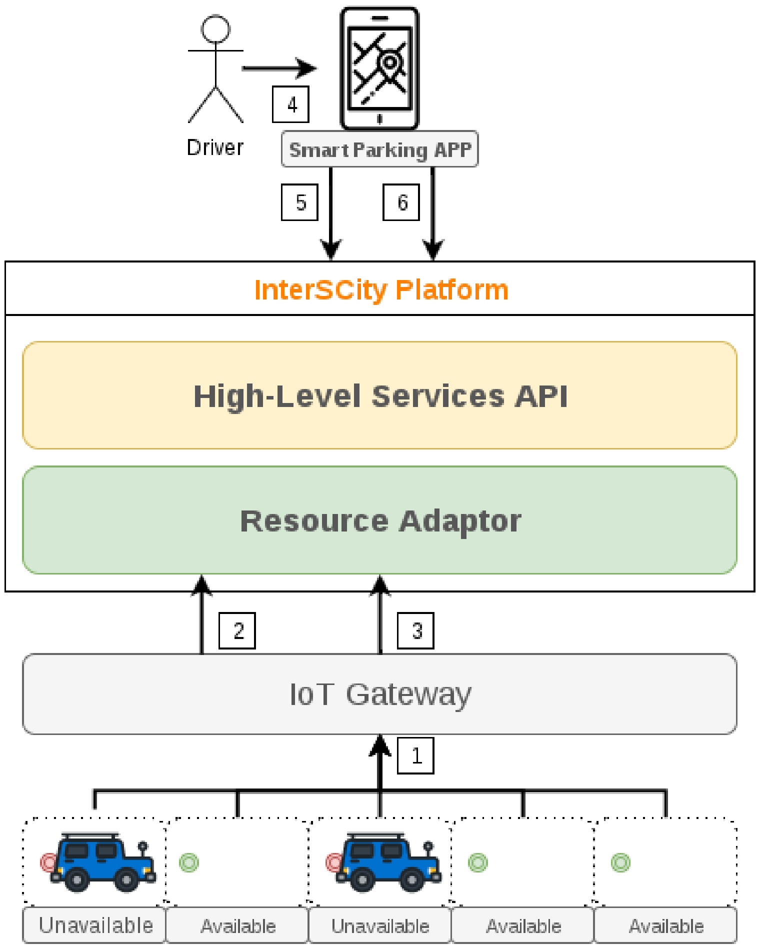


Figure 3.17: *Smart Parking application life cycle.*

features, the system provides: a map that evidences the distance that patients must travel to perform certain medical procedures; categorization of medical procedures; graphs for better visualization and understanding of data; advanced filters.

Although the available health data come from records of health services instead of coming from cyber-physical systems, the InterSCity abstractions are flexible enough to properly integrate them. Therefore, we modeled each health facility of the city as a resource with a single capability (*medical_procedure*), while each medical procedure perfomed in a given health facility is a context data of this capability. Figure 3.19 presents the entire lifecycle of running the Health Dashboard on the top of the InterSCity platform. We only have access to historical data that is made available through static files annually. Consequently, we had to create a script to parse those files (Step 1) and behaves like an IoT Gateway, registering the resources (Step 2) and the medical procedures data (Step 3) according to our model. Currently, the Health Dashboard App uses the InterSCity to discover all health facilities of the city (Step 4) and to collect the whole historical data of their medical procedures (Step 5).

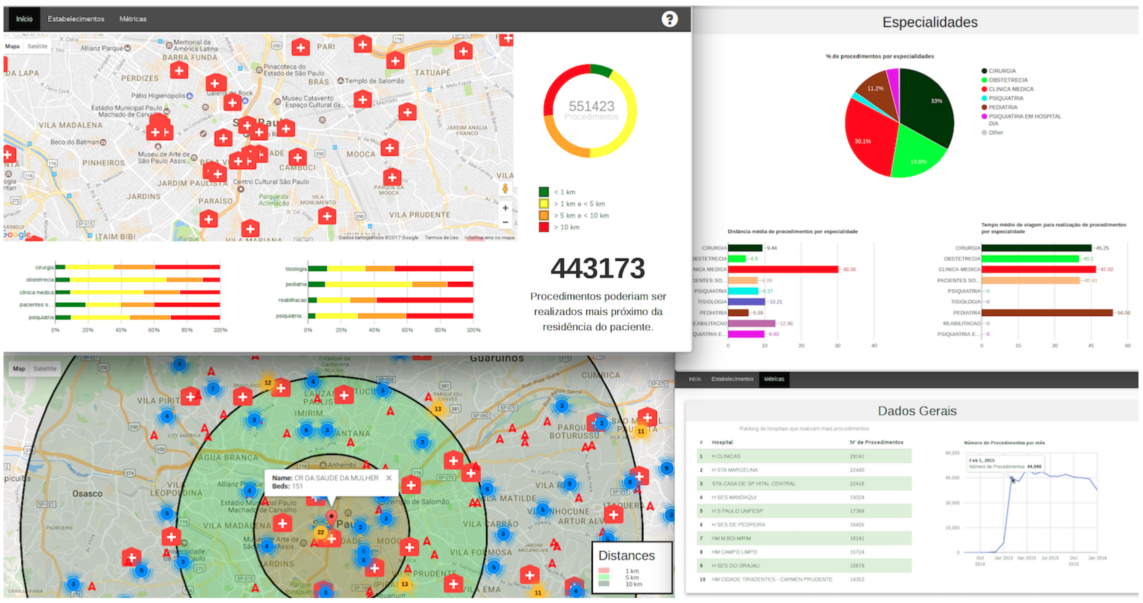


Figure 3.18: *Screenshot of the São Paulo Health Dashboard.*

Acknowledgements

We acknowledge the student members of the São Paulo Health Dashboard group who developed the application on top of the InterSCity platform: Eduardo Pinheiro, Debora Lina, Lucas Moura, Macartur Sousa, Lucas Kanashiro, and Rosangela Pereira.

3.6.3 SancaLights

SancaLights[26] is the winner project of the 2017 USP Hackathon on Smart Cities and it used the InterSCity as backend middleware. This project aimed at providing a full-stack solution for monitoring the public lighting in real-time. The project members advocate that through the use of low-cost sensors to read the electricity consumption of each public lamppost in real time, the CIP tax (contribution to the cost of public lighting services) will be calculated more fairly, generating cheaper electricity bills for the entire population. Currently, the government calculates the CIP using estimates which assumes all lampposts are lit for a certain period. Such estimates neither considers problems with broken lampposts nor contemplate the difference of the quality of illumination in different districts of the city. Improving the maintainance process of lampposts is another key advantage of this application since city managers can identify the exact location of faulty lampposts in real time.

To implement the SancaLights, the project members built their own hardware prototype using a Raspbarry PI microcontroller with a sensor to measure the eletric current of the lamppost. Also, the group developed its own IoT Gateway to handle the sensor readings and post these data periodically on the InterSCity platform. Finally, the group developed a Web Application based on the InterSCity APIs to show the status of city lampposts in real time, as shown in Figure 3.20.

[26]https://jornal.usp.br/universidade/alunos-da-usp-criam-hardware-que-pode-diminuir-valor-da-conta-de-luz

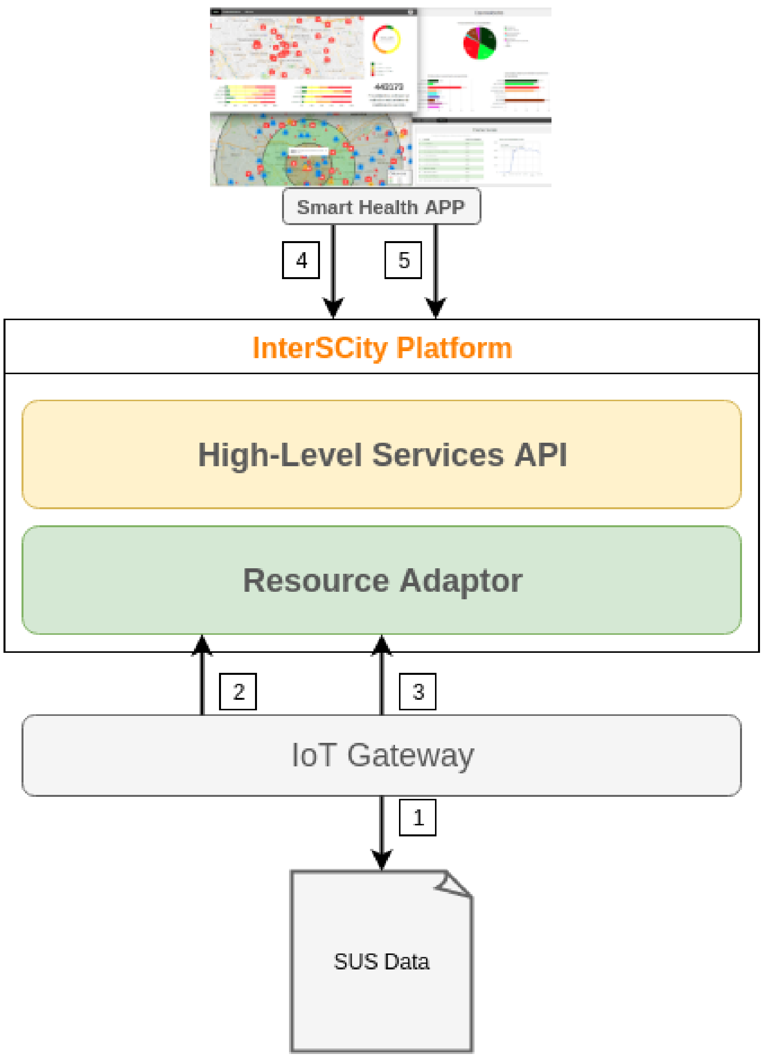


Figure 3.19: *São Paulo Health Dashboard application lifecycle.*

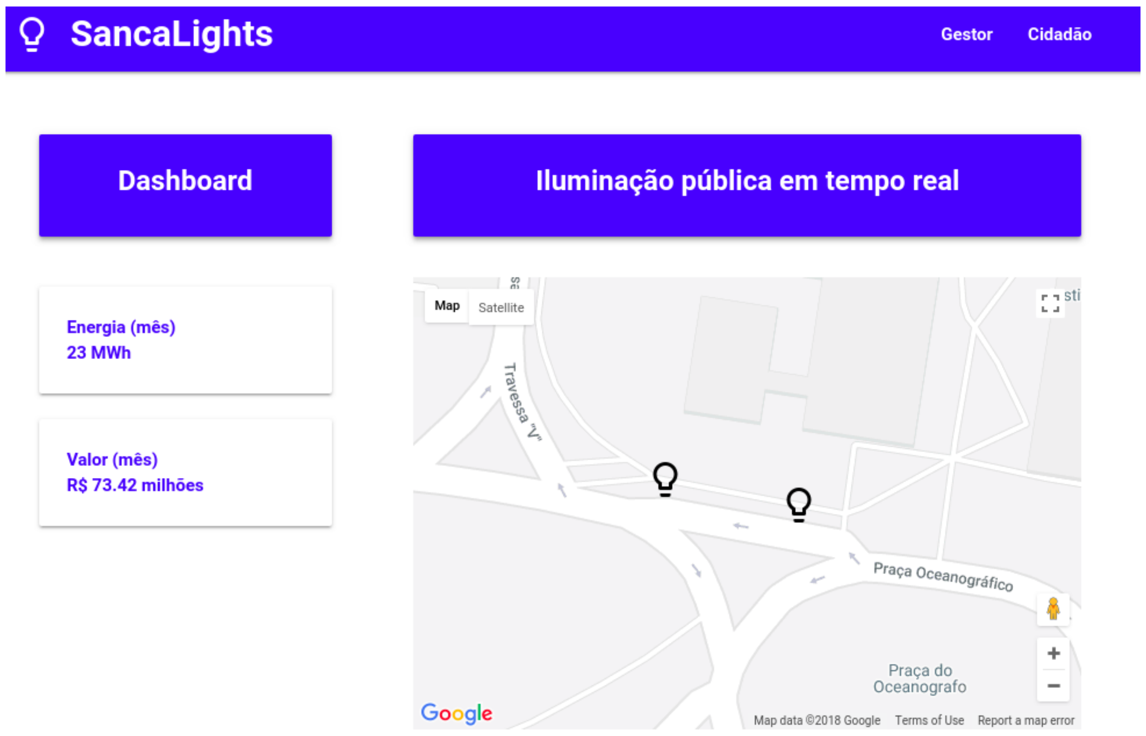
Acknowledgements

We acknowledge the student members of the SancaLights group who developed the application on top of the InterSCity platform: André Perez, Daniel Fernandes, Guilherme Rocha, and Leonardo Parente

3.6.4 Recomenda SP

Another notable project created during the 2017 USP Hackathon on Smart Cities is the Recomenda SP[27]. This project collected data from multiple city resources periodically from the InterSCity platform to produce useful recommendations about the everyday life in the city of São Paulo through a Twitter bot called SPoiler[28]. Citizens with a Twitter account can follow hashtags of interest to filter the SPoiler recommendations, as shown in Figure 3.21.

