Informatics

Quality of Service aware Resource Management for Edge Systems

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

M.Sc. Cosmin Florin Avasalcai
Registration Number 11743103

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar

External reviewers:
Frank Leymann. University of Stuttgart, Germany.
George Pallis. University of Cyprus, Cyprus.

Vienna, 20th July, 2021

Cosmin Florin Avasalcai — Schahram Dustdar

Declaration of Authorship

M.Sc. Cosmin Florin Avasalcai

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 20th July, 2021

Cosmin Florin Avasalcai

Acknowledgements

Many persons have helped me along the way to be the person that I am today. First, I would like to thank my supervisor Univ. Prof. Dr. Schahram Dustdar for having me in the Distributed Systems Group and offering me continuous support and guidance throughout my PhD. Furthermore, I would like to thank Dr. Christos Tsigkanos for all the countless meetings and discussions. Many thanks go to the DSG group for the nice working environment and great chats.

I would also like to thank Prof. Paul Pop, from DTU Compute, for guiding me during my Master studies and giving me the possibility to have my first contact with research. Finally, my greatest thanks go to my wife and my family for their unconditional love and support.

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA–Fog Computing for Robotics and Industrial Automation.

Kurzfassung

Anspruchsvolle latenzempfindliche Anwendungen stellen hohe Anforderungen wie hohe Verfügbarkeit und niedrige Latenz. Die aktuellen Cloud-zentrierten Systeme stehen vor der Herausforderung, die strengen Anforderungen der Anwendung zu erfüllen. Infolgedessen haben Forscher zwei neue Paradigmen vorgeschlagen, i.e., Edge und Fog Computing, als Alternative zur Bereitstellung anspruchsvoller IoT-Anwendungen, die näher am Rand des Netzwerks liegen. Durch die Erweiterung des Cloud Systems um diese beiden Paradigmen erhalten wir ein Kantensystem. Edgesysteme wurden als Lösung identifiziert, um mehr Ressourcen näher am Endbenutzer zu verteilen, da die Erfüllung der Anwendungsanforderungen zur Laufzeit, mit Unsicherheiten und auf dezentrale Weise erfolgen muss. Die verteilte Natur der Edge-Systeme macht die Anwendungsentwicklung jedoch schwieriger, da der Entwickler die Funktionalität der Anwendung in mehrere Microservices aufteilen muss. Darüber hinaus definiert eine hohe Volatilität das Kantensystem, da Kantenknoten durch (i) Heterogenität und (ii) Mobilität gekennzeichnet sind, was einen Knoten unzuverlässig macht - ein Knoten kann ausfallen oder das Netzwerk unerwartet verlassen. Infolgedessen ist die Bereitstellung und Verwaltung von Anwendungen unter Volatilität schwieriger. Dies erfordert neuartige Methoden zur Anwendungsentwicklung und Ressourcenverwaltungstechniken, die den Anforderungen der Anwendung entsprechen und dem Entwickler helfen, eine Anwendung im Zielkantensystem zu entwickeln, bereitzustellen und zu verwalten. Die Entwicklung dieser Techniken ist jedoch keine triviale Aufgabe.

In dieser Arbeit stellen wir neuartige Methoden und Ressourcenmanagement Frameworks zur Verfügung, um die effiziente Nutzung der verfügbaren Ressourcen des Randknotens zu ermöglichen und die korrekte Anwendungsfunktionalität während der gesamten Ausführung aufrechtzuerhalten. Unser Ziel ist es, (i) den Entwickler während des Anwendungsentwicklungsprozesses zu unterstützen, (ii) die latenzempfindlichen Anwendungen im Zielkantensystem bereitzustellen und (iii) sicherzustellen, dass bereitgestellte Anwendungen während ihrer Lebensdauer betriebsbereit bleiben. Wir schlagen zunächst EdgeFlow vor, eine neue Methode für die Entwicklung und Bereitstellung latenzempfindlicher IoT-Anwendungen auf dem Edge-System. Der Zweck von EdgeFlow besteht darin, den Entwickler während des Anwendungsentwicklungsprozesses zu unterstützen, indem dem Entwickler ermöglicht wird, die Anwendungsanforderungen zu definieren und diese zur Entwurfszeit zu validieren. Drei verschiedene Phasen charakterisieren unser vorgeschlagenes Framework, i.e., (i) Entwicklung, (ii) Validierung und (iii) Bereitstellung. Zu diesem Zweck schlagen wir eine Erweiterung des Flow-Based-Programming-Paradigmas um neue

Timing- und Ressourcenanforderungen vor. Darüber hinaus bieten wir eine Ressourcenverwaltungstechnik an, die Sie bei der Bereitstellung und Validierung unterstützt. Im nächsten Teil unserer Arbeit konzentrieren wir uns auf die Bereitstellung einer neuartigen dezentralen Ressourcenverwaltungstechnik und des dazugehörigen technischen Rahmens für die Bereitstellung von Anwendungen auf Geräten mit eingeschränkten Ressourcen. Das vorgeschlagene Framework ermöglicht die effiziente Nutzung verfügbarer Ressourcen, die auf ressourcenbeschränkte Randknoten verteilt sind. Das Ressourcenverwaltungsframework verwendet die Erfüllbarkeit, um zur Laufzeit eine Bereitstellung für eine Anwendung zu finden. Das erzeugte Mapping entspricht den Ressourcenanforderungen von Microservices und den konstruktionsbedingten Einschränkungen der Anwendungslatenz. Unser Ansatz gewährleistet eine nahtlose Bereitstellung zur Laufzeit, sofern keine Kenntnisse der Geräteressourcen zur Entwurfszeit vorliegen. Wir schlagen ferner ein neues robustes IoT-Anwendungsmodell vor, das einer hierarchischen Architektur folgt. Wir modellieren eine Anwendungskomponente mit mehreren Konfigurationen - jede Konfiguration hat ein anderes Funktionsniveau und unterschiedliche Ressourcenanforderungen. Darüber hinaus erweitern wir das dezentrale Ressourcen-Framework, um das neue Anwendungsmodell bereitstellen zu können. Schließlich schlagen wir ein adaptives Framework vor, mit dem eine Microservice-Anwendung auf einem Edge-System effizient bereitgestellt und gewartet werden kann. Unser Framework befasst sich mit zwei miteinander verflochtenen Problemen: (i) finden einer Microservice-Platzierung zwischen Geräten und (ii) erstellen eines Aufrufpfads, der der bereitgestellten Anwendung dient. Für dieses Framework ist es unser Ziel, die korrekte Funktionalität der Anwendung aufrechtzuerhalten, indem die vorgegebenen Anforderungen hinsichtlich End-to-End-Latenz und Verfügbarkeit erfüllt werden.

Abstract

Demanding latency-sensitive applications have stringent requirements such as high availability and low latency. The current cloud-centric systems face challenges in satisfying the application's stringent requirements. As a result, researchers have proposed two new paradigms, i.e., edge and fog computing, as an alternative to deploying demanding IoT applications closer to the edge of the network. By extending the cloud system with these two paradigms, we obtain an edge system. Edge systems have been identified as a solution to distribute more resources closer to the end-user since meeting application demands must occur at runtime, facing uncertainty, and in a decentralized manner. However, the edge systems' distributed nature makes the application development more challenging since the developer must divide the application's functionality into multiple microservices. Furthermore, high volatility defines the edge system due to edge nodes being characterized by (i) heterogeneity and (ii) mobility, making a node unreliable – a node may fail or leave the network unexpectedly. As a result, the application deployment and management under volatility is more challenging. This calls for novel application development methodologies and resource management techniques that comply with the application's requirements and aids the developer to develop, deploy, and manage an application in the target edge system. However, developing these techniques is not a trivial task.

In this thesis, we provide novel methodologies and resource management frameworks to enable the efficient utilization of the edge node available resources and maintain the correct application functionality throughout its entire execution. Our objective is to (i) aid the developer during the application development process, (ii) deploy the latency-sensitive applications in the target edge system, and (iii) ensure that deployed applications remain operational during their lifespan. We start by proposing EdgeFlow, a new methodology for latency-sensitive IoT applications development and deployment on the edge system. The purpose of EdgeFlow is to assist the developer during the application development process by allowing the developer to define the application requirements and validate them at design time. Three different stages characterize our proposed framework, i.e., the (i) development, (ii) validation, and (iii) deployment. To this end, we propose an extension of the Flow-Based Programming paradigm with new timing and resource requirements. Moreover, we provide a resource management technique to assist with the deployment and validation stages. In the next part of our thesis, we focus on providing a novel decentralized resource management technique and accompanying

technical framework to deploy applications on resource-constrained devices. The proposed framework enables the efficient utilization of available resources distributed between resource-constrained edge nodes. The resource management framework uses satisfiability to find at runtime a deployment for an application; the mapping produced is compliant with microservices resource requirements and the application latency constraints by construction. Our approach ensures seamless deployment at runtime, assuming no design-time knowledge of device resources. We further propose a new robust IoT application model that follows a hierarchical architecture. We model an application component using multiple configurations – each configuration has a different functionality level and resource requirements. Additionally, we extend the decentralized resource framework to be able to deploy the new application model. Finally, we propose an adaptive framework capable of efficiently deploying and maintaining a microservice application on an edge system. Our framework tackles two intertwined problems – (i) finding a microservice placement across devices and (ii) building an invocation path that serves the deployed application. For this framework, our objective is to maintain the correct application's functionality by satisfying its given requirements in terms of end-to-end latency and availability.

Contents

List of publications

The contents of this thesis is based on the research work published in the following conferences, journals, and book chapters. For a full list of publications please visit the following website [1]

[1] C. Avasalcai, C. Tsigkanos, and S. Dustdar. Resource management for latency-sensitive iot applications with satisfiability. *IEEE Transactions on Services Computing*, 2021.

[2] Cosmin Avasalcai and Schahram Dustdar. Latency-aware distributed resource provisioning for deploying iot applications at the edge of the network. In Kohei Arai and Rahul Bhatia, editors, *Advances in Information and Communication*, pages 377–391, Cham, 2019. Springer International Publishing.

[3] Cosmin Avasalcai and Schahram Dustdar. Edge computing: Use cases and research challenges (to appear). In *Handbook Industrie 4.0 Vol. 5*, 2021.

[4] Cosmin Avasalcai, Ilir Murturi, and Schahram Dustdar. *Edge and Fog: A Survey, Use Cases, and Future Challenges*, chapter 2, pages 43–65. John Wiley & Sons, Ltd, 2020.

[5] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Decentralized resource auctioning for latency-sensitive edge computing. In *IEEE International Conference on Edge Computing (EDGE)*, 2019.

[6] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Adaptive volatile edge systems management at runtime with satisfiability. *ACM Transactions on Internet Technology (accepted)*, 2021.

[7] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Edgeflow - developing and deploying latency-sensitive iot edge applications. *IEEE Internet of Things Journal (accepted)*, 2021.

[8] Cosmin Avasalcai, Bahram Zarrin, Paul Pop, and Schahram Dustdar. Efficient hosting of robust iot applications on edge computing platform. In *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, 2020.

[1] https://dsg.tuwien.ac.at/team/cavasalcai/

CHAPTER 1

Introduction

The adoption rate of Internet of Things (IoT) devices has lead to a solid global infrastructure of internet-connected devices, capable of monitoring and controlling the surrounding environment via sensors and actuators as well as sending data to other devices over the Internet [KKSF10]. Since IoT devices are resource-constrained devices not capable of processing the generated data, researchers have proposed cloud computing as a paradigm to deliver and consume digital resources [AFG$^+$10]. Combining the cloud paradigm with the IoT has converged to a cloud-centric geographically distributed system capable of executing different IoT applications. However, with the recent increase in connected IoT devices, new emergent latency-sensitive IoT applications have been developed. A latency-sensitive IoT application[1] has stringent requirements, e.g., low latency, high availability, better privacy and security, and application resource requirements that current cloud-centric solutions fail to satisfy since high volumes of data must be transferred to the cloud [WZZ$^+$17].

To address the shortcoming of cloud computing, researchers have proposed two new paradigms, i.e., edge computing [Sat17] and fog computing [BMNZ14]. Fog computing enables the utilization of available computational resources found at the edge of the network [BMNZ14] – a paradigm consisting of multiple geo-distributed fog devices capable of hosting deployed IoT applications. The purpose of fog computing is to extend the cloud-centric system such that it may satisfy the stringent requirements of IoT applications. Similarly, the edge computing underlying purpose is to assist the cloud in executing IoT applications. However, in contrast to fog computing, edge computing enables computational resources at or near the location of data, pushing even closer to the user the execution of IoT applications [AAY$^+$17, SCZ$^+$16]. As a result, by combining

[1]In this thesis, the term IoT applications is used to refer to latency-sensitive IoT applications, unless otherwise stated.

the three paradigms, we obtain a new system[2] consisting of four layers, i.e., cloud, fog, edge, and IoT devices (see Figure 1.1) – an edge system that retains the advantages of all involved paradigms. Nevertheless, deploying and maintaining an IoT application on an edge system is challenging since heterogeneity, uncertainty, and limited resource capabilities define an edge and fog node.

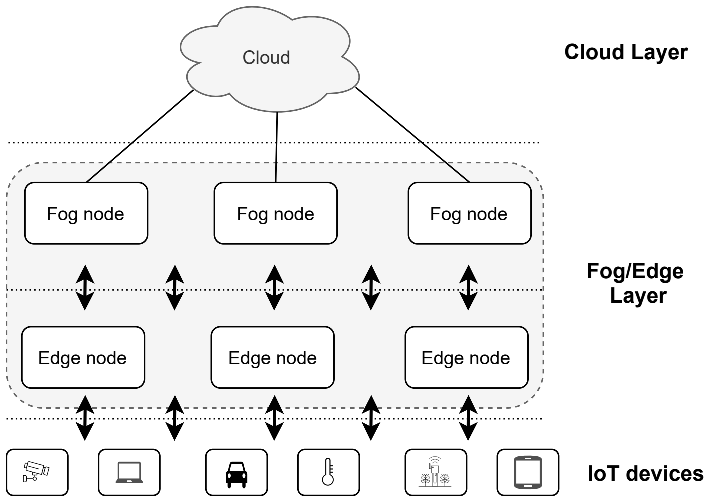


Figure 1.1: Cloud, Fog, and Edge Computing architecture overview.

Consider an IoT application that is particularly data-intensive and requires low latency communication to function properly. To satisfy its requirements, a deployment strategy should as much as possible take advantage of the available resources distributed between the edge and fog layers and avoid utilization of the cloud layer, as latency would be prohibitive and uplink bandwidth may be saturated. Besides the initial deployment, an adaptive strategy should manage the application and ensure correct functionality during its execution. However, benefiting from distributed computational resources is not trivial and requires novel resource management techniques. We discuss the IoT application's deployment and management challenges in detail in section 1.1.

1.1 Problem Statement

The emerging edge systems facilitate the deployment of IoT applications closer to the end-user. However, these systems make the deployment and management of IoT applications more challenging, creating new research problems that must be addressed. In this section, we define the scope of this thesis by presenting a detailed description of the addressed challenges.

[2]In this thesis, we refer to this new system as edge system since it aims to migrate computation closer to the edge of the network.

The purpose of this thesis is to enable the migration of IoT applications from cloud-centric systems to distributed edge systems. More precisely, we enable the efficient utilization of the distributed available computational resources found in an edge system and maintain the correct application functionality throughout its entire life cycle. We envision an ecosystem providing methodologies and frameworks to (i) aid the developer to develop emergent IoT applications, (ii) deploy these applications in the target edge system, and (iii) manage the application at runtime to ensure correct functionality during its execution. The proposed system reduces the complexity of deploying an application in a distributed edge system and provides additional guidance to the developer during the application development process. Furthermore, we provide a seamless application deployment and management process that requires no technical knowledge from the user – an approach that enables both companies and private developers to rapidly develop, deploy, and manage IoT applications in an edge system.

The distributed nature and the location of fog and edge nodes bring many advantages in an edge system, enabling the deployment of IoT applications closer to the end user [AMD20]. At the same time, the edge and fog computing paradigms introduce new challenges during the IoT application development process, like dividing the application functionality into different microservices. Previously, in cloud-centric systems, the IoT application was always deployed at the same location, i.e., in the cloud, where enough resources are available to execute the entire application. In an edge system, the nodes share the total computational resources available in the system, where often a single node cannot host the IoT application alone. Therefore, a new IoT application model is proposed in the research literature [MB18] that breaks the IoT application functionality into different computational entities. This is in line with typical microservice-based applications, i.e., ones defined as a microservice composition, where interconnected microservices create a workflow to achieve a certain goal [WIH16]. An IoT application is divided into small, modular, and easily deployable microservices; a microservice can be a stateful or serverless function. The overall IoT application functionality is then defined in terms of a communication flow among those microservices. Individual microservices may have resource requirements – for instance, storage, memory, and some dependency on local data or dedicated hardware, to support, e.g., machine learning functionalities. Considering the new IoT application model, the development process has become more complex and challenging. Currently, the developer lacks the proper methodology to develop such IoT applications and define the microservices' resource requirements as well as the applications' objectives.

Once the developer models the IoT application and provides all the information required, the deployment process can start. Deploying an IoT application on an edge system is not a trivial task and requires novel resource management techniques to achieve the proper execution of an IoT application. Resource management, in this context [TN18a], aims at enabling collaboration between edge and fog nodes by sharing their available computational resources. In such a setting, IoT applications are deployed on possibly resource-constrained devices and in dynamic networks where high uncertainty is introduced by (i) node mobility, (ii) node heterogeneity (i.e., a node can be a resource-constrained device as well as a powerful server), and (iii) lack of knowledge at design time of network topology and edge nodes' available resources. Previously, deployment of applications in an edge system has been generally tackled from two perspectives: (i) task offloading from resource-constrained devices to improve objectives such as energy consumption [MB18] or (ii) relying on the cloud to perform resource allocation [RGXZ17]. Still, such approaches do not sufficiently take into account latency application requirements, do not consider the node's preferences, and assume knowledge of participant nodes' internals.

The successful deployment of an IoT application in an edge system is not enough to ensure correct functionality since edge computing is characterized by a high degree of distribution that introduces volatility – computational nodes that participate on edge systems are spatially distributed [TGBG20] and may fail or leave the system often [TND19]. As a result, we need to ensure that during the execution of an IoT application, we can adapt to any changes in the system. To achieve this, we must combine resource allocation with resource migration techniques to deploy and manage an application in an edge system.

1.2 Research Questions

The aforementioned challenges represent the motivation behind the research conducted during the course of this thesis. In this section, we present the key research questions that we investigate and answer in this thesis:

Q1: *What is a suitable programming model and methodology to develop novel microservice-based applications efficiently, providing sufficient information to enable its deployment?*

The recent adoption of edge and fog computing has created an environment where available computational resources are distributed among different nodes. As a result, to successfully deploy an IoT application in an edge system we must efficiently use these resources. The application development process plays an instrumental role in finding a deployment strategy since it acts as the means to collect as much information as possible regarding the deployed application. The challenge for the application's developer stays in: (i) defining and validating the microservices' resource requirements, (ii) defining constraints for the application's communication flow, and (iii) set the application's overall objectives; information that impacts the application deployment process. Therefore, helping the developer to understand, during the development process, if the defined constraints and requirements are suitable for the target edge system is crucial for designing and deploying IoT applications on different systems.

Q2: *How to efficiently deploy an application on resource-constrained edge nodes, particularly in the absence of cloud resources?*

To deploy an IoT application in an edge system, where computational resources are distributed among multiple edge and fog nodes, has several challenges. The main challenge is to efficiently utilize the available resources present in the edge layer (see Figure 1.1), since at this layer nodes often have limited computational resources. Due to its proximity to end-devices, the edge layer provides the fastest response times for applications deployed on it. As a result, we can find edge nodes placed in remote locations as well, where connection to the cloud is not stable, e.g., on an offshore platform. Furthermore, since the edge layer is characterized by uncertainty, we must employ our deployment technique at runtime to consider the current internal status of edge nodes (i.e., available computational resources at the deployment time). Therefore, it is important to build novel resource management techniques capable of deploying an application on resource-constrained edge nodes at runtime.

Q3: *How to deploy and manage an application in a volatile edge system?*

As mentioned in Section 1.1, edge computing is characterized by a high degree of distribution that introduces volatility and uncertainty. Under these conditions, finding a deployment strategy is not enough, we require adaptive techniques to ensure the correct functionality of the deployed application. This volatility of edge systems represents the challenge when preserving the application's functionality throughout the entire life span. Therefore, the deployment framework must adapt to node failure, maintaining the application stable at the edge of the network.

1.3 Scientific Contributions

In this section, we present the contributions that this thesis makes to the state of the art by addressing the aforementioned research questions. These contributions are:

C1: *Programming models, techniques, and methodologies for developing and deploying IoT applications in an edge system.*

With the new IoT application model, the application developer must correctly define the application resource requirements, communication flow, and objectives. We propose an IoT application development and deployment methodology to address the challenges associated with the application development process. The methodology consists of two stages, i.e., the development stage and the validation and deployment stage – both taking place at design time. For the former, we propose an extension to the Flow-Based Programming (FBP) paradigm with timing and resource requirements, providing the means to collect all critical information associated with an application. For the latter, we propose a resource management technique to validate and find a deployment strategy that satisfies all application's requirements, considering the target edge system. The contribution is presented in detail in Chapter 4 and was originally presented in [AZD20].

C2: *A resource management technical framework for application deployment on resource-constrained devices.*

We tackle the issue of application deployment on resource-constrained nodes found in an edge system. In this contribution, we present a decentralized resource management technical framework, aiming to deploy IoT application on the edge layer, guaranteeing adherence to (i) defined application requirements and objectives and (ii) resource preferences of participating nodes. With our framework, we are able to find a satisfiable deployment strategy at runtime, considering the current status of participant nodes. We present the contribution originally in [ATD20b] and [ATD19a] and provide a detail description in Chapter 5.

C3: *A novel IoT application model and resource management technical framework for application deployment on resource-constrained devices.*

Taking advantage of the available resources found in the edge layer is highly dependent on the resource requirements of the application's microservices. If a microservice has demanding requirements that cannot be fulfilled by a single resource-constrained device, then it is impossible to find a deployment strategy exclusively on the edge layer. As a result, to improve the successful deployment of stringent IoT applications on the edge layer, we propose a robust IoT application model where the application's functionality is divided into composite computational entities. A composite entity offers more granularity, enabling the deployment of the application fully at the edge of the network. Furthermore, we extend the decentralized resource management technical framework with a new technique to enable the participant nodes to better express their preferences. We present the contribution originally in [AZPD20] and provide a detail description in Chapter 6.

C4: *An adaptive technical framework for deployment and management of IoT applications in edge systems.*

In this contribution, we provide an adaptive technical framework enabling the edge system to manage the deployed IoT application throughout its execution phase. We tackle the issue of satisfying application constraints like availability and latency in an edge system characterized by uncertainty and volatility. Our technical framework considers two different problems, i.e., application deployment and runtime management, employed at different times during the application's life cycle. For the application deployment, we propose a resource placement technique to ensure the correct placement of microservices according to their resource requirements. In contrast, for the runtime application management, we employ a technique to recover the application's functionality after the appearance of disruptions in the edge system. This contribution is presented in detail in Chapter 7 and was originally presented in [ATD20a].

1.4 Organization of the Thesis

This thesis is based on the contributions presented in the original research papers published during the author's doctoral studies. If have re-worked and extended each contribution to fit the overall context of the thesis.

The remainder of the thesis is organized as follows: Chapter 2 describes the background information and introduces the concepts and terminology used throughout the entire thesis. Chapter 3 summarizes the related work categorized based on the main contributions of this thesis, i.e., application development and resource management techniques. The main contributions of the thesis, outlined in Section 1.3, are presented in the next four chapters. In Chapter 4, we present our first contribution, i.e., a latency-sensitive application development and deployment methodology. For the next two chapters, we describe contributions to successfully deploying an application on the target edge system. Chapter 5 presents a distributed resource management framework to deploy latency-sensitive applications specifically on resource-constrained devices. Chapter 6 presents a robust application model to enable the efficient utilization of available resources found at the edge of the network and an extension of the decentralized resource management framework to help the participant nodes to better express their preferences. Chapter 7 describes an adaptive framework capable of efficiently deploy and maintain an application on the target edge system. Finally, in Chapter 8 we conclude the thesis with a reflection of our contributions and an outlook of the future work.

CHAPTER 2

Background

In this chapter, we introduce the core concepts and technologies, which represent the basis for our work presented in this thesis. We start by presenting the target architecture consisting of three computing paradigms, i.e., cloud, fog, and edge computing. Finally, we present an overview of the resource management used for application deployment and management as well as the microservice-based architectures and containerization used to model an IoT application.

2.1 Cloud Computing

In the last couple of years, cloud computing has become one of the most important paradigms for hosting internet applications [AFG$^+$10], mainly because of the cloud capabilities to deliver applications as services as well as the hardware and systems that provide the services. According to the National Institute of Standard and Technology (NIST) [MG$^+$11], cloud computing is defined as follow: *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

Cloud computing is a centralized paradigm with many advantages like high availability and resource scalability, offering virtually unlimited resources to the user [Ley09]. More concretely, cloud computing provides the following advantages: (i) manages the deployed application seamlessly without the involvement of the user, (ii) provides computational resources on-demand, and (iii) requires payment on actual resource usage [AFG$^+$10]. Cloud computing makes use of technologies and concepts like virtualization and elasticity to bring these benefits. The former provides service isolation that allows the cloud server to be used more efficiently by different users and applications with different requirements. In contrast, the latter ensures service scalability and high-availability by providing the required resources for a service, i.e., the utilized resources grow or shrink with the service requirements at a certain time.

Due to its wide adoption, many cloud providers appeared, like Amazon, Google, and Microsoft, offering three different service models, i.e., Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Service as a Service (SaaS). However, as explained in Section 1.1, the cloud's centralized nature has become a disadvantage. All the data generated by the IoT devices results in huge amounts of data sent to the cloud, increasing the application's communication latency. In this thesis, we consider that the cloud is in charge of hosting all high computational microservices part of an application; microservices that cannot be deployed on edge or fog nodes.

2.2 Fog Computing

Fog computing is a paradigm introduced by Cisco [BMNZ14] with the purpose of extending the cloud capabilities closer to the edge of the network. Multiple definitions of fog computing are found in the literature, however the most relevant is presented in [YHQL15]. According to them, fog computing is a geographically distributed computing architecture connected to multiple heterogeneous devices at the edge of the network, but at the same time not exclusively seamlessly backed by cloud services. Hence, fog is an additional computational layer placed between cloud and the edge of the network (see Figure 1.1), consisting of heterogeneous fog devices distributed at different geographical locations [OCY$^+$17]; a fog device is a highly virtualized IoT node that provides computing, storage, and network services.

By migrating computational resources closer to the end devices, fog enables the deployment of new latency-sensitive applications directly on devices like routers, switches, small data centers, and access points. Depending on the application, this can impose stringent requirements like fast response time and predictable latency (e.g., smart connected vehicles, augmented reality), location awareness (e.g., sensor networks to monitor the environment), and large-scale distributed systems (smart traffic light, smart grids).

To fill the technological gap in the current state of the art where cloud computing paradigm is at the center, fog collaborates with the cloud to form a more scalable and stable system across all devices. From this union, the developer benefits the most since an application can be deployed either on fog or cloud. For example, taking advantage of the fog node capabilities, we can process and filter streams of collected data coming from heterogeneous devices, located in different areas, taking real-time decisions and lowering the communication network to the cloud. As a consequence, fog computing introduces effective ways of overcoming many limitations that cloud is facing, e.g., latency constraints, network bandwidth constraints, and better security and privacy [CZ16].

Multiple high-level fog architectures have been proposed in the literature [DB16], [SCM18], [SDW$^+$15] which describe a three-layer architecture containing (i) the smart devices and sensor layer which collect data and forward it to the fog layer for further processing, (ii) the fog layer applies computational resources to analyze the received data and prepares it for the cloud, and (iii) the cloud layer which performs high intensive analysis tasks. In this thesis, we consider that there is a fourth layer in the target architecture, i.e., the edge computing layer.

2.3 Edge Computing

Edge computing represents a new paradigm that aims at facilitating the operation of computing, storage, and networking services closer to the edge of the network [GD18]. As a result, the underlying premise of edge computing is to create a bridge, by adding an additional layer of nodes, between IoT devices (i.e., sensors and actuators) and cloud [SD16]; an edge layer consists of multiple interconnected and distributed edge nodes capable of assisting the cloud when trying to satisfy the stringent requirements of new emerging IoT applications. An edge node is characterized by (i) heterogeneity and (ii) mobility, representing any resource-constrained node found on the communication path between IoT devices and cloud, e.g., a smartphone, a gateway, a micro data server, or a car.

Multiple definitions of edge computing are found in the research literature, however, in our opinion, the most relevant is presented in [SCZ$^+$16]. The authors define edge computing as an enabler for technologies to process data near end-users, i.e., on downstream data for cloud services and upstream data for IoT services. The edge computing paradigm brings many advantages like:

1. *Proximity* – since computational resources are available in the proximity of end-users, both cloud and IoT devices can benefit from allocating parts of the IoT application to the edge nodes. Components can be mapped to the edge layer to pre-process the collected data before being sent to the cloud for storage and data analysis; a strategy that helps lowering data congestion and bandwidth waste inherent from cloud centralization [WZZ$^+$17]. In contrast, it can help resource-constrained devices by offering the possibility of offloading high computational tasks to edge devices found in the proximity of the user.