[27] https://devpost.com/software/recomenda-_sp
[28] https://twitter.com/recomenda_sp



Energia (mês)
23 MWh

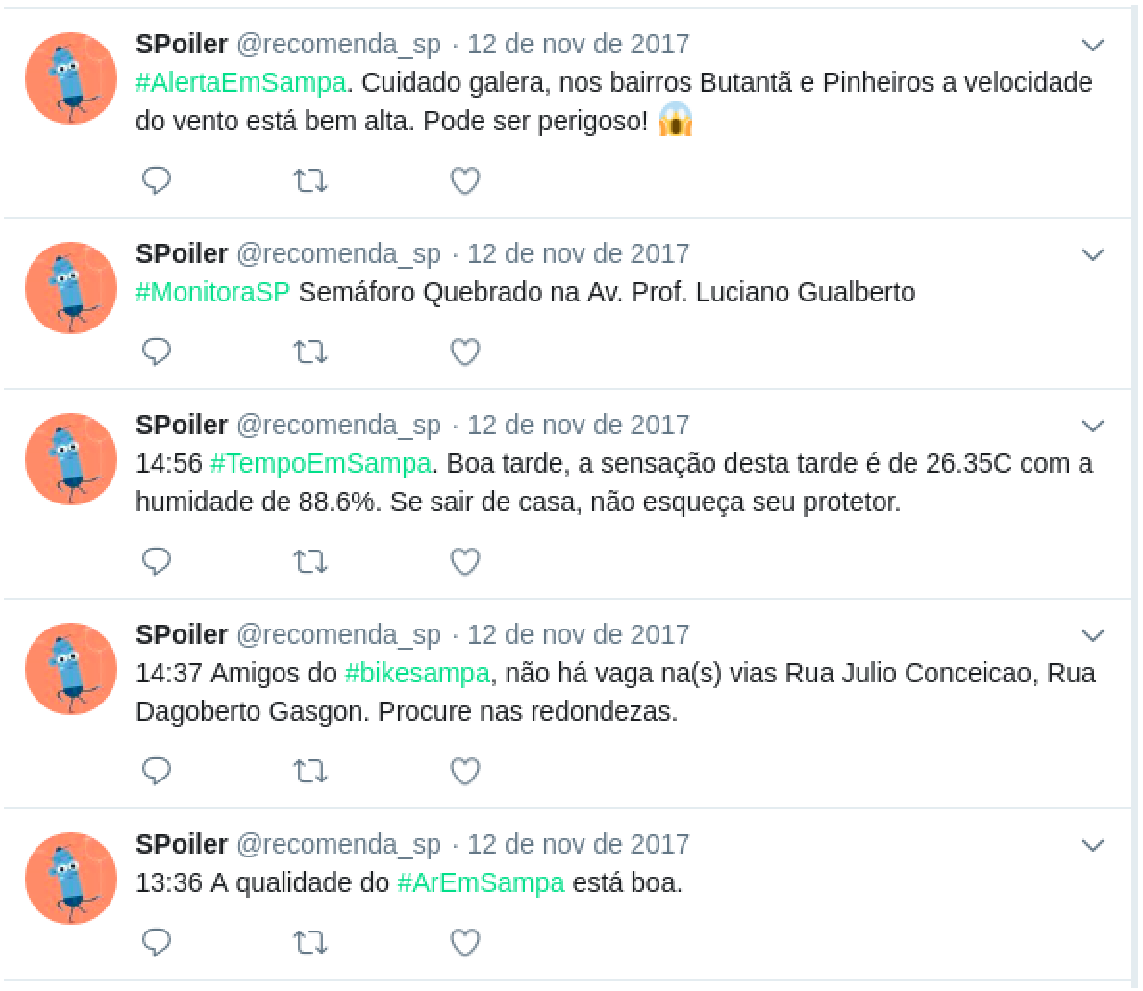
Valor (mês)
R$ 73.42 milhões

Figure 3.20: *Screenshot of the SancaLights project.*

To produce the recommendations, the Recomenda SP used a Complex Event Processing (CEP) engine to identify events to automatically produce alerts through the SPoiler bot. In this sense, they used data from city environment monitoring sensors (windspeed, temperature, humidity, atmospheric pressure, UV), data from shared bike stations (BikeSampa), and data from other city resources.

Acknowledgements

We acknowledge the student members of the SancaLights group who developed the application on top of the InterSCity platform: Vitor Silva, Italo Alberto, and Victor Harano.



SPoiler @recomenda_sp · 12 de nov de 2017
#AlertaEmSampa. Cuidado galera, nos bairros Butantã e Pinheiros a velocidade do vento está bem alta. Pode ser perigoso! 😱

SPoiler @recomenda_sp · 12 de nov de 2017
#MonitoraSP Semáforo Quebrado na Av. Prof. Luciano Gualberto

SPoiler @recomenda_sp · 12 de nov de 2017
14:56 #TempoEmSampa. Boa tarde, a sensação desta tarde é de 26.35C com a humidade de 88.6%. Se sair de casa, não esqueça seu protetor.

SPoiler @recomenda_sp · 12 de nov de 2017
14:37 Amigos do #bikesampa, não há vaga na(s) vias Rua Julio Conceicao, Rua Dagoberto Gasgon. Procure nas redondezas.

SPoiler @recomenda_sp · 12 de nov de 2017
13:36 A qualidade do #ArEmSampa está boa.

Figure 3.21: *Screenshot of the SPoiler Twitter bot.*

Chapter 4

Scalability-seeking Experimental Method

The primary design objective of the proposed architecture of the InterSCity platform is to meet scalability. From the first implemented versions of InterSCity to the current architecture presented in Chapter 3, several evolutions have been incorporated to the project, ranging from solving specific implementation problems to redesigning certain elements. In this sense, during this master's research, we have introduced the Scalability-seeking Experimental Method in our development life-cycle, an iterative method based on performance testing with the objective of continuously improving the proposed solution and identifying possible bottlenecks to solve them beforehand.

Therefore, in this chapter, we present the above method and navigate through the main improvements that were implemented from its application. As a result, based on our experience, we deepen the discussion on the impact of microservices architecture in the system evolvability.

Our approach is shown in Figure 4.1, which illustrates our Scalability-seeking Experimental Method. The method aims at providing an iterative set of pragmatic steps to guide the development and evaluation of the platform to meet performance and scalability requirements. Firstly, we must have a fully functional version of the platform; then we run performance tests or a lightweight experiment to evidence a possible bottleneck, which is identified through an analysis of the usage of computational resources and performance metrics. From this point, the possibilities are twofold. If a bottleneck is identified, we move back to the *design & development* phase to address it, completing a cycle of improvement. Otherwise, we run a comprehensive set of scalability experiments to evaluate the platform concerning performance and scalability. It is worth noting that running experiments to find bottlenecks and improve the system design does not require such high scientific rigor (i.e., repeated executions of the same experiment), as the objective of these performance test is to identify existing problems as faster as possible. In turn, for more conclusive results on the performance and scalability properties of a stable version of the system, we must run repeated experiments and perform a more careful analysis.

Both stages that depend on experiments, the early *lightweight experiments* to identify bottleneck and the *comprehensive experiments* to enable more sophisticated

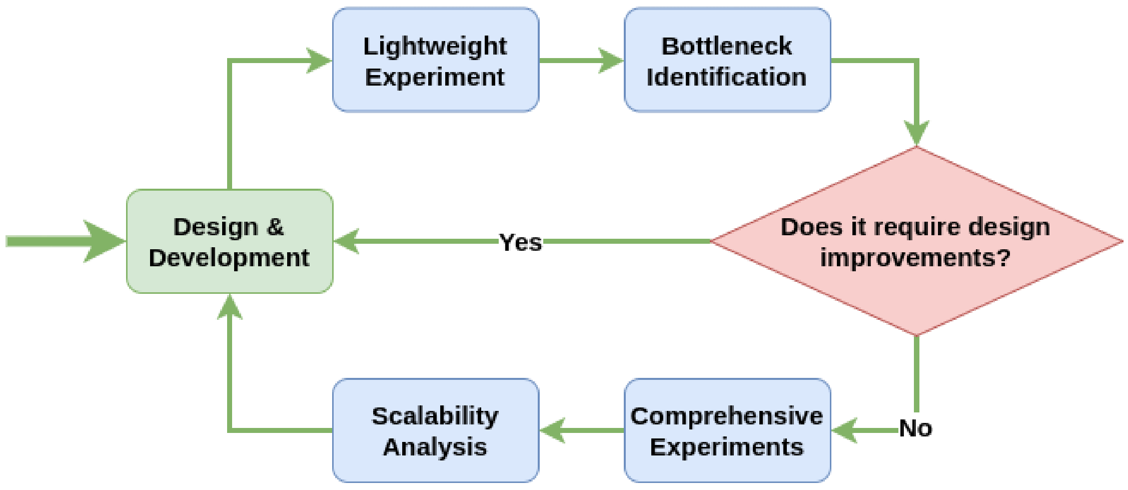

Lightweight Experiment
Bottleneck Identification
Design & Development
Yes
Does it require design improvements?
No
Scalability Analysis
Comprehensive Experiments

Figure 4.1: *Scalability-seeking Experimental Method*

scalability analyzes, aims at understating how variations of the workload and aspects of the system impacts on the platform performance and scalability. Therefore, it is important to define some elements to enable a correct analysis: the objective of the experiment; how to generate a significant workload for the system; the techniques and metrics that will be used; which are the parameters that will change during the execution of the experiment and which will not. However, these aspects are more stringent for running comprehensive experiments since they demand higher scientific rigor.

In spite of the fact that the described method is based on scientific techniques, several of its steps are empirical, such as identifying bottlenecks and making technical improvements. Moreover, although the method could be used in other contexts, we only present them as a pragmatic process to guide us towards achieving scalability in the InterSCity project. In this sense, we neither formally define the method nor apply scientific rigor to all its stages since many results of its application are merely technical.

During this master's research, we were able to run two complete rounds of the Scalability-seeking experimental method, within which we have made several cycles of improvements. We noticed that many scalability and performance issues are not identifiable in common tests at implementation time while identifying and addressing them at the end of the development life-cycle could be more complicated and costly. In this chapter, we focus on the empirical aspects of developing the InterSCity platform by presenting the results of some improvement cycles. In turns, Chapter 5 focuses on discussing the scientific validations by presenting results of the comprehensive experiments we ran at the end of two complete cycles.

4.1 Improvement Cycles of the First Round

During the first complete round, we conducted three improvement cycles, which supported us in making important architectural changes. In each of these cycles, we ran experiments to evaluate how the performance of the InterSCity platform degrades with the increase of concurrent IoT gateways (underlying client layer) sending sensor data continuously over an extended period. The main objective of these experiments are to evaluate the individual behavior and consumption of hardware resources of each microservice so that we could identify potential bottlenecks.

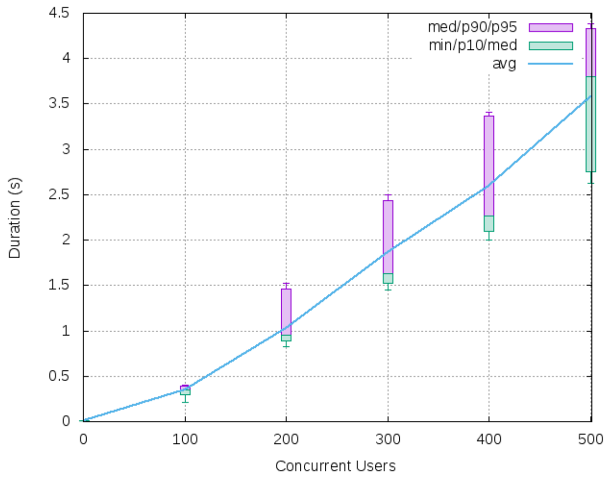
In summary, for each experiment, we ran an instance of the InterSCity platform in the Digital Ocean[1], a public cloud provider. We used several single-core GNU/Linux Debian 8.6 machines with 512MB RAM having 2.0GHz of clock speed in the same private network hosting one single containerized instance of a service, such as the Resource Adaptor, Resource Catalog, Data Collector, and RabbitMQ services, without any replication. We performed the degradation analysis by benchmarking the platform against different workloads supported by the Funkload[2] load testing tool in all the three cycles. Workloads were characterized by the number of concurrent emulated IoT Gateways that continuously send sensor data from city resources to the platform. Each client (the IoT Gateways) runs a loop sending synchronous requests to the platform throughout the experiment as fast as it can. We ran each of the load tests for four minutes, with a 30-second interval between them, collecting both response time and throughput metrics, while keeping the same capacity and configuration of the platform during all workload tests. We repeated each run of the experiment 20 times.

Figure 4.2a shows the results of the first improvement cycle regarding the performance degradation of the platform as the number of concurrent IoT Gateways increases. It can be seen that the response time increased significantly with increasing workload. The average response time was already longer than 1 second with 200 concurrent customers, reaching 3.6 seconds with 500. This high latency happened due to some characteristics of the tested InterSCity version, represented by Figure 4.3. In this version, whenever a new city resource was registered in the platform, the Resource Adaptor provided a new endpoint (URL) to enable other services to request for the most recent data or send actuator commands. Thus, every time a client sent a sensor data, the Resource Adaptor recorded this data in its local database and provided an API so that the Data Collector could retrieve the data later by polling HTTP requests each second. The requests made by the clients and the polling mechanism required redundant and costly I/O operations, turning the Resource Adaptor into the bottleneck.