2. *Context-Awareness* – edge nodes may enter or leave the network at any time without prior notice, changing the environment where the application resides – an environment that is incapable of satisfying the application's requirements. Therefore, to recover from this unstable state, we can use context data to understand the application's environment. Context data consists of knowledge of device location, environmental characteristics (e.g., temperature sensor, video, and images), and network information.

3. *Low latency* – edge computing enables applications to respond in real-time by execution them near the end-user; a characteristic that aids the cloud in meeting the stringent requirements of latency-sensitive IoT applications.

4. *Increase availability* – even in the absence of a stable connection with the cloud, edge computing ensures that deployed IoT applications work properly.

Edge computing capabilities shine when converging with IoT and cloud creating novel techniques for IoT systems. By enabling more computational resources in the proximity of IoT devices, edge allows customers to develop and deploy new IoT applications on edge nodes, taking advantage of lower latency and increased privacy and security, processing data at the edge without the need of transferring it to a remote location like a cloud. As a result, we consider edge computing as an extension of cloud, helping cloud to meet the stringent requirements of IoT applications, e.g., smart connected vehicles or augmented reality which requires low latency and fast response times, sensors networks that requires location awareness, and smart grids which require large-scale distributed systems.

Edge computing fills the technological gap found in cloud-centric IoT systems by collaborating with the cloud to create a more scalable and reliable system where IoT applications may be deployed. From this collaboration, new possibilities to deploy the application appears, letting the developer to choose if a component should be placed in the cloud or on an edge device, depending on that component requirements. An action that can be done manually by the developer or automatic using resource management techniques.

Since many devices can fill the role of an edge device, similar paradigms such as mobile edge computing (MEC) [BWFS14] and fog computing are introduced in the research literature, to move more computational resources closer to the edge of the network. MEC considers that an edge device is a micro data server placed at a telecommunication relay station which aids resource-constrained devices (i.e., a smartphone) to compute high computational tasks. In contrast, fog computing focuses more on the infrastructure side by providing more powerful fog nodes (i.e., a fog node may be a high computation device, access points, or cloudlet). We observe that in both cases, the underlying principle is the same, i.e., to extend the cloud and allow the deployment of IoT applications closer to the end-user. The key difference between edge computing and fog computing is where the computation resides. While fog computing pushes processing into the lowest level of the network, edge computing pushes computation into devices such as smartphones or devices with limited computation capabilities. In this thesis, we consider that our target architecture is a four-layer architecture as presented in Chapter 1. We call this architecture an edge system since both fog and edge computing have the same underlying premise of moving computational resources closer to the edge of the network. Throughout the thesis, we will specify if we target a specific layer or the entire edge system.

2.4 Microservice-based architecture

As seen in the previous sections, the target computing architecture changed from a cloud-centric architecture to a decentralized architecture where computational resources are distributed among different nodes – a change that influenced the application development process. Traditionally, an application was developed as a monolithic entity containing the entire functionality into a single component. However, this approach does not take advantage of the distributed nature of the new computing architecture. Therefore, the application's software architecture must change to take advantage of the available distributed computational resources.

The microservice architecture envisions an application as a collection of weakly coupled, lightweight, small contained services that work together to achieve a specific goal [New15]. According to Lewis and Fowler [LF], the microservice architectural style is *"an approach to developing a monolithic application as a suite of small services, each running its own process and communicating with lightweight mechanisms, often an HTTP resource API"*. An approach that fits rather well with the distributed nature of the target computing architecture obtained from the integration of cloud, fog, and edge computing – allowing an application to use all the available resources shared between nodes. As a result, with the adoption of IoT as well as the other computing paradigms, the microservice-based architecture has been widely adopted as an alternative to the monolithic architectures [LF].

Three key concepts stay at the core of every microservice-based architecture, i.e., containerization, heterogeneity, and decentralization. Containerization places a microservice together with all required dependencies in a lightweight container. Placing each microservice in a container guarantees that it will run uniformly and consistently on any infrastructure. Containerization enable multiple key benefits like (i) scalability, allowing the developer to scale microservices individually depending on the requirements, (ii) ease of deployment, the developer can make a change to a single container independently, (iii) replaceability, i.e., developers can easily replace microservices with better implementations, and (iv) composability, meaning that the developer can reuse a microservice in different applications – giving opportunities to developers to reuse functionality [New15]. Heterogeneity refers to the possibility of using different technologies, platforms, and programming languages. Finally, decentralization allows the use of the distributed available resources in a more efficient manner. However, to deploy and manage these microservice-based applications is not a trivial task. We require new resource management techniques to be able to deploy and manage an application on the new computing architecture.

2.5 Resource management

As previously stated in Chapter 1, to deploy and manage an IoT application in an edge system, we require novel resource management techniques. Since IoT devices are resource-constrained devices, applying resource management techniques at the edge will allow edge nodes to optimize their resource utilization (e.g., energy-aware smart devices that increase their battery levels by offloading computation to nearby nodes), improve data privacy, and enable devices to collaborate and share resources to execute IoT applications. Based on this, resource management is divided into five different categories, each performing a specific task [TN18b].

Resource estimation. To deploy successfully an IoT application in an edge system, one of the fundamental requirements in resource management is the ability to estimate the application's component resource requirements, i.e., how many computational resources a certain component requires. This is important for handling the uncertainties found in an IoT network and providing at the same time a satisfactory Quality of Service for deployed IoT applications.

Resource discovery. Since an edge system is very volatile – nodes are mobile and may join or leave the system at any point in time, it is important to know what are the current computational available resources of all nodes found in the system. For this reason, we require *resource discovery* techniques to discover the currently available resources before starting the IoT application development process. *Resource discovery* together with *resource estimation* provides the core knowledge required to deploy an IoT application in an edge system.

Resource sharing. In an edge system, the total available computational resources are divided between fog and edge nodes. As a result, to deploy an application, participant nodes must collaborate and share their resources to achieve a common goal, i.e., the execution of an IoT application. Therefore, resource sharing aims at enabling collaboration between nodes and maybe offer incentives to make the nodes more willing to share their resources.

Resource placement. With the previous three categories, we manage to (i) learn valuable information regarding the target edge system, (ii) define application's resource requirements, and (iii) ensure that participant fog and edge nodes will share their resources. Based on this knowledge, we can start devising an IoT application deployment strategy using a *resource placement* technique. A *resource placement* technique utilizes the knowledge of node's available resources to deciding *where* to place the application's components such that its objectives are satisfied.

Resource migration. This category provides techniques to decide, at runtime, how to change the initial application deployment to continue to ensure that the deployment still satisfies the application's objectives. The migration process reacts to sudden changes in the edge system and tries to find a migration strategy that moves the application's components on different nodes. To migrate an IoT application, we can either use (i) specific *resource migration* techniques that are capable of adapting based on context information or (ii) the same *resource allocation* techniques used to find an initial deployment.

By combining the aforementioned resource management approaches we manage to optimize the usage of available computational resources found in an edge system according to the IoT application objectives. In this thesis, we focus on combining *resource placement* with *resource migration*, in order to deploy an IoT application on an edge system satisfying its objectives.

CHAPTER 3

Related Work

In this chapter, we discuss existing techniques found in the research literature to develop, deploy, and manage IoT applications in an edge system. We divide the related work into two different sections, i.e., application development and resource management. In the first section, we present existing solutions to assist with the IoT application development process. In the second section, we focus on multiple aspects of resource management, i.e., (i) service placement, (ii) service offloading, and (iii) service migration, discussing techniques found in the research literature that target the placement of applications closer to the edge of the network and the migration of resources when nodes have failed.

3.1 Application development

The adoption of fog and edge computing paradigms as well as the demanding requirements of emergent IoT applications have introduced new challenges in the application's development process. Due to the distributed nature of edge systems, as presented in previous chapters, the IoT application model has shifted from a monolithic architecture to a microservice-based architecture – the consensus among researchers depicts an IoT application model as a collection of microservices [MB18, BKA$^+$20, GVGB17, YHZ$^+$17]. In the research literature, researchers focused on proposing resource management techniques to efficiently use the distributed available resources found in an edge system – considering as given the application model and its associated timing and resource requirements. However, developing an IoT application model and defining all requirements is not a trivial task.

Only recently, researchers have proposed techniques to aid with the IoT application development process. Giang et al. [GBLL15] present a distributed dataflow programming model for fog computing that aids the developer during the application development process. In this paper, the authors propose a two stage application development process, i.e., developing individual application microservices and defining the communication path between them. Wang et al. [WZH+20] propose a stream processing approach, i.e., Edge-Stream, for building new applications for edge computing systems. Edge-Stream represents data flows between the application's microservices as streams to make the approach more user-friendly. Frasad [NTBG15] is another framework that helps with the IoT application development and makes use of a model-driven design approach to enhance the reusability, flexibility, and maintainability of sensor software. Rafique et al. [RZY+20] develop an IoT application development framework using model-driven development and attribute-driven design. The framework transforms the application's requirements into a solution architecture using the attribute-driven design and then uses model-driven development to generate models to transform the application's components into software artifacts. Other papers make use of the Flow-Based Programming (FBP) paradigm to develop new IoT applications. Szydlo et al. [SBS+17] introduce a heuristic data flow transformation technique to successfully distribute flows on the target network. The technique receives as input an application model, i.e., a data flow graph, and decides where to place a process – either cloud or edge devices. Furthermore, the authors develop uFlow, a flow-based processing framework that enables the developer to define data flows for different IoT applications. Belsa et al. [BSPE18] present a solution to enable the interoperability between applications deployed on different IoT platforms. The authors introduce a methodology, based on FBP, to create the communication flow between the available IoT services. The methodology is an extension to Node-RED and allows the application developers to create nodes that enable access to IoT services and create a communication flow between multiple such nodes. Jain et al. [JT17] propose a mapping technique composed of two stages: (i) the IoT application is modeled into multiple different tasks annotated with target location information and (ii) each task is deployed on an edge node based on its location. The authors extend Node-RED to allow the development of IoT applications and deployment of defined microservices to their predefined location, i.e., cloud or edge. Compared to the related IoT development approaches, we focus on aiding the developer in defining IoT applications' timing and resource requirements during the development process. Furthermore, we provide validation of defined requirements considering the target edge system – we enable the means to capture valuable application model information required by resource management techniques to find satisfiable deployment strategies.

3.2 Resource management

Multiple papers found in the research literature tackle the resource placement problem in an edge and fog computing setting. According to the definition of edge computing presented in Chapter 2, in an edge system, an application can be migrated from the cloud closer to the origin of data or offloaded from resource-constrained devices. Besides the two cases, in volatile edge systems, there is the need for adaptive frameworks that make use of service migration techniques. Therefore, we divide this section into three parts, i.e., service placement, service offloading, and service migration. In the first part, we focus on presenting resource management techniques that aim at assisting the cloud in satisfying IoT applications' demanding requirements by placing microservices on resource-constrained devices. In contrast, in the second part, we present resource management techniques to move high computational services from resource-constrained devices to nearby edge nodes – enabling the execution of applications on end-user devices. Finally, in the third and last part, we present resource management techniques to provide an edge system with adaptive capabilities.

3.2.1 Service placement

Brogi et al. [BF17] propose FogTorch, a service placement technique capable of providing deployment strategies for IoT applications on a target fog computing infrastructure. The main goal of the paper is to provide a general and extensible model to capture valuable characteristics of both IoT applications and fog infrastructure. The proposed approach deploys an IoT application such that its Quality of Service (QoS) requirements, i.e., latency and bandwidth, are satisfied. For a similar problem formulation, Salaht et al. [ASDL$^+$19b] propose a constraint programming model, extendable in terms of deployment constraints and objectives, which can obtain a competitive result in relation to heuristics and meta-heuristic algorithms. Scoca et al. [SAB$^+$18] propose a latency, bandwidth, and resource-aware scheduling algorithm aiming at mapping services to edge nodes – the main objective of the technique is to guarantee optimal service quality. The approach uses a score-based technique to compute a scoring mapping for each service by evaluating the target edge nodes as well as the communication links. Redowan et al. [MRB18a] introduce a latency-aware technique aiming to deploy the application's modules on fog computing such that it satisfies all objectives. In the target fog system, the authors consider two types of applications to be deployed, i.e., latency-sensitive and latency-tolerant applications. The proposed decentralized placement algorithm aims to place the applications between fog node clusters and cloud considering the applications' latency requirements. Furthermore, to optimize the fog node's resource utilization, an optimization technique using linear programming is proposed that minimizes the number of computational active fog nodes found in a cluster. Skarlat et al. [SNS$^+$17] propose an optimization service placement algorithm to place services on fog nodes. The authors first organize its computational node into a hierarchical logic, where fog nodes are grouped into colonies. The main idea is to distribute any application submitted to a fog colony to the controlled fog nodes found closer to the end-user. In case that there are no available resources in the current fog colony, the technique always tries to map the application to its neighbor colony. Ultimately, the application is sent to the cloud, if there are insufficient available resources on the fog layer. Wobker et al. [WSMB18] introduce a fog computing platform to deploy and manage fog applications. The service placement technique is based on Kubernetes and is using a labeling system that enables the mapping of application's components on fog nodes based on application's requirements like memory, computational power, and storage and node's available resources.

Petri et al. [PRZR19] propose an edge orchestration technique focusing on providing task deployment based on resource proximity. The orchestrator is deployed on a router and dynamically find a task allocation based on the application requirements forwarding the tasks to either cloud or edge devices. Breitbach et al. [BSEB19] describe a three-level resource allocation technique for edge computing. The first level optimizes the distribution of data replicas on multiple devices to minimize the execution of a task at the cost of increased data management. A context-aware replication technique is used to devise the number of replicas and the data allocation considering multiple characteristics, i.e., data size, current fluctuation of the system, the available storage, and application requirements. The second level distributes the application's tasks based on different scheduling strategies, while the third level provides data adaptive capabilities by monitoring the task deployment. Dzhang et al. [ZMZ$^+$18] describes a competitive-cooperative game-theoretic resource allocation framework to deploy latency-sensitive applications at the edge of the network. In this work, the authors ensure cooperation between nodes by offering incentives based on their work. In this case, edge devices are considered to be rational actors that have no desire of sharing their resources and collaborate with other nodes if proper payoff is not offered. Other works undertake the problem of service placement in different scenarios [LBDP19, HLW$^+$20, EPR20]. As can be observed, the main objective of all technical papers targeting service placement is to ensure latency fulfillment upon placement of services. However, there is one more objective that is equally important and must be considered in a fog and edge setting, i.e., applications availability.

Zhu et al. [ZH18] present EdgePlace, a heuristic technique capable of finding a placement strategy that can minimize the resource unit cost and increases the application's availability. Depending on the application's demands the approach makes a trade-off between resource utilization, bandwidth utilization, and application availability. The technique receives as input a service graph consisting of multiple service chains that must be deployed on MEC servers; a service is a VM. In this case, to achieve service availability during the service placement phase, the authors make use of constraints like affinity and anti-affinity when deploying a service graph. However, in the case of host failure, the application's functionality is recovered by migrating the service chain to another host. Sangolli et al. [SRP$^+$19] propose an edge platform aiming to guarantee high service availability and minimize latency. For this purpose, the authors describe a real-time service migration technique to migrate services among nearby edge nodes when an edge node becomes unresponsive or has a high resource utilization. Daneshfar et al. [DPPA18] proposed a service allocation technique with a central controller capable of mapping users' requests to fog nodes where services reside. In this case, service availability is inherited from nodes' availability. Furthermore, the authors do not consider the possible dependencies between services, each service representing a standalone entity. Lera et al. [LGJ19] present an availability-aware service placement to ensure objectives as application e2e latency and availability. The technique consists of two different stages: (i) break the fog network into smaller and better-connected communities and choose a community where the entire application is placed and (ii) map the application's services between the nodes found in the chosen community. Availability is achieved by deploying an entire application in one fog community since the fog nodes have high connectivity.

Furthermore, the authors consider that in a fog system only the communication link may fail. As a result, the application can continue to operate, by reaching the fog node using a different communication path. However, if a node fails and becomes unreachable, the system cannot ensure the correct application functionality.

Multiple resource auctioning techniques were proposed in the literature to distribute applications between devices. In [BBG18], an auction-based technique is proposed that enables users to bid for the available computational resources of an edge server. Once a bid arrives, the server computes the price for the requested resources taking into account its location, i.e., cloud or edge. Similarly, in [SLYZ18] an auction-based solution is presented to map the requests of bidders (i.e., mobile devices) to the available resources of an edge server. Khan et al. [KVRF16] propose a distributed auctioneer for resource allocations on distributed systems, aiming to combat device trust and their operators' true intentions. A set of distributed protocols are shared between multiple participants with the intent to simulate a centralized auctioneer. By doing a decentralized auctioneer the problem of trust disappears since all operators have a say in how the resources are distributed; eliminating the need of different participant nodes to get an unfair advantage. The solution considers both theoretical and practical implications of a decentralized auctioneer, by using game theoretical perspective as well as limiting the communication overhead.

3.2.2 Service offloading

All service placement techniques discussed in Section 3.2.1 focused on migrating an application from the cloud closer to the edge of the network. However, such techniques may be used for service offloading as well. Recent novel directions in distributed systems have demonstrated advanced service placement techniques aiming to offload parts of an IoT application from resource-constrained devices like smartphones to nearby edge servers (e.g., mobile edge computing (MEC) nodes). By offloading high computational demanding microservices, applications can run on end-user devices.

Avgeris et al. [ADAP19] propose a service offloading technique focusing on the efficient utilization of edge servers. For this purpose, the authors introduce a two-level resource allocation mechanism that allows users found in the area of an edge server to offload computational tasks. However, the mechanism considers a cluster of edge servers without taking into account other edge clusters or the mobility of users. Brandic et al. [MB18] tackles the same problem of service offloading from mobile devices considering parameters like application runtime, battery lifetime, and user cost as its objectives. In this case, the authors propose a heuristic task offloading aiming to migrate computational tasks from mobile devices to a heterogeneous architecture composed of both cloud and edge devices. Another multi-objective task offloading technique is presented in [DMB19] aiming at finding a balance between users' satisfaction and providers' profit. Chen et al. [CCW+19] propose an intelligent resource allocation technique based on deep reinforcement learning to offload tasks from resource-constrained devices to MEC servers. The objective is to find an optimal resource allocation strategy that considers latency, energy consumption, and radio transmission bandwidth. Cicconetti et al. [CCP19] propose a low-latency distributed computation offloading technique that aims at distributing tasks in a pervasive system. By combining serverless and edge computing, a fully distributed domain is created consisting of three different entities: (i) clients who wish to offload tasks, (ii) dispatchers who are in charge of distributing the incoming tasks from clients to a group of computers, and (iii) computers which provides the computational resources. The system is divided into two main categories: an online phase where the dispatcher decided the distribution of all incoming tasks and an offline phase where important functions, like the setup of containers or dispatcher configuration, are performed. Liu et al. [LYWG20] propose a task offloading technique that aims to minimize the system cost, i.e., energy and latency. The technique groups the users into clusters based on their priorities and decided if a cluster should run all its tasks locally or should be offloaded to an edge server.

With regards to resource auctioning techniques, in scientific literature, we can find several proposals that focus on offloading tasks to edge devices. Grosu et al. [BBG18] introduce an auction-based mechanism to perform resource allocation to MECs and compute the related price for each resource. The premise is that mobile users compete for computational resources available at the edge servers to execute IoT applications. Once a user submits a request to the nearest edge server, the pricing mechanism computes the price for that particular resource both in the cloud and edge. Based on the cost, the user decides the deployment location of their application, i.e., edge server or cloud. Sun et al. [SLYZ18] describes an auction-based solution where users bid for resources sold by edge servers. For this purpose, the authors develop a two-sided approach to model the interaction between a MEC server, i.e., the seller, and the bidders, i.e., the end-user devices. To this end, a double auction scheme is used to efficiently map computational resources of a MEC server to the needs of a mobile device by determining the matchmaking between the bidders and sellers. Cao et al. [CZP18] present another resource auctioning mechanism aiming to determine the optimal content placement on mobile edge devices, based on user's bids. Since the auctioneer determines the content allocation based on user input (which may be untruthful), the authors propose a mechanism that can find true valuations from the users and promote participation. Duan et al. [DLC17] propose a reverse auction that considers partial fulfillment of tasks, as well as attribute and price diversity. A distributed auction framework where each task owner is in charge of hosting his/her auction without the need of collecting global information. The authors offer two different auction schemes, i.e., the cost-preferred auction that schedules tasks according to users' asking price and time scheduled-preferred auction that considers their arrival time.

3.2.3 Service Migration

Generally, resource placement techniques do not account for the volatility of edge systems, where nodes have a high failure probability. As a solution to this challenge, researchers propose a new adaptive mechanism, based on migrating services between nodes. Govindaraj et al. [GJAK19] present an approach to perform smart resource allocation allowing them to achieve live migrations on demand. Since performing a migration is an expensive task, the solution tries to minimize the migrations required while maintaining the round trip time for each device under a certain threshold.

Gonçalves et al. [GVC$^+$18] introduce a VM migration and placement technique for fog environments. The technique consists of two parts, a proactive VM migration approach that uses user mobility predictions and a VM placement approach based on Integer Linear Programming. The latter aims at improving the VM placement on selected fog nodes. Kassir et al. [KdVW$^+$20] propose an adaptive distributed service placement technique based on Least Ration Routing capable of migrating VMs to other locations when there are changes in the network, i.e., a change in node location, a change in network size, or network congestion appears. Mseddi et al. [MJEA19] describe an online intelligent resource allocation solution capable of determining the optimal service mapping on fog nodes and find a migration strategy considering the node's available resources. The main objective of this work is to maximize the number of satisfied user requests considering latency as a QoS requirement. Rossi et al. [RCP20] introduce a self-adaptive technique for deploying microservice-based applications in the cloud. The solution is a two-layered hierarchical approach, based on MAPE cycles, capable of self-adapting based on a learned microservices scaling threshold. In this case, the authors choose a decentralized approach, where the first layer provides application-level feedback to the second layer which takes decisions at the microservice level.

From the above resource management related works, we can observe that different approaches are applicable for different problem settings. Heuristic approaches are applicable in situations where the target edge system does not exhibit high uncertainty, allowing for the necessary execution time to find near-optimal solutions. Typically, heuristic approaches are used when high scalability and maximization/minimization of available resources or application requirements are required. In contrast, in situations where specific types of application and systems training data can be obtained, learning approaches appear promising. Such learning approaches are also suitable for finding optimal resource offloading in a distributed manner or adapting to changes found in the application or the target system. Finally, in situations where constraints and objectives can be expressed as constraints and the objective is to find an optimal solution, resource management techniques based on ILP seems like a good fit. In this thesis, for our adaptive and resource management frameworks, we make use of Satisfiability Modulo Theories (SMT) and Constraint Programming (CP) – in our problem setting, we desire qualities like fast reaction and providing guarantees that a found solution satisfies the application's requirements. In conclusion, in this thesis, we distinguish ourselves from the resource management works previously mentioned by proposing a decentralized resource management framework and an adaptive framework. The former differs from other service placement solutions from three perspectives, i.e., (i) we guarantee that if there is a solution possible, it is always found, (ii) we maintain device resource preferences by enabling local decisions related to shared resources, and (iii) we perform resource management in a decentralized manner, on resource-constrained devices without requiring any knowledge about available resources of participant nodes. The latter is capable of (i) ensuring applications' availability on target edge systems, (ii) avoiding container migration and instead changing invocation paths, and (iii) when an application is placed on edge nodes, the required number of replicas is adjusted considering nodes failure probabilities and the application's availability requirement.

CHAPTER 4

EdgeFlow - Developing and Deploying Latency-Sensitive IoT Edge Applications

The rise of the IoT paradigm has made the current cloud-centric systems face challenges in satisfying the demanding requirements of latency-sensitive IoT applications, i.e., low latency, privacy, and availability. To solve this challenge, researchers introduced two new paradigms to extend cloud capabilities closer to the edge of the network, forming an edge system where available computational resources are distributed among interconnected heterogeneous and resource-constrained devices. This shift in architectural structure, from a centralized architecture to a distributed one, has changed the IoT applications model – an IoT application consists of multiple interconnected components[1]. A component is capable of executing one part of the application's functionality. However, developing and deploying such an application model is not a trivial task since the developer must (i) define and validate the application's requirements at design time and (ii) find a deployment strategy such that it satisfies all application requirements.

As described in Chapter 1, edge systems bring many advantages but introduce new challenges as well. One of these challenges is related to the application development process since the developer must divide the application's functionality and define different requirements for each component. This application model is in line with the flow-based programming (FBP) paradigm [Mor10] concepts; an application consists of multiple components that communicate via a communication flow to achieve its overall functionality.

[1]In this chapter, to be consistent with the FBP notation, we will refer to an application microservice as a component.

Several FBP tools like noFlo [2], node-RED [3], and drawFBP [4] exist to aid the developer in creating new IoT application models and define their communication flow. However, it is still challenging to define and validate the application's resource requirements during the development stage.

In this chapter, we propose EdgeFlow, a new IoT framework for latency-sensitive IoT applications deployment and development. Our main contribution is a methodology for aiding the developer to build new IoT applications, at design time, by (i) defining new timing and resource requirements, (ii) validating all requirements, and (iii) finding a deployment strategy. More concretely, the contributions of this chapter are as follows:

- *EdgeFlow.* A methodology for latency-sensitive IoT applications development and deployment. Our proposed methodology aids the developer in defining and validating timing and resource requirements as well as finding optimal and feasible deployment strategies.
- *Development stage.* We propose an extension of the FBP programming paradigm with new concepts like timing and resource requirements. By introducing new timing requirements, we support the definition of multiple e2e delays for different communication flows for the developed application.
- *Deployment stage.* We introduce a novel resource allocation technique capable of finding optimal or feasible deployment strategies. Our main objective is to find a deployment strategy that satisfies all timing and resource requirements defined in the development stage. At the same time, with the deployment stage we provide validation for all application requirements introduced during the development stage. Therefore, the developer can redefine the application's requirements considering the target edge system capabilities.

The remainder of the chapter is structured as follows. Section 4.1 provides an overview of EdgeFlow and defines the application model, the target edge system, and the application's communication flows constraints, given as input files to the deployment stage. In Section 4.2, we describe the implementation details of our proposed framework. Section 4.3 presents the application development methodology, while Section 4.4 shows the results of our deployment stage evaluation. Finally, Section 4.5 concludes this chapter.

4.1 EdgeFlow: application development and deployment framework

EdgeFlow provides a methodology for developing IoT applications and validates the requirements by finding a deployment strategy on the target edge system – the proposed

[2]https://noflojs.org/
[3]https://nodered.org/
[4]https://github.com/jpaulm/drawfbp

methodology takes place at design time. As a result, depending on the type of the target edge system, i.e., static or volatile, the deployment strategy devised during the deployment stage can be more or less efficient. A static edge system represents a system deployed in a controlled environment, e.g., in a smart manufacturing scenario, where edge nodes have limited mobility and the edge system characteristics are know at design time. In this setting, following the feasible and optimal deployment strategies devised by our resource allocation technique, the developer can successfully deploy an application on the edge nodes. However, if we target volatile edge systems, where nodes may fail or leave the network, then the deployment stage may only provide application requirements validation; a volatile system may change during the deployment stage while we search for an optimal deployment strategy, rendering the deployment strategy unreliable. As a result, we require a resource allocation technique capable of finding deployment strategies at runtime. We introduce a decentralized resource management framework in Chapter 5 capable of deploying applications, at runtime, on resource-constrained devices. Figure 4.1 presents an overview of our EdgeFlow methodology consisting of three distinct stages, i.e., (i) the development stage, (ii) the deployment stage, and (iii) the validation stage.

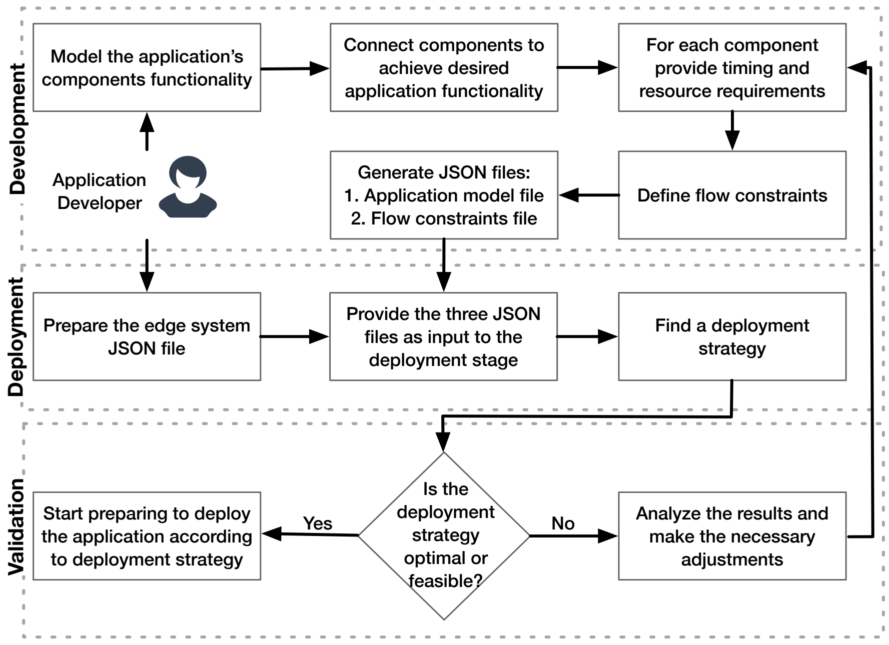

Development
Model the application's components functionality
Connect components to achieve desired application functionality
For each component provide timing and resource requirements
Application Developer
Generate JSON files:
1. Application model file
2. Flow constraints file
Define flow constraints
Deployment
Prepare the edge system JSON file
Provide the three JSON files as input to the deployment stage
Find a deployment strategy
Validation
Start preparing to deploy the application according to deployment strategy
Yes
Is the deployment strategy optimal or feasible?
No
Analyze the results and make the necessary adjustments

Figure 4.1: EdgeFlow methodology overview.