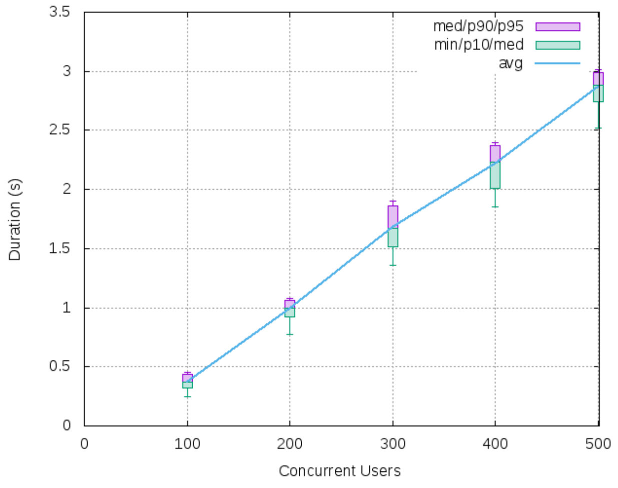
To improve the performance of the platform, we started a new cycle of the method by returning to the *design & development* stage. The major modification was that we changed the Resource Adaptor to save the data posted by a resource to memory instead of using a disk-based database. In this way, we have been able to reduce the latency of client requests and to consume less resource when Data Collector polled the Resource Adaptor for new data, as demonstrated in Figure 4.2b

Despite the improvements in results, this design would not scale to support a large number of IoT gateways sending data in parallel to the platform. Moreover, by monitoring the microservices, we noticed that both the Resource Adaptor and the Data Collector services were becoming bottlenecks. The former consumed a lot of memory and CPU due to a large number of requests and the latter due to the continuously thread-based work to poll the Resource Adaptor for the increased number

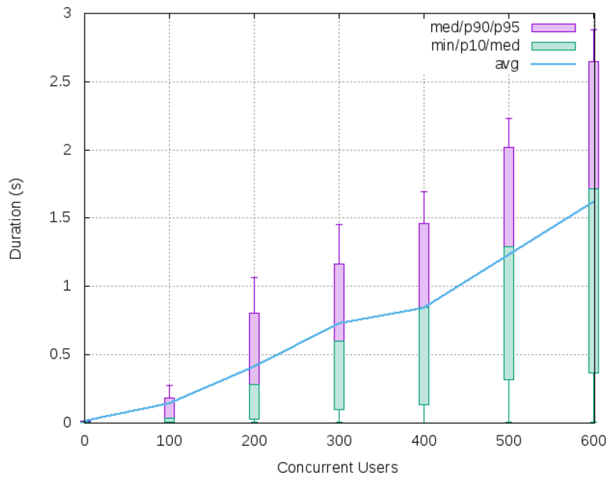
[1]https://www.digitalocean.com
[2]http://funkload.nuxeo.org



(a) *First improvement cycle - Response time degradation.*



(b) *Second improvement cycle - Response time degradation.*



(c) *Third improvement cycle - Response time degradation.*

Figure 4.2: *Improvement Cycles of the First Round of the Scalability-seeking Experimental Method*

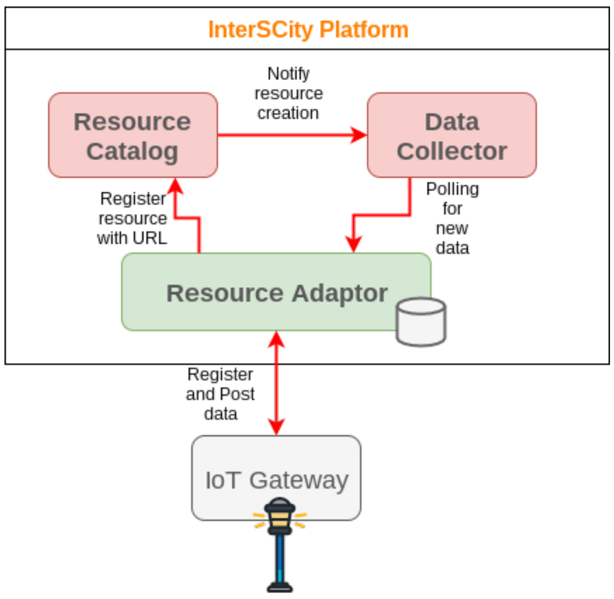


Figure 4.3: *First improvement cycle - InterSCity design.*

of city resources.

Therefore, we started the third cycle of development and experimentation that has led to the first stable architecture of the platform, presented at the beginning of this chapter. Among other modifications, we highlight three major improvements: (I) the prioritization of asynchronous calls over HTTP request-response method; (II) the substitution of the polling mechanism by the event-oriented approach and background workers; and (III) the removal of states related to the city resources from the Resource Adaptor. Consequently, we have reduced the coupling between services and improved the cohesion of the Resource Adaptor, which now has the single responsibility of providing an API for integrating IoT resources with other platform services.

Figure 4.2c shows the performance degradation of the platform as the number of concurrent IoT Gateways increases. The best average response time occurs for a workload of 50 concurrent clients, which was less than 60 milliseconds.

The average response time remained below 1 second with a workload of up to 400 parallel clients. However, the tests with 250 or more clients in parallel start to have requests with the latency above 1 second; for this particular application, this is not a problem, but this is an interesting indication of the limitations of the platform in case of applications with more stringent real-time requirements. It is important to note that the number of failed requests (returned with an error code or with a timeout) varied from 0.01% for 350 concurrent clients to 0.16% for 600 concurrent clients. With up to 250 concurrent IoT Gateways, 100% of the requests were successful.

In addition to evaluating the degradation in the overall performance, we also monitored the use of hardware resources on each machine to identify potential bottlenecks. For this purpose, we used the Linux-based Collectl[3] tool, as it can be used to monitor a broad set of subsystems such as CPU, disk, memory, processes, and network. Among the four machines used to deploy the platform in the experiment, the one that hosted the Resource Adaptor microservice presented the highest load

[3]collectl.sourceforge.net

in CPU and memory usage, both close to 100% during the entire duration of the experiment, being identified as the main bottleneck of the first stable version of the platform considering the tested scenario. The Data Collector host machine also maintained a high level of CPU usage, as well as intensive use of I/O operations to store the sensed data. The other two machines did not characterize bottlenecks.

After three improvement cycles, we achieved the first stable version of the InterSCity platform on which we performed the remaining steps of the scalability-seeking experimental method through controlled scalability experiments. However, the results of these experiments will only be shown in Section 5.1

4.2 Improvement Cycles of the Second Round

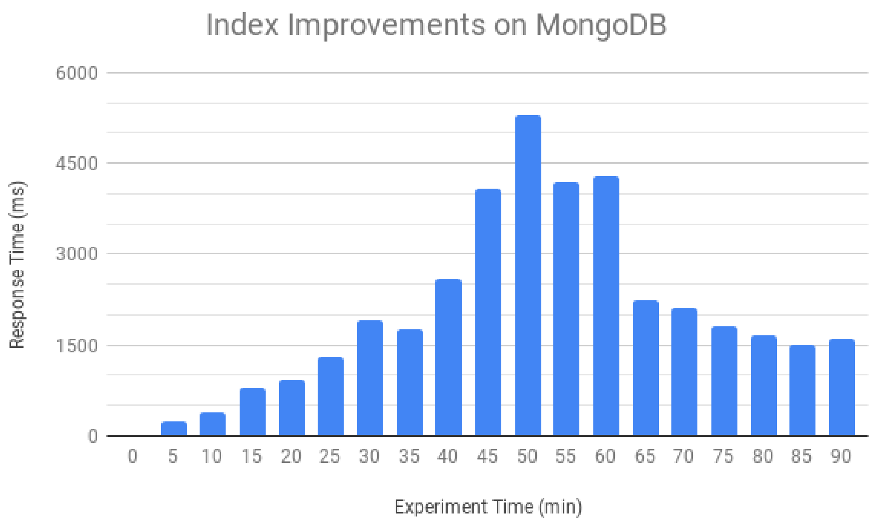
After achieving a stable version of the InterSCity platform, we continued our development process by adding new features and refactoring the existing infrastructure. In this way, we were able to run another complete round of scalability-seeking experimental method to evaluate the impact of our changes on the system overall performance.

During the second round, we conducted four improvement cycles. However, we changed the experimental setup and objectives to assess the platform performance considering a more realistic workload and covering a larger set of functionalities. To generate the workload to assess the platform scalability, we integrated the InterSCity platform with the InterSCSimulator, a scalable smart city simulator (**santanainterscsimulator**). We implemented a Smart Parking scenario in the InterSCSimulator in which drivers could use the Smart Parking App (explained in Secion 3.6.1) to discover available parking spots in São Paulo city. Each driver actor works as a client for the platform as they perform a set of requests to it to obtain the nearest available parking spots. Similarly, the parking spots are city resources registered to the platform, being responsible for notifying the platform whenever it becomes available or unavailable. By using the simulator, we could produce a more appropriate workload to the platform since the simulator is capable of managing the behavior of both the IoT infrastructure and the client applications at the same time while considering the consequences of their interactions during the simulation in contrast to the previous approach based on simpler benchmarking tools.

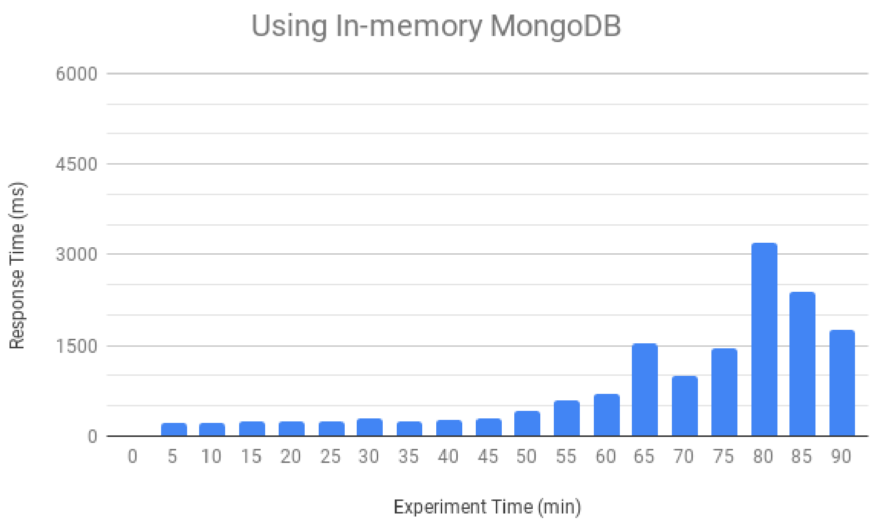
In summary, in each improvement cycle performed the following steps: (I) running a production-like instance of the platform in a cloud environment; (II) enabling an auto-scaling mechanism for the platform's microservices based on the variation of the workload; (III) setting up the simulator in an isolated environment; (IV) starting the simulation of the Smart Parking scenario; (V) monitoring the platform's performance and its usage of the computational resources during the entire simulation; and (VI) analyzing the obtained results.

By using the early described scalability-experimental method, we were able to considerably improve the platform design from a scalability and performance point of view. Figure 4.4 presents the sequence of four improvement cycles during the second round of the proposed method.

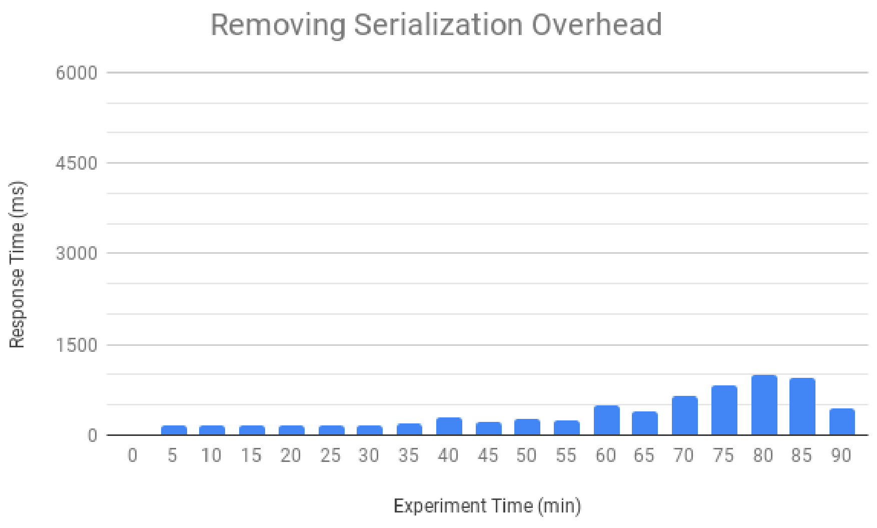
Using a functional version of the InterSCity platform, which contains a series of modifications relative to the stable version obtained in the first round of the scalability-



(a) *Index improvements on MongoDB*



(b) *Using in-memory MongoDB on Data Collector*



(c) *Removing serialization overhead*

Figure 4.4: *Improvement Cycles of the Second Round of the Scalability-seeking Experimental Method*

seeking experimental method, we executed the first improvement cycle. This first cycle evidenced very poor performance, with response times in excess of two minutes. By monitoring the use of computational resources, we observed an unexpectedly high CPU usage by MongoDB, indicating that it was using poor indexing strategies. This was identified as a possible bottleneck, so we went back to *design & development* phase. After monitoring the queries submitted by the Data Collector to MongoDB, we were able to improve our database indexes.

The second cycle of the experiment (Figure 4.4 (a)), showed considerable improvement in response times compared to the first cycle (Figure 4.4 (b)). However, the mean response time remained unsatisfactory, with some of them reaching close to 6 seconds. Such latency of responses is poor for smart parking applications, as it can result in traffic generation and longer driver unseating. By monitoring the computational resources used by each service, we notice that the Data Collector became the main bottleneck. After examining the logs of this second experiment, we learned that the database lock was delaying the responses to the queries on Data Collector. This happened due to multiple concurrent writes of the data generated by the simulated IoT infrastructure when the state of parking spots changed. We solved this new bottleneck by caching the latest data samples received from the IoT infrastructure in an in-memory MongoDB instance. After that, read queries were able to access recent data in memory so that the multiple concurrent writes to the MongoDB in-disk instance no longer affected their performance.

Such design enhancements led to performance improvements, with response times as shown in Figure 4.4 (b). In this version, we observed that some queries that were performed during microservices communication were returning a considerable amount of unnecessary data. As a consequence, a considerable latency was added to serialize, send and then deserialize the data. More precisely, replies of the requests performed by the Resource Discovery microservice to the Data Collector only required the list of *UUIDs* of the matched IoT resources. However, the previous version was serializing all the data returned by MongoDB into Ruby objects (including all metadata) before transforming them into JSON responses. Therefore, we changed the strategy to avoid unnecessary serialization steps to reduce CPU usage on an increasing load, especially when mapping database query responses into Ruby objects.

After several cycles of improvement, we came up with the latest version of the InterSCity platform, which shows a noticeable improvement in performance and scalability compared to the version we had before the adoption of the method. This can be seen from the response times shown in Figure 4.4 (c). Consequently, as the next chapter presents, we were able to perform the second complete iteration of our method to produce a more conclusive and comprehensive analysis of the scalability and performance of InterSCity.

Chapter 5

Scalability Evaluation

We designed a series of extensive experiments to evaluate the scalability properties of the proposed platform. During this master's research, we performed two sets of experiments, which are detailed later in separate sections.

In the earlier stages of this research, we carried out experiments to evaluate the first stable version implemented of the InterSCity platform, which is the result from the application of the first round of scalability-seeking experimental method, as explained in Chapter 4. These experiments aimed at evaluating the behavior of the platform in handling data streams from multiple sensors in terms of performance degradation and the improvement of performance in the addition of more computational resources.

In the second set of experiments, we evaluated the latest stable version implemented during this research in the final of the second round of the scalability-seeking experimental method. In comparison to the early experiment, these experiments are quite more comprehensives since they are based on a more realistic smart city scenario, with a varying workload, and considering both client applications and underlying IoT gateways interacting with the platform. In these experiments, we deployed the InterSCity in a cloud infrastructure and used DevOps tools to enable the auto-scaling of its microservices to properly support the varying demand.

5.1 Evaluating the Preliminary InterSCity Version

This first set of experiments is composed of two preliminary experiments. First, we evaluated how the performance of the InterSCity platform degrades with the increase of concurrent IoT gateways (clients) sending sensor data continuously over a long period of time. The main objective of these experiments is to evaluate the individual behavior and consumption of hardware resources of each microservice so that we could identify potential bottlenecks. The second experiment aim at assessing the scalability of the platform by applying supported scalability strategies to the bottlenecks we identified.

To conduct the experiments, we ran a production-like instance of the InterSCity platform in the Digital Ocean[1] cloud. Both InterSCity microservices and external

[1]https://www.digitalocean.com

services, such as PostgreSQL, RabbitMQ, and Redis, were deployed within Docker containers. However, each service instance was hosted on its virtual machine for isolation purposes, guaranteeing a fixed amount of machine resources per service. In the experiments, we consider a common smart city scenario where distributed sensors continuously collect observations from the city and send the sensed data to IoT gateways. The scripts used in the experiment are available in the repository for reproducibility [2].

5.1.1 Degradation Analysis

For the first experiment, we used several single-core GNU/Linux Debian 8.6 machines with 512MB RAM having 2.0GHz of clock speed in the same private network hosting one single instance of a service, such as the Resource Adaptor, Resource Catalog, Data Collector, and RabbitMQ services, without any replication. We performed the degradation analysis by benchmarking the platform against different workloads supported by the Funkload[3] load testing tool in all the three cycles. Workloads were characterized by the number of concurrent emulated IoT Gateways that continuously send sensor data from city resources to the platform. Each client (the IoT Gateways) runs a loop sending synchronous requests to the platform throughout the experiment as fast as it can. We ran each of the load tests for four minutes, with a 30-second interval between them, collecting both response time and throughput metrics, while keeping the same capacity and configuration of the platform during all workload tests. We repeated each run of the experiment 20 times.

Figure 5.1 shows the performance degradation of the platform as the number of concurrent IoT Gateways increases. The best average response time occurs for a workload of 50 concurrent clients, which was less than 60 milliseconds.

The average response time remained below 1 second with a workload of up to 350 parallel clients. However, the tests with 250 or more clients in parallel start to have requests with the latency above 1 second; for this particular application, this is not a problem, but this is an interesting indication of the limitations of the platform in case of applications with more stringent real-time requirements. It is important to note that the number of failed requests (returned with an error code or with a timeout) varied from 0.01% for 350 concurrent clients to 0.16% for 600 concurrent clients. With up to 250 concurrent IoT Gateways, 100% of the requests were successful.

In addition to evaluating the degradation in the overall performance, we also monitored the use of hardware resources on each machine to identify potential bottlenecks. For this purpose, we used the Linux-based Collectl[4] tool, as it can be used to monitor a broad set of subsystems such as CPU, disk, memory, processes, and network. Among the four machines used to deploy the platform in the experiment, the one that hosted the Resource Adaptor microservice presented the highest load in CPU and memory usage, both close to 100% during the entire duration of the experiment, being identified as the first bottleneck in the tested scenario. The Data Collector host machine also maintained a high level of CPU usage, as well as intensive

[2]www.github.com/LSS-USP/smart-city-platform-experiments
[3]funkload.nuxeo.org
[4]collectl.sourceforge.net

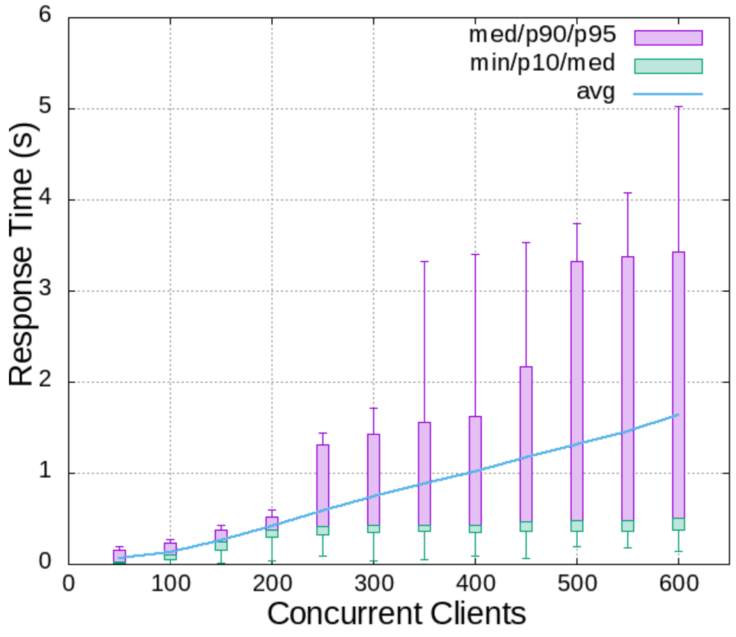


Figure 5.1: *Response time degradation.*

use of I/O operations to store the sensed data. The other two machines did not characterize bottlenecks.

5.1.2 Scalability Analysis

The objective of the second experiment on the preliminary version of InterSCity was to evaluate the scalability of the platform, as well as to demonstrate the flexibility of our architecture to address scalability issues. The same smart city scenario was considered, with concurrent IoT gateways continuously sending sensor data from city resources to the platform. For this experiment, we kept a fixed workload with 500 concurrent clients to evaluate the platform's speedup and scale-up metrics. The speedup metric measures how the performance improves with the addition of new resources to the system, while the scale-up metric measures the throughput gain.

The loosely coupled microservices architecture allows us to increase only the resources of the identified bottleneck microservices. We benchmarked the platform with the fixed workload during six 4-minute cycles applying a round-robin load balancing strategy by adding a replica of the Resource Adaptor microservice for each new cycle. The first cycle used exactly the same deployment setup of the first experiment described before. Both the Load Balancer (NGINX[5]) and the new Resource Adaptor instances were deployed on Docker containers hosted by separate single core, GNU/Linux Debian 8.6, 512MB RAM, 2.6GHz machines. These tests were performed using the Apache Benchmark[6] Linux tool.

As a result of the scaling strategy, the average response time decreased from 1725 milliseconds (with 1 instance) to 320 milliseconds (with 6 instances). Figure 5.2

[5]www.nginx.com

[6]httpd.apache.org/docs/2.4/programs/ab.html

shows the performance improvement measured in the experiment. We can observe a significant performance gain when scaling horizontally only one of the microservices that make up the platform; the speedup is very close to optimal. Since all messages received by Resource Adaptors are published through the RabbitMQ message service, the use of CPU by this service increases considerably, indicating another possibility for improving the speedup even a little more. RabbitMQ offers horizontal scalability natively, and we plan to enhance the platform making use of this feature, further improving its scalability.

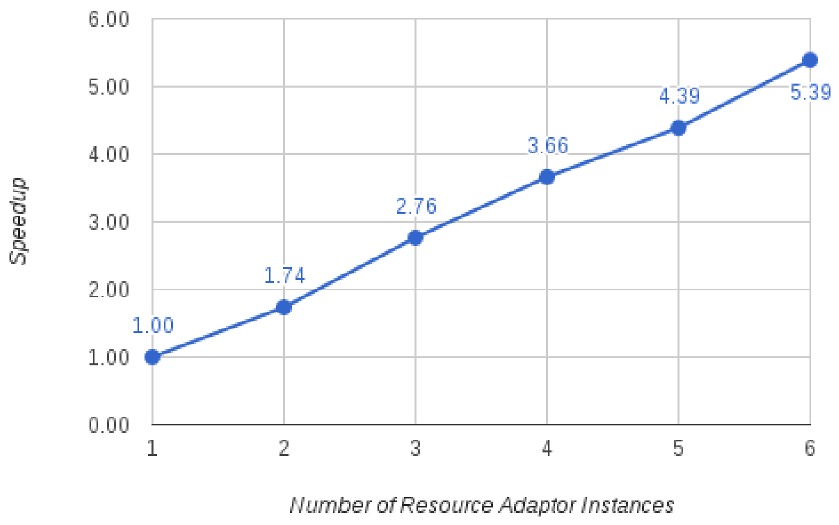


Figure 5.2: *Speedup - performance improvement varying the number of Resource Adaptors.*

As can be seen in Figure 5.3, the mean throughput of the platform increased substantially by horizontally scaling the Resource Adaptor in the tested scenario. With 6 instances, the platform answered an average of 1546 requests per second, 5.5 times more than when using the configuration with a single Resource Adaptor. Similarly to the speedup metric, the scale-up value increases almost at the same rate at which new instances are added, which is an excellent result.

5.1.3 Threats to Validity and Limitations

Although the above experiments point towards the applicability of our approach in the context of smart cities by demonstrating that the platform can support different scalability demands while keeping acceptable performance, they are preliminary since they have a set of limitations.

Regarding scope, the experiments are limited to the interactions among underlying IoT systems with the platform, disregarding the services provided to client applications. Consequently, most of the features provided by our platform were not tested. Also, it is not possible to analyze the impact of the multiple interactions from different actors to the platform on the overall system performance. For instance, a large number of writing operations could impact negatively on the performance of reading operations.

We only considered a simple scenario of concurrent clients sending sensor data to the platform continuously. Such an approach is limited due to the lack of variation

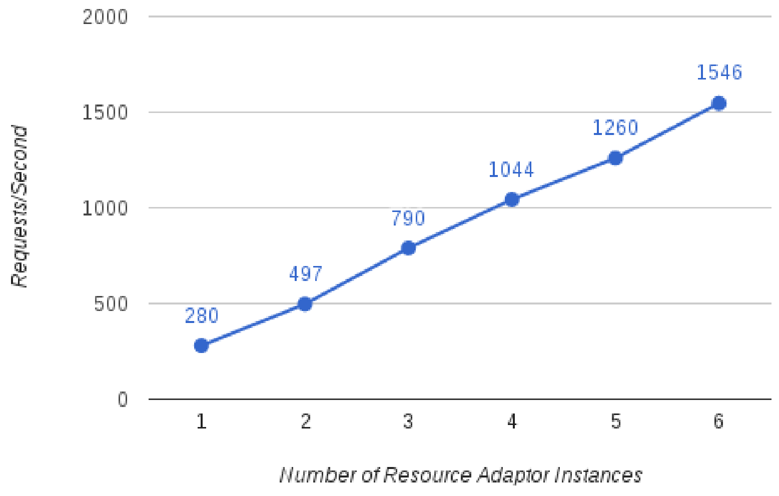


Figure 5.3: *Throughput improvement varying the number of* Resource Adaptors*.*

of data being transmitted, as well as their frequency. Also, due to our experimental setup limitations, the number of concurrent clients and the computational resources available for running the tested platform instance is immensely lower than what is expected for a production environment in a real smart city. Future smart cities will comprise a huge diversity of IoT devices which will produce a massive volume of data in different formats with varying intervals. Also, smart cities will integrate non-sensor devices, such as actuators. In this sense, to properly evaluate the InterSCity platform it is required to consider more realistic smart city scenarios and to use appropriate tools to generate significant workload to the platform.