4.1.1 Development stage

During the *development stage*, we ensure that the developer can provide as much information as possible regarding the currently developed application – information that improves the chances to successfully deploy an application on the target edge system. We employ FBP to develop new latency-sensitive IoT application and extend it to capture new timing and resource requirements. Furthermore, we provide a higher granularity when defining the application's timing requirements, i.e., the developer can add a maximum end-to-end (e2e) delay constraint to different communication flows. An IoT application has multiple communication flows, ranging from a communication flow defined by the communication link between two components to a flow containing all application's components (if possible).

The FBP paradigm views an application as a network of processes, i.e., components, interconnected via predefined communication links. Each component runs asynchronously and communicates via streams of data chunks, i.e., Information Packets (IPs) [Mor10]. FBP is component-oriented, allowing the developer to develop different applications using the same network of components by connecting them differently – a practice that improves the application development process and enhances the reusability of components. However, FBP is not a coding language, making this paradigm ideal to build applications by choosing predefined components from a library. Therefore, providing this separation between components and application development, facilitating an application development environment where it is not required for an application developer to have technical background – the developer can build the application using the existent components.

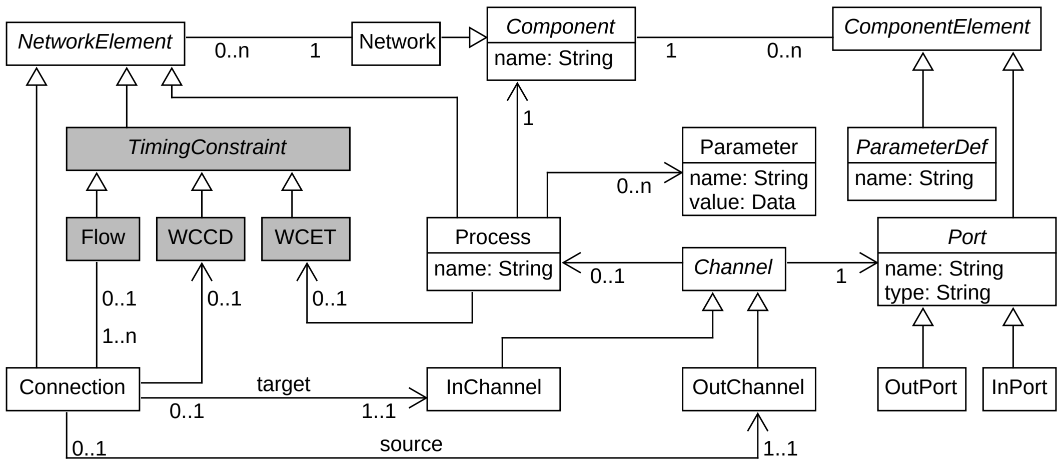


Figure 4.2: The FBP paradigm extension metamodel.

Currently, the FBP paradigm does not provide the possibility to define Quality of Service (QoS) requirements, i.e., timing requirements, data locality, affinity and anti-affinity constraints between components, privacy [TAD19, RB19], and security [XJL+19], during the application's modeling stage. In this chapter, we target the development and deployment of latency-sensitive IoT applications – one of the fundamental concerns of these applications is latency. To this end, we propose an extension of the current FBP paradigm with new timing requirements[5]. Three essential timing requirements define a latency-sensitive IoT application, i.e., worst-case execution time (WCET), worst-case communication delay (WCCD), and the ability to define a maximum e2e delay for different communication flows. In the application model, a communication flow consists of two or more components and their communication links. Therefore, to compute the e2e delay of a communication flow, we must know the component's WCET and the communication's WCCD. The WCET represents the time required by a component to produce a result, while the WCCD is the time an IP needs to travel from its source to its destination. In [ZBS18], authors formalized the syntax and semantics of Flow-Based languages, proposing a metamodel for FBP. Figure 4.2 presents our extended metamodel based on their formalism.

Besides the ability to define the application's timing and resource requirements, the *development stage* allows the developer to generate two input files used by the deployment stage, i.e., the application model file and the flow constraints file. As the name suggests, in the former, we store the application's resource requirements as well as the communication links used by the components into a JSON file. In contrast, in the latter, we save each communication flow defined by the developer during the application development process.

Application model file

An application model defined with FBP consists of a set of components $C=\{c_1, c_2, ...\}$ that collaborates to perform a certain goal. An application may have one or more source components (i.e., the component that provides the required data) and at least one sink (i.e., a component that acts upon the environment according to the received information). In Figure 4.3, we present an example of an application model having one source and two sinks, where we can observe an example of a communication flow starting with the *source component*, i.e., c_0, and finishes with two *sink components*, i.e., c_4. There are other possible communication flows, e.g., starting with c_0 and ending with c_6.

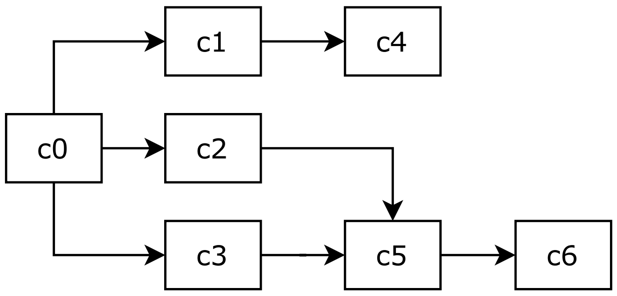


Figure 4.3: Latency-sensitive IoT application model.

[5]We identify all other QoS requirements as an interesting path for future work. Furthermore, researchers can contribute to the IoT application modeling paradigm by providing extensions for new requirements.

A component $\mathsf{c_i}$ performs a certain functionality and represents a containerized microservice. Each component is characterized by a set of timing and resource requirements, $\mathsf{C_{req}}$=$\{\mathsf{r_1}, \mathsf{r_2}, ... \}$ as well as a set of input and output ports, $\mathsf{C_{in}}$=$\{\mathsf{in_1}, \mathsf{in_2}, ... \}$ and $\mathsf{C_{out}}$=$\{\mathsf{out_1}, \mathsf{out_2}, ... \}$ – the developer defines these requirements according to the application's goals. A resource requirement represents the generic memory (i.e., RAM), computational power (i.e., CPU), and storage (i.e., HDD) requirements, while the WCET of a component is an example of a timing requirement. To fit better the application's needs, in Chapter 5, we extend the components' resource requirements with specific requirements, e.g., hardware requirements like GPUs for high computational components or specific data that must be present on the host node.

Flow constraints file

We offer the developer the possibility to define e2e delay constraints for multiple communication flows found in the application model. Recall that to compute the e2e delay of a communication flow, we must consider both the WCET of each participant component as well as the communication latency associated with the links used by components to exchange IPs. Therefore, a communication flow will always consist of at least two components and one communication link. For example, for the application model shown in Figure 4.3, the developer can create multiple flow constraints. There are three big flows consisting of the following components: (i) $\mathsf{c_0} - \mathsf{c_2} - \mathsf{c_5} - \mathsf{c_6}$, (ii) $\mathsf{c_0} - \mathsf{c_3} - \mathsf{c_5} - \mathsf{c_6}$, and (iii) $\mathsf{c_0} - \mathsf{c_1} - \mathsf{c_4}$ respectively. However, the developer can add a constraint even for a smaller flow consisting of a minimum of two components, e.g., $\mathsf{c_0} - \mathsf{c_1}$. In this chapter, we assume that the developer provides constraints for the minimum number of communication flows required to involve all components found in the application model. If any component remains outside of a defined flow constraint, then our deployment stage will consider it as a single component with no dependencies.

Grammar 4.1: Flow Constraint Language

```
<Delay> :: <Number> ms | <Number> ns
<BooleanOp> :: < | > | >= | <= | =
<FlowSource> :: <InPortID> | <DataPacketID>
<FlowSink> :: <OutPortID> | <ComponentFlow>
<ComponentFlow> :: <InPortID> <ComponentID> <OutPortID>
<FlowPath> :: <FlowPath> -> <ComponentFlow> | <ComponentFlow>
<Flow> :: <FlowSource> -> <FlowPath> -> <FlowSink> | <FlowSource> -> <FlowSink>
<FlowConstraint> :: <FlowID> : <Flow> <BooleanOp> <Delay>
<FlowConstraints> :: <FlowConstraints> ; <FlowConstraint> | <FlowConstraint>
<FlowConstraintsDef> :: flow constraints <AppID> <FlowConstraints> end
```

EdgeFlow utilizes a mini-language to specify the flow constraints, see Grammar 4.1. The developer can specify the *flow constraints*, i.e., the communication flow's path and the maximum e2e delay, using the proposed mini-language. In Formula 4.1, we present an example of a flow containing two components $\mathsf{c_1}$ and $\mathsf{c_2}$. In the flow declaration, the IN and OUT represent the name of the input and output ports used by each component. As we can observe, the colon separates the flow's path declaration from its id, while $\leq$ shows the relation between the path and the e2e delay and $\rightarrow$ represents the direction of the communication. Furthermore, the last component does not need an output port, this highlighting that the path is ending.

$$\mathsf{path_1} : \mathsf{IN}\,\mathsf{c_1}\,\mathsf{OUT} \rightarrow \mathsf{IN}\,\mathsf{c_2} \leq \mathsf{e2eDelay} \tag{4.1}$$

4.1.2 Deployment and validation stages

The *deployment stage* provides validation for defined application constraints by determining eligible deployments (if any) of the designed application to the target edge system. We cast our deployment technique within CP paradigm [RVBW06], where we define the deployment constraints as a constraint satisfaction problem. Consequently, the deployment stage can discover feasible or optimal deployment strategies. A deployment strategy that (i) satisfies each component resource requirements while not exceeding the device's available resources and (ii) meets the communication flow constraints, i.e., ensuring that the e2e delay of each communication flow does not exceed the allocated one. Notice that CP fits rather well with our deployment stage since our primary focus is to validate the application's requirements – CP provides guarantees that if a deployment strategy is possible, then it satisfies all requirements. To deploy an application, the deployment stage requires information regarding the application model and the target edge system. The developer provides all required information as three input files, i.e., *application model file*, *flow constraints file*, and *edge system file*; the developer can generate the first two input files using the *development stage*, while the third file can be created manually or obtained from the system administrator.

Edge system file

Let $\mathsf{E_N}$={$\mathsf{E_1}$, $\mathsf{E_2}$, ... } be a set of edge nodes found in the target system. Each node is characterized by a set of available resources, $\mathsf{E_{res}}$={$\mathsf{r_1}$, $\mathsf{r_2}$, ... }, like RAM, CPU, and HDD, and a list of communication links $\mathsf{Link_{com}}$={$\mathsf{link_1}$, $\mathsf{link_2}$, ... } – each $\mathsf{link_i}$ having associated a bandwidth. Depending on the characteristics of the target edge system, we identify two different types of systems, i.e., a volatile edge system and a static edge system. The former consists of edge nodes characterized by mobility and heterogeneity, introducing a high uncertainty level that makes the deployment of an application at design time impractical. Such an edge system can be found in a typical smart city scenario. In contrast, the latter is characterized by a low uncertainty since nodes' behavior is predictable and their characteristics are known at design time – a system seen in a smart city scenario. Since the edge system is not modeled with the FBP paradigm, we assume that the developer obtains the edge system file from the system administrator, i.e., considering the static platform scenario. In the case of a volatile edge system, the developer can create the file using techniques that can estimate the current edge system's available resources from the historic data stored in the cloud.

4.2 Application development and deployment stages

EdgeFlow consists of two core stages, i.e., the *development stage* and the *deployment stage*. For the former, we develop a prototype to prove the proposed application development concept, while for the latter, we propose a resource allocation technique capable of providing deployment strategies such that it satisfies all application requirements and constraints. In this section, we discuss in detail the two stages and their implementation.

4.2.1 Development stage prototype

To prove the benefits of creating a new latency-sensitive application using EdgeFlow, we develop an *application development prototype* based on *drawFBP*. DrawFBP uses FBP at its core and allows developers to create diagrams using blocks, i.e., components [Mor10]. An advantage of drawFBP is represented by component reusability – developers can share their components with other developers to create different applications. A component can be created using multiple programming languages like Java, C#, or even JSON. As a result, the developer can use existing components from the drawFBP library during the application development process. In this scenario, the development process resumes combining the selected components with communication links such that the applications perform the desired functionality. However, defining the application's communication flow and choosing the components is not enough; the developer must define specific requirements for both the communication flows and for each component.

We extend drawFBP with new options, like *set component requirements*, *set flow constraints*, *application model: generate JSON file*, and *flow constraints: generate JSON file*, offering developers the possibility of adding timing and resource requirements. Using the *set component requirements*, the developer can describe for each component the following characteristics, i.e., WCET, period, IP size, and resource requirements (RAM, CPU, HDD). Furthermore, the developer can define different e2e delay constraints for custom communication flows using the *set flow constraints* option. Finally, the developer can generate the two input files using the *application model: generate JSON file* and *flow constraints: generate JSON file* respectively.

4.2.2 Deployment stage technique

Our deployment technique helps the developer to validate the application's requirements considering the target edge system. Using the output of the deployment stage, the developer gets more clarity in defining the component's resource requirements and the application's constraints. Two possible cases can lead to deployment failure, i.e., (i) the deployed application has very stringent requirements or (ii) the target edge system lacks the required available resources to host the application. Under these conditions, if either of the two cases is true, the developer can make the required adjustments accordingly to enable the deployment on that target system. Therefore, the developer can use the *deployment stage* to understand if the application requirements can be satisfied by the edge system's available resources. Using EdgeFlow, the developers can create better application models suitable for deployment on a large variety of systems.

As mentioned in the previous section, we implement the resource allocation technique using CP. Depending on the strategy found, CP can return one of the four different status values, i.e., (i) *optimal*, (ii) *feasible*, (iii) *unknown*, and (iv) *infeasible*. If the CP solver returns one of the first two, i.e., status (i) and (ii), then a deployment strategy that satisfies the application's requirements is found. In contrast, if the returned status is (iv), then the CP solver cannot find a deployment strategy that meets all requirements – a deployment strategy does not exist. Finally, since the CP solver allows the developer to set a predefined time to search for a solution, there is another state, i.e., (iii), returned when the solver cannot find a deployment strategy in the allocated time. However, compared to state (iv), the CP solver is unable to guarantee that with more time a deployment strategy will not be found. Therefore, in this case, the developer should increase the time or try to find the optimal solution. To find a deployment strategy we need to provide a CP model of the problem to the CP solver. Therefore, we model the problem using *decision variables*, *constraints*, and a *global objective*.

Decision variables

Based on the information received as input, we can create a set of *decisions variables* used in the CP model. A decision variable represents a variable for which the CP solver tries to assign a value chosen from a predefined domain such that it satisfies the application's requirements. In our case, we identify four different decision variables, i.e., *component variables*, *latency variables*, *WCET variables*, and *resource variables*.

In the *component variables*, we define for each component a variable domain containing a list of edge nodes that are suitable hosts for the current component. For example, let us consider that component c_1 can be mapped only on three edge nodes, i.e., E_1, E_2, and E_3; hence, a valid domain for the decision variable of c_1 is $\mathsf{D}=\{\mathsf{E}_1, \mathsf{E}_2, \mathsf{E}_3\}$. Under these conditions, the solver can only map c_1 on one of the nodes available in D. To decide what node to choose from D, the CP solver uses the other variables as support. Using the *resource variables* we can filter out the nodes that lack the available resources to host a component by keeping track of the edge node's available resources. Every node starts with a predefined set of available resources; resources that are diminished with the resource requirements of already mapped components. An approach that ensures the correct distribution of components on nodes, without exceeding the node's available resources. However, using only the available resource of nodes does not guarantee that a deployment strategy fulfills the application's requirements defined in the flow constraints file. As a result, we introduce two new decision variables to successfully validate the flow constraints, i.e., the latency variables and the WCET variables.

The *latency variables* are in charge of saving the communication latency between two components considering their mapping. Let us consider that two components c_1 and c_2 communicate with each other – c_1 is mapped on E_1 and c_2 is mapped on E_2. To compute the latency required to transfer an IP between the two components, we can use the IP size and the communication link bandwidth. To build the variable's domain, we use the dependencies between components described in the *flow constraints* input file and all possible locations from their respective domains devised in the *component variables*. As a result, to compute the communication latency between two dependent components, $\mathsf{c_i}$ and $\mathsf{c_j}$, we take all possible distinct edge node combinations from their associated domains.

The *WCET variables* serves a similar purpose as the *latency variables* but has a different effect, i.e., to store the WCET given to each component. Since the WCET of a component is strictly dependent on the host's internal status, obtaining the exact WCET of a component is challenging; the edge system consists of multiple heterogeneous devices, requiring a complete analysis of the WCET of a component on every edge node. We consider such analysis as out of scope for the current approach. Therefore, to lower the challenge in finding a suitable WCET, we assume the developer can provide a lower and an upper bound for the WCET of each component.

We can continue with the introduction of our constraints since we have added all the decision variables to our CP model. With these constraints, we enforce a set of rules to which the CP solver must abide – these rules guide the solver towards a feasible deployment strategy that satisfies the application's objectives. For this purpose, considering the EdgeFlow's objectives, we define two different constraints, i.e., *components constraints* and *flows constraints*.

Constraints

We start by introducing a set of constraints for each component, i.e., *components constraints*, to ensure that the distribution of components on edge nodes does not exceed the node's available resources. To achieve such purpose, the *components constraints* make use of the following decision variables, i.e., *component variables* and *resource variables*. Formula 4.2, Formula 4.3, and Formula 4.4 guarantee that a deployment strategy does not exceed nodes' available resource, where $\mathsf{n_c}$ represents the total number of components mapped on the current node.

$$\mathsf{usedCPU} = \sum_{\mathsf{i=1}}^{\mathsf{n_c}} \mathsf{t_{CPU}} \leq \mathsf{availalbe_{CPU}} \tag{4.2}$$

$$\mathsf{usedRAM} = \sum_{\mathsf{i=1}}^{\mathsf{n_c}} \mathsf{t_{RAM}} \leq \mathsf{available_{RAM}} \tag{4.3}$$

$$\mathsf{usedHDD} = \sum_{\mathsf{i=1}}^{\mathsf{n_c}} \mathsf{t_{HDD}} \leq \mathsf{available_{HDD}} \tag{4.4}$$

By validating the *components constraints*, we can successfully deploy the application on the target edge system. However, for the moment, we deploy an application considering only its resource requirements – we do not make use of the other decision variables to help us compute the e2e delay of a communication flow. Therefore, we introduce a new set of constraints, i.e., *flows constraints*, to reinforce the constraints introduced in the *flow constraints file.* We enforce the flow constraints on the deployment strategy by combining three decision variables, i.e., the *components variables*, the *WCET variables*, and the *latency variables.* In conclusion, by adding these constraints to the CP model, we consider both the flow's constraints and the component's resource requirements. Equation 4.5 guarantees that the flow's e2e delay does not exceed the maximum e2e delay associated with it; remember that the e2e delay of a flow is the sum of all participant components' WCET and their communication latency. These constraints are captured in Formula 4.5, where $\mathsf{l_f}$ represents the total number of communication links found in a flow *f*, $\mathsf{c_f}$ is the total number of components part of a flow *f*, and $\mathsf{maxE2Edelay_f}$ represents the maximum e2e delay allowed for flow *f*.

$$\mathsf{e2eDelay} = \sum_{\mathsf{link}}^{\mathsf{l_f}} \mathsf{link_{latency}} + \sum_{\mathsf{c}}^{\mathsf{c_f}} \mathsf{c_{wcet}} \leq \mathsf{maxE2Edelay_f} \tag{4.5}$$

Global objective

The purpose of this objective is to minimize the e2e delay of each flow. In doing so, we obtain a solution that offers an optimal deployment strategy if no time limit is imposed. We capture the global objective in Formula 4.6, where $\mathsf{n_f}$ represents the total number of flow constraints defined in the *flow constraints file* and $\mathsf{flowE2E_i}$ is the current e2e delay of flow *i*.

$$Min(\sum_{\mathsf{i=1}}^{\mathsf{n_f}} \mathsf{flowE2E_i}) \tag{4.6}$$

4.3 Application Development methodology

In this section, we evaluate EdgeFlow applicability in a real scenario by presenting the *application development experience.* We present the entire process required to build and deploy a latency-sensitive application (see Figure 4.1), i.e., we present the (i) application development stage, (ii) deployment stage, and (iii) validation stage. We start by describing the development process where we build the application's model and define its timing and resource requirements. Furthermore, we conclude this stage by generating the two input files required by the *deployment stage*, i.e., the application model file and the flow constraints. With the two files ready, we can proceed with the deployment stage. However, remember that this stage requires information regarding the target edge system – therefore, the developer must prepare and provide this file. Once the edge system file is ready, we can start finding a deployment strategy for our application. Based on the results, we can validate the defined application's requirements. The application development prototype and the deployment stage technique are available in our online appendix [6] and our git repository[7].

As a running exemplar, we model a public safety IoT application deployed in a smart city scenario. The application aims at preventing any possible disaster scenario by analyzing all the images and videos from an area. The application consists of components capable of analyzing both the environment as well as people. For example, the application sends an emergency signal to the police department if a suspicious package is found in the monitored area. Considering the safety implications, the application must adhere to some strict timing requirements such as low e2e delay to be able to provide alerts without delay. Thus, the application must execute at the edge of the network. As a consequence, a prerequisite for the developer is to validate the timing and performance requirements on the target edge system before deploying the application. As we will show, EdgeFlow is capable of performing such validation. Since our focus is to show the extensions and improvements we bring with our proposed IoT framework, we assume that the public safety application's components are available in the drawFBP library. In this setting, the developer must connect the components and add the timing and resource requirements.

Development stage

Using the *application development prototype*, the developer can create all components required for the application, add their functionality by choosing it from the drawFBP library, and connect them via ports to create the desired application's goal. In our case, the public safety application consists of four components, each enacting a specific functionality (see Figure 4.4) – for example, *MotionDetection.class* represents a java class where the functionality of c_1 resides. By following the described steps, the developer creates the application model without defining the timing and resource requirements.

The developer can specify the application's timing and resource requirements using our FBP extension options presented in Section 4.2.1 – the developer can add these requirements at any time, either after the application's model is complete or when components are added. To assign the component's requirements, the developer can use the option *set component requirements* available in the component menu; right-click on the target component to access this menu. The process of setting the component's requirements goes through each requirement and asks the developer to provide a value or a range (in the case of WCET). To create new flow constraints, the developer can select the *set flow constraints* option from the *file menu* and define a new flow constraint using the mini-language presented in Section 4.1.1. We have added multiple support options as well, e.g., with *display flow constraints* the developer can see what flow constraints are currently defined and with *delete flow constraints* the developer can delete any flow constraint that is no longer desired. Similarly, we provide a display option for the components as well to see the current requirements. Finally, the developer can store the

[6] https://dsg.tuwien.ac.at/team/cavasalcai/projects/EdgeFlow
[7] https://github.com/cavasalcai/EdgeFlow

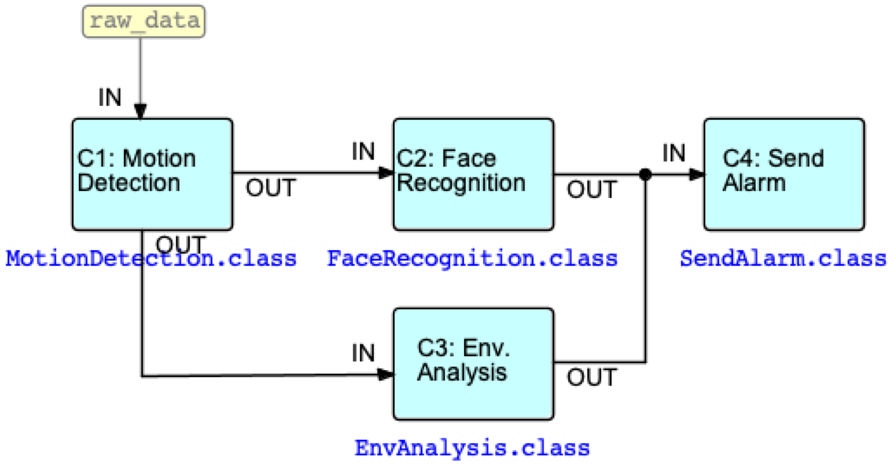


Figure 4.4: Public Safety Application DrawFBP model.

information into the two required input files by using the following options available in the *file menu*, i.e., *Application Model: generate JSON file* and *Flow Constraints: generate JSON file*.

Deployment stage

In Table 4.1, Table 4.2, and Figure 4.5 we present the contents of the three input files, i.e., the two generated from the development stage and the edge system file given by the developer. Table 4.1 shows the target edge system, where we can see the node's available resources, connections with other nodes, and the bandwidth of each communication link. For example, E_0 can reach E_1 and E_2 – the communication link between E_0 and E_1 has a bandwidth of 10 *units/sec*, while the communication link between E_0 and E_2 has associated a bandwidth of 15 *units/sec*. We choose for each available resource, i.e., RAM, CPU, HDD, a value between 15 and 30 *units* – the deployment stage can operate with different units, e.g., MB or GB, as long as there is consistency between the available and required resources.

Nodes	Available Resources			Connections	
	RAM	CPU	HDD	destination	bandwidth
E_0	20	22	15	E_1	10
				E_2	15
E_1	17	30	18	E_0	10
				E_2	15
E_2	15	25	16	E_0	15
				E_1	15

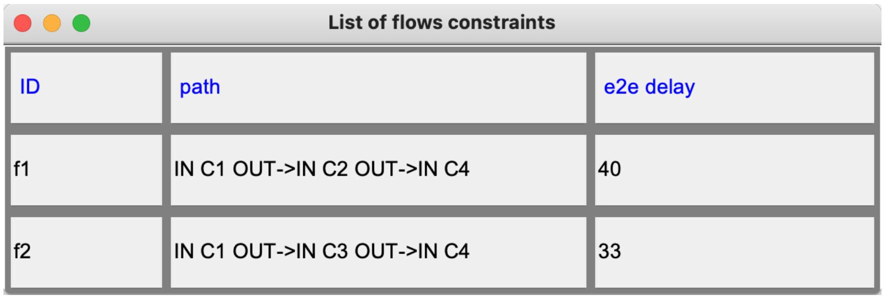
Table 4.1: Edge Computing platform characteristics.

In Table 4.2, we present the application's timing and resource requirements. For our public safety application, we choose for each component the following: all resource requirements have a value between 1 to 15 units, we randomly select a data size value between 30 and 115 units, and add a custom range for the WCET considering each component functionality.

Components	Resource Requirements			WCET	Data size
	RAM	CPU	HDD		
c_1	6	7	5	[5, 9]	60
c_2	13	15	7	[20, 35]	90
c_3	10	13	10	[15, 25]	75
c_4	3	2	3	[2, 4]	30

Table 4.2: Public Safety Application resource and timing requirements.

Finally, in Figure 4.5, we present the two flow constraints, i.e., f_1 and f_2, added for the public safety application. As we can observe, to define a flow constraint we must provide an id, the communication flow path, and the maximum e2e delay. For f_1, we choose the following communication path $\mathsf{c}_1 - \mathsf{c}_2 - \mathsf{c}_4$ and an e2e delay equal to *40 ms*, while f_2 is having the following path $\mathsf{c}_1 - \mathsf{c}_3 - \mathsf{c}_4$ and a maximum e2e delay equal to *33 ms*. To choose the maximum e2e delay, we consider the sum of the lower bound of the WCET of all components found on the communication flow, to which we add 10 more ms; this reflects the impact of the communication latency between components.


List of flows constraints

ID	path	e2e delay
f1	IN C1 OUT->IN C2 OUT->IN C4	40
f2	IN C1 OUT->IN C3 OUT->IN C4	33

Figure 4.5: Communication Flow constraints for public safety application.

With the three files ready, the developer can start the process of finding a satisfiable deployment strategy. Considering the target edge system, the deployment technique tries to find an optimal or feasible deployment strategy. Depending on what status the CP solver returns, the developer must decide if the application's requirements must be changed to accommodate the target edge system capabilities or try to find a more suitable edge system for the deployed application. In Figure 4.6, we present the deployment strategy returned by the deployment stage. We highlight the host node of each component by placing the node's id on the top left corner. For example, component c_1 is mapped on E_0 and component c_2 is mapped on E_1. In this case, the deployment stage finds the optimal deployment strategy in *10 ms*.