5.2 Evaluating the Latest InterSCity Version

To evaluate the latest InterSCity's stable version, we conducted a comprehensive experiment to provide an extensive analysis of the platform's scalability properties. Such experiments resulted in the final validations of the proposed platform in the context of this masters' research. To properly validate the proposed platform, we have addressed most of the limitations identified in our preliminary experiments.

However, demonstrating the actual scalability of the InterSCity platform presents significant challenges due to the lack of available infrastructure for real experimental setups, as well as the lack of comprehensive datasets. This state of affairs derives from the difficulty in assessing platform characteristics in real-world scenarios.

One challenging related problem is the generation of representative workloads to assess the performance and scalability of smart city software platforms. Two distinct fundamental actors must be accounted for: (1) client applications, which generate requests using the platform facilities, and (2) the underlying IoT devices, which continuously produce sensed data and/or receive actuation commands. In this context, the use of randomly generated datasets or a failure to consider either of these external actors can result in experiments that do not represent realistic scenarios. In a real smart city, citizens dynamically interact with the cyber-physical infrastructure, triggering multiple events and changing the context observed by pervasive devices and systems. Similarly, platform clients, such as end-user applications and city management services, may vary their interaction behavior with the platform due to events observed in the physical world and real-time data provided by the city. This dynamic behavior of citizens and platform clients needs to be captured and modeled to generate significant and representative synthetic workloads.

Available benchmark tools focus on generating workloads for Smart Cities or IoT platforms by extrapolating real sensor traces from various contexts or by emulating the behavior of users based on Web traces. In this sense, the InterSCity research community proposed the use of the InterSCSimulator to generate a large-scale workload to platform evaluation based on realistic smart city scenarios (E. F. Z. Santana et al., 2017). Although data produced by a simulator is still synthetic, it reflects the individual behavior of involved actors and their interactions, emulating the dynamic behavior of a city and adapting the simulation accordingly.

The InterSCSimulator is a smart city simulator for the behavior of city actors and their interactions in large-scale settings. It implements models based on city actors, such as citizens, IoT devices, vehicles, buildings, and roads. Previous work shows that InterSCSimulator is capable of simulating an entire city such as São Paulo, with more than 10 million software agents virtually moving across tens of thousands of streets.

The tasks we performed to enable the simulation-based experiments on the InterSCity platform are twofold: (I) Integrate the simulator with the platform enabling two-way communication among them; and (II) Design and implement the smart city scenario on the simulator. The integration of the two systems is the focus of another master's research, and therefore we will not delve into its details here. The following is the details of the scenario used in the evaluation of the InterSCIty platform.

5.2.1 Smart City Scenario

To evaluate the InterSCity platform, we implemented a Smart Parking scenario on InterSCSimulator, which uses the existing mobility models in addition to new parking spot actors. This scenario considers several car drivers using the Smart Parking mobile application backed by the InterSCity platform to assist in the difficult task of finding a free parking spot in a large city like São Paulo, such as the application we presented in Section 3.6.1. This application offers geolocated real-time information about parking spaces. When using the mobile app, the car sends its current location to the server application, which then answers with a list of the closest available parking spots. In this scenario, we simulated the behavior of drivers that use the mobile application, along with the behavior of the IoT devices installed in parking spots and the interaction between them.

We extended the simulator with models to support the monitoring of parking spots and to allow drivers to find parking spots in the city. A parking spot actor generates events when a car parks on it or leaves it. It simulates the behavior of a real smart parking infrastructure supported by cyber-physical systems to detect the presence of cars based on technologies that are commonly used in smart parking solutions, such as Wireless Sensor Networks (WSN), Light Dependable Resistor (LDR) sensors, Infra-Red (IR) sensors, and magnetic sensors.

The simulation model consists of the following flow for a single car agent: (I) the car starts its trip from an origin to a destination point; (II) when the car is close to arriving at its destination, it requests the nearest free spots to the Smart Parking application; (III) the application asks the platform to find the nearest available spots considering the user's location; (IV) with the data returned by the platform, the simulator changes the route of the car to the closest parking spot returned; (V) the driver arrives at the spot and finishes its trip; (VI) the simulator updates the status of the chosen parking spot on the InterSCity platform, marking it as unavailable.

If the platform does not find any parking spots that match the request of a car driver, the corresponding simulation agent may return to Step **II** increasing the search radius. After three failed attempts, the agent stops using the application and completes its execution in the simulation. Another special case might occur during Step **V**, as the target parking spot might become unavailable (e.g., by being taken by another vehicle), requiring the car to return to Step **II**.

The interactions between the InterSCSimulator and the InterSCity platform required the integration of these two systems. Such interactions impose workloads on the platform at the top layer, as the simulator consumes data as a client application, and at the bottom layer, as it updates the status of the underlying IoT infrastructure of smart parking spaces.

To generate a significant workload to evaluate the platform, we modeled a realistic scenario based on real data from the huge city of São Paulo. This data was used as input to the simulator to define the number and distribution of parking spots around the city and the trips undertaken by drivers, including their origins, destinations, and departure times. The data used to generate workloads is detailed below:

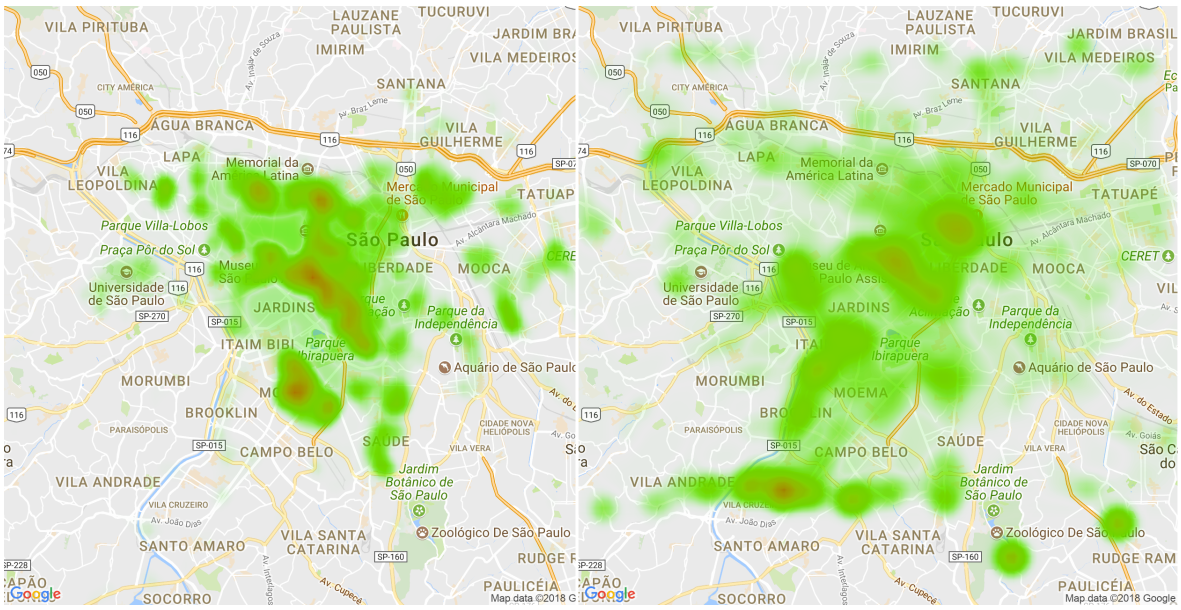
- **Origin-Destination (OD) Survey:** we created the simulated trips based on the OD survey performed by the subway company of São Paulo[7]. This survey describes the trips of 200,000 people and extrapolates the data to the entire population of the city. The survey includes information on the origin, destination, transportation mode, and departure time. We used this data to define the behavior of car agents. To generate the load for the platform experiments, we simulated the traffic in São Paulo during peak hours, from 5:40 am to 8:40 am. In the OD survey, there are 492,976 cars that start their trips during the considered time interval.

- **OpenStreet Maps:** to create the city graph used in the simulation, we used the map from OpenStreet Maps. This map contains all the streets and junctions of the city, together with a number of attributes, such as length, capacity, and speed limit. Such information is used by the simulator to define the routes taken by cars as they perform their trips, as well as to simulate the impact of traffic on the speed of cars.

- **Parking Spots:** we created the simulated parking spots based on data from OpenStreet Maps and from *Zona Azul*[8] (the rotary parking service of the city of São Paulo).

[7]Origin-Destination Survey - http://goo.gl/Te2SX7
[8]http://www.cetsp.com.br/consultas/zona-azul/mapa-zona-azul/mapa-zona-azul.aspx

To show the distribution of parking spaces and drivers destinations throughout the city, Figure 5.4 presents two heat maps: (a) the simulated distribution of parking spots across the city; and (b) the distribution of trip destinations throughout the entire simulation. It is worth noting that parking spots with IoT infrastructure are significantly more concentrated than trip destinations. This may lead to situations where drivers do not find available parking spots after three attempts. In such cases, the user agent stops using the application and finishes its execution.



(a) *Parking spots distribution* (b) *Destinations distribution*

Figure 5.4: *Heat maps of the distributions of parking spots and car trip destinations in the City of São Paulo*

5.2.2 Experiment Configuration

Using the integrated environment described above, we performed a comprehensive experiment to assess the scalability properties of the InterSCity platform. The experiment consisted of: (I) running a production-like instance of the platform in a cloud environment; (II) enabling an auto-scaling mechanism for the platform's microservices based on the variation of the workload; (III) setting up the simulator in an isolated environment; (IV) starting the simulation of the Smart Parking scenario; (V) monitoring the platform's performance and its usage of the computational resources during the entire simulation; and (VI) analyzing the obtained results.

To conduct the experiments, a production-like instance of the InterSCity platform pre-populated with the resources available in the city (parking spots) and their initial states (available) is required. Only the microservices Resource Catalog, Resource Discovery, and Data Collector are used in the Smart Parking scenario.

We used Docker containers for InterSCity microservices and supporting services. In all experiments we used the Google Cloud Platform[9] (GCloud), which is an appropriate

[9]https://cloud.google.com

infrastructure to run InterSCity in a production environment as the platform is a cloud-native system. More specifically, we used the Kubernetes Engine, a set of tools provided by GCloud based on Kubernetes, to schedule the deployment of containers throughout a GCloud cluster. Among other tasks, we used Kubernetes to also automate the restarting, replication, and scaling of containers. Thus, Kubernetes increases the reproducibility of our experiments by ensuring the correct application of deployment rules, configuration, and state. All the code used to perform the experiments, as well as the Kubernetes configuration files, are publicly available in an online repository[10].

We divided the cluster into 5 different node pools so that Kubernetes could schedule the containers to the appropriate pools. Figure 5.5 presents these node pools and the number and type of the machines used by each one on GCloud[11]. The Platform Pool comprises 25 machines of type *n1-standard-2* (2 virtual CPUs and 7.5GB of memory) and runs both InterSCity microservices and Kong. There are three additional node pools composed of *n1-highmem-2* machines (2 virtual CPUs and 13GB of memory), which execute the support services of the InterSCity environment. Both MongoDB and PostgreSQL pools have 5 nodes running distributed, fault-tolerant instances of their respective database systems, whereas RabbitMQ has a dedicated machine in an isolated node. MongoDB is deployed using the replica set strategy[12] to distribute *read* operations among secondary nodes (slaves), although *write* operations are always performed on the primary node (master). The same strategy is adopted for the PostgreSQL instance for optimizing the *read* operations performed by the Resource Catalog. Finally, the InterSCSimulator runs on its own *n1-highmem-16* machine (16 virtual CPUs and 104GB of memory), isolated from the rest of the services.

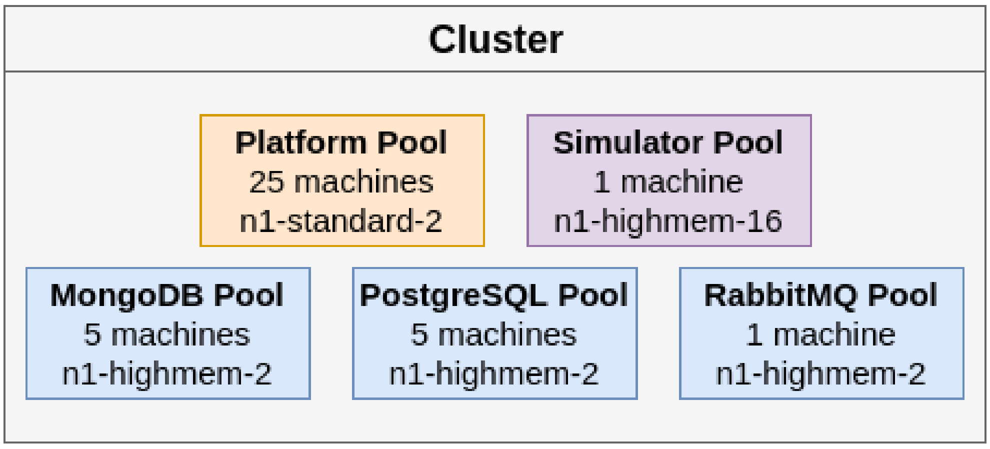


Figure 5.5: *Cluster Node Pools for the experiments*

For the Platform Pool, Kubernetes may schedule several containers for the same host depending on the available computational resources. The distribution of containers across the 25 nodes may differ from one experiment to another and is a variable that we did not control for during the performed experiments. To evaluate the impact of such variations on the analysis, we repeated the experiment 15 times and studied the variability of the results.

[10] https://github.com/LSS-USP/interscity-k8s-experiment
[11] https://cloud.google.com/compute/docs/machine-types
[12] https://docs.mongodb.com/manual/tutorial/deploy-replica-set

As we were interested in assessing the platform scalability considering a smart city scenario with a varying workload, we used automatic scaling for the Resource Catalog, Resource Discovery, Data Collector, and Kong services, as they are designed to scale horizontally. For this purpose, we specified a target of 60% of CPU usage for each of those services, enabling the system to increase or decrease the number of containers per service. The system balances the workload to match the target CPU usage considering the average CPU usage of the running containers, which is measured every 30 seconds. Initially, each service has four containers, which is set as the minimum number of running containers. This number may increase as long as computational resources are available in the Platform node pool. We run the containers behind a load balancing service.

Although we could benefit from GCloud's elasticity properties by automatically adding and removing nodes to our cluster using its auto-scaling feature, this would introduce another level of uncertainty in our experiments, since in our experience the time taken to create new VMs may vary considerably. Thus, we created all the nodes in advance, before starting the experiments, keeping them running throughout the experiment.

5.2.3 Scalability Analysis

To better analyze the behavior of the InterSCity platform, we ran multiple rounds of the experiment. Our objective was to minimize the effects of uncontrollable variables inherent to the environment in which the tests were performed, so as to evaluate important aspects of the system and to ensure that the observed results are good estimates for the general behavior of the system.

We performed a total of 15 rounds of the experiment. Each round ran for 3 hours, corresponding to the simulation of the morning peak traffic hours in São Paulo according to the scenario described in Section 5.2.1. Figure 5.6 represents the average workload generated by InterSCSimulator on the platform during the experiment and the standard deviation (black lines on top of each bar). It is worth noting the constant increase in the workload during the first 80 minutes. In the approximate interval of one hour between 60 and 120 minutes, we observed the highest load period of the experiment, whereas the maximum peak of requests happens after 80 minutes, corresponding to more than 113,000 requests in 10 minutes. In total, more than one million requests were performed to the platform during the experiment time. Since fulfilling each of these requests requires a complex set of operations with multiple internal steps, this translates to a very large computational load.

Figure 5.7 shows the dynamic creation and destruction of InterSCity containers due to the application of the auto-scaling strategy in a single round. The initial replication of Kong instances was enough to support the entire workload during the entire experiment since it only performs the low-latency task of forwarding the incoming requests to the proper microservices. In turn, the three InterSCity microservices, which are truly responsible for handling the requests, were replicated according to the increasing workload. Thus, the number of containers for these services varied from 4 to 25. It is important to mention that the InterSCity elasticity mechanism also reduced the number of containers as the demand decreased. As can be seen in

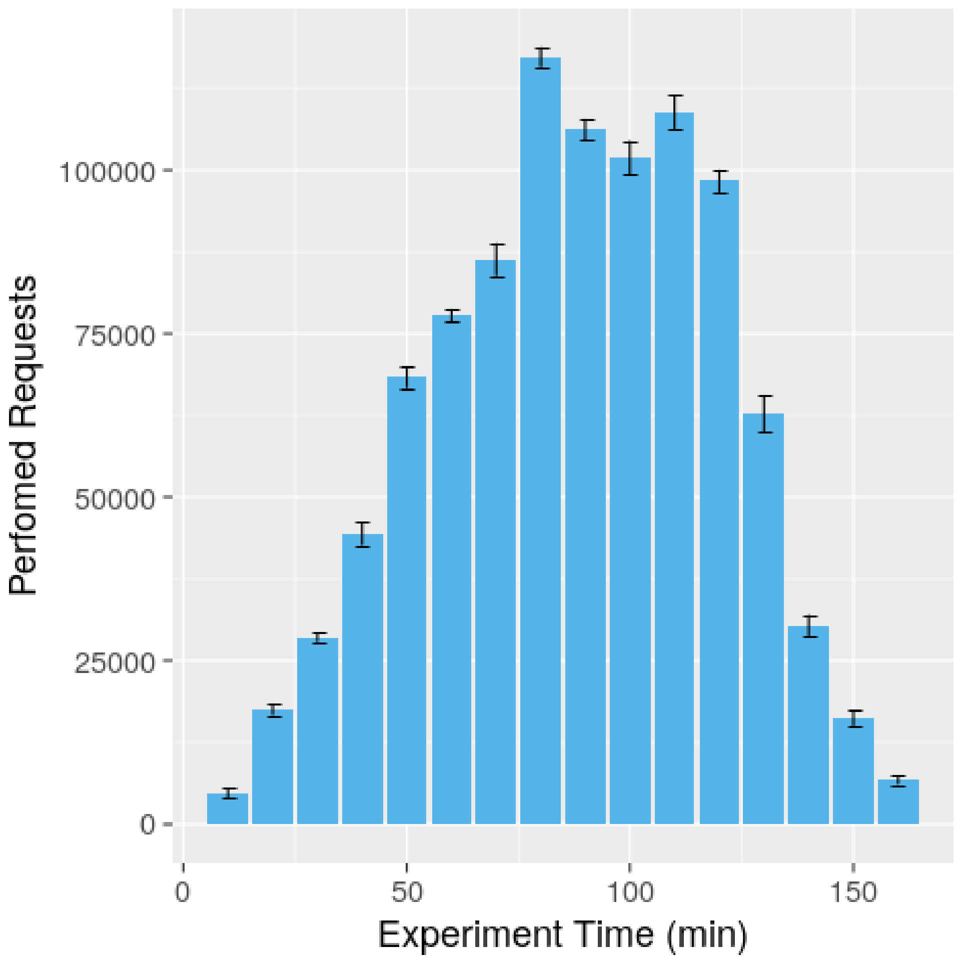


Figure 5.6: *Average workload generated by the InterSCSimulator*

Figure 5.7, among the InterSCity microservices, Data Collector was the microservice that consumed the least CPU time.

Figure 5.8 shows the average throughput of the InterSCity platform over the duration of the experiment. The throughput is defined as the rate of successful responses received by the simulator. The result indicates that the throughput closely matches the generated workload, as can be seen by comparing Figures 5.6 and 5.8. Despite the simulated variations in the drivers' behavior throughout the experiment, the platform was able to handle the varying demand thanks to its scalability and autoscaling support features, described in Section 3.2. However, we should mention that the throughput did not match the generated workload exactly, since some of the requests failed, representing nearly 0.6% of all requests on average. Failed requests include those which had responses with an HTTP error code, as well as those that were not completed due to connection refusal or timeout. But we consider that being able to handle over 99.5% of the requests under high load is satisfactory; a typical user would see a failure every 200 requests, which is very good for this kind of real-time smart urban application.

Another fundamental aspect of the system assessment is to analyze the performance of the platform to handle application requests with a varying workload. In this respect,

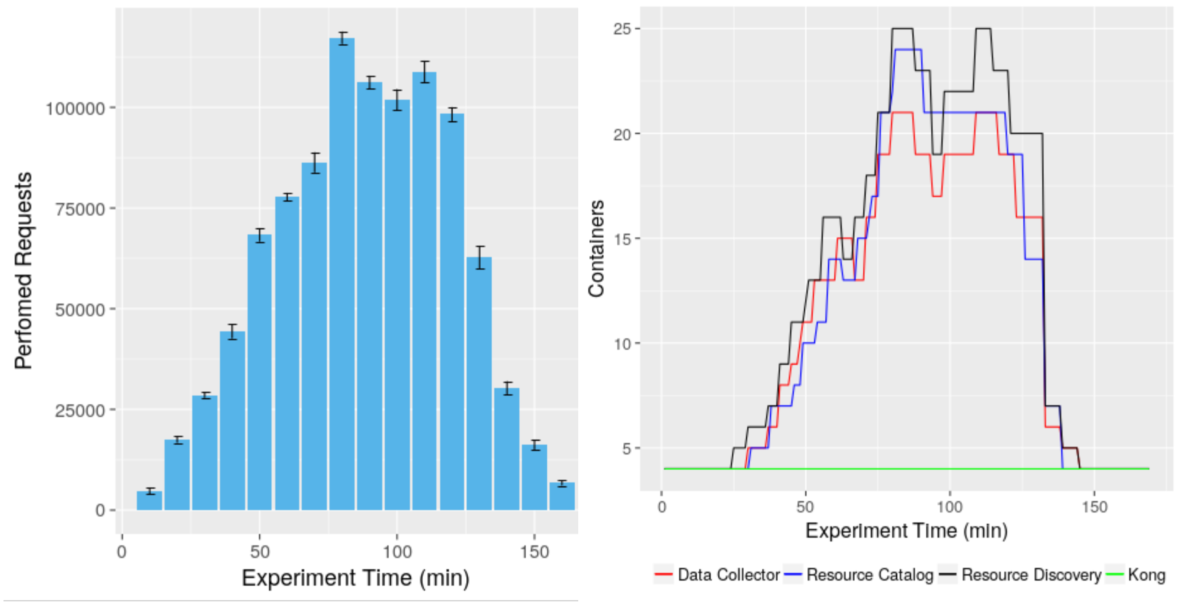


Figure 5.7: *InterSCity services autoscaling*

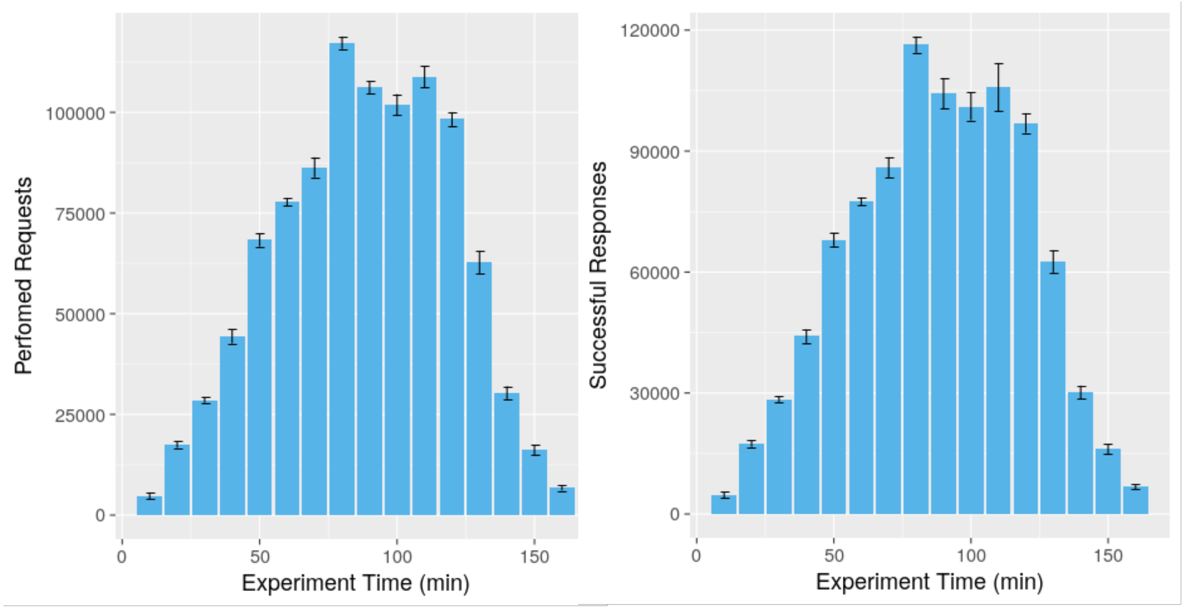


Figure 5.8: *Average InterSCity throughput*

we are mainly interested in analyzing the performance degradation and verifying whether the platform is scaling appropriately to serve its clients within acceptable response times. For this purpose, we collected the response time from the client's point of view, as shown by Figure 5.9. During most of the experiment duration, the platform was able to respond in less than one second. However, differently from the throughput, the impact of the highest demand period on the observed response time is noticeable since, during a small time interval (after 110 minutes of execution), the average response time was greater than 1 second. The response time went back down to 500 milliseconds after that. But, we can see that even in periods of high-load, the response time was kept under 2s, which is an excellent result for this kind of application.

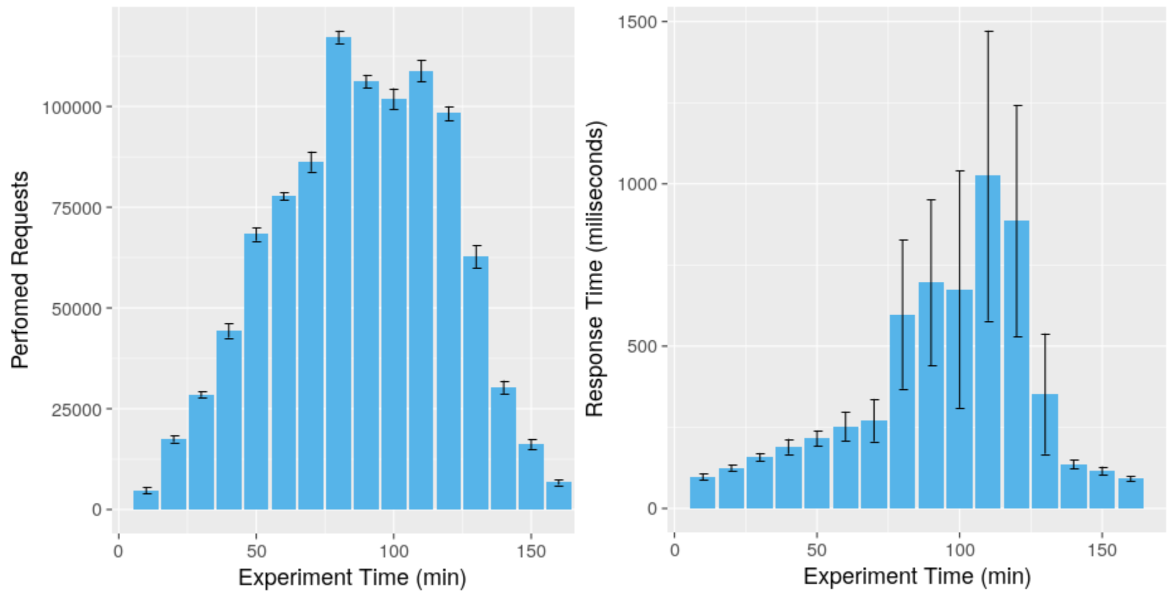


Figure 5.9: *InterSCity Average Response Time*

We should recall that the distribution of containers on the available nodes may impact the response time, as several containers may compete for computational resources if they are running on the same host machine. Moreover, although the system handles the autoscaling task every 30 seconds, we have no control over the time it takes for a container to be scheduled and become ready to receive requests. On the other hand, this distribution may also introduce a beneficial effect due to the scheduling of services that constantly interact with each other to the same machine, reducing network latency and unpredictability.