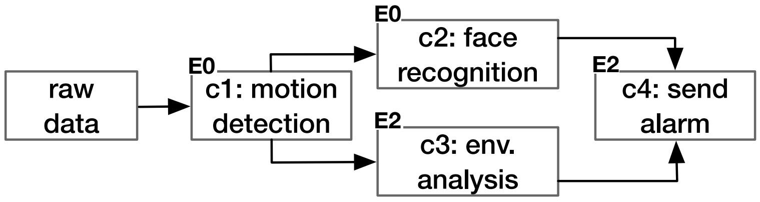


Figure 4.6: Deployment strategy for public safety application.

The *deployment stage* returns a detail report showing the communication latency between components and their WCET concerning each communication flow constraint. In Table 4.3, we show the report in which we present the actual e2e delay of each flow, the communication latency for dependent components, and the components' WCET. For example, for f_1 the communication latency between components c_2 and c_4 is equal to *6* ms. To conclude, we can observe that for the optimal solution, our deployment technique manages to minimize the flows' e2e delay – the actual e2e delay of flow f_1 is *33* ms, while for f_2 is *26* ms.

Flows	Components		Communication Latency		e2e delay
	ID	WCET	destination	latency	
f_1	c_1	5	c_2	0	33
	c_2	20	c_4	6	
	c_4	2	-	-	
f_2	c_1	5	c_3	4	26
	c_3	15	c_4	0	
	c_4	2	-	-	

Table 4.3: Flows actual e2e delay and communication latency.

Validation stage

In the validation stage, the developer makes a decision regarding the defined application's requirements based on the information presented in the deployment report. In our case, the deployment stage has found an optimal deployment strategy that fulfills all application requirements – therefore, there is no need to redefine the timing requirements. However, if the *deployment stage* cannot find a solution, then the developer can change the requirements and employ the deployment stage again.

4.4 Evaluation

To quantitatively evaluate our deployment stage, we consider as a performance metric the execution time required to obtain an optimal mapping of components to the target edge system. We are interested in finding how certain markers like (i) the application size, (ii) the total number of edge nodes, and (iii) the number of flow constraints impact the deployment stage performance. Considering our evaluation objective, we propose three distinct scenarios, each having a different application model and flow constraints input files. Furthermore, in every scenario, we increase the number of available edge nodes present in the target edge system; each system has a different number of available edge nodes. We can obtain these files using the *application development stage*, as proven in Section 4.3. However, considering our evaluation objective, it is not feasible nor required to develop the applications and add each timing and resource requirements manually – for each scenario, we randomly generate all input files using different procedures.

Application model file: generation

All considered applications follow the same model, i.e., each has one source component and one sink component. A decision that does not alter the evaluation objective since an application with multiple sources and sinks only implies a higher initial number of flows. We choose a different number of components for each scenario – the application in the first scenario has 10 components, while the other two scenarios have 20 and 30 components respectively. We model the component's resource requirements as a tuple, i.e., (RAM, CPU, HDD), choosing for each resource a random value between *[5, 15]* units. Similarly, for the component's WCET range, i.e., *[l, u]*, we choose a value for *l* and *u* from *[4, 10]* ms and *[10, 12]* ms respectively. We set the other component requirements as follows: the period has a value between *[10, 30]* ms, the IP's data size is between *[30, 120]* bytes, and we define a total of two input and two output ports.

Edge system file: generation

For our evaluation, we create multiple target edge systems, each having a size between 10 and 500 nodes. For each application size, we gradually increase the size by 10, generate the edge system file, and employ the deployment stage to find an application deployment strategy – this approach yields a total of 50 deployments for each scenario. Similar to the components' resource requirements, we model the available resources of an edge node as a tuple and choose for each resources a value between *[15, 30]* units. Furthermore, we choose for each communication link an available bandwidth between *[30, 90]* bytes/ms.

Flow constraints file: generation

Our intent is to define randomly different flow constraints for each scenario – flows that have different communication paths and sizes. For every scenario, we have defined three flow constraints files, i.e., a file containing (i) one flow, (ii) three flows, and (iii) five flows. Remember that the developer must define flow constraints such that it involves all communication links and components at least once. As a result, in our procedure, the first flow will always traverse the application from the source component to the sink component; involving all other components in between.

We have developed a procedure to help us build the flow constraints file. The procedure takes as input the total number of flow constraints and the maximum e2e delay. We set the maximum e2e delay to a high value, i.e., *500* ms, for all flows. Choosing a smaller e2e delay does not impact the deployment stage's performance; however, it may influence its ability to find a deployment strategy if we set the e2e delay to a very stringent value. Moreover, in Section 4.3, we have demonstrated the capability to generate deployment strategies under demanding e2e delay requirements.

To create a communication path between the participating components, the procedure creates a pair of two components, i.e., (*src*, *dest*), starting from the source component and selects the next destination components. Next, we create a new pair using as *src* the *dest* component from the previous pair and choosing as the new *dest* a new component. The procedure continues until the destination becomes the sink component. For example, let us consider that we want to build flow *f1* from Section 4.3. In this case, we have four components involved in *f1*, i.e., C=\{$\mathsf{c_0}$, $\mathsf{c_1}$, $\mathsf{c_2}$, $\mathsf{c_4}$ \}. To build the flow constraint, the procedure starts from $\mathsf{c_0}$ and chooses the destination $\mathsf{c_1}$ forming the first pair ($\mathsf{c_0}$, $\mathsf{c_1}$). Next pair is formed by making $\mathsf{c_1}$ as the source and choosing $\mathsf{c_2}$ as the new destination, resulting in the new pair ($\mathsf{c_1}$, $\mathsf{c_2}$). Finally, the procedure stops with the pair ($\mathsf{c_2}$, $\mathsf{c_4}$), since $\mathsf{c_4}$ is the sink component. To build the other flows, we randomly select the number of participating components, i.e., C, and restart the procedure.

To evaluate the performance of our deployment stage, we find an optimal deployment strategy for each scenario, gradually increasing the system size and defining only one flow constraint. In Figure 4.7, we present the execution time required by the deployment stage to find an optimal deployment strategy for all three scenarios. The *x-axis* represents the total number of nodes found in the target system, while *y-axis* represents the execution time in seconds.

In Figure 4.7, we show the total execution time required by the deployment stage to yield an optimal deployment strategy. However, the deployment technique consists of two different parts, i.e., (i) building the CP model and (ii) solving the model using a CP solver. As a result, we are interested in finding how much time the deployment stage requires for each part (see Figure 4.8). In Figure 4.8b, we present the execution time required to generate the CP model, while Figure 4.8a presents the time required by the CP solver to find an optimal deployment strategy.

In all experiments presented above, we have kept the number of flow constraints equal to 1. However, we are interested in observing the impact of multiple flow constraints on the execution time of both the CP solver and the CP model. Therefore, we remake the evaluation increasing the number of flow constraints as well – we perform 50 deployments using a total of 3 and 5 flow constraints respectively. In Figure 4.9, we show the execution time required to solve a model for each scenario, while in Figure 4.10 we show the time required to build the CP model.

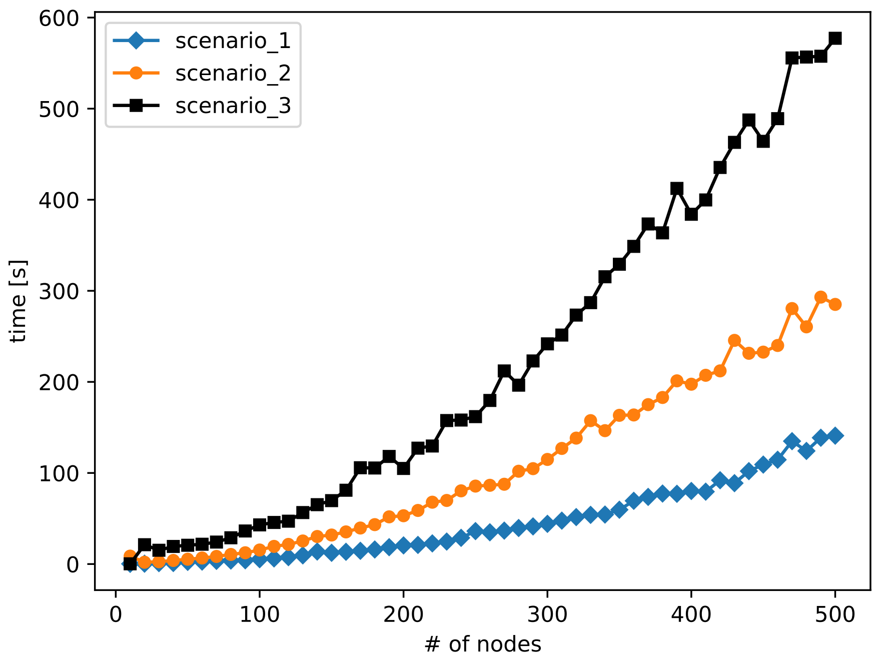
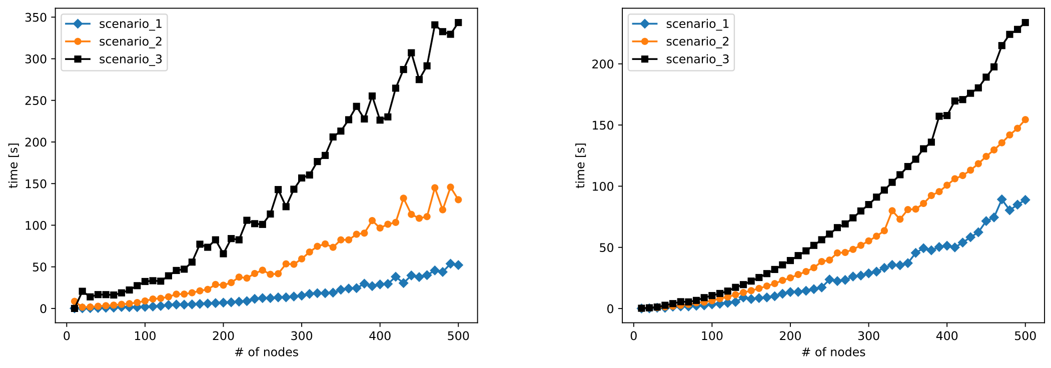


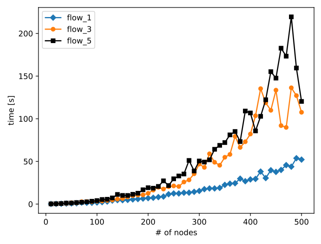
Figure 4.7: Execution time of the deployment stage for different scenarios over different edge systems sizes, considering only one flow constraint.



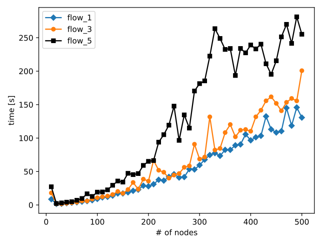
(a) CP solver.

(b) CP model.

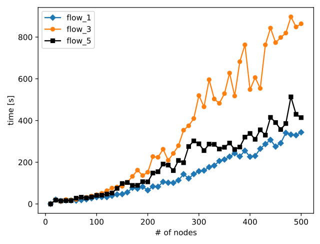
Figure 4.8: Execution time of (a) finding a deployment strategy and (b) building the CP model over different edge system sizes, considering one flow constraint.



(a) Scenario 1.

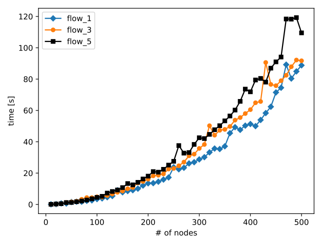


(b) Scenario 2.

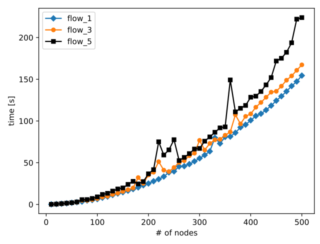


(c) Scenario 3.

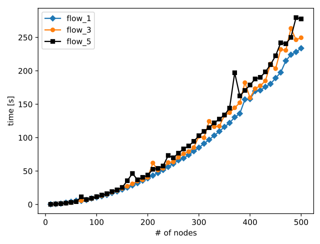
Figure 4.9: Impact of the number of flow constraints on solver execution time considering all three scenarios.



(a) Scenario 1.



(b) Scenario 2.



(c) Scenario 3.

Figure 4.10: Impact of the number of flow constraints on model execution time considering all three scenarios.

4.4.1 Discussion

We have demonstrated that with the proposed deployment technique, we can successfully find an optimal deployment strategy. Contrary to how we chose the flows' maximum e2e delay in Section 4.3, we have decided to choose a less stringent maximum e2e delay since this does not impact our evaluation results; we use the same e2e delay for all scenarios. Results of Figure 4.7 show that (i) the number of nodes found in the target system and (ii) the application's size impacts the execution time required to find an optimal deployment strategy. Breaking down the execution time, see Figure 4.8b and Figure 4.8a, we can conclude that the time required to build the CP model represents around *42 %* from the total execution time while solving the model requires the remaining *58 %*. In both cases, we can see an increase with the number of nodes and components.

In Figure 4.9, we have demonstrated that the number of added flow constraints impacts the execution time required to solve a CP model. In contrast, we can see from Figure 4.10 that an increase in flows will marginally increase the time required to generate the CP model. It is normal to have an increase in execution time for the CP solver; with the addition of new constraints, the problem becomes harder to solve, hence requiring more time. However, this is not the case when building the CP model since we only create more variables for the added constraints.

In our evaluation tests, we can observe that the execution time for building the CP model gradually increases with an increase in the (i) number of nodes, (ii) number of components, and (iii) number of flow constraints. An expected behavior, since the number of model variables and constraints increases in the CP model. As a result, we can conclude that the constraints themselves do not impact the execution time for building the CP model. In contrast, the execution time required by the CP solver has the same trend, but it fluctuates between different deployments when the application size grows. A trend that is the result of all the optimizations the CP solver has. The CP solver's execution time depends more on how complex the problem is; the complexity depends on the node's available resources, the application size, the component's resource requirements, and the defined flow constraints. For example, in Figure 4.9c, we can observe that the solver manages to find a deployment strategy faster when there are 5 flows than when we have 3 flow constraints. However, since we built the scenarios randomly and the CP solver has its own optimizations, we cannot say with certainty why this behavior appears or the fluctuations in execution time.

Finally, one advantage of using CP for our deployment technique is the ability to allow the developer to set a maximum execution time for the CP solver. Consequently, if the developer is not interested in finding the most optimal application deployment, then the developer can limit the CP solver's execution time. For example, we can find a feasible deployment strategy for scenario 3, having one flow constraint and an edge system size of 500 nodes (see Figure 4.8a), in *120* ms. An approach that can further lower the total execution time required to find a deployment strategy, in return lowering the time required by the developer to validate all defined requirements. It is important to mention that if we do not give enough time to the CP solver to arrive at a conclusion, then the solver returns the 'UNKNOWN' status – the solver does not have enough knowledge to determine if the solution is infeasible or feasible. As a consequence, the developer should pick a reasonable time for complex problems.

We acknowledge the high computational demands of our deployment stage when finding optimal deployment strategies for scenarios where the problem becomes too complex. We can see in Figure 4.7 that the deployment stage requires around 600 seconds to find the optimal deployment strategy for scenario 3. However, we argue that the execution time is not an issue since the deployment stage takes place at design time when the application is not operational – thus, it does not impact the application performance. Furthermore, EdgeFlow's primary objective is to assist the developer to develop new IoT applications at design time. Later in this thesis, we will present resource management techniques capable of deploying and managing applications at runtime.

4.4.2 Challenges and Limitations

We identify two types of latency-sensitive applications that would benefit from edge computing, i.e., the hard real-time IoT applications and soft real-time IoT applications.

Both applications are similar since their correct functionality relies on having a low e2e delay and meeting their deadlines. However, there is an important distinction between the two, i.e., violating the deadline of a soft real-time IoT application may be acceptable since it only impacts the application's performance; in contrast, missing the deadline of hard real-time applications can have catastrophic events. Hence, in this chapter, we focus on the development of soft real-time IoT applications offering the possibility to validate only the e2e delay set for each flow, i.e., it does not violate its maximum allowed e2e delay deadline for a certain flow. We do not provide time analysis strategies for validating the component's WCET on the host node.

There are two main challenges for the developer during the application modeling stage, i.e., assigning the WCET and the resource requirements for each component. The former plays an important role in the overall e2e delay while the latter is critical for the deployment technique – without knowing the component's resource requirements, the deployment technique cannot devise a deployment strategy. Finding the WCET is not a trivial task. The WCET of a component is directly dependent on the host node, i.e., the developer must know the internal status of the node (i.e., the current load and the available resources) and the location of the component. An approach to determine the component's WCET is to compute it at deployment time. We can integrate the WCET analysis into the *deployment stage* similar to how we do for the latency communication computation. An approach that automates the process of finding of WCET and simplifies the application developer tasks. In this chapter, we assume that the developer provides the WCET using an external tool; the implementation of an automatic approach is our target for future work. Similar to the WCET computation, finding the component's resource requirements is a challenging task. One option to find and estimate these resources (i.e., RAM, CPU, HDD) is to benchmark the application on multiple edge systems and take the maximum usage as an estimate.

Finally, besides finding a mapping of components to nodes, we must map the input and output virtual ports as well. There are two approaches that we can follow to achieve port mapping, i.e., manual and automatic. The former requires that the engineer performs manually the mapping of virtual ports to the host node real ports following the deployment strategy suggestion; a scenario that is possible only if the target edge system is known and has a relatively small size. In comparison, in the latter approach, the resource allocation technique is in charge of mapping the ports and the components without requiring the help of an engineer.

4.5 Conclusion

To enable any resource management framework to find deployment strategies that satisfy the application's requirements, it is important to have as much information as possible regarding the received application like (i) the application's communication flow, (ii) the application's components descriptions, including the resource requirements, and (iii) the application's objectives. As a result, the application developer requires an approach to build an application and to provide all this information. EdgeFlow fulfills the same purpose and assists the developer throughout the entire application development and deployment process. For this framework, we propose a methodology for latency-sensitive IoT applications consisting of three important stages, i.e., the *development stage*, *validation stage*, and *deployment stage*. For the *development stage*, we propose an extension of the FBP paradigm with timing and resource requirements; requirements that are crucial to the successful deployment of the application on the target edge system. Further, we consider multiple communication flow constraints, ensuring that the e2e delay of a certain communication flow does not exceed a certain e2e delay. For the *deployment and validation stages*, we introduce a new resource allocation technique capable of finding feasible or optimal deployment strategies. It is important to mention that the deployment stage can find feasible and optimal deployment strategies at design time. Therefore, it cannot account for any changes that take place while the technique searches for a deployment strategy. Furthermore, it cannot ensure the correct application's functionality during its life cycle, if the target edge system has changed after the application deployment stage – we will target these two issues in the following chapters.

CHAPTER 5

Resource Management for Edge Computing Services

Satisfying the software requirements of emerging microservice-based IoT applications has become challenging for cloud-centric architectures, as applications demand fast response times and availability of computational resources closer to end-users. Those shortcomings can be tackled by taking advantage of distributed computational resources in the spirit of edge computing, where data processing occurs locally by functionality deployed on edge nodes, with advantages including data locality and fast response times [Sat17]. Edge systems consist of multiple heterogeneous computing nodes often running containerized microservices. Utilizing distributed computing resources within an edge system is challenging, as applications often have stringent performance and deployment requirements. As such, meeting application demands must occur at runtime, facing uncertainty and in a decentralized manner, something that must be reflected in system deployment.

Resource management in this context [TN18a] aims at enabling collaboration between edge nodes by sharing their available computational resources. In such a setting, IoT applications are deployed on possibly resource-constrained devices and in dynamic networks where high uncertainty is introduced by (i) node mobility, (ii) node heterogeneity (i.e., an edge node can be a resource-constrained device as well as a powerful server), and (iii) lack of knowledge at design time of network topology and edge nodes' available resources. We propose a novel resource management technique focusing particularly on resource sharing and allocation for application deployment. Previously, deployment of applications at the edge of the network has been generally tackled from two perspectives: (i) task offloading from resource-constrained devices to improve objectives such as energy consumption [MB18] or (ii) relying on the cloud to perform resource allocation [RGXZ17]. Still, such approaches do not sufficiently take into account latency application requirements, do not consider node's preferences, and assume knowledge of participant nodes' internals.

In this chapter, we propose a decentralized resource allocation technical framework aiming to deploy applications at the edge of the network, guaranteeing adherence to (i) defined latency Service Level Agreements (SLAs) and (ii) resource preferences of participating nodes. We focus solely on microservice placement performed by a resource-constrained device, by providing optimized procedures, and by fully utilizing the available resource found at the edge of the network. Specifically, our contributions are:

- A decentralized resource allocation technique for sharing IoT resources with nearby nodes based on application requirements;
- A scheme where participating nodes on the network may utilize multiple decision strategies, making their own choices regarding their contribution of their local resources, including data;
- We advocate decentralization, as the system can operate without an assumed connection to the cloud, if there are enough available resources at the edge of the network.

Our framework encodes the resource allocation problem within Satisfiability Modulo Theories (SMT [BT18]), where the placement of microservices on edge nodes generates constraints in first-order logic while latency SLAs are encoded with integer linear arithmetic. Thus, we provide guarantees – if a mapping exists, it is always found at runtime by a solver situated in some edge node deploying the application, and is always correct, i.e., it satisfies latency SLAs, preferences of participating nodes, and other constraints. We evaluate the applicability and performance of our technique, especially compared to the absence of cloud resources. Our obtained results demonstrate its efficiency for relevant problems, particularly on resource-constrained edge devices. Our experiments show that our framework is capable of deploying IoT applications entirely on resource-constrained devices, the SMT solver providing the mapping being deployed on a resource-constrained device as well.

The remainder of the chapter is structured as follows. In Section 5.1 we present an overview of our solution and introduce a motivational example. Section 5.2 defines the IoT application and architecture considered in this chapter. In Section 5.3, we describe implementation details of our proposed technique, while Section 5.4 presents the methodology and results of our evaluation regarding applicability and performance. Finally, Section 5.5 concludes this chapter.

5.1 Decentralized Resource Allocation

In our framework, an IoT application to be deployed consists of interdependent containerized microservices which need to be executed in some specific way to provide the application's functionality [WIH16, MB18]. Compared to the application's considered in Chapter 4, the considered IoT applications have a single point of entry (the initial microservice) and some sink (or final) microservice, signifying the result of the computation. The overall application has certain (strict) latency requirements, while other concerns may impose further constraints over where a microservice may be deployed as well. In this chapter, latency is understood as an adherence to certain defined Service Level Agreements (SLAs) and represents the time required for a message to traverse the application's communication flow (i.e., from the input data microservice to a sink microservice). Within the edge setting, the application's microservices may be deployed on different networked physical nodes. Nodes participating in the system individually select microservices they may host – perhaps based on some incentive scheme [WD19, STD15].

Our framework provides seamless deployment of applications with latency requirements; as illustrated in Figure 5.1, the application designer defines the building blocks of an application (as microservices) as well as their dependencies in an application model, at design time – essentially the microservice composition. When the system is operational, microservices are deployed to appropriate edge nodes so that application requirements are satisfied, without requiring any knowledge of the runtime network topology or node's internal status.

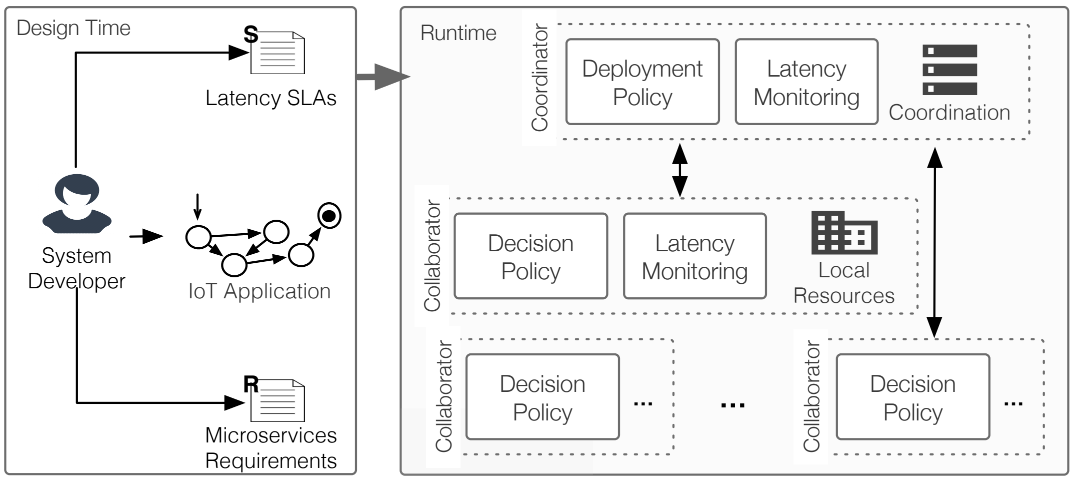


Figure 5.1: Decentralized Resource Allocation: Overview.

The functionality of the framework we advocate revolves around two key concepts; decentralization and node participation. To encourage edge nodes to participate and share resources with applications on the network, we assume that an incentive mechanism exists that offers rewards based on the involvement of a participant node. As such, the framework establishes collaboration between nodes to achieve the deployment goal. Two key components can be found in our solution: the *coordinator* node, which seeks to deploy some application and the *collaborator* nodes, which are the system participants – any edge node can in principle take any of those roles. Once a request for deployment of an application arrives, a list of participants is created. We consider some limited number of participants, chosen based on proximity to the coordinator node. The coordinator serves as the local decision making authority, by advertising the IoT application to the nearby participant nodes and eventually deciding a microservice distribution that satisfies latency, preferences of participants and other application requirements. Collaborator nodes offer resources for parts of the application, at their own preference and based on their current state of available computational resources.

Running example

Consider a public safety application that aids police officers to identify wanted persons in a crowd by performing video analysis utilizing available resources found in their proximity; this is performed by processing video captured from the police officer's body camera, or video and images stored in nearby edge nodes (i.e., other smartphones, tablets, dash board cameras, etc.). Such video analysis is computation-intensive and may require specialized hardware running machine learning workloads. The officer's smartphone (or dash cam) represents the application coordinator which is connected to his/her body camera. The application consists of multiple distinct microservices including (i) motion detection, (ii) object detection, (iii) object tracking, and (iv) result generation; arrows indicate the invocation of microservices within the application workflow.

Deploying such an application to a centralized location is not desirable due to its stringent requirements as well as inherent privacy concerns – video data should not be stored in a remote centralized location. First, we can observe that the decentralized nature of our technique fits rather well with the application requirements since it handles video without sending it to a central facility for processing. Furthermore, the application is particularly data-intensive, as vast amounts of generated data are analyzed. The centralization imposed by a cloud design has implications for both network congestion as well as latency. Moreover, some microservices may require specialized device resources (e.g., the object detection microservice may require machine learning supporting hardware), making deployment on a single edge node which does not possess such capabilities infeasible. Considering this, deployment nearby the system's end-devices is required – in this manner, computation and data management can be performed closer to the targeted area and distributed among participating nodes.

5.2 Problem formulation

In this section, we outline our system model and the assumptions behind it, as well as the objectives that we consider. Similar to the previous chapter, we aim to deploy an IoT application on a target edge system – the model of the two remains unchanged. Therefore, we briefly mention the IoT application and edge system model, focusing more on new characteristics and objectives that are relevant to the current resource management framework, i.e., the definition of communication latency and some edge node characteristics.

5.2.1 Application and System Model

As previously stated in Chapter 4, the target edge system is a distributed system consisting of multiple, possibly heterogeneous and mobile edge nodes, i.e., E_N={E_1, E_2, ... }. Nodes with software stacks capable of executing microservices and communicating with other edge nodes. Each edge node has a certain set of available resources E_{res}={r_1, r_2, ... } that can be shared within the system to facilitate the deployment of IoT application. From the entire edge system, the application coordinator selects a group of edge nodes, i.e., participant nodes, E_P={E_{p1}, E_{p2}, ... }, found in its vicinity. Each node may represent a mobile device such as a smartphone or static devices such as a server hardware. Finally, we assume that nodes share resources and collaborate without the need for incentives. However, we acknowledge the need and importance of suitable incentive mechanisms to reward nodes that share resources and behave cooperatively instead of competitively – we identify the development of such incentive mechanisms as future work.

To fully utilize the nearby available computational resources, an application may be deployed on different edge nodes. Partitioning the application functionality into microservices is typical within distributed edge systems, where the execution of an entire application may not fit on a single edge node [DTF16]. An application model is defined by the developer at design time and consists of a set of microservices M={m_1, m_2, ... }, and communication links. We assume that the application has only one communication flow that starts with a source which provides source data e.g., from an IoT sensor, and ends with a sink, i.e., an actuator microservice, to take actions on the obtained results. More concretely, we assume that an application model is described by a direct acyclic graph (DAG), G_{app} = (V, E), where vertices represent microservices and edges show the links between them. Considering this, we can model our motivational example as shown in Figure 5.2. It is important to mention that we abstain from application particulars such as how coordination occurs at the business logic level; our approach is concerned with finding a suitable deployment strategy across the edge system. Given the application model, our technique is agnostic about the inner workings of the deployed application.