5.2.4 Threats to Validity and Limitations

In this section, we clarify and discuss some important aspects of our experiment design as they may threaten the validity of our results and affect reproducibility. Most of the threats are inherent to the environment where the experiments were carried out, as there is considerable uncertainty concerning the provisioning of resources and services in cloud environments (Tchernykh et al., 2015). As the Google Cloud

Platform is not a controlled environment, it is impossible to have complete knowledge of the system. For instance, we cannot correctly specify all properties related to the communication network used within the cluster, such as network capacity, nor ensure constant bandwidth. Such variables may directly impact results such as the observed response time.

Although Kubernetes plays a crucial role in our experiments, it also raises some concerns for their validation. Firstly, we do not control the way Kubernetes distributes new containers across the available hosts of a node pool, except for the definition of the minimum computing resource requirements for executing a specific container. As a consequence, on each round of the experiment, Kubernetes may distribute microservices differently, which may lead to a different load distribution among the hosts in use. For instance, consider the case of a new container that has the minimum requirement of 30% of available CPU and is allocated on a host whose CPU usage is already at 60%. The container would probably have less CPU time than if it had been allocated on a host with less competition for resources or, at least, manifest higher latencies. Another important aspect regarding Kubernetes is the variation in the time spent on scheduling new containers, especially in the auto-scaling task. In periods of increased workload, the delay in the allocation of new containers may directly impact the number of failed requests and the response time.

Finally, since we advocate for the use of more appropriate scenarios to test smart city platforms, it is worth mentioning the aspects related to the realism of the smart city simulations performed as part of the experiments. The conformance of the simulations with real-world future smart cities scenarios depends on good models. In this sense, the refinement of the models used to simulate car trips, as well as the behavior of drivers in search for parking spots in a large city might change the workload generated on the platform. Moreover, in addition to Smart Parking, it is essential to evaluate the platform using other smart city scenarios, which may pose different demands and require other functionalities, such as actuation on city resources.

Chapter 6

Conclusion

Smart city platforms play a key role in the development of future smart cities as they support cross-domain solutions, interoperability among multiple city systems, and resource and data sharing. However, due to various technical, practical, and methodological challenges, the community still lacks robust solutions that can be shared across smart city initiatives, as well as production environments to support scientific validation of existing proposals.

Despite the existence of numerous smart city projects, developing architectures to support city-scale environments and evaluating them is still challenging. In addition to implementing a set of functionalities to support the development of applications, smart city platforms must meet a number of important non-functional requirements. In particular, the scalability and evolvability are critical for the wide adoption of these platforms, and yet these characteristics are often ignored or superficially addressed. Most of the works that propose a new smart city platform that supposedly meets scalability requirements only present superficial discussions of design and implementation decisions that can lead to a scalable architecture. Moreover, they often do not provide scientific evidence of the feasibility of their approaches, but restricted proofs-of-concept.

This masters' research aimed at achieving both technical and scientific contributions through the proposal of a novel microservice-based open source smart city platform that provides facilities to build next-generation scalable smart city solutions and to integrate heterogeneous IoT systems. In particular, our work brings the following contributions:

- The initial design and implementation of the InterSCity platform
- Advances in the state-of-art by exploring the impact of a microservices architecture in the design, development, and deployment of scalable smart city platforms supported by experimental results and our early experience
- Advances in performance evaluation of smart city platforms with the use of a simulation-based approach for workload generation and DevOps techniques for reproducible experiments

Our early experience with the development of InterSCity shows that microservices can be properly used to build smart city solutions providing finer-grained, single-purpose building blocks that can be more easily and independently evolved compared to traditional SOA approaches. The loosely-coupled architecture provided by the microservices approach enabled us to iteratively improve the InterSCity design and implementation throughout the whole period of this research. However, it also introduces new challenges due to an increase in the overall complexity that requires the proper use of DevOps techniques, automated tests, and design patterns from the beginning of the project.

We conducted two sets of experiments to validate the proposed platform. Despite the limitations of our preliminary experiments, their results pointed towards the applicability of our approach in the context of smart cities, since the platform were able to support different scalability demands while keeping acceptable performance.

For the final evaluation of the InterSCity scalability properties, we adopted the InterSCSimulator to generate a large-scale workload considering a realistic smart city scenario. The experiment showed that InterSCity's architecture is capable of scaling up and down horizontally to handle a varying workload. More precisely, it was able to handle more than one million complex requests during approximately 3 hours, considering a Smart Parking scenario during São Paulo's rush hour. Also, the platform response time remained acceptable, mostly 1 second or below, even at the highest demand time interval. We highlight the use of modern tools such as Kubernetes and Docker containers as essential means to achieve these results.

Finally, we highlight that our open source and open science approach encourage the community to leverage the contributions described in this work and to contribute to the evolution of the field.

6.1 Future Work

Although we have evaluated the InterSCity scalability properties comprehensively, we did not cover all of its functionalities. We need to perform further experiments to continue exploring the scalability and performance properties of the platform, mostly by considering other smart cities scenarios and features. Also, we still need to advance more in methods and tools for evaluating smart city platforms to allow comparison between several existing solutions. The InterSCity research community advanced in this sense by using the InterSCSimulator to generate realistic workload to evaluate such platforms. However, there are considerable challenges such as integrating the simulator with other platforms. Other notable works in this area include the RIoTBench(Shukla and Yogesh Simmhan, 2017), a real-time IoT benchmark for distributed stream processing platforms, and the CityBench(Ali et al., 2015), which is focused on the evaluation of RDF stream processing engines within smart city applications.

There are still several open technical challenges regarding the InterSCity platform, including: (I) meeting other critical non-functional requirements, such as security and privacy; (II) extending the core functionalities to better support application development; and (III) deploying a production instance of the platform on a real city.

Although InterSCity has a prototype of a visualization service for presenting the city's resources, there are several necessary improvements related to the visualization of data. In addition, various issues related to usability and support for real-time visualization should be considered in the evolution of this service. Moreover, although the resource and capabilities abstractions are simple and flexible, they do not contemplate several aspects of the city. For example, concepts such as streets and neighborhoods are quite difficult to represent in the currently stage of the platform. Another example is related to events that may occur in the city that, although they can be identified by resources with sensor capabilities, they are not related to such resources necessarily.

Regarding extending InterSCity's functionalities, we believe two fundamental aspects require priority: Integration of Static Resources and Big Data Processing. In the following, we expose the main ideas on how to extend InterSCity in each of these areas.

Integration of Static City Resources

Currently, InterSCity does not provide an appropriate way to serve static data about city resources - resources without functional capabilities. For example, The city hall of São Paulo provides an online visualization tool, called GeoSampa[1], to represent several layers of static city resources, including: the mapping of city facilities and natural spaces, the political-administrative divisions, and mobility information. Each of these resources has a set of specific attributes which could be used by different smart city applications. Although it is possible to download the data used by GeoSampa, there is no API available to access them in a standardized and automatic way. To this end, we need to either improve the Resource Catalog or add a new microservice to manage data from static resources through a NoSQL database. Thus, InterSCity could support the discovery of static resources through Resource Discovery API and the standard access to regular city resources abstraction provided by all other city resources. Moreover, such work could also enhance the support for geolocation services by adding support for GeoJSON format, described at RFC 7946 (Butler et al., 2016), as it encodes a variety of geographic data structures and geometry types such as:

- Point - represents a single point in the world based on coordinates. This is the only format currently supported by InterSCity;
- LineString - represents a sequence of linked points. This data structure is appropriate to describe paths, such as bus lines and bike paths;
- Polygon - represents a series of linked coordinates with four or more positions, where the first and last positions are equivalent. Polygons are used to describe the delimitations of neighborhoods, buildings, and public squares.

Big Data Processing

Sharing computational resources and reusable services across several applications are some of the main purposes of building smart city platforms. More specifically, Big Data services will be vital for future smart cities as they could potentially integrate a large volume of data from different sources which must be timely processed to

[1]geosampa.prefeitura.sp.gov.br

provide valuable outcomes for cities (Al Nuaimi et al., 2015). In this sense, add Big Data support in the InterSCity platform is a major contribution that requires design and implementation effort. To this end, InterSCity must address the following two challenges: (I) implementing a generic, scalable and evolvable Big Data architecture to enable both streaming and historical data processing; and (II) providing a reusable service that can support several client applications' requirements.

Among other possibilities, client applications could use the proposed Big Data architecture to perform the following tasks:

- Creation of Composite Resources - composite resources are city resources whose context data are the aggregation of observations of two or more city resources. Composite resources must have a UUID and could be used to represent non-physical resources from city. For example, a bus line could be modeled as a composite resource that aggregates data from individual buses and stops, such as average speed, number of stops, and average waiting time.

- Aggregation of Resource Observations - In some cases, client applications could use Big Data services to access either aggregated or raw context data of city resources to perform low latency queries. For instance, an application that needs to retrieve the average speed of city roads every 30 seconds to propose faster routes for drivers.

- Resource Observations Filtering - Applications could use the proposed data processing service to filter events based on custom parameters in near real time fashion, for example, an application that is only interested in the occurrences of hospitalizations in the city's hospitals whose specialty is "pediatrics".

Bibliografia

[Abbott and Fisher 2009] Martin L. Abbott and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. 1st. Addison-Wesley Professional, 2009. isbn: 0137030428, 9780137030422 (cit. on pp. 12, 13).

[Ali et al. 2015] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. "CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets". In: *The Semantic Web - ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*. Cham: Springer International Publishing, 2015, pp. 374–389. isbn: 978-3-319-25010-6. doi: 10.1007/978-3-319-25010-6_25. url: http://dx.doi.org/10.1007/978-3-319-25010-6_25 (cit. on p. 80).

[Apolinarski et al. 2014] W. Apolinarski, U. Iqbal, and J. X. Parreira. "The GAMBAS middleware and SDK for smart city applications". In: *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)*. Mar. 2014, pp. 117–122. doi: 10.1109/PerComW.2014.6815176 (cit. on p. 15).

[Al Nuaimi et al. 2015] Eiman Al Nuaimi, Hind Al Neyadi, Nader Mohamed, and Jameela Al-Jaroodi. "Applications of big data to smart cities". In: *Journal of Internet Services and Applications* 6.1 (2015) (cit. on p. 82).

[Amaral et al. 2015] Leonardo Albernaz Amaral, Ramão Tiago Tiburski, Everton de Matos, and Fabiano Hessel. "Cooperative Middleware Platform As a Service for Internet of Things Applications". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: ACM, 2015, pp. 488–493. isbn: 978-1-4503-3196-8. doi: 10.1145/2695664.2695799. url: http://doi.acm.org/10.1145/2695664.2695799 (cit. on p. 26).

[Batista et al. 2016] D. M. Batista et al. "InterSCity: Addressing Future Internet Research Challenges for Smart Cities". In: *7th International Conference on the Network of the Future*. IEEE, Nov. 2016 (cit. on pp. 5, 25).

[Bauer et al. 2013] Martin Bauer et al. *Internet of Things – Architecture IoT-A Deliverable D1.5 – Final architectural reference model for the IoT v3.0*. Tech. rep. July 2013 (cit. on p. 9).

[BREIVOLD et al. 2012] Hongyu Pei BREIVOLD, Ivica CRNKOVIC, and Magnus LARSSON. "A systematic review of software architecture evolution research". In: *Information and Software Technology* 54.1 (2012), pp. 16–40. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2011.06.002. URL: http://www.sciencedirect.com/science/article/pii/S0950584911001376 (cit. on p. 11).

[BALALAIE et al. 2016] A. BALALAIE, A HEYDARNOORI, and P. JAMSHIDI. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52. ISSN: 0740-7459. DOI: doi.ieeecomputersociety.org/10.1109/MS.2016.64 (cit. on pp. 12, 35, 37).

[BONINO et al. 2015] D. BONINO et al. "ALMANAC: Internet of Things for Smart Cities". In: *2015 3rd International Conference on Future Internet of Things and Cloud*. Aug. 2015, pp. 309–316. DOI: 10.1109/FiCloud.2015.32 (cit. on p. 15).

[BACHANI et al. 2016] Mamta BACHANI, Umair Mujtaba QURESHI, and Faisal Karim SHAIKH. "Performance Analysis of Proximity and Light Sensors for Smart Parking". In: *Procedia Computer Science* 83 (2016). The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops, pp. 385–392. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2016.04.200. URL: http://www.sciencedirect.com/science/article/pii/S1877050916302332 (cit. on p. 51).

[BELLABAS et al. 2013] Alia BELLABAS, Fano RAMPARANY, and Marylin ARNDT. "Fiware Infrastructure for Smart Home Applications". In: *Evolving Ambient Intelligence*. Ed. by Michael J. O'GRADY et al. Cham: Springer International Publishing, 2013, pp. 308–312. ISBN: 978-3-319-04406-4 (cit. on p. 20).

[BUTLER et al. 2016] H. BUTLER et al. *The GeoJSON Format*. RFC 7946. RFC Editor, "August" 2016. DOI: 10.17487/RFC7946. URL: https://tools.ietf.org/html/rfc7946 (cit. on pp. 42, 81).

[CARAGLIU et al. 2009] A. CARAGLIU, C. DEL BO, and P. NIJKAMP. "Smart Cities in Europe". In: *Journal of Urban Technology 18(0048)*. Taylor & Franci, 2009, pp. 45–59. DOI: 10.1080/10630732.2011.601117 (cit. on p. 7).

[CHENG et al. 2015] Bin CHENG, Salvatore LONGO, Flavio CIRILLO, Martin BAUER, and Ernoe KOVACS. "Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander". In: *Big Data (BigData Congress), 2015 IEEE International Congress on*. IEEE. 2015, pp. 592–599 (cit. on p. 17).

[COLA et al. 2015] Simone Di COLA, Cuong TRAN, Kung-Kiu LAU, Antonio CELESTI, and Maria FAZIO. "A Heterogeneous Approach for Developing Applications with FIWARE". In: *Service Oriented and Cloud Computing.*

Lecture Notes in Computer Science. Vol. 9306. Springer, Cham. 2015. DOI: https://doi.org/10.1007/978-3-319-24072-5_5 (cit. on pp. 20, 21).

[DRAGONI et al. 2016] Nicola DRAGONI et al. "Microservices: yesterday, today, and tomorrow". In: *CoRR* abs/1606.04036 (2016). URL: http://arxiv.org/abs/1606.04036 (cit. on pp. 11, 13, 14).

[FAZIO et al. 2012] M. FAZIO, M. PAONE, A. PULIAFITO, and M. VILLARI. "Heterogeneous Sensors Become Homogeneous Things in Smart Cities". In: *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*. July 2012, pp. 775–780. DOI: 10.1109/IMIS.2012.136 (cit. on pp. 2, 8, 26).

[FREIRE et al. 2012] Juliana FREIRE, Philippe BONNET, and Dennis SHASHA. "Computational Reproducibility: State-of-the-art, Challenges, and Database Research Opportunities". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 593–596. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213908. URL: http://doi.acm.org/10.1145/2213836.2213908 (cit. on p. 3).

[GOPU et al. 2016] Arvind GOPU et al. "Trident: scalable compute archives: workflows, visualization, and analysis". In: vol. 9913. 2016, 99131H-99131H-12. DOI: 10.1117/12.2233111. URL: http://dx.doi.org/10.1117/12.2233111 (cit. on p. 14).

[GONZÁLEZ and ROSSI 2011] J. A. GONZÁLEZ and A. ROSSI. *New trends for smart cities, open innovation Mechanism in Smart Cities*. Tech. rep. European commission within the ICT policy support programme, 2011 (cit. on p. 7).

[HERNÁNDEZ-MUÑOZ et al. 2011] José M. HERNÁNDEZ-MUÑOZ et al. "The Future Internet". In: ed. by John DOMINGUE, Alex GALIS, Anastasius GAVRAS, Theodore ZAHARIADIS, and Dave LAMBERT. Berlin, Heidelberg: Springer-Verlag, 2011. Chap. Smart Cities at the Forefront of the Future Internet, pp. 447–462. ISBN: 978-3-642-20897-3. URL: http://dl.acm.org/citation.cfm?id=1983741.1983773 (cit. on pp. 2, 8, 26).

[HUMMER et al. 2013] Waldemar HUMMER, Florian ROSENBERG, Fábio OLIVEIRA, and Tamar EILAM. "Testing Idempotence for Infrastructure as Code". In: *Middleware 2013*. Ed. by David EYERS and Karsten SCHWAN. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 368–388. ISBN: 978-3-642-45065-5 (cit. on p. 38).

[HOHPE and WOOLF 2003] Gregor HOHPE and Bobby WOOLF. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683 (cit. on p. 34).

[ISO 1988] ISO. *ISO 8601:1988. Data elements and interchange formats — Information interchange — Representation of dates and times.* Geneva, Switzerland: International Organization for Standardization, 1988, p. 14. URL: http://www.iso.ch/cate/d26780.html (cit. on p. 45).

[Joy 2015] A. M. Joy. "Performance comparison between Linux containers and virtual machines". In: *2015 International Conference on Advances in Computer Engineering and Applications.* Mar. 2015, pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727 (cit. on p. 38).

[Krylovskiy et al. 2015] A. Krylovskiy, M. Jahn, and E. Patti. "Designing a Smart City Internet of Things Platform with /Microservice Architecture". In: *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on.* Aug. 2015, pp. 25–30. DOI: 10.1109/FiCloud.2015.55 (cit. on pp. 3, 21, 22).

[Le et al. 2015] V. D. Le et al. "Microservice-based architecture for the NRDC". In: *2015 IEEE 13th International Conference on Industrial Informatics (INDIN).* July 2015, pp. 1659–1664. DOI: 10.1109/INDIN.2015.7281983 (cit. on p. 14).

[Lewis and Fowler 2014] James Lewis and Martin Fowler. *Microservices: a definition of this new architectural term.* 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 03/06/2017) (cit. on pp. 11, 12, 14).

[Leach et al. 2005] Paul J. Leach, Michael Mealling, and Rich Salz. *A Universally Unique IDentifier (UUID) URN Namespace.* RFC 4122. http://www.rfc-editor.org/rfc/rfc4122.txt. RFC Editor, July 2005 (cit. on p. 30).

[Moura et al. 2016] P. Moura, F. Kon, S. Voulgaris, and M. van Steen. "Dynamic resource allocation using performance forecasting". In: *2016 International Conference on High Performance Computing Simulation (HPCS).* July 2016, pp. 18–25. DOI: 10.1109/HPCSim.2016.7568311 (cit. on p. 38).

[Nations 2014] United Nations. *World Urbanization Prospects: The 2014 Revision.* Tech. rep. ST/ESA/SER.A/352. New York: Department of Economic and Social Affairs of the United Nations, 2014 (cit. on p. 1).

[Neirotti et al. 2014] P. Neirotti, A. De Marco, A. C. Cagliano, G. Mangano, and F. Scorrano. "Current trends in Smart City initiatives: Some stylised facts". In: *Cities* 38 (2014), pp. 25–36. DOI: 10.1016/j.cities.2013.12.010. URL: http://dx.doi.org/10.1016/j.cities.2013.12.010 (cit. on pp. 1, 7).

[Newman 2015] S Newman. *Building Microservices.* O'Reilly Media, 2015. ISBN: 978-1-4919-5035-7 (cit. on pp. 11, 12, 37).

[Partridge 2004] Helen L. Partridge. "Developing a human perspective to the digital divide in the 'smart city'". In: *Australian Library and Information*

Association Biennial Conference. Ed. by Helen Partridge. Gold Coast, Queensland, Australia, 2004. URL: http://eprints.qut.edu.au/1299/ (cit. on p. 7).

[PCAST 2016] PCAST. *Technology and the Future of Cities, Report To The President*. Tech. rep. Executive Office of the President, United States, Feb. 2016, p. 99. URL: https://www.whitehouse.gov/sites/whitehouse.gov/files/images/Blog/PCAST%5C%20Cities%5C%20Report%5C%20_%5C%20FINAL.pdf (cit. on p. 2).

[Peng 2011] Roger D. Peng. "Reproducible Research in Computational Science". In: *Science* 334.6060 (2011), pp. 1226–1227. ISSN: 0036-8075. DOI: 10.1126/science.1213847. eprint: http://science.sciencemag.org/content/334/6060/1226.full.pdf. URL: http://science.sciencemag.org/content/334/6060/1226 (cit. on p. 3).

[Perera et al. 2014] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. "Context Aware Computing for The Internet of Things: A Survey". In: *IEEE Communications Surveys Tutorials* 16.1 (First 2014), pp. 414–454. ISSN: 1553-877X. DOI: 10.1109/SURV.2013.042313.00197 (cit. on p. 9).

[Pereira et al. 2018] Carlos Pereira, João Cardoso, Ana Aguiar, and Ricardo Morla. "Benchmarking Pub/Sub IoT middleware platforms for smart services". In: *Journal of Reliable Intelligent Environments* 4.1 (Apr. 2018), pp. 25–37. ISSN: 2199-4676. DOI: 10.1007/s40860-018-0056-3. URL: https://doi.org/10.1007/s40860-018-0056-3 (cit. on p. 21).

[Salhofer 2018] Peter Salhofer. "Evaluating the FIWARE Platform: A Case-Study on Implementing Smart Application with FIWARE". In: *Proceedings of the 51st Hawaii International Conference on System Sciences*. 2018, pp. 5797–5805. DOI: 10.24251/HICSS.2018.726. URL: http://hdl.handle.net/10125/50615 (cit. on p. 20).

[L. Sanchez et al. 2011] L. Sanchez et al. "SmartSantander: The meeting point between Future Internet research and experimentation and the smart cities". In: *2011 Future Network Mobile Summit*. June 2011, pp. 1–8 (cit. on p. 3).

[Luis Sanchez et al. 2014] Luis Sanchez et al. "SmartSantander: IoT experimentation over a smart city testbed". In: *Computer Networks* 61 (2014). Special issue on Future Internet Testbeds ¿ Part I, pp. 217–238. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.bjp.2013.12.020. URL: http://www.sciencedirect.com/science/article/pii/S1389128613004337 (cit. on p. 16).

[E. F. Z. Santana et al. 2017] E. F. Z. Santana, N. Lago, F. Kon, and D. S. Milojicic. "InterSCSimulator: Large-Scale Traffic Simulation in Smart Cities using Erlang". In: *18th Workshop on Multi-agent-based Simulation (MABS)*. 2017 (cit. on p. 70).

[SANTANA et al. 2017] SANTANA, A. P. CHAVES, M. A. GEROSA, F. KON, and D. S. MILOJICIC. “Software Platforms for Smart Cities: Concepts, Requirements, Challenges, and a Unified Reference Architecture”. In: *ACM Computing Surveys* 50.6 (Nov. 2017), 78:1–78:37. ISSN: 0360-0300. URL: http://doi.acm.org/10.1145/3124391 (cit. on pp. 2–4, 8, 10, 25).

[SERRANO et al. 2015] M. SERRANO et al. *IoT Semantic Interoperability: Research Challenges, Best Practices, Recommendations and Next Step*. Tech. rep. Mar. 2015 (cit. on pp. 9, 10).

[SILVA et al. 2013] Welington M. SILVA et al. “Smart Cities Software Architectures: A Survey”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC ’13. Coimbra, Portugal: ACM, 2013, pp. 1722–1727. ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480688. URL: http://doi.acm.org/10.1145/2480362.2480688 (cit. on p. 8).

[SOLDATOS et al. 2015] John SOLDATOS et al. “OpenIoT: Open Source Internet-of-Things in the Cloud”. In: *Interoperability and Open-Source Solutions for the Internet of Things: International Workshop, FP7 OpenIoT Project, Held in Conjunction with SoftCOM 2014, Split, Croatia, September 18, 2014, Invited Papers*. Ed. by Ivana PODNAR ŽARKO, Krešimir PRIPUŽIĆ, and Martin SERRANO. Cham: Springer International Publishing, 2015, pp. 13–25. ISBN: 978-3-319-16546-2. DOI: 10.1007/978-3-319-16546-2_3. URL: http://dx.doi.org/10.1007/978-3-319-16546-2_3 (cit. on p. 15).

[STEINMETZ et al. 2017] C. STEINMETZ et al. “Ontology-driven IoT code generation for FIWARE”. In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. July 2017, pp. 38–43. DOI: 10.1109/INDIN.2017.8104743 (cit. on p. 20).

[SHUKLA and YOGESH SIMMHAN 2017] Anshu SHUKLA and Shilpa Chaturvedi and YOGESH SIMMHAN. “RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms”. In: *CoRR* abs/1701.08530 (2017). URL: http://arxiv.org/abs/1701.08530 (cit. on p. 80).

[TAIBI 2013] Fathi TAIBI. “Reusability of Open-source Program Code: A Conceptual Model and Empirical Investigation”. In: *SIGSOFT Softw. Eng. Notes* 38.4 (July 2013), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/2492248.2492276. URL: http://doi.acm.org/10.1145/2492248.2492276 (cit. on p. 26).

[TCHERNYKH et al. 2015] Andrei TCHERNYKH, Uwe SCHWIEGELSOHN, Vassil ALEXANDROV, and El-ghazali TALBI. “Towards Understanding Uncertainty in Cloud Computing Resource Provisioning”. In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, pp. 1772–1781. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.05.387. URL: http://www.sciencedirect.com/science/article/pii/S1877050915011953 (cit. on p. 77).

[Villanueva et al. 2013] F. J. Villanueva, M. J. Santofimia, J. Barba, and J. C. Lópes. “Civitas: The Smart City Middleware, from Sensors to Big Data”. In: *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*. 2013, pp. 445–450 (cit. on pp. 1, 2, 8, 15).