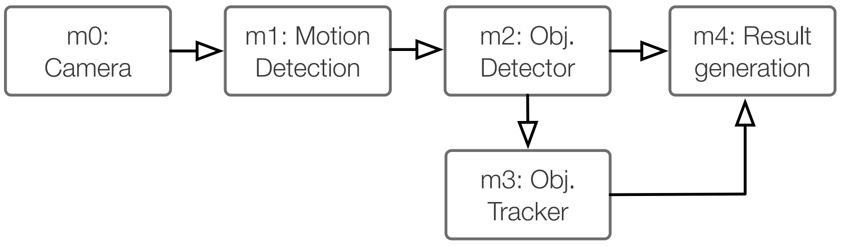


Figure 5.2: Public Safety application model.

A microservice m_i implements a set of instructions that performs a specific function of an application. Besides the set of computational resource requirements, i.e., M_{req}={r_1, r_2, ... }, a microservice may require specific requirements like data, domain-specific hardware, sensors, or actuators. For example, a particular requirement, for the *motion detection* *microservice* of our example, is represented by the collected data from a specific area during a time frame.

We define the latency between two dependent microservices as the time required to send a message from the source to the destination microservice. In an edge system, edge nodes communicate via communication links – each communication link between two nodes, $\mathsf{E_{p1}}$ and $\mathsf{E_{p2}}$, has an associated latency $\mathsf{l_{E_{p1},E_{p2}}}$. Therefore, microservices inherit the communication latency from their host node. For example, if $\mathsf{m_1}$ is mapped on $\mathsf{E_{p1}}$ and $\mathsf{m_2}$ on $\mathsf{E_{p2}}$, the communication latency of $\mathsf{l_{m_1,m_2}}$ is equal to $\mathsf{l_{E_{p1},E_{p2}}}$. Furthermore, we compute the application's latency as the sum of all communication delays between dependent microservices along the application's communication flow. Acknowledging the important role of latency in our technical framework, a latency monitoring module is imperative to the overall functionality. We present an example of a simple monitoring module in Chapter 7. However, proposing a more complex monitoring module is out of our scope; we assume that latency is adequately measured and provided.

5.2.2 Objectives

In an edge system, we deploy applications across multiple connected edge nodes, which makes latency induced due to network and distribution a prime concern. A secondary concern highly pertinent to distributed systems, is edge node resource preferences; participant nodes should be able to take decision on managing their available resources, according to internal strategies defined by the administrative entity and incentives received. We treat those two concerns as key drivers of our resource management approach, which must be satisfied upon deployment.

Our first objective targets one of the fundamental concerns of contemporary applications, i.e., latency – one of the main arguments for edge computing. We focus on a particular manifestation of latency, which is the e2e delay of an application when operational. In contrast to the e2e delay presented in the previous chapter, we define the e2e delay as the duration of time required by an application to produce a result from received source data – it does not include the microservices' WCET. For example, the e2e delay of our example application (Figure 5.2) captures the duration of time for $\mathsf{m_4}$ to generate a result once $\mathsf{m_1}$ collects data from its sensors. We assume that the desired e2e delay (as an SLA) for an application is defined by the developer.

Our second objective is to respect resource preferences of participating nodes. Each node has authority on how its resources (including hardware or sensing capabilities) are shared with others – data that may reside locally are similarly treated. We achieve this behavior by enabling edge nodes to take decisions locally, which guarantees the mapping of microservices where data or resources required reside.

We want to mention that a centralized solution where the coordinator resides in the cloud is generally possible. However, in this chapter, we target decentralized edge-intensive systems, where (i) a connection to the cloud (for all participating nodes) cannot be assumed, and (ii) we seek to avoid the single point of failure that such an arrangement would introduce – in fact, any participating edge node (with or without a cloud connection) can serve the role of an application coordinator.

5.3 Resource management Technical Framework

In Figure 5.3, we present an overview of the coordinator's internal communication and the communication exchange with the selected collaborators. Our resource management technical framework consists of two major modules, i.e., (i) *the deployment policy module* and (ii) *the decision policy module*. The former implements a novel decentralized resource allocation technique that aims to deploy an application without prior knowledge of edge nodes' available resources. In contrast, the latter uses multiple decision strategies to take local decisions considering their current available resources.

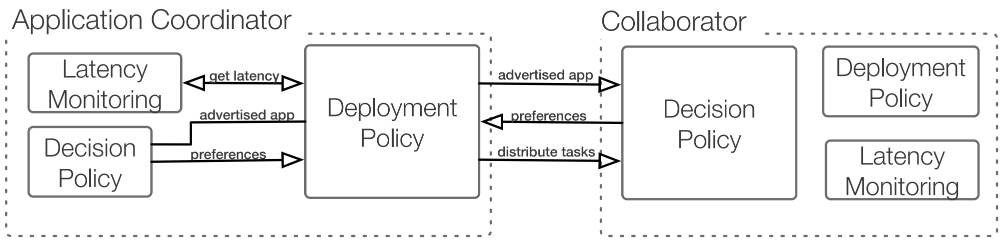


Figure 5.3: Decentralized resource management: technique overview.

5.3.1 Deployment policy module

The deployment policy module purpose is to distribute microservices on a set of edge nodes collaborators such that the overall application requirements are satisfied. Therefore, the functionality of this module represents the coordinator's capabilities and consists of two different stages:

1. *Advertising stage.* Once the application coordinator receives an inquiry for application deployment, a message containing a description of the application model, i.e., a list of microservices and their characteristics, is advertised to each participant node. The coordinator allocates a bounded time frame for receiving the node's preferences; if during this period a node does not send its preferences, then the coordinator does not consider the node during the deployment stage.
2. *Deployment stage.* After the advertisement time frame in which the coordinator collects all node preferences, the deployment stage starts. The coordinator finds a satisfiable allocation of microservices to participating nodes by considering the application requirements and participant preferences; a node preference can be partial or fully fulfilled.

We note that, for complex resource allocation problems, researchers have proposed solutions based on metaheuristic optimization algorithms. Such solutions typically yield a near-optimal deployment strategy over a longer period of time, are applicable in situations where the target edge system is not defined by high uncertainty, and require a host with powerful capabilities. As a result, a centralized solution where the coordinator is deployed in the cloud or on nodes with powerful computational resources is followed [SA17, AH15]. We differentiate ourselves from these approaches as follows. Firstly, our target domain is edge computing where (i) edge nodes may have limited computational resources, and (ii) latency and nodes preferences are first-class concerns. Secondly, we aim for guarantees, i.e., we provide a satisfiable solution that does not represent the near-optimal microservice allocation, but it is more valuable when deploying an application in a volatile edge system where the topology can change multiple times during the execution of an application – rendering an optimal solution not valid anymore. Furthermore, in metaheuristic approaches, there are no guarantees that the generated solution satisfies the application's objectives, especially if simulation time is limited. Consequently, we choose to cast our problem within SMT – we provide guarantees that if a solution exists, it is found and correctly satisfies the application requirements.

Notice that SMT fits our resource allocation problem particularly well; (i) the placement of microservices on nodes are essentially constraints over the space of deployment options, which can be encoded in first-order logic, and (ii) numerical latency SLAs can be encoded by integer linear arithmetic. Consequently, to solve our resource allocation problem, we divide our SMT formula into four different encodings, i.e., the *microservice facts*, the *domain facts*, *preferences constraints*, and *constraint formulation.* These capture different constraints over the desired solution, and are illustrated in the following.

Microservice Facts

In the *microservice facts* encoding, we define a set of rules for the correct logical placement of microservices on the target edge system. As a rule, only one instance of a microservice $\mathsf{m_i}$ can exists on the edge system at a time, e.g., we cannot deploy the same microservice $\mathsf{m_i}$ twice. Furthermore, we can allocate a microservice $\mathsf{m_i}$ on an edge node $\mathsf{E_{pn}}$ only if it is part of the set of microservice sent as preferences by that particular node. For example, consider that we desire to deploy the motivational example application presented in Section 5.1, composed of five microservices $\mathsf{m_0}$, $\mathsf{m_1}$, $\mathsf{m_2}$, $\mathsf{m_3}$, and $\mathsf{m_4}$, on two participant nodes, i.e., $\mathsf{E_{p1}}$ and $\mathsf{E_{p2}}$. Now, let us assume that the coordinator receives, during the advertising stage, the following node preferences: node $\mathsf{E_{p1}}$ sent $\mathsf{P_1} = \{[\mathsf{m_3}, \mathsf{m_4}]\}$ and $\mathsf{E_{p2}}$ choose to host $\mathsf{P_2} = \{[\mathsf{m_1}, \mathsf{m_2}, \mathsf{m_3}]\}$. Based on the rules enforced by this encoding, each participant node can receive the microservices that are not common between $\mathsf{P_1}$ and $\mathsf{P_2}$, i.e., $\mathsf{m_1}$, $\mathsf{m_2}$, and $\mathsf{m_4}$. However, the common microservices, i.e., $\mathsf{m_3}$, can be deployed only on one node, independent of how many nodes prefer to receive it. The general encoding is shown in Formula 5.1, where n_M represents the total number of microservices. The semantics of *map()* is to provide a microservice allocation between $\mathsf{m_i}$ and one participant node, where *participants* represents the set of nodes that preferred to share resources for that particular microservice.

$$\mathsf{microserviceFacts}: \bigwedge_{i=1}^{n_M} (\exists!\, E : \mathsf{map}(m_i = E)),\ \forall E \in \mathsf{participants}. \tag{5.1}$$

Domain Facts

This encoding captures the latency between two dependent microservices which are mapped on different nodes. As described in Section 5.2.1, the latency is found by giving an analogy between the microservice mapping derived from the *microservice facts* and their associated host node latency. As a result, if a microservice m_1 is deployed on E_{p2} and m_2 is deployed on E_{p1}, then the communication latency between m_1 and m_2 is equal to the communication latency of the two edge nodes, i.e., E_{p1} and E_{p2}. The general formula is shown in Formula 5.2, where l_{m_i,m_j} represents the latency between two microservices, while $l_{E_{pi},E_{pj}}$ represents the latency between two nodes, and n_{E_P} represents the total number of participant nodes.

$$\begin{aligned} \mathsf{domainFacts}: \bigwedge_{k=1}^{D} (m_i = E_{pi},\ m_j = E_{pj}) \Rightarrow (l_{m_i,m_j} = l_{E_{pi},E_{pj}}) \\ \text{for } D = \{i,j\} \text{ where } i \in [0,\ n_M] \text{ and } j \in [0,\ n_{E_P}]. \end{aligned} \tag{5.2}$$

Preferences Constraints

The node's preferences, $P = \{p_1,\ p_2,\ ...\}$, consists of multiple groups of microservices that a particular node can host. Each decision strategy introduced in the *decision policy module* generates a group of microservices – for example, one strategy derives p_1, while another strategy generates p_2. Furthermore, our decision strategies guarantee that choosing microservices from the same group does not exceed the node's available resources. Recall that the application coordinator finds a deployment strategy using only the node's preferences. Therefore, the coordinator cannot ensure that when combining microservices from different lists, e.g., from p_1 and p_2, the node can still host them. We must create an encoding that will guide the coordinator to choose microservices from the same group. Creating such an encoding is not trivial since each participant computes its preferences locally and the coordinator does not know the node's available resources. As a result, the coordinator must decide based on the node preferences received during the advertising stage.

In the *preferences constraints*, we define a set of rules to aid the application coordinator in finding a valid deployment strategy, considering only the node's preferences. More concretely, the mapping rules limit the coordinator to choose microservices to the boundary of an individual group. Therefore, the coordinator can choose microservices from only one group from the preferences sent by a participant node. Recalling our motivating example, a participant E_{p1} receives in the advertisement message the motivation example application model and based on its own decision strategies creates the following set of preferences $\mathsf{P} = \{\mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3, \mathsf{p}_4\}$, where p_i contains a group of preferred microservices, e.g., $\mathsf{p}_1 = [\mathsf{m}_3, \mathsf{m}_4]$, $\mathsf{p}_2 = [\mathsf{m}_1, \mathsf{m}_2]$, $\mathsf{p}_3 = [\mathsf{m}_1, \mathsf{m}_4]$, and $\mathsf{p}_4 = [\mathsf{m}_2, \mathsf{m}_3]$. As a consequence, the coordinator can choose microservices from a single group to be mapped on a participant node. Let us consider that the coordinator chooses p_2 as the best group sent within P. In this case, choosing p_2 means that every microservice that is not part of this group is not considered in the deployment strategy, since it may exceed the available resources of that node. However, there may be common microservices between groups, e.g., there are common microservices, i.e., m_2, between p_2 and p_4. In this scenario, we must guarantee that if m_2 is mapped first on E_{p1}, we do not block the microservices from p_2 and p_4 since, with the current information, the coordinator can choose both groups. Let us consider that the next microservice mapped on E_{p1} is m_1 – only then the remaining microservices part of p_4 cannot be chosen anymore. In contrast, if m_3 is selected, then the coordinator cannot choose microservices from p_2. The encoding is shown in Formula 5.3, where n represents the total number of node's preferences received.

$$\mathsf{prefConstaint}: \bigwedge_{\mathsf{i}=1}^{\mathsf{n}}((\mathsf{p}_1 \vee \mathsf{p}_2 \vee \mathsf{p}_3 \vee \mathsf{p}_4) \wedge (\mathsf{p}_1 \implies !(\mathsf{p}_2 \vee \mathsf{p}_3 \vee \mathsf{p}_4)....)). \tag{5.3}$$

Constraint formulation

The last encoding of our SMT formula ensures that the deployment meets the latency SLA of the application; *constraint formulation* captures rules that account for the latency in the e2e delay. To instrument a complete application, the developer should additionally account for its execution overhead as well. As a result, the encoding presented in Formula 5.4 guarantees that the sum of the communication latency $\mathsf{l}_{\mathsf{m}_1,\mathsf{m}_2}$ does not exceed the total required SLA, where n_e represents the number of edges.

$$\mathsf{e2eConstraint}: \sum_{\mathsf{i}=1}^{\mathsf{n}_\mathsf{e}} \mathsf{l}_\mathsf{i} \leq \mathsf{SLA}. \tag{5.4}$$

By combining the aforementioned constraints, we obtain the complete formula $\mathcal{F}$ used by the coordinator to find a satisfiable allocation according to application requirements:

$$\mathcal{F}: \mathsf{microserviceFacts} \wedge \mathsf{domainFacts} \wedge \mathsf{prefConstraints} \wedge \mathsf{e2eConstraint}. \tag{5.5}$$

Solving $\mathcal{F}$ incurs an energy cost, something which has to be accounted for since we target possibly resource constrained edge settings. The energy cost amounts to the execution of an SMT solver against the formula – later, we demonstrate that it is quite feasible to do so on single-board computers for relevant problems. The energy draw depends on the CPU power draw to solve $\mathcal{F}$ – full CPU usage for certain amount of time, depending on the problem size. Furthermore, executing the deployment policy also introduces a communication cost, for which we can calculate bounds for the exchanges required for each stage. In the *advertising stage*, the coordinator node sends a message to all participants and waits for their preferences. This stage requires a total of *2*m* messages, where *m* is the number of nodes. In the *deployment stage*, the coordinator informs only the nodes that will receive microservices, with a maximum of *m*.

5.3.2 Decision policy module

The decision policy module concerns strategies that a participant uses to create a set of preferences, i.e., P, for an advertised application. As mentioned before, these strategies enable a collaborator node to create groups of preferred microservices based on their preferences and current internal state. As such, node preferences are enforced since every decision is made locally without sharing information with other nodes in the system. Besides the property of considering the collaborators' microservice preferences, the strategies play a more fundamental role in the overall functionality of our framework – to ensure coverage of microservices. Generally, to ensure that the application coordinator receives at least one preference for every advertised microservice, a consensus model is preferred where participant nodes communicate with each other to decide for what microservices to share their resources. However, in this scenario, there is an increased communication overhead and a node does not make decisions by itself; forcing a participant node to compromise according to the preferences of other nodes. Therefore, in the following, we outline some indicative strategies that participants may use to create P.

We especially note that the coordinator has no control over the participants' microservice preferences; in our conception, they are free to contribute (any) resources by sending microservices preferences of the advertised application. The rationale of giving participants *free rein* about their resource contribution to the system is as follows. Every participant may decide to adopt four default, indicative tactics to increase the number of groups sent in P. By choosing four tactics, we intend to aim for greater coverage without requiring any information from other nodes. Each tactic has a different role in creating a group of preferred microservices. Hence, we conceptually group them based on their role in two different strategies.

Maximization Strategy

This strategy aims to maximize the number of microservices, placed in a group (i.e., $\mathsf{p_i}$), by maximizing the utilization of all available resources of a node. In this strategy, we present two independent tactics. The first tactic is based on the well-known , i.e., *0-1 knapsack* dynamic programming algorithm. We note that this fits well since it yields the near-optimal solution. However, although near-optimal, this tactic has high computational demands – as we focus on microservice allocation, we consider efficient tactic development as an interesting avenue of future work. An alternative can use heuristics to approximate knapsack-like solutions. The second tactic adopted is based on a random selection of microservices. Notice that even though the overall strategy offers great coverage, the changes that the application coordinator will distribute an entire group are small, since these tactics do not consider dependencies between microservices.

Dependencies Strategy

Compared to the previous strategy, with the dependencies strategies we create more microservice groups that considers the dependencies between microservices. To achieve this purpose, we employ two graph-theoretical algorithms as tactics, i.e., *strongly connected components* and *fan-out*. The former finds the largest strongly connected component into the application model and builds a group by selecting the microservices from the component until it reaches the node's maximum available resources. In contrast, the latter selects microservices from the biggest edge fan-out found in the application model.

It is important to stress that these strategies are indicative to participant nodes and are built to offer suitable node preferences for a wide range of applications. The above indicative strategies aim to bootstrap choices for a collaborator, which then can amend based on its internal resource sharing rules. Each collaborator utilizes the strategies above to generate its preferences within an advertised application. As we observed, the coordinator then proceeds to calculate a satisfiable mapping based on the technique presented in Sec. 5.3.1. Note that the preferences of a participant for certain microservice might be fully or partially satisfied, based on the application-wide objectives.

Energy costs for collaborators can be adjusted by selecting different strategies to capture the node's preferences. We especially note that different collaborators' strategies would be interesting to develop in tandem with incentive mechanisms; in essence, to encourage participants to consider more microservices, something we identify as future work. Finally, the actual microservices have to be deployed in nodes. In practice, this entails downloading containers from a container repository. Costs arising from this are application dependent; size of containers comprising the application microservices and downlink bandwidth are key such factors.

5.4 Evaluation

To evaluate our technique and accompanying technical framework capabilities to devise deployment strategies that efficiently use the edge node's available resources, we consider two evaluation goals; applicability and performance. For the former, we present and deploy four different application models obtained from literature at the edge – we are interested in demonstrating the resource management technical framework's ability to deploy realistic applications. For the latter, we follow a quantitative approach to evaluate the performance of our technique on resource-constrained devices. To concretely support evaluation, we realized a prototypical tool based on the CVC4 SMT solver [BCD$^+$11], which we deploy on a resource-constrained device acting as the coordinator. Furthermore, we present the performance of our framework when a more powerful device is hosting the application coordinator. After the applicability aspects, we describe the experimental setup and finally discuss the obtained results.

5.4.1 Applicability: IoT Applications at the Edge

We consider that a deployment technique may be intended for three different scenarios, i.e., offloading, mapping, and job assignment. Offloading refers to the possibility of deploying computational microservices of an application to nearby edge nodes to ensure better functionality and optimize the energy consumption of resource-constrained devices. Such a practice is usually employed for smartphone applications. The second scenario represents a mapping of microservices permanently deployed on an edge system. A scenario most useful in cases where the target edge system does not have a high uncertainty – however, if we target a volatile edge system, then a self-adaptive technique must assist the resource management framework to ensure correct application functionality even after changes in the system occurs. Finally, the third scenario is suitable for applications that are instance-based, meaning that a deployment occurs only when the coordinator receives a request to do so. In this scenario, we capture the utilization of available resources of nearby edge nodes only for a limited time. Our technique is capable of deploying an application for all three scenarios, however, we consider that, due to its nature, it provides the most benefits in the context of microservice offloading; we can enable a resource-constrained device to make decisions locally on what microservices to offload and where, without the need of a central entity. To evaluate the applicability of our framework, we model four realistic applications, usually deployed on resource-constrained devices incapable of executing an entire application locally. Considering this classification, we select two mobile applications, i.e., (1) an antivirus application and (2) a face recognition application, one application that fit the second scenario, i.e., (3) the public safety motivational example of Section 5.1, and (4) a team building application representing an example of an instance-based application, to be deployed on an edge system composed of five edge nodes.

A set of computational resource requirements, i.e., a tuple (RAM, CPU, HDD, *{OTHER}*), characterize each microservice and node, where *OTHER* represents a set of special requirements of a microservice or special resources available on a node, capturing their functionality and capabilities. We can map the microservices which require no other specific resources (i.e., shown with $\emptyset$) on any node if there are available computational resources. Furthermore, for every individual application model, we distribute on participant nodes a set of available computational resources of random values between 5 to 10 units (for the first three applications) and between 10 to 20 units (for the last application). For the team building application, we increase the nodes' available resources since there is an increase in size. In addition, we deploy other resources required by the application on different nodes. For illustration purposes, we adopt a generalized 'unit' for resource quantification – in practice, this would be refined per application (e.g., in MB/GB for RAM or GHz for CPU). Finally, we randomly assign a communication latency between 1 and 10 ms. For all application deployments, we deploy the coordinator on an ARMv8 R-Pi3 device featuring a 1.2GHz CPU and 1GB RAM – a device serving as the edge node. In contrast, we simulate the participant nodes on a machine with a dual-core Intel i5 2.3GHz processor. We make available the application models, further details, and evaluation results in our online appendix [1]. Moreover, we provide the implementation and technical details in our git repository[2].

Antivirus Application (A1)

This mobile application behaves like a software antivirus and is modeled using a DAG graph composed of 5 microservices and 5 edges [MB18]. We can observe from Figure 5.4 that two microservices require special resources to ensure correct application functionality, i.e., microservice m_0 which requires a set of files that must be scanned and m_5 that requires a node with a display to present the results to the user. Besides the computational resource requirements of each microservice, we assign as the application objective an *SLA = 30.* The application is deployed on an edge system composed of 5 collaborators. In Figure 5.4, one can see the deployment strategy found, as well as the available resources of each edge node and the resource requirements of each allocated microservice. The successful mapping (indicated on Figure 5.4 with the dotted edge nodes) is found in *434 ms* and has an *SLA = 23.*

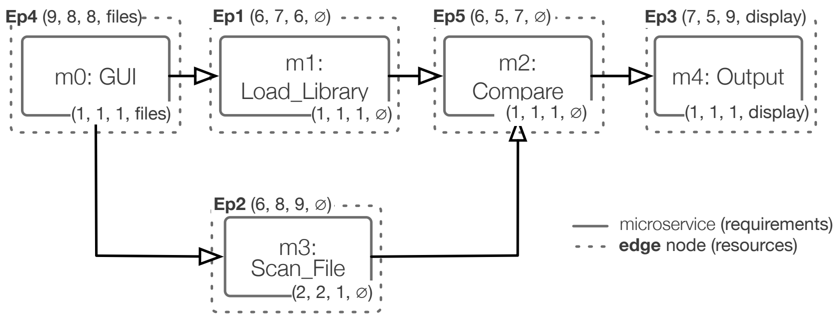


Figure 5.4: Antivirus application and deployment (in overlay).

Facerecognizer Application (A2)

The second mobile application we deploy represents a *facerecognizer* that models an image processing application able to identify a face in an image. Similar to the antivirus application, the model contains 5 microservices and 5 edges [MB18]. In the same manner, two microservices require special resources to be in place: m_0 requires a set of images as input and m_5 needs a display. We choose for this application an *SLA = 30* and we deploy it on an edge system, having the same size as the previous application, but with a different distribution of available resources. In Figure 5.5 we show the microservices' resource requirements, the nodes' available resource, and the satisfiable deployment strategy found. We require *422 ms* to find a solution to deploy the *facerecognizer application* at the edge in the absence of cloud resources – the application has an *SLA = 19.*

[1]https://dsg.tuwien.ac.at/team/cavasalcai/projects/ResourceManagement
[2]https://github.com/cavasalcai/Decentralized-Resource-Management

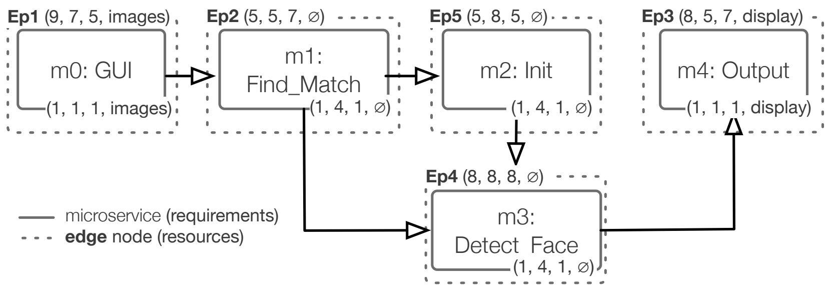


Figure 5.5: Facerecognizer application and deployment (in overlay).

Public Safety Application (A3)

The *public safety application* model is defined by a set of 5 microservices and 5 edges and requires an *SLA = 30.* The application is deployed on an edge system composed of 5 edge nodes. In Figure 5.6, we present the microservices' resource requirements and nodes' available resources. Furthermore, we can observe from Figure 5.6, that m_1 represents the only microservice that requires some special resource, i.e., raw video data – microservice m_0 represents the input sensing microservice and provides m_1 with input video data. A satisfiable allocation is found in *407 ms* and has an *SLA = 24*.

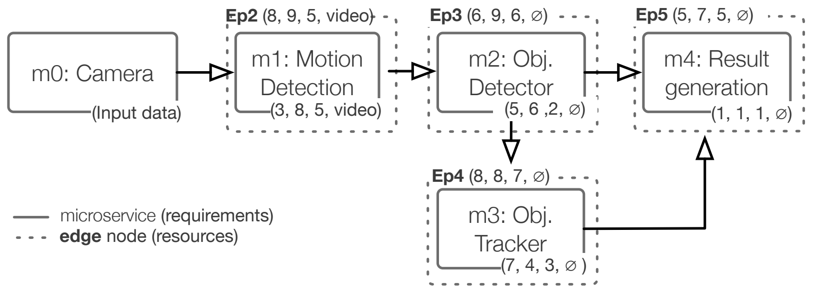


Figure 5.6: Public safety application and deployment (in overlay).

Team Building Application (A4)

We propose an application to help companies finding the perfect team building location based on an analysis of user personal data. The application has an *SLA = 50*, contains 8 microservices, each with a different role, and 11 edges. Similar to the other applications, we present the microservices' resource requirements and the collaborators' available resources in Figure 5.7; microservice m_0 represents the source microservice and symbolizes the need for different types of data required by the 4 dependent microservices. The dependent microservices that analyze the input data are: m_1, which performs analysis of the employee's stored health information, m_2 analyzes the stored video files by performing motion detection, m_3 performs object detection on all stored images, and m_4 analyzes the environment where the employee works. Based on this information received from previous microservices, m_5 generates a list of possible locations on which m_6 computes the budget required. Finally, m_7 represents the end microservice and creates a report suggesting possible destinations as well as an estimation of travel cost. Considering the defined experimental setup, a solution is found in *510 ms* with *SLA = 31*.

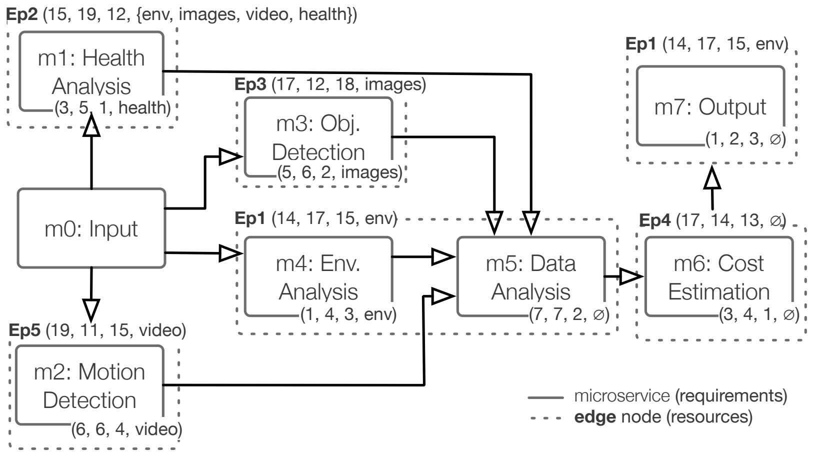


Figure 5.7: Team building application and deployment (in overlay).

5.4.2 Performance: Experiments Setup and Results

To quantitatively evaluate our resource allocation framework, we consider as a performance metric the execution time required to obtain a distribution of microservices to the participant edge nodes. We demonstrate the coordinator's ability to find a satisfiable deployment strategy at the edge, considering the technique's highly computationally-demanding solving component. Furthermore, to understand the impact of each encoding on the overall framework performance, we perform an analysis of the SMT formula by examining the number of symbols of each encoding presented in Formula 5.5, i.e., *domainFacts*, *microserviceFacts*, and *prefConstraints*. For this purpose, we design an experimental setup of an application and a target edge system.

For the application model, we adopt *montage* [BSM10], a real-world DAG workflow. The application consists of 24 microservices, each having resource requirements between 1 to 10 units and 50 edges. To accurately evaluate performance and avoid discarding satisfiable deployment strategies due to randomness in the distribution of specific resources on nodes and other limitations introduced by factors like SLA, we set the SLA to a large value and set each microservice to require no specific resources. Regarding the edge system, we randomly assign to each edge node a set of available resources chosen in the range of 10 to 20 units.

The overall objective of our experiment setup is to map the application to an edge system in which we gradually increase the available resources (i.e., by increasing the total number of collaborators). We adopt the same test environment used in the applicability scenario from Section 5.4.1. We perform 500 tests for each newly created edge system, on which

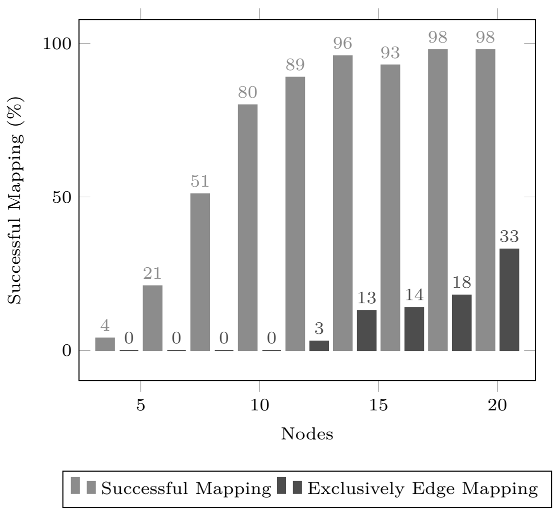


Figure 5.8: Successful mapping over number of participant nodes.

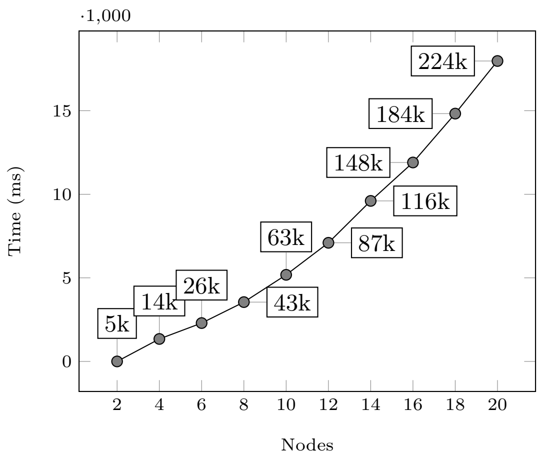


Figure 5.9: Mapping time over number of participant nodes and formula size.

we measure: (i) the percent of successful mapping and exclusively at the edge mapping for different numbers of participating nodes, (ii) the time required by the coordinator to find a successful mapping of microservices to different size groups of participants, and (iii) the number of symbols each encoding of the SMT formula requires. Our results are presented in Figures 5.8, 5.9, and 5.10. In Figure 5.8, we observe that as the number of nodes increases, the successful mapping rate improves. We can observe a similar behavior in Figure 5.9, where both the execution time and the number of symbols required by $\mathcal{F}$ increase with the number of participant nodes. Finally, in Figure 5.10, the relation between the total number of symbols of $\mathcal{F}$ and each encoding is presented.

Recall that, in all experiments, the application coordinator resides on a resource-constrained device, i.e., an ARMv8 R-Pi3 device featuring a 1.2GHz CPU and 1GB

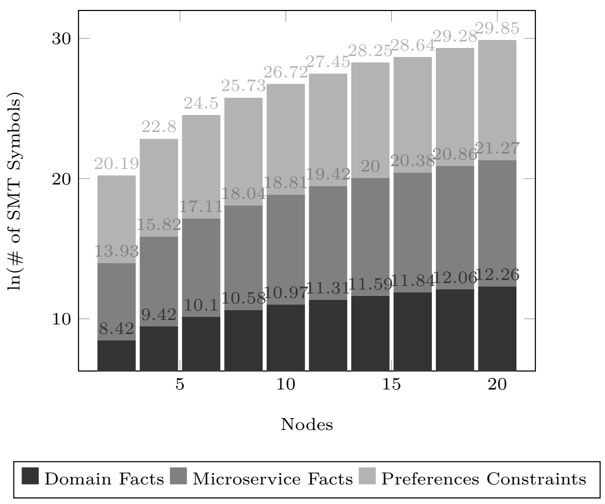


Figure 5.10: Contribution in symbols of different problem components to the overall formulae, over increasing node count.

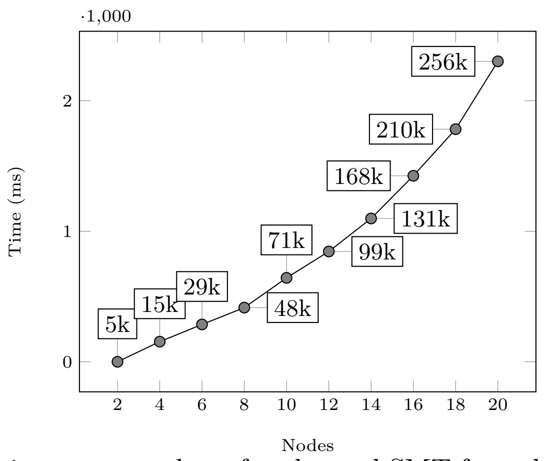


Figure 5.11: Mapping time over number of nodes and SMT formula size when coordinator resides on a powerful device.

RAM. However, we are interested in observing how the performance will change if a more powerful device, i.e., a machine with a dual-core Intel i5 2.3GHz processor, hosts the application coordinator. Therefore, similar to Figure 5.9, we present in Figure 5.11 the total time required by the coordinator to find a satisfiable deployment strategy.

5.4.3 Discussion

We have demonstrated that by using our resource management framework, we can successfully deploy applications in a decentralized manner. We note that for all applications considered (Section 5.4.1), a successful microservice allocation at the edge (without resorting to cloud resources) is achieved in under *510 ms*, i.e., A1 is determined in *434 ms*, A2 in *422 ms*, while A3 is found in *407 ms*, and A4 in *510 ms*. Therefore, we successfully demonstrate that our proposed technical framework is efficient in deploying different types of realistic applications in under a second, a duration suitable for all three scenarios involving offloading, mapping, and job assignment. Note that the deployment time of an application is negligible for the overall application's lifecycle – during the deployment stage, the application is not operational yet. Moreover, a deployment time of 500 ms is small compared to the overhead of moving the containers to their assigned location.

In Figure 5.8, we have recorded the successful rate of our framework to find an allocation of microservices at the edge with or without the need for computational resources found in the cloud. For each group of nodes, we illustrate (i) the total number of satisfiable solutions found and (ii) the number of solutions found using only edge resources. As one can observe, the latter is influenced by the available resources shared between the participants. However, the decision strategies used by the participants have a bigger impact on the number of solutions fully mapped at the edge. As a result, to fully utilize the available resources found in an edge architecture, we must optimize the strategies to ensure greater microservice coverage. We propose a solution to this issue in the next chapter.

Figure 5.9 illustrates the impact that an increase in the number of participant nodes has on the execution time required by the coordinator to yield a microservice allocation. In this case, we can observe that the growth of the number of nodes influences the SMT formula size, which ultimately impacts the time required to find a solution. We note that with respect to previous preliminary work [ATD19b] (see Figure 5.11), the memory requirements of the SMT formulae produced are significantly reduced – for instance, consider the reduction of a problem size of 20 nodes concerning the SMT formula size, where we observe a decrease of *12.5%* in the formula size. This increase in efficiency grows with the number of participating nodes (e.g., for an edge system of 4 nodes, we see a decrease of *6.67%*, while for 12 nodes, a decrease of *12.12%*). Memory is a significant factor in an edge system because resource-constrained devices typically have limited amounts. Thus, the encoding used in this chapter presents obvious scalability improvements over previous work [ATD19b]. Furthermore, comparing Figure 5.9 with Figure 5.11, we can observe that the time required to find a satisfiable deployment strategy is dependent on the coordinator's host node. Even without the SMT formula optimization, the time is significantly lower than when the coordinator resides on a resource-constraint device. We highlight that if faster deployment times are desired, then more powerful nodes should host the application coordinator. Nevertheless, as proven with our evaluation, our resource management framework works with all types of nodes.

To better understand which encoding has the biggest impact on the overall SMT formula size, we performed an analysis on the three encodings, i.e., *domain facts*, *microservice facts*, and *preferences constraints*. In Figure 5.10, we illustrate the total number of symbols required for each encoding, on which we apply the base e logarithmic function for presentation purposes – we can observe that their size increases with the number of nodes in the system. For example, to deploy the considered montage graph application of 24 microservices on an edge system consisting of 20 edge nodes, we have an SMT formula size of *224K* symbols divided between the three encodings as follows; the *domain facts* has a total number of *211K*, the *microservice facts* has *8K* symbols, and the *preferences constraints* has a total of *5K* symbols. We can conclude that encoding the latency objective in the SMT formula is highly expensive (i.e., 90% of the total number of symbols) since the generated encoding is additive for every microservice latency. To properly capture the application's e2e delay, the formula requires all possible microservice mappings to nodes and their associated communication latency.

The framework can support any microservice-based IoT application that can be decomposed as a DAG. The first part of the evaluation (Section 5.4.1) demonstrates applicability on a diverse set of applications, intended to represent different scenarios obtained from the literature like offloading, mapping, and microservice assignment. These applications have different characteristics such as variable number of microservices and communication links. Furthermore, to quantitatively assess the effect of the major computational-demanding factors – the application's size, node's available resources, and number of nodes – we considered a real-world workflow, intended to stress the framework (Section 5.4.2). We believe this shows that results are generalizable, even in large applications – the extent of which is demonstrated on the time it takes to find deployment strategies. As such, we believe that the framework performance is acceptable for relevant problems – considering that for the experiments performed the coordinator resides on a single-board computer having an ARMv8 1.2GHz CPU.

We acknowledge the high computational demands of our proposed technique, but we note that it offers guarantees – something demonstrated in the realistic scenarios A1-A4. Our evaluation results show that the framework can successfully deploy, in absence of cloud resources, realistic applications with distributed resources at the edge of the network. We note that scaling up to higher numbers of participants is hindered by the sizable encoding of the e2e delay objective (Figure 5.10). If the focus is on other objectives that scale linearly with the fan-out degree of a service (such as bottleneck avoidance), the overall performance improves significantly. As a result, we conclude that our proposed technique performs best in scenarios where only a slice of the entire system is considered for an application deployment (e.g., a disaster scenario or microservice offloading) or an extra module is introduces that is capable of creating a group of participants, selecting them from the proximity of the coordinator node. Finally, we note that the technique does not guarantee the best latency, only one that is less than the required latency. For example, let us consider mobile application (A1). In this case, it is possible to improve the SLA by mapping dependent microservices on the same node if there are available resources. In this case, some microservices like m_3 and m_4 can be mapped on the same node E_{p3}, yielding a lower overall SLA. However, the deployment solution is guaranteed to be correct (by virtue of SMT), and is obtained quite fast (typical applications obtained from the literature – Section 5.4.1 – are mapped at times of under a second). Other solutions tackling similar problems in literature have different goals; here a further guarantee is that if there is a solution possible, it is always found.

5.5 Conclusion

Taking advantage of available resources closer to end-devices calls for novel resource management techniques that comply with latency, node preferences, and decentralization demands of IoT applications. In this chapter, we propose a decentralized resource allocation technique and accompanying technical framework for the deployment of latency-sensitive applications on an edge system – our application coordinator can reside on any node as long as there are enough computational resources. We specifically focus on deploying applications in the absence of cloud resources, where the coordinator is deployed on a resource-constrained device. We have demonstrated that our technique can efficiently utilize available resources at the edge and provide guarantees – if a solution that satisfies application latency objectives and task requirements exists at the edge, it will be found. Finally, in this chapter, we provide an improvement upon the resource allocation technique presented in Chapter 4, by being able to find a deployment strategy in a decentralized manner and at runtime; thus, considering all changes that may occur during the deployment stage.

CHAPTER 6

Efficient Hosting of Robust IoT Applications on Edge Systems

In the previous chapter, i.e., Chapter 5, we have proposed a decentralized resource management technique and technical framework that enables the deployment of a microservice-based IoT application to an edge system. With the previous approach, we manage to fully deploy an application on an edge system, i.e., without the help of the cloud. To efficiently use the distributed available resources found at the edge of the network, we must divide the IoT application functionality into multiple smaller components [1] – we can model such IoT application as a DAG, where vertices represent an atomic component, while edges represent the dependencies between them [MB18]. An atomic component represents a small part of the application's functionality, with a fixed set of resource requirements, that cannot be further divided into smaller components. As a result, combining the DAG application model with resource management techniques [HV18] enables the successful deployment of an IoT application at the edge of the network. However, the successful deployment of an application entirely on resource-constrained devices is highly dependent on the atomic components' resource requirements. We can deploy an application at the edge only if we have, for each atomic component, at least one edge node capable to host it – a limitation given by the inability to further decompose atomic components. As a result, finding a deployment strategy for an IoT application is highly dependent on the nodes' available resources and the components' resource requirements – forcing the mapping of some components to the cloud while leaving available resources unused at the edge.

To address the aforementioned issues, we propose a robust application model and extend the resource management technique to lower the impact of available resources on deployment success. We say an IoT application is a *robust application* if its functionality

[1]Similar to Chapter 4, we consider that the IoT application is composed of components.

can be adapted based on the edge system's available resources. Such an application can still be functional, with a different functionality level, if it faces a resource shortage on edge devices. Therefore, to achieve deployment flexibility, we extend the DAG application model such that it can include composite components and contains information that enables the deployment strategy to choose the best functionality level of the application considering the target edge system's capabilities. Such a modeling approach is inspired by the aspect-oriented flow-based programming [ZB15]. In this case, the developer can define multiple aspects for each composite component to ensure different functionality levels. Combining the proposed application model with our novel decentralized resource management is imperative to achieve efficient deployments. Furthermore, we develop a new decision policy module to enable participants to efficiently utilize their available resources and increase coverage of components between participant edge nodes.

In this chapter, we propose a decentralized resource auctioning technical framework to deploy IoT applications on an edge system. Our objective is to find a satisfiable deployment strategy that meets the application requirements, i.e., the e2e delay latency, and efficiently utilizing all computational available resources found on resource-constrained devices. Our contributions are:

- An improved IoT application model that allows for better resource utilization and enables configurable application functionality based on the node's available resources.
- An extended decentralized resource auctioning that considers the proposed robust application model, to seamlessly deploy IoT applications at runtime, assuming no design-time knowledge of network topology or devices' available resources. Furthermore, it enables the developer to push component updates in the form of new aspects, without any downtime required.
- A new decision policy that empowers a device to take local decisions regarding its available resources. Our strategy is capable of providing both feasible and optimized solutions, having the following objectives: (i) maximizing component coverage, (ii) maximizing the available resource utilization, and (iii) maximizing the application functionality.

The remainder of the chapter is structured as follows. In Section 6.1, we present the overview of our proposed solution and introduce a motivational example. Section 6.2 defines the application and architecture considered. In Section 6.3, we describe the implementation details of our proposed technique. Section 6.4 presents the methodology and results of our evaluation regarding both deployment and decision strategy. Finally, Section 6.5 concludes the chapter.

6.1 Technical Framework Overview

Our technical framework provides decentralized resource management at runtime, focusing on seamlessly deploying latency-sensitive IoT applications on the edge system such that we efficiently utilize all available resources found at the edge of the network. Our main objective is to enable developers to deploy their applications on any target system without needing any knowledge of the current network topology and nodes' available resources. In Figure 6.1, we present an overview of our technical framework. The application developer defines the robust application model, at design time, by providing the data-flow between composite components as a DAG, the components' aspects, and the application requirements. During the development process, the developer can use the EdgeFlow framework to model the composite components and provide the application communication flow. Furthermore, the developer can define for each composite component a set of aspects – aspects that give a different component functionality. Similar to the application model proposed in the research literature, we define an aspect as a DAG where vertices represent atomic components connected by edges. Once the application model is ready, the developer chooses an edge node from the target edge system to host the application.

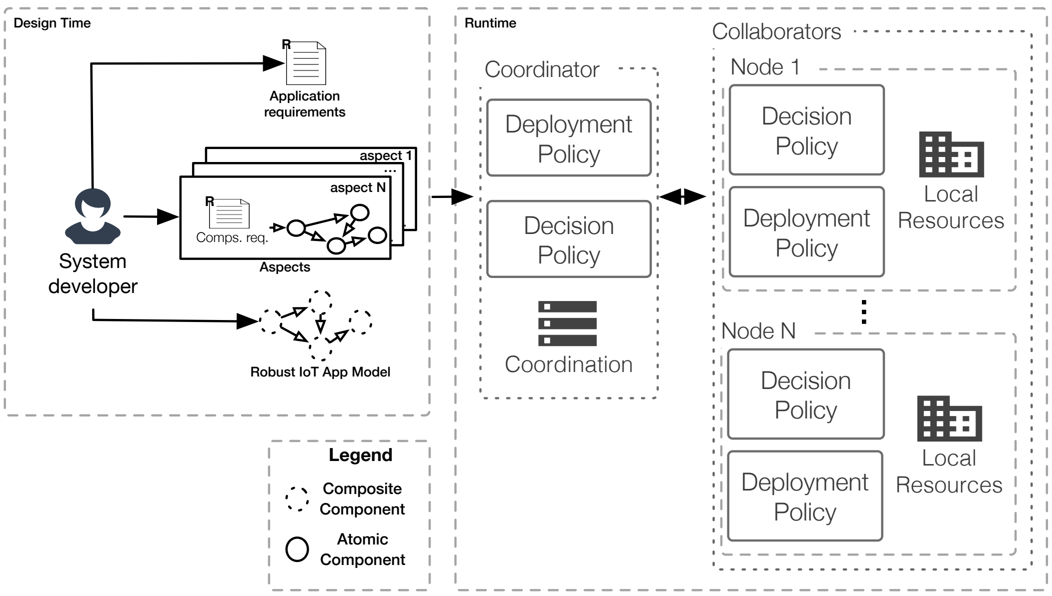


Figure 6.1: Decentralized Resource Auctioning: Overview.

We identify two roles that an edge node can have during application deployment, i.e., coordinator and collaborator. An edge node becomes a coordinator when an application deployment request arrives. The coordinator's objective is to ensure that the application deployment satisfies the application's resource requirements. To achieve this objective, the coordinator has three different phases: (i) find collaborators, (ii) find deployment strategy, and (iii) perform application updates. In the first phase, the coordinator advertises the application model to all reachable nodes to find the required computational resources needed to deploy the application. The second phase finds a satisfiable component allocation based on the received nodes' preferences – the coordinator considers its available resources when finding a deployment strategy. Finally, in the third phase, the coordinator considers the new updates received from the developer – updates provided as new aspects for the composite components.

A collaborator is an edge node that shares its available resources with the coordinator based on the information received in the advertisement message. A collaborator has total control over its available resources, being able to create a specific set of preferences based on its current internal state – the objective of each preference sent is to maximize the use of available resources and provide high component coverage. The former objective aims at maximizing the allocated available resource for the advertised application; since we empower nodes to take local decisions, the strategies used to decide how many resources to share are defined by its administrative entity and can be different from the total available resources of a node. The latter objective represents the ability of a node preference to cover all advertised components, i.e., there is at least one collaborator that can and wants to host each component. Both equally important in aiding the application coordinator to devise a satisfiable deployment. As a result, we manage to solve the limitations seen in the previous chapter. To implement the collaborator decision module, we change the two strategies presented in Section 5.3.2 with a new strategy based on CP. In the case when a collaborator receives multiple concurrent advertisement messages from different coordinators, a first-in-first-out synchronization strategy can be used; if the collaborator sent a preference for the first message received, then the resources used are locked until a decision is made by the coordinator node.

Finally, we want to mention that a collaborator becomes a coordinator for each composite component received. In this case, the node can use the deployment policy module to deploy the composite components instead of the entire application. By becoming a coordinator, the collaborator can improve the composite component's functionality based on the available resources found locally or in its proximity. This is an important feature that ensures the best application functionality possible at the current time considering the host's available resources. Furthermore, it shows that there is no need to obtain the optimal deployment strategy during the application deployment stage since (i) the application can achieve maximum functionality when more resources become available on each collaborator and (ii) the application is deployed in a volatile system where uncertainty is introduced by device heterogeneity and mobility.

Motivational Example. Serving as our running example, we consider a public safety application, deployed in a smart city scenario, with the purpose of monitoring public areas. Although the main objective of the application is, for example, finding wanted criminals and discovering stolen cars, it is also desirable to aid the authorities with finding missing persons, if enough resources are available in the edge system. For this purpose, the application is composed of distinct composite components, including *people analysis* and *environment analysis* components.

Privacy and low e2e delay latency define our public safety application, requirements that may introduce significant challenges when deploying to the cloud. First, to ensure privacy requirements and low latency, data must be processed near its origin, making a cloud deployment less desirable. Furthermore, the application should ensure correct functionality even when a stable connection to the cloud is missing. Consequently, a full deployment close to the origin of data is desirable; where the IoT application is distributed among multiple edge nodes. In our scenario, an edge node may be a static device such as a CCTV camera or a mobile device that may leave the system, at any time, e.g., a dash camera found in a car or images saved in a smartphone.

6.2 Problem Formulation

The current IoT application model presented in the research literature [MRB18b], i.e., an application is modeled as a DAG, still faces some deployment challenges since it is dependent on the available resources of an edge node; a component cannot be further divided to accommodate the target edge system's lack of available resources to host the entire application on edge nodes. In our conception, an application model is composed of interconnected composite components, where each component has defined a set of different configurations. Each configuration has a set of resource requirements that must be fulfilled upon deployment. Since we extend our decentralized resource management technical framework, essentially the system model is the same. Therefore, in this section, we focus on describing the robust application model, the considered objectives, and our assumptions.

6.2.1 Application and System Models

We target application deployment in volatile edge systems, where uncertainty is introduced by mobile and heterogeneous nodes – each edge node has limited available resources, i.e., $\mathsf{E}_{\mathsf{res}}$. Besides the available resources, a node has sensors to collect data from its surroundings and actuators to perform different actions. Let E_{N} represents the total number of nodes available in the target edge system and $\mathsf{E}_{\mathsf{P}} = \{\mathsf{E}_{\mathsf{p1}}, \mathsf{E}_{\mathsf{p2}}, ...\}$ be the number of chosen participant nodes. In Figure 6.2, we present an overview of a group of nodes considered for deployment, where each device can be a coordinator or collaborator.

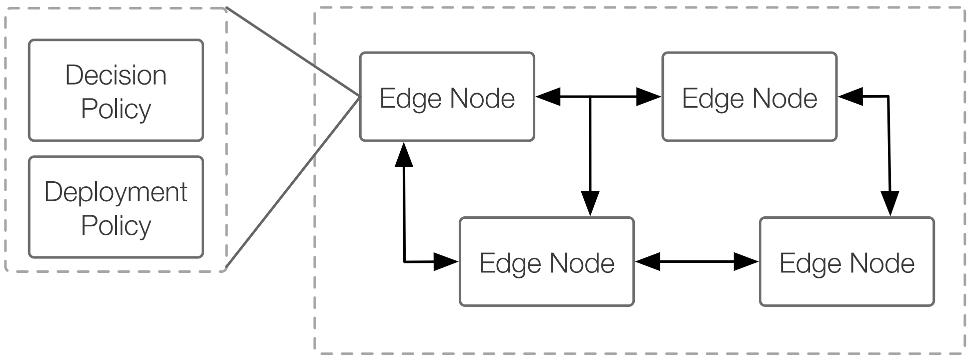


Figure 6.2: Edge System Architecture: Overview.

Robust Iot application model

In our conception, a robust IoT application model consists of multiple interconnected composite components – a model defined as a hierarchical graph. Our application model is inspired by the aspect-oriented flow-based modeling paradigm [ZB15], where cross-cutting concerns of an application, e.g., security, persistence, synchronization, fault detection, can be modularized as an aspect. In this chapter, we see the functionality level of an IoT application as a cross-cutting concern which is impacted by the resource availability of edge nodes; therefore, it can be modeled as an aspect to adopt the application's behavior and functionality according to the available resources. The application developer specifies these aspects for different levels of functionalities, e.g., from a very critical operating mode to the fully functional operating mode, along with their priorities. Our resource management technique takes these aspects into account and tries to host the application according to available resources and the given functionality aspects of each composite component.

To be concrete, we assume that an application model consists of a set of composite components $\mathsf{C} = \{\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3, ...\}$, where each component has defined a set of aspects (i.e., a configuration), e.g., $\mathsf{c}_1 = \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, ...\}$. The model uses a DAG, $\mathsf{G}_{\mathsf{app}}$ = (V, E), to model the data flow between composite components. The application model for our motivational example is presented in Figure 6.3.

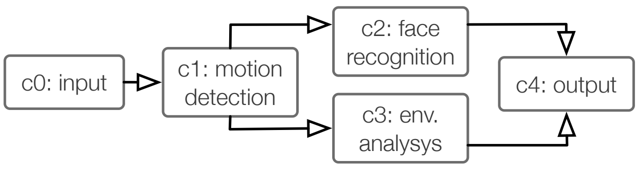


Figure 6.3: Smart Building Application model.

A composite component c_i performs a specific application feature that is integral to the overall application performance. To this end, we define each composite component as a set of aspects from which it inherits their resource requirements; resource requirements and functionality are dependent on the chosen aspect. Let us consider that the *people analysis* component, from our motivational example, has a set of 2 predefined aspects, $\mathsf{c}_2 = \{\mathsf{a}_1, \mathsf{a}_2\}$; aspect a_1 represents the desired functionality, having the highest resource requirements, while a_2 represents the minimum functionality level with the lowest resource requirements (see Figure 6.4 and 6.5).

As we can observe, an aspect a_i defines the configuration and functionality of a composite component. Each aspect is developed at design-time and specifies, at least, the minimum functionality required by that particular composite component. Moreover, an aspect can enable a range of different functionalities that a composite component should perform. As a result, we have established that aspects have different priorities based on their functionality and resource requirements. For example, the base configuration for a component is considered a_1, which always has the largest resource requirements and provides the maximum functionality. As such, we assign to this aspect the highest

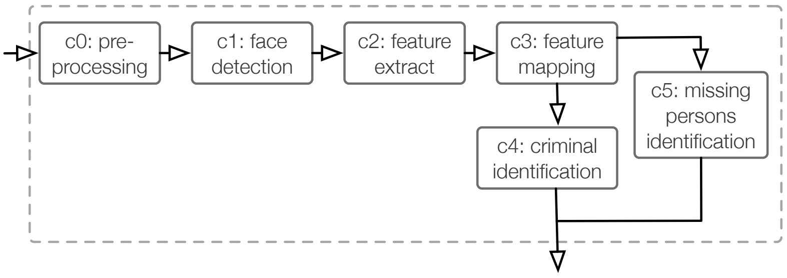


Figure 6.4: Aspect 1 (a_1): Face recognition for missing persons.

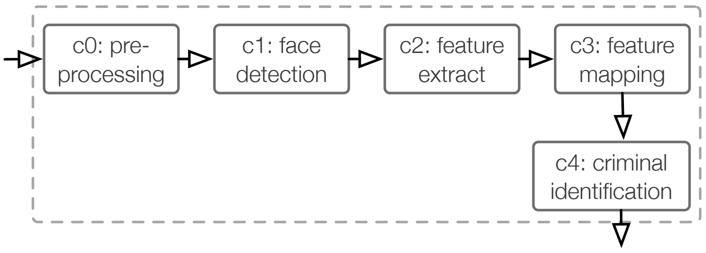


Figure 6.5: Aspect 2 (a_2): Face Recognition for wanted persons. [ASDL+19b].

priority. At the end of this spectrum, the last aspect, i.e., a_2 in the aforementioned example, has the lowest priority; since it has the smallest resource requirements and minimum functionality. In conclusion, an aspect is defined by a set of resources, $R_a = \{r_1, r_2, r_3, ...\}$, and a DAG modeling the workflow between its atomic components. With such a modular approach, a composite component can have different aspects depending on the host node's available resources. As a result, the deployment technique can find deployment strategies that efficiently use the edge nodes' available resources.

Finding the application's e2e delay is critical to the overall deployment strategy, hence we define the two important components required for its computation, i.e., (i) the communication latency and (ii) the component's WCET. The former, as defined in Section 5.2.1, is dependent on the component location since the latency between c_i and c_j, i.e., l_{c_i,c_j}, is equal to the latency of their host nodes, i.e., $l_{E_{pi},E_{pj}}$. The latter is dependent on the host's internal status and must be computed locally when the collaborator node prepares the preferences list. We discussed in Chapter 4 that finding the component's WCET deployed in a volatile edge system, at design time, is challenging since we do not have any knowledge about the network topology or the current status of each node. However, with our decentralized solution, where nodes are empowered to make their own decisions, these challenges are overcome. In conclusion, we can find the component's WCET by computing its value based on the current CPU load when the preferences list is computed. However, since our focus is to efficiently use the node's available resources, we assume that the latency and WCET are provided by a latency monitoring module as well as a WCET module.

6.2.2 Objectives

There are two objectives in our technical framework by which a node abides depending on the role it plays, i.e., a coordinator or a collaborator. A coordinator node objective did not change, i.e., we try to satisfy the application's *e2e delay* requirement upon deployment on the edge system. However, in this chapter, the *e2e delay* consists of multiple parameters, i.e., the communication latency between components and an overhead latency introduced by the component allocation on edge nodes. The overhead latency is composed of the component's WCET plus the e2e delay of that composite component if the node becomes its coordinator when it cannot host that particular composite component anymore. For example, consider the *authentication* component $\mathsf{c_2} = \{\mathsf{a_1}, \mathsf{a_2}\}$, where after the deployment strategy is mapped on node $\mathsf{E_{p1}}$, having as its configuration $\mathsf{a_2}$; which has the lowest resource requirements and minimal functionality. In this case, to improve the performance of the application the component can be upgraded to a higher functionality level if $\mathsf{E_{p1}}$ becomes its coordinator and finds the required resources in its neighborhood. As a result, the overhead latency for $\mathsf{c_1}$ can be equal to the WCET of the component if the further deployment has failed, or WCET + e2e delay of the further deployment result. In contrast, when the node is a collaborator, the objective is to provide a list of preferences for the advertised application model such that it maximizes (i) the utilization of available resources, (ii) the component coverage, and (iii) the application functionality. All are based on the local status of the collaborator (i.e., available resources and CPU load) and may differ at different points in time.

6.3 Application deployment framework

We identify two major components for our extended decentralized resource management technique, i.e., *the deployment policy module* and *the decision policy module*. The former implements an extension of the decentralized resource allocation technique, presented in Chapter 5, which aims at deploying an application entirely on edge devices. The latter empowers each participant node to take local decisions concerning their available resources and improves upon the previous decision strategies by offering better component coverage and optimized preferences.

6.3.1 Deployment policy module

The deployment policy module retains its functionality proposed in previous chapter, i.e., to find a component allocation on edge nodes that satisfies the application requirements. However, we extend it with more functionality to support our robust application module and update the application functionality at runtime. Therefore, besides the two stages already presented in Section 5.3.1, we enable a third stage that allows the application developer to provide application updates, at runtime. This is possible once the application is deployed and operation on the edge system. In this case, the deployment policy module still keeps track of the edge nodes that hosts application components throughout the application lifespan. As a result, a developer can send new aspects for each composite component to expand the application functionality. When a new aspect is received, the coordinator sends it to the location of its target composite component. Consequently, we can perform on the fly updates of the application without any downtime. Finally, we still use SMT to allow the application coordinator to device satisfiable deployment strategies. Therefore, we translate our component allocation objectives into an SMT formula, using the following encodings: *component facts*, *domain facts*, *latency facts*, *preferences constraints*, *aspect facts*, *aspect constraints*, and *constraint formulation.*

We briefly introduce the first four encodings since these represent the base that is defined in the previous chapter. The *component facts* encoding ensures that the SMT solver maps a component on a single participant node. To be considered as a valid choice, the component must be found in the preference list sent by the target edge node – the coordinator cannot map a component on a node that did not show its desire to host that particular component. Since there is no consensus between participant nodes, multiple nodes may desire to share their resources and host a certain component. Therefore, we must guarantee that only one node is chosen as the host of each composite component. In contrast, the *latency facts* and *domain facts* help to determine the latency between two dependent components – latency that is dependent on the components' host nodes. Be aware that in this encoding we must contain all communication latency between dependent components since the SMT formula aims at providing the solver with complete information about the search space – information that guides the SMT solver to devise a satisfiable deployment strategy that fulfills the application's objectives. Finally, with the *preferences constraints* encoding, we ensure that only one group of components is chosen from the preferences list sent by a collaborator node. As a result, we know that a solution does not exceed the available resources of a collaborator. We continue to present the remaining encodings that represent our contribution to the deployment policy module.

Aspect Facts

The *aspect facts* encoding provides the possible aspects a composite component can have; information important for the e2e delay computation, since each aspect has associated a WCET. Hence, knowing the aspects of each component provides a knowledge base used by the SMT solver to enforce the next two encodings, i.e., *aspect constraints* and *constraint formulation.* We present the encoding in Formula 6.1, where *latency()* returns the overhead latency of aspect a from component c.

$$\mathsf{aspectFacts} = \mathsf{OR}(\mathsf{c_a} = \mathsf{latency(a)}))\,\forall\,\mathsf{c} \in \mathsf{C}. \tag{6.1}$$

Aspect Constraints

With the *aspect constraints* encoding, we help the SMT solver to associate the correct WCET to a component based on its mapping. Similar to the *latency facts*, this constraint helps the solver to create a relation between the mapping of a component and its WCET. As we know, the WCET is closely related to the current node status. Hence, we need to know where the component is mapped to consider its WCET in the *constraint formulation*. Furthermore, since a preference list contains multiple groups of components, we must ensure that we get the correct WCET for a component according to the rules presented in *preferences constraints*. For example, node $\mathsf{E_{p1}}$ sent its preferences $\mathsf{P} = \{\mathsf{p_1}, \mathsf{p_2}\}$, where $\mathsf{p_1} = \{\mathsf{c_1}, \mathsf{c_2}\}$ and $\mathsf{p_2} = \{\mathsf{c_1}\}$. Each component has associated an aspect and its WCET – therefore, if $\mathsf{c_1}$ is mapped on $\mathsf{E_{p1}}$, then we have two possible WCET associated with it (one for $\mathsf{p_1}$ and one for $\mathsf{p_2}$). If $\mathsf{c_1}$ has the same aspect in both $\mathsf{p_1}$ and $\mathsf{p_2}$, then the component's WCET has the value of the aspect. However, if the two have different aspects for $\mathsf{c_1}$, then the solver chooses the aspect for the component that satisfies the *preferences constraints* rules.

Constraint Formulation

Finally, the *constraint formulation* ensures that the found deployment strategy satisfies our objectives. Such encoding accounts for both the communication latency and WCET of a component; the result must be smaller or equal to the application requirements. A rule that helps the solver to verify the satisfiability of a solution found using the aforementioned encodings. We show the encoding in Formula 6.2, where *d* represents the total number of dependencies between two components.

$$\mathsf{e2eDelay} = \sum_{\mathsf{i=1}}^{\mathsf{d}} \mathsf{l_i} + \sum_{\mathsf{c}} {}^{\mathsf{C}}\mathsf{c_a} \leq \mathsf{app\,requirements} \tag{6.2}$$

By combining the aforementioned constraints, we obtain the complete formula $\mathcal{F}$ used by the coordinator to find a satisfiable allocation according to application requirements:

$$\begin{aligned}\mathcal{F} : \mathsf{componentFacts}\ \wedge \mathsf{domainFacts} \wedge\ \mathsf{prefConstraints} \wedge\ \mathsf{e2eConstraint}\\ \wedge \mathsf{aspectFacts}\ \wedge\ \mathsf{aspectConstraints}\ \wedge\ \mathsf{e2eDelay}.\end{aligned} \tag{6.3}$$

6.3.2 Decision policy module

The *decision policy module* has the purpose of aiding the collaborators in creating their preferences based on the information received from the advertised message. The procedure starts once a node receives the advertised message from the coordinator. Based on its current status, the node can become a collaborator if it has the required resources to host at least one component. To aid in its decision, we have developed a strategy to create a list of preferences, i.e., P, for each advertised message .P may contain one or more groups of components and has the following three objectives: (i) maximize the utilization of available resources for each group, (ii) maximize the coverage of components per P, and (iii) maximize the overall application functionality. Finally, once the collaborator creates P, the required resources allocated for P are reserved until the coordinator finds a deployment strategy to avoid any conflicts when multiple applications are deployed at the same time. Furthermore, for each component sent in P, it computes and sends the components' WCET. In the end, the freshly created preferences list is sent to the coordinator.

To obtain a node's preference list that satisfies all objectives aforementioned above, we have developed a new decision strategy capable of finding a feasible or optimal P in the given amount of time. The strategy yields an optimal preference list only if the solution can be found in the available computational time given by the coordinator. If the time has expired, then the strategy produces a feasible solution that guarantees the fulfillment of the constraints. To achieve this behavior, we have decided to use CP [RVBW06] to find our list of preferences for each node. With CP, we can describe a model using *decision variables*, *constraints*, and *global objectives*.

Decision variables

We create our *decision variables* based on the information received in the advertisement message. A decision variable aims at defining for each component a domain representing all the possible aspects the component can have. For example, if $\mathsf{c_1}$ has a set of two different aspects, i.e., $\mathsf{c_1} = \{\mathsf{a_1}, \mathsf{a_2}\}$, then the domain is composed of the two aspects. Under these conditions, the CP solver can assign only one aspect, chosen from the domain, for the decision variable of $\mathsf{c_1}$. Besides the component variables, to be able to maximize the usage of the node's available resources when creating a group of preferred components, we must define three new decision variables to keep track of the resource requirements of a chosen aspect – the domain of a resource decision variable, i.e., RAM, CPU, and HDD, has the same length as the domain of the component. For example, the domain for the RAM resource variable of $\mathsf{c_1}$ is $\{\mathsf{a_{1_{RAM}}}, \mathsf{a_{2_{RAM}}}\}$. Finally, remember that we are interested in considering the aspects with the highest functionality in our deployment strategies. As a result, we define a penalty variable to aid the solver in considering the highest priority aspects. We have assigned a penalty of 0 for the aspects with the highest priority – we increase the penalty by 2 for every other aspect assigned to a component. In the end, if a component is not placed in P, then a penalty of 20 is given. The construction of the *decision variables* has a time complexity of $\mathcal{O}(n_c n_a)$, where $\mathsf{n_c}$ is the total number of components sent in a group and $\mathsf{n_a}$ is the total number of aspects a component has. It is important to mention that the *decision variables* are created for each $\mathsf{p_i}$ found in P.

Constraints

We added all required decision variables to the CP model, so we can start defining *constraints* to guide the CP solver in its quest to find a feasible solution. For this purpose, we have created two major constraints, i.e., (i) *group constraints* and (ii) *max coverage constraints.* With *group constraints*, we ensure that any group of components, i.e., $\mathsf{p_1}$, sent in P, does not exceed the available resources of its node. We divide the *group constraints* into two important parts, i.e., *set resource decision variables* and *check available resources.* The former creates a logical relation between an aspect of a component and its resource requirements (see Formula 6.4). The latter ensures that the sum of all resources required for $\mathsf{p_1}$ does not exceed the available resources of that node (see Formula 6.5). The time complexity required to create the *group constraints* is $\mathcal{O}(n_c)$, while for finding the *max coverage constraints* is $\mathcal{O}(n_c n_{cp})$ where $\mathsf{n_{cp}}$ represents the number of times a component appears in P. For example, if we have $\mathsf{P} = \{\mathsf{p_1}, \mathsf{p_2}\}$ and $\mathsf{c_1}$ is part of both $\mathsf{p_1}$ and $\mathsf{p_2}$, then $\mathsf{n_{cp}}$ for $\mathsf{c_1}$ is *2*.

$$\begin{aligned} \mathsf{setResource} : (\mathsf{c} = \mathsf{a}) \Rightarrow (\mathsf{c_{CPU}} = \mathsf{a_{CPU}} \wedge \mathsf{c_{RAM}} = \mathsf{a_{RAM}} \ \wedge \ \mathsf{c_{HDD}} = \mathsf{a_{HDD}}) \\ \forall\, \mathsf{c} \in \mathsf{C}\, \mathsf{and}\, \forall\, \mathsf{a} \in \mathsf{c_a}. \end{aligned} \tag{6.4}$$

$$\begin{aligned} \mathsf{maxCPU} &= \sum_{\mathsf{i=1}}^{\mathsf{n_c}} \mathsf{c_{CPU}} \leq \mathsf{availabe_{CPU}} \\ \mathsf{maxRAM} &= \sum_{\mathsf{i=1}}^{\mathsf{n_c}} \mathsf{c_{RAM}} \leq \mathsf{availabe_{RAM}} \\ \mathsf{maxHDD} &= \sum_{\mathsf{i=1}}^{\mathsf{n_c}} \mathsf{c_{HDD}} \leq \mathsf{availabe_{HDD}} \end{aligned} \tag{6.5}$$

With the *max coverage constraint*, we want to maximize the component coverage in P. Remember that component coverage is one of the main issues that we must solve when giving nodes the possibility to decide how to share their resources. In Chapter 5, we use four different tactics to account for this problem. In contrast, in this chapter, we solve this issue by defining the *max coverage constraint.* To achieve this, we verify that across all groups of components sent in P at least once each component appears (we call this a *strict preference*). A *strict preference* achieves *100 %* component coverage; however, the CP solver will consider all other preference lists, with a lower component coverage, as infeasible. Enforcing this constraint has different side effects, i.e., we might lower the components' functionality to satisfy this constraint. However, we can make the constraint soft if we change the > 0 to ≥ 0 in Formula 6.6 and obtain what we call a *permissive preference.* In this case, a *permissive preference* considers feasible all solutions independent of the component coverage. We present the constraint in Formula 6.6, where n_p is the total number of groups of components sent in P and *priority()* returns the priority of the aspect assigned to component *c*.

$$\mathsf{maxCoverage} = \sum_{\mathsf{i=1}}^{\mathsf{n_p}} \mathsf{priority(c)} > 0\, \forall\, \mathsf{c} \in \mathsf{C}. \tag{6.6}$$

Global objective

Finally, we define a *global objective* to enable our decision policy module to yield the best solution under the given time. The purpose of this objective is to minimize the penalty at the node's preference list level. As a result, we obtain a preference list that prioritizes the higher priority aspects for each component while maximizing the utilization of all available resources. We present the global objective in Formula 6.7.

$$Min(\sum_{\mathsf{i=1}}^{\mathsf{n_p}} \mathsf{penalty}) \tag{6.7}$$

6.4 Evaluation

A major goal of the extended decentralized resource management framework is to efficiently utilize the available computational resources found on resource-constrained devices when deploying a latency-sensitive IoT application on an edge system. To this end, we evaluate our proposed application model in terms of obtained successful mappings at the edge of the network. We first evaluate the decision policy module to understand the effects of various applications and nodes available resources on (i) the total number of component groups sent in a preference list and (ii) computational time required to find a feasible solution. Finally, we proceed to deploy three different IoT applications, (i) *montage*, a real-world DAG workflow [BSM10], (ii) a cognitive application [ASDL19a], and (iii) a mockup application defined by us – each application has a different size to better assess the framework's capabilities.

6.4.1 Decision policy module: Experiment and Results

We first evaluate the decision policy module to demonstrate the collaborator's ability to provide feasible and optimal preferences. The purpose of this evaluation is to understand how the following impact component coverage: (i) edge node available resources, (ii) application size (i.e., the total number of composite components), (iii) available search time (i.e., the maximum time allowed to find the node's preferences), and (iv) number of groups sent in P. Furthermore, we evaluate both strict and permissive preferences by making the *max coverage constraint* a hard or a soft constraint. Depending on the type of the *max coverage constraint*, the maximum number of groups sent in P plays an important role in the overall performance of the deployment technique – the number of groups may increase or decrease the framework performance in finding feasible deployment strategies. On the one hand, a strict constraint requires a minimum number of groups to find a feasible P; the minimum number of groups is dependent on the node available resources and application size. On the other hand, a permissive constraint requires more computational time and available resources to yield a P with a component coverage close to 100%. To evaluate the decision module, we have implemented the bid strategy using *CP-SAT solver* provided by Google OR-Tools [2]. We perform our measurements by deploying the collaborator on a machine with a dual-core Intel i5 2.3GHz processor.

In our experiments, we choose to find the node's preferences when the application coordinator advertises the *montage* application – an application with the highest number of components from the three we consider, making it harder for the decision strategy to build a preference list. During our experiments, we send an advertising message

[2]https://developers.google.com/optimization

containing the application model to the collaborator node and record its preferences. We repeat the experiment multiple times, every time changing a set of different metrics, i.e., available search time (i.e., 1, 5, 15, 30, 45, and 60 seconds), number of groups sent in a preference list (i.e., 2, 4, 6, and 8), node available resources (12, 36, and 60 units for each considered resource), and max coverage constraint type (permissive or strict). We proceed with our evaluation as follows: we set the available resources of a node and record the component coverage changing the search time and the number of groups sent in P. For example, for every node's available resources set, we record a total of 48 preference lists, i.e., for each number of groups sent in P, we change the available search time resulting in 6 total preferences list per group size. Furthermore, we perform this approach twice, once when we use the strict preference constraint and the second one when we opt for a more permissive constraint. We present our results in Table 6.1 to Table 6.3 where we show the component coverage considering the number of groups sent in P and the available search time. For example, in Table 6.1, the decision policy module can find a preference list with component coverage of 25 % when we set the constraint type to permissive, we allow for a maximum of two groups in P, and we limit the search time to 1 second.

Constraint Type	# groups	search time (seconds)					
		1	5	15	30	45	60
Permissive	2	25	25	25	30	29	29
	4	46	36	46	50	50	50
	6	55	58	50	50	42	42
	8	63	67	63	63	58	58
Strict	2–4	0	0	0	0	0	0
	6–8	100	100	100	100	100	100

Table 6.1: Component coverage percentage of permissive and strict constraints with 12 units available resources

In Table 6.1, we can observe that the node's available resources and the number of groups prevent the decision strategy to find a feasible P – it is impossible to find the node preferences when the number of groups is 2 and 4. A consequence of making the *max coverage constraint* strict – a constraint that considers that P is feasible only when its component coverage is *100 %*. In comparison, we can see that by setting this constraint to permissive, the decision strategy finds a preference list for every group size – having a component coverage between *0 %* and *100 %*. In this case, the only chance to not find a feasible P is when the collaborator lacks the available resources required to host at least one component from the application.

In Table 6.2 and 6.3, we can observe the impact of the node's available resources on the decision strategy's capability to find feasible solutions. We can conclude that by increasing the node's available resources, it is possible to find feasible solutions for all groups if we have a strict *max coverage constraint*. Similarly, for permissive constraints,

Constraint Type	# groups	search time (seconds)					
		1	5	15	30	45	60
Permissive	2	58	63	50	50	58	58
	4	63	58	71	71	71	66
	6	66	66	83	79	79	79
	8	86	88	100	100	100	83
Strict	2–4–6–8	100	100	100	100	100	100

Table 6.2: Component coverage percentage of permissive and strict constraints with 36 units available resources

we can observe an improvement in the overall component coverage across all solutions. Furthermore, by increasing the search time, we can achieve *100 %* component coverage for permissive constraints as well.

Constraint Type	# groups	search time (seconds)					
		1	5	15	30	45	60
Permissive	2	71	71	75	71	71	71
	4	75	88	92	92	92	92
	6	92	92	100	92	92	96
	8	83	83	83	83	96	100
Strict	2–4–6–8	100	100	100	100	100	100

Table 6.3: Component coverage percentage of permissive and strict constraints with 60 units available resources

6.4.2 Deployment policy module: Experiments and Results

To evaluate the deployment policy module, we consider as a performance metric the deployment strategy's ability to find a satisfiable mapping by using only the edge nodes' available resources – without using the available resources found in the cloud. To do so, we compare our robust application model and the DAG model by looking at their impact on the deployment strategy – we compare the deployment efficiency of both cases, i.e., when we set the *max coverage constraint* to strict or permissive. It is important to understand the impact of a strict constraint on the deployment strategy, considering that some nodes may not be able to send their preferences because there are not enough resources to offer maximum component coverage or the number of groups is too small. We simulate the deployment of the three aforementioned IoT applications on an edge system, evaluating both the applicability and performance. The implementation of our coordinator is based on *Z3* SMT solver [DMB08] and is deployed on the same device as described in Section 6.4.1 – we simulate all collaborators used for the performance evaluation on the same device.

Applicability: Cognitive Application Deployment

To prove the applicability of our decentralized resource management framework, we deploy a realistic cognitive application consisting of 8 components. Based on the results presented in Section 6.4.1, for our applicability evaluation, we have decided to set the decision policy module configuration as follows: (i) the available resources on each collaborator node is randomly set between *12 and 24 units*, (ii) a total number of two groups of components can be sent in the node's preference list, (iii) the search time is *1 sec*, (iv) we use strict constraints, aiming to have maximum component coverage in each P, and (v) the e2e delay is set to *50*.

We model the *cognitive application* [ASDL19a] using our proposed robust model, assigning each composite component a set of four different aspects. Each aspect has a set of resource requirements chosen in the range of 1 to 9 units; the lowest priority aspect has resource requirements closer to 1, while the highest priority aspect has resources close to 9. For evaluation purposes, we choose the WCET of an aspect randomly between 1 and 10. Our target edge system consists of two resource-constrained devices, that can be both coordinators and collaborators for the deployed applications – $\mathsf{E_{p1}}$ has the following available resource, $\mathsf{E_{res}}$ = {RAM = 19, CPU = 21, HDD = 18}, while $\mathsf{E_{p2}}$ has $\mathsf{E_{res}}$ = {RAM = 21, CPU = 22, HDD = 15}. In Figure 6.6, we present the cognitive application robust model and the application mapping, where in the left corner we see the host node and on the right corner we see the chosen aspect for that composite component. For example, edge node $\mathsf{E_{p1}}$ hosts the composite component, i.e., *vision capture*, with the aspect $\mathsf{a_4}$.

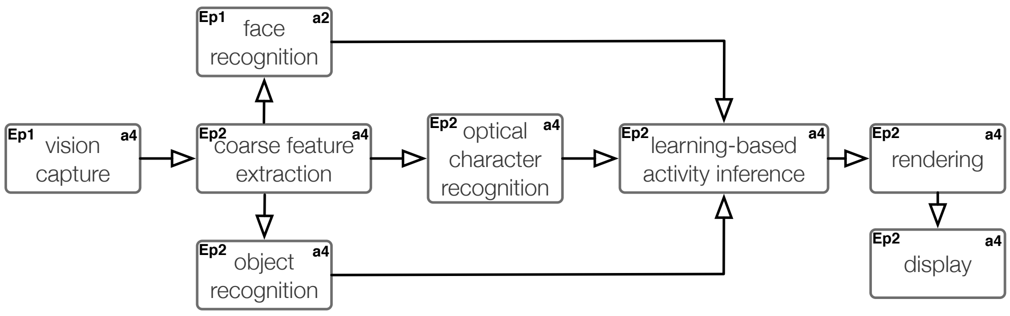

Ep1 face recognition a2
Ep1 vision capture a4
Ep2 coarse feature extraction a4
Ep2 optical character recognition a4
Ep2 learning-based activity inference a4
Ep2 rendering a4
Ep2 object recognition a4
Ep2 display a4

Figure 6.6: Cognitive Application model [ASDL19a]

The coordinator manages to find a satisfiable mapping in *40 ms*, which includes the time required to find a mapping after the node's preferences were received. To this time we add the time required to find the node's preference list, i.e., *1 sec*. The overall application's *e2e delay* is equal to 49 ms, from which 12 ms represents the communication latency and the remaining 37 ms comes from the components' WCET. To have a better picture of the current component's functionality, we present in Table 6.4 the current component's resource requirements based on their chosen aspect and related WCET.

Composite Components	Resource Requirements			WCET
	RAM	CPU	HDD	
c_1	3	1	1	1
c_2	3	2	4	7
c_3	4	5	5	1
c_4	3	1	2	7
c_5	3	1	3	3
c_6	3	1	1	9
c_7	3	1	3	6
c_8	3	1	1	3

Table 6.4: Cognitive Application resource requirements and WCET.

Performance evaluation

To quantitatively evaluate the impact of the proposed robust IoT application model and the new decision policy module, based on the results presented in Section 6.4.1, we choose the following configuration for our decision policy module: (i) we randomly select the nodes' available resources between 26 and 50 units, (ii) we set the maximum number of groups sent in the preference list to 6, (iii) we keep the search time equal to *1 sec*, and (iv) we set the application's e2e delay to a high value to avoid losing any deployment due to a demanding e2e delay. Furthermore, we deploy two different applications, i.e., the *montage graph* with a total number of 24 components and the *mockup application* having 16 components. Deploying different application models allows us to evaluate the impact of total available resources found in the target edge system and the application size on the deployment efficiency. Similar to the cognitive application model, we assign to each composite component a set of four distinct aspects. However, for our edge system, we consider five different sizes, i.e., 2, 4, 6, 8 and 10 collaborators.

We present our evaluation results in Table 6.5 where we compare three different deployments based on the total number of successful application mappings on edge devices – without resorting to cloud resources. In the first deployment, we aim to achieve 100% component coverage in each node's preference list sent to the coordinator – we set the *max coverage constraint* to strict. In the second deployment, we change to constraint type to permissive. It is important to mention that, for the first two deployments, we use the proposed robust IoT application model to define the applications. Finally, to show the improvements over the normal DAG application model used in Chapter 5, we add the deployment results presented in Section 5.4.2 for comparison purposes – in this case, the DAG application model does not have aspects.

6.4.3 Discussion

In Section 6.4.1, we have demonstrated that our decision policy module enables a collaborator to provide feasible preferences under strict and permissive constraints. In the former case, finding the node's preferences is dependent on the number of groups sent

Application	Type	Number of nodes				
		2	4	6	8	10
Montage	strict	0	81	100	100	92
	permissive	0	0	8	92	97
	DAG model	0	0	0	0	0
Mockup	strict	44	100	100	100	98
	permissive	0	6	100	100	99
	DAG model	0	0	0	0	0

Table 6.5: Successful mapping on edge.

in P as well as the node's local available resources. We can observe this in Tables 6.1, 6.2, and 6.3, where with an incremental increase in both metrics we achieve 100% component coverage. Considering these findings, we can conclude that a node with 36 units of available resources represents the optimal node available resources suitable to deploy the montage application. In contrast, a permissive bid is more flexible, allowing collaborators to submit their preferences even when the component coverage is not close to maximum coverage – it finds a feasible preference list for all configurations. We can conclude, from the three tables, that the component coverage increases with the available resources of a node. Furthermore, compared to a strict constraint, where the objective is to maximize coverage, a permissive bid aims at providing better application functionality, i.e., chooses the highest priority aspect for each component. As a result, we see a fluctuation in component coverage given by the available search time.

In Section 6.4.2, we show the coordinator's ability to successfully deploy an application to resource-constrained devices. With the aid of our experiments, we prove that the deployment policy module can find a satisfiable deployment strategy for a realistic IoT application, even when there are not enough available resources to host the entire application at maximum functionality. The coordinator manages to find a satisfiable deployment strategy for our cognitive application by keeping its functionality close to the minimum. By choosing the strict constraints, which achieves 100% component coverage, the decision module does not maximize the application's functionality. Additionally, we set the number of groups to 2, which further limits the possibility of increasing the application functionality. However, the host node can improve the application functionality when more resources become available. In conclusion, we demonstrate that our proposed robust application model in combination with an efficient decision policy module can achieve better results compared to the decentralized resource management framework presented in Chapter 5.

From Table 6.5, we can observe that even for a large application size, we can provide full edge deployments for more than *80%* of total tests if the target edge system has at least the available resources required to host the application with the lowest functionality. In this case, each component uses the lowest priority aspect. Considering this, we do not find a satisfiable deployment on the system composed of two collaborators because it lacks the application's minimum required resources. In contrast, by increasing the number of collaborators, we considerably increase the chances of finding feasible solutions. From the deployment of the two applications, i.e., mockup and montage, we can conclude that deploying an application with a smaller number of components on the same edge system will result in the deployment strategy performing better. For example, on an edge system with 10 nodes, our framework can find deployment strategies for both permissive and strict types. Moreover, the deployment using bids with 100% component coverage yields much better results for a lower number of nodes, compared to the other two deployments. In conclusion, we can observe that by deploying an application using our robust model we can achieve far better results compared with the DAG application model. When we use the DAG model, our framework cannot find one successful mapping exclusively at the edge. As proven in Chapter 5, we can deploy an application using the DAG model in edge systems with more nodes.

By comparing the different deployments, i.e., permissive, strict, and DAG model, we notice that deployments using strict constraint outperform the others on smaller system sizes while the deployment using a DAG model cannot find one satisfiable solution. Furthermore, when deploying our *mockup* application model, we can see that our framework can find feasible deployment strategies on a small edge system size – we can see that by lowering the size of the application, we lower the required available resources, enabling the coordinator to find a mapping on a system with a smaller size. In conclusion, making each node provide full component coverage enables the coordinator to fully utilize the available resource found on edge nodes.

As a final note, we acknowledge that the current evaluations were performed on a laptop and not on a resource-constrained device (e.g., a Raspberry Pi). Even though the computation times for creating a node preference list may increase, the evaluation performed is still valid. One limitation of our deployment technical framework is the inability to automatically decide at runtime, based on the application size and the available resources, what is the optimal number of groups sent in P by a node. In this case, we had to perform multiple evaluations for all deployed applications, having the number of groups hardcoded into the decision policy module. To solve this problem, we intend, in future work, to develop such a method that decides at runtime the optimal performance list size considering the deployed application.

6.5 Conclusion

In this chapter, we substantially extend our previous work with a new decision policy module and provide a novel robust IoT application model. The former consists of developing a decision strategy that empowers collaborator nodes to make better decisions regarding their available resources – a policy that aims at providing feasible and optimized node's preferences. The latter enables the developer to define different application functionality levels. An approach that (i) enables the resource management technique to efficiently use the available resources found on edge devices, (ii) allows updating the application functionality at runtime, and (iii) empowers each collaborator to adapt the component's functionality based on the available resources found locally or in the proximity of its host node. To conclude, we proved that significant improvements are seen when deploying an IoT application defined using our proposed model. Moreover, we demonstrate that finding preferences that achieve 100% component coverage yields better results compared to permissive preferences, when the target edge system has limited available resources.

CHAPTER 7

Adaptive Management of Volatile Edge Systems at Runtime With Satisfiability

Recent developments in DevOps and software design advocate dividing applications' functionality into small, modular, and easily deployable microservices. We define the overall application's functionality in terms of an invocation sequence among those microservices. An IoT application, as a composition of those microservices, may need to adhere to specific requirements. Fundamental ones, the importance of which is exacerbated on edge computing settings, are exhibiting a certain availability and enjoying low latency. During the application's lifespan, deployment should take such requirements into account, something that is nontrivial and requires novel resource management techniques to deploy the application and maintain its correct functionality at runtime.

Resource management in edge computing has considered various aspects, resource placement, migration, and discovery among others [TN18b]. In previous chapters, we have proposed techniques (i) to develop the IoT application and find deployment strategies at design time (Chapter 4), (ii) to seamlessly deploy applications, at runtime, on the target edge system such that the application's requirements are satisfied – we focus on efficiently using all the distributed nodes' available resources (Chapter 5), and (iii) to improve the utilization of resources found at the edge of the network, we propose a new IoT application model and further improve our decentralized resource management technique (Chapter 6). However, in all situations, we only perform resource placement, i.e., we find an allocation of microservices on the target edge nodes before the application becomes operational – a practice that does not ensure the correct application's functionality during its lifespan. Therefore, in this chapter, we focus on combining *resource placement* with *resource migration*, in order to deploy and host an IoT application on an edge system satisfying its latency and availability requirements. Existing approaches to tackling availability have been based on scaling up resources in pools/clusters of nodes hosting microservices, such as OpenFaas [1], Kubernetes [2], Docker Swarm [3], etc. In our case, we are not concerned with scalability or elasticity – but with systems where edge nodes hosting microservices may fail or leave the network. Thus, the systems we target for deployment are highly volatile, raising challenges on the dependable execution of IoT applications spanning multiple hosting nodes. To this end, we tackle the problems of volatility and distribution of resources in edge systems, where typically microservices located on various hosting nodes need to be executed in some specific sequence. Nodes may fail, and the application's invocation path may need to change but still satisfy the latency and availability requirements. We interpret the above problem as two separate goals in the systems we target; (i) placing appropriately necessary resources by deploying microservices to nodes, (ii) and reacting to instabilities caused by node failure.

During the *microservice placement stage*, we place microservices on edge nodes such that the applications' resource requirements are met. This stage considers the desired availability of the application along with known failure probabilities of individual hosting nodes. We pursue availability by replicating microservices accordingly throughout the system. This is a costly step, as it involves migrating and moving around microservices – as such it should be performed as infrequently as possible. During the *invocation path stage*, we ensure the correct application's functionality by finding an invocation path among nodes hosting microservices such that it satisfies the application's requirements. We deploy the application in a volatile edge system, where nodes may fail or leave the system at any point in time – a change in the system may disrupt the current application's invocation path among microservices. Therefore, we must find a new invocation path between the remaining available microservices to adhere again to the application's availability requirements and the e2e latency. Observe that, during the placement stage, we do not have as objective the application latency – a practice that solves the computational issues faced by the decentralized resource management framework when deploying an application. Considering the application's latency as an objective in the *invocation path stage* lowers the high computational demands of modeling the communication latency between dependent microservices. During this stage, we consider only a small part of the edge system, i.e., only the edge nodes where the application's microservices and their replicas reside.

Our framework builds on the premise that an application model consists of microservices that should be invoked in some specific sequence to fulfill the desired functionality. A set of requirements define the application model, i.e., the microservices' resource requirements, the maximum e2e latency that an invocation path should exhibit across the system, and the desired application and microservices availability. Based on these, the proposed framework deploys and maintains the application on the edge system. We consider that the target edge system consists of geo-distributed edge devices. However, we are not in a

[1]https://docs.openfaas.com/
[2]https://kubernetes.io/docs/home/
[3]https://docs.docker.com/engine/swarm/

static setting similar to Mobile Edge Computing, where edge servers reside at certain locations. Instead, we consider volatile edge systems where service operators have no control over nodes joining/leaving the system, which end-users operate. In Figure 7.1, we illustrate the problem setting treated in this chapter. We define the edge system as (i) microservices hosted on nodes (ranging from single-board computers to server-class data center hosts) forming a network, (ii) every node is network-reachable from any other node, (iii) microservice instances are replicated to ensure availability, (iv) nodes have high failure probability, and (v) several invocation paths among microservices hosted are possible, each characterized by an e2e latency.

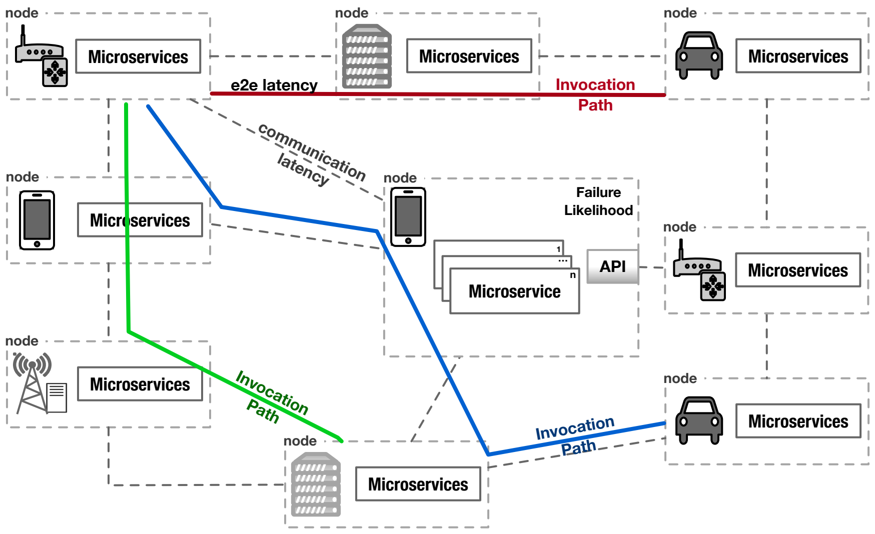


Figure 7.1: Overview of a volatile edge system illustrating (i) interconnected edge nodes featuring different communication latencies, (ii) microservices being replicated across nodes, (iii) multiple invocation paths characterized by different e2e latencies.

Similar to our previous resource management framework, we encode the two problems encountered – application deployment and runtime management – within SMT [BT18]. Thus, we provide guarantees – if a mapping exists, it is always found at runtime by an SMT solver situated in some edge node and is always correct as it satisfies application availability and latency requirements. Those two problems correspond to two Monitoring-Analysis-Planning-Execution (MAPE) loops [KC03]. Finally, we evaluate our framework's performance by measuring the execution time required to deploy and maintain an application on an edge system as well as investigate its recovery from emergence of volatility.

The remainder of the chapter is structured as follows. In Section 7.1 we present the overview of our proposed solution and introduce a motivational example. Section 7.2 defines the application and network considered in this chapter. In Section 7.4 we describe the implementation details of our resource provisioning technique, while in Section 7.5 we describe our stabilization technique. Section 7.6 presents the methodology and results of our evaluation regarding both provision and stabilization. Finally, Section 7.7 concludes the chapter.

7.1 Framework for Adaptive Management of Volatile Edge Systems

Edge-intensive systems typically heavily take into account runtime aspects, as unforeseen as well as emergent system behaviors might hinder system stability; computational nodes hosting microservices may fail or leave the system at any point – as such, deployment decisions previously made may be rendered obsolete. Therefore, we cast our proposal within self-adaptive systems – we monitor the runtime state, construct a precise representation amenable to analysis, and devise potential counteractions in a self-adaptive manner. In essence, we tackle the problems of volatility and distribution of resources in edge systems – microservices need to be executed in some way depending on resources that are located in nodes that may fail. Microservices require specific resources, and "some way" refers to an invocation path among them that needs to satisfy some desired application property. Within our approach, we use two MAPE [AZ10] cycles to address the system goal, i.e., the placement stage and the invocation path stage, where both take place at system runtime.

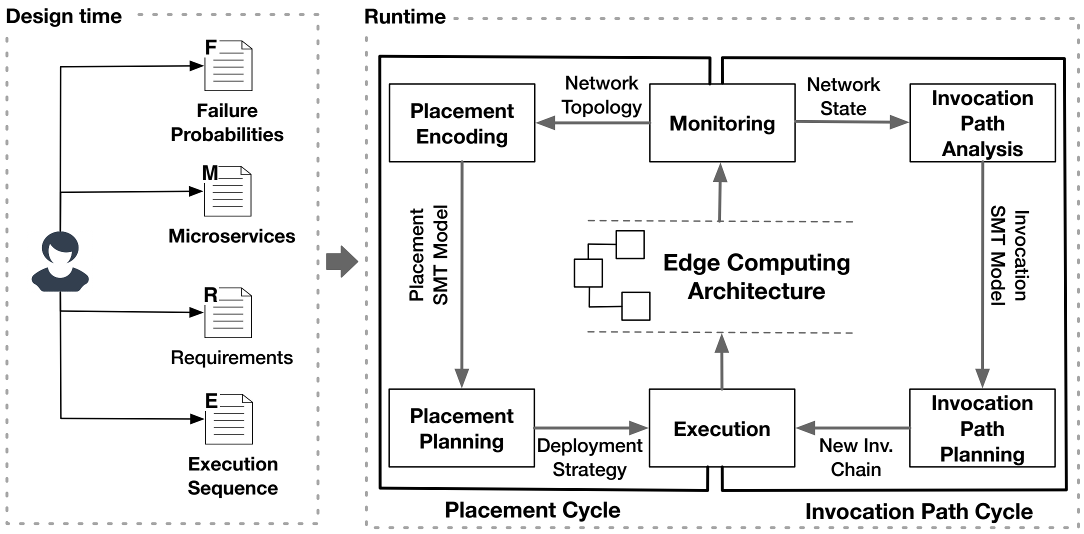


Figure 7.2: Adaptive framework overview featuring two adaptive cycles at runtime in charge of placing the application and managing the invocation path across microservices respectively.

The *placement cycle* finds a mapping of microservices on the participating edge nodes and ensures the fulfillment of applications resource requirements and objectives. To achieve this, this cycle monitors the state of the system, namely which nodes are reachable, available, and what resources they currently have, through the Monitoring activity. Subsequently, the Placement Encoding activity considers some desired availability factor of the application and known failure probabilities of nodes and models the problem in a representation amenable to analysis. In this chapter, we assume that the system engineer knows the node's failure probability at design time – nodes failure probabilities may be also learned, at runtime, something which we identify as an interesting avenue of future work. Based on the developed model, the Placement Planning activity produces a *plan* by replicating microservices accordingly throughout the system, ensuring that the overall application availability is satisfied. Finally, the plan is executed upon the edge system by actual allocation of microservices upon nodes. Note that the adaptive framework uses the *placement cycle* in two different situations, i.e., initially and sporadically. In the former, the cycle maps the required microservices on the edge system before the application is operational on the system. In the latter, the framework triggers the placement cycle when radical changes in the system appear – there are not enough available microservices left in the system to ensure application recovery by using the invocation path cycle alone.

The *invocation path cycle* ensures the correct application's functionality by finding a sequence of nodes where its microservices reside, within the system, that satisfies all objectives, i.e., e2e latency and availability requirements. As the system is assumed to be unstable, the cycle uses the Monitoring Activity to identify any system changes that may disrupt an existing invocation path. If the system changes disrupt the application's invocation path, then the Invocation Path Planning activity must find a new invocation path and keep the application operational. Recall that nodes host replicas of various microservices, but nodes themselves have communication links of various qualities between them, which the Monitoring activity monitors. In contrast to the *placement cycle*, the *invocation cycle* produces a requirements-satisfiable invocation path among available microservices with every change in the system state, targeting non-radical system state changes, when it merely needs to be *stabilized.* The difference among the two adaptive behaviors – targeting infrequent and radical change versus minor instability in invocation paths – is exploited in the design of the framework's runtime behavior, as described in the following sections.

Running Example

Consider a public-safety application deployed in a smart city scenario and within the circumstances of a special event, such as a festival taking place within the city. Usually, large crowds of people attend these events, leading to an increase in different types of applications deployed in the area. Since this is a special event, the sudden request for hosting multiple applications on the current infrastructure renders it inadequate to meet each application's requirements. Furthermore, building the required computing infrastructure is not efficient since the event is only for a limited amount of time – rendering the extra infrastructure idle outside of the event window, leading to resource waste. However, there are many unused resources at the event owned by the participants, e.g., within smartphones and cars among others. Hence, since these are connected to the same system, they can be used for the benefit of deployed applications. Note how an application operator does not control the users' devices – devices may leave the system at any time.

In our example, the public-safety application must be operational, in the system, for the duration of the entire event. The application provides a safe environment by analyzing the surroundings to find suspicious cases such as an unattended package or a dangerous activity in the crowd. Thus, once the data is analyzed, the application helps the authorities to prevent dangerous scenarios. As we can imagine, response time is critical in such situations. As a result, the application cannot reside entirely in the centralized cloud due to requirements as low latency and high availability – requirements that we can fulfill by deploying the application closer to the event location.

Our adaptive framework fits particularly well in this decentralized scenario. Using our proposed framework, we can maintain the edge system in a stable state throughout the entire event – we can adapt to changes found in the system and network. With the *placement cycle* we can distribute the application's microservices throughout the edge system, while with the *invocation path cycle*, we can establish an initial application's invocation path and provide the possibility to recover from node failure. To have a better understanding of our framework, we present a step-by-step process of managing the public-safety application on the target edge system. The application consists of different microservices like (i) *face recognition*, (ii) *environment analysis*, and (iii) *data analysis* and has a given microservice execution sequence (see Figure 7.3). The application has two overall requirements, i.e., a maximum e2e latency and desired availability. The target system consists of multiple nodes ranging from powerful server nodes (e.g., in the event premises) to mobile nodes of users. Each node has an associated failure probability and communication link – each communication link has a latency associated, assumed to be known beforehand. Our purpose is to satisfy all application's requirements during its lifespan at runtime.

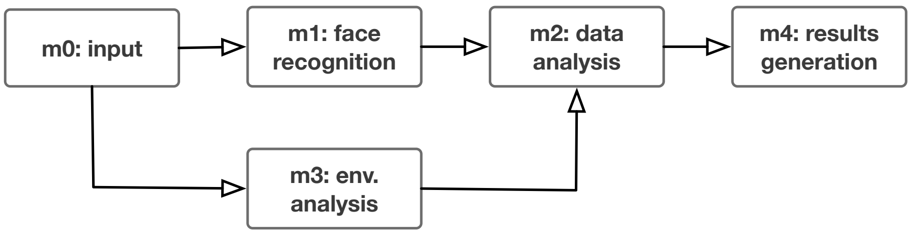


Figure 7.3: Public-safety application model and its execution sequence.

We distinguish three characteristic stages that an edge system may be found in during the lifespan of an application. Those represent the initial placement and start of the application's execution, the emergence of some disruption due to node failure and then the stable system state after the corrective action employed by our approach. Those are illustrated in Figure 7.4; specifically:

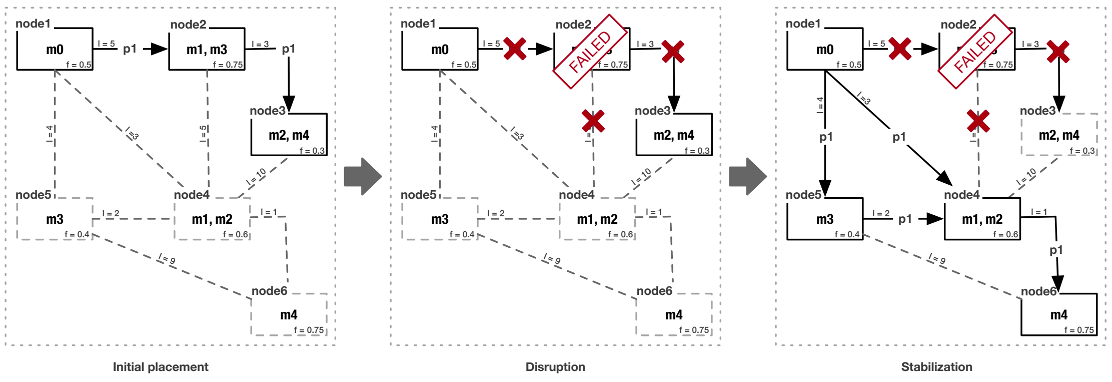


Figure 7.4: Three distinct stages in which the edge system may be found: (i) *initial placement* – the application becomes operational, (ii) *disruption* – a node failure disrupts correct functionality, and (iii) *stabilization* – the system adapts to changes by finding a new invocation path.

Initial placement

During the initial placement, the framework triggers the *placement cycle* to find an allocation of the application's microservices on edge nodes accordingly (Figure 7.4). This includes replication of certain microservices across nodes, to account for possible future node failures. Notice that the application is not operational after the invocation of the *placement cycle*. Therefore, the framework uses the *invocation path cycle* to find a satisfiable invocation path for the application, among the already allocated microservices, such that it satisfies the application's requirements. The *invocation path cycle* chooses one invocation path from many possible paths, formed between all available microservices. Using both cycles in a sequence, we ensure that the application function as intended – therefore, we can observe that we achieved a *stable edge system* where all nodes and placed microservices are operational and an invocation path is derived. In Figure 7.4 the dashed line represents the communication path between two nodes, while f shows the node's failure probability, while l is the latency associated with a certain communication path. Finally, *p1* represents the invocation path chosen by the *invocation path cycle*.

Disruption

During the application's lifespan, a disruption occurs due to the failure of one or more nodes (Figure 7.4) resulting in an unresponsive application – all microservices that were hosted by the failed nodes cannot be reached. For example, let us assume that during the execution of our public-safety application *edge node 2* has failed – the participant owning the device has left the event. Note that on this node, two microservices m_1 and m_3 reside – microservices that were part of the application's invocation path, i.e., *p1*. Under these conditions, the previous invocation path is no longer viable and a new one must be devised. The new invocation path makes use of the replicated microservices across the system, guaranteeing that the latency and availability desired are maintained.

Stabilization

After the new path is calculated, the application's operation is restored (Figure 7.4). Note how this new stable state may be fragile; if more nodes fail, the invocation path planning activity may fail. If an invocation path cannot be calculated (e.g., because multiple nodes hosting critical microservices disappeared), the system is unable to return to a stable state and a placement must be triggered again. With the placement cycle, we populated the system with new microservices to enable the application to be operational again.

7.2 Application and System Models

In this section, we outline the application and system models for which we created our framework. We start by defining the target edge system model and its components, i.e., the edge node. Next, we present the application model along with its defining components, such as microservices and execution sequence. Finally, we introduce the framework objectives, i.e., the application's latency and availability.

Edge System Model

In our target edge system, we consider that there is a central node (i.e., a coordinator node) that governs over all other nodes. Let $\mathsf{E_N}$ =$\{\mathsf{E_1}, \mathsf{E_2}, \ldots\}$ represent the total number of devices found under the supervision of a coordinator node. Be aware that in contrast to edge nodes that may have limited computational resources, the coordinator node is a powerful device, like an edge server, and represents the location where our adaptive framework resides. Furthermore, the coordinator does not host microservices, its purpose being only to manage the application on the supervised nodes. An edge node may have different capabilities ranging from resource-constrained devices like a smartphone or Raspberry Pi to single-board computers and server-class data center hosts. Each edge node has a *failure probability* representing the probability of a node to leave the network or fail. The failure probability can be estimated at runtime with the help of a failure probability monitoring module – a module that can use historic failure data of each type of edge device to predict the failure probability of a new node [YZW21]. Furthermore, a set of available resources, $\mathsf{E_{res}}$=$\{\mathsf{r_1}, \mathsf{r_2}, \ldots\}$ denotes the node's current capabilities. In addition to the computational resources like RAM, CPU, and HDD, a node may have locally stored different running microservices; microservices that a deployed application may require and must be discovered with the help of a resource discovery technique [MATD19]. For this chapter, however, we assume that there are no microservices already deployed in the system, helping us to consider the worst-case scenario, i.e., deploying all microservices required to fulfill the application's requirements.

Application model

In this chapter, we follow the IoT application model presented in Chapter 1, i.e., an application model follows a microservice-based architecture where its functionality is divided into multiple linked microservices, each being characterized by a set of requirements. We define an application model by a set of three different characteristics: (i) a set of microservices $\mathsf{M_N} = \{\mathsf{m_1}, \mathsf{m_2}, ...\}$, (ii) an execution sequence, and (iii) a set of functional and resource requirements. The functional requirements represent the overall requirements of the application, e.g., latency and availability. In contrast, the resource requirements represent the computational resources each microservice needs to achieve its purpose on an edge node. Similar to the definition of a microservice presented in Chapter 5, $\mathsf{m_i}$ is defined, at design-time, by a set of resource requirements, $\mathsf{M_{res}} = \{\mathsf{res_1}, \mathsf{res_2}, \mathsf{res_3}\}$. The set of requirements associated with a microservice consists of CPU, RAM, and HDD and ensures proper execution on an edge node. Specific to the adaptive framework, each microservice $\mathsf{m_i}$ can have multiple replicas, i.e., $\mathsf{Rp} = \{\mathsf{rp_1}, \mathsf{rp_2}, ...\}$, that guarantee the fulfilment of the availability requirements. The number of replicas required for a microservice is found during the *placement cycle* and is dependent on the application's availability requirements and the node's failure probability. By replicating microservices on the edge system, we increase the application's resilience to disruptions, enabling the coordinator to stabilize the application in the event of node failure. In the end, the application's functionality is defined by its associated execution sequence – a sequence that represents the order in which each microservice is invoked. Keeping the same format of the application model as presented in Chapters 5 and 6, the application model has one execution sequence – a sequence that starts with a source (i.e., the starting microservice was chosen by the developer) and ends with a sink (i.e., an actuator microservice that consumes all processed data). The *invocation path cycle* uses the execution sequence to find a new application's invocation path at runtime.

Objectives

Since we focus on latency-sensitive applications, the communication latency between two dependent microservices plays an important role as the total e2e latency of the application's invocation path is a key requirement. This is the same e2e latency we used as objective in Chapter 5, i.e., a communication link between two edge nodes, $\mathsf{E_i}$ and $\mathsf{E_j}$, has an associated communication latency $\mathsf{l_{E_i,E_j}}$. We measure the e2e latency as the duration of time required to execute the application's invocation path; starting from the source microservice and ending with the sink microservice. Note that we do not consider the WCET of microservices. We assume that the communication latency is monitored at runtime (i.e., through the Monitoring activities of the MAPE loop of Figure 7.2). The communication latency of a node with another is inherited by all their hosted microservices, $\mathsf{m_i}$ and $\mathsf{m_j}$.

A second objective that was not considered in the decentralized resource management framework, but it is very important when managing applications in an edge system, is *availability*. Similar to the communication latency, the microservice's availability represents a characteristic that is inherited from its location. In this case, each microservice has an availability that is a reflection of the failure probability of its host node. Let us consider that $\mathsf{m_i}$ is placed on $\mathsf{E_i}$. We know that $\mathsf{E_i}$ has a failure probability of 0.6%, resulting in an availability of 1 - 0.6% = 0.4% for component $\mathsf{c_i}$. All components residing on $\mathsf{E_i}$ will have the same availability.

7.3 Monitoring and Execution activities

The *monitoring and execution* activities guarantee that any devised strategy given by our adaptive framework considers the up to date information retrieved from the current target edge system and is capable of enforcing the required changes into the system. Consequently, both the placement cycle and the invocation path cycle share the two activities. In this section, we present the characteristics of the *monitoring and execution* activities concerning the two cycles.

7.3.1 Monitoring activity

Considering the uncertainty found in our target edge system, where nodes can fail or leave the system freely, the monitoring activity represents a crucial activity for the two MAPE cycles. The overall objective is to provide a global view of the monitored system, by giving information about three key groups with respect to (i) overall system changes, (ii) edge nodes, and (iii) communication links.

Overall system changes

The monitoring activity provides valuable information to the coordinator node regarding changes in the system. To be effective in determining when a change occurs, the monitoring activity finds a system change by keeping as a reference the last known stable state – the state when the application was successfully operating at the desired parameters. A change in the status of the system appears when a microservice used in the current invocation path is no longer reachable due to node failure or a new node has joined the edge system. The former has a disruptive effect on the system's stability which triggers the *invocation path cycle* or in the worst case, the *placement cycle* as well. In the latter scenario, there are no changes in the application's functionality; the addition of a new node provides advantages to the system by increasing the total available resources.

Edge nodes monitoring

The monitoring view has transitioned from the system level to the node level. At the node level, there is crucial information that the coordinator requires when deploying and maintaining an application in the system. First, it requires knowing the current node's available resources every time it makes use of the *placement cycle*. By available resources, we mean both resources (such as RAM, CPU, or specialized hardware) as well as existing microservices. As a result, by monitoring each node, the location of already placed microservices may be discovered. To achieve a level of freshness in the system, the desire is to always use the currently available resources of a node at the application's deployment time.

Communication links monitoring

Since one objective is to satisfy latency, monitoring the communication links between nodes is crucial. Not only do these links provide the communication latency between dependent microservices, but they are part of the application's invocation path alongside nodes. However, communication links may fail. As such, whenever a link fails, the coordinator must start the *invocation path cycle* to find a new satisfiable invocation path. In this chapter, we assume that a communication link fails if either the source or the destination node has failed.

To be more concrete, let us consider our running example, where the monitoring activity monitors the system. First, the monitoring activity recognizes the appearance of new nodes when a new participant arrives at the event – nodes used for further application deployments or system stabilization. Furthermore, when a node joins the system, the monitoring activity starts monitoring the node's internal status as well as its communication links. Thus, knowing what resources each node has and what is the communication latency when sending a message to other nodes. After the public-safety application is operational, the activity starts to monitor closely the nodes that participate in the application's invocation path, generating event triggers for the *invocation path cycle* when one or more of these nodes become unresponsive.

7.3.2 Execution activity

The last step in any MAPE cycle is the execution activity. Once either of the two cycles finds a successful plan, the execution activity may start. During this activity, the framework enacts the plan received upon the edge system. After the execution activity finishes, the application becomes operational.

Placing microservices

Based on the placement plans conceived during the *placement planning activity*, the execution activity can start distributing the microservices on nodes. In our case, the execution activity works above the node level by sending microservices to the nodes according to the placement strategy without providing techniques to ensure correct execution on the host node. In this chapter, we assume that each node guarantees the correct execution of the received microservice on the node, scaling up and down the microservice depending on the node's available resources and the number of microservice requests by using some container framework. A container has a small footprint, providing all the required dependencies a microservice needs to function as intended. Furthermore, notice that using a container fits rather well in a distributed edge system, where nodes are heterogeneous and mobile, the only requirement being that each node in the system runs locally a container engine.

Building invocation paths

Once all microservices are available in the system, the execution activity enacts the newly generated invocation path by informing each participating node on how to reach their destination node, i.e., the node where a dependent microservice resides. As a result, when building a new invocation path, there is no need for performing a microservice migration – an expensive action. Instead, the execution activity locates the new nodes that participate in the invocation path and informs the nodes about addresses used for communication. For an example of this, consider executing an invocation path found for the public-safety application (see Figure 7.4). First, during the *initial placement* the invocation path cycle finds p_1 as the satisfying invocation path. In this case, to execute the plan, the execution activity finds the addresses of participating nodes, i.e., E_1, E_2, and E_3. Next, it informs E_1 that m_0 communicates with its dependent microservice, i.e., m_1, using the address of E_2. Notice that if two dependent microservices share the same node, as in the case of m_1 and m_3, then the communication is internal and no address is required. Next, the activity informs E_2 about the address of E_3, since there are the last two microservices required. In conclusion, during system *stabilization*, the execution activity can enact the new path by simply informing the participant nodes what address to use.

7.4 Placement cycle

The *placement cycle* ensures that the edge system where applications must execute satisfies all requirements in terms of availability constraints and microservices' resource requirements. To provide correct deployment, the edge nodes' available resources must be considered; this is the information that the monitoring activity provides for all available nodes found in the system. Since we are in a centralized environment, we assume that the coordinator node can monitor any changes regarding the node's internal status – information regarding current available resources. The placement cycle is bootstrapped with a given application-wide availability requirement, an application model, and the target edge system – those are specified per use case. Note how this cycle is not concerned with invocation path calculation and its associated e2e latency; the *placement cycle* merely aims to derive the needed number of replicas to satisfy availability, and subsequently map them on the edge system. Specifically, we distill the following *placement cycle's* objectives:

- Which is the minimum number of microservice replicas required to satisfy the application's availability requirement?
- Given the application model and the number of replicas, what placement strategy satisfies the microservices' resource requirements?

7.4.1 Placement Encoding activity

The *placement encoding activity* uses the knowledge received from the monitoring activity, i.e., node's available resources, communication latency, and node's failure probability, to find a satisfiable mapping of microservices in a volatile edge system full of uncertainty. Remember that in such a system, where some nodes have high failure probabilities, computing an optimal microservice placement is not necessary (due to increased computation required) – the network configuration may change frequently, sometimes even before the optimal solution is found. As a result, we seek to find a microservice placement that satisfies the given requirements, i.e., the application's availability and microservices' resource requirements, with respect to the target edge system. To this end, we adopt SMT [BT18], since deciding if a first-order formula satisfies a set of constraints fits rather well the placement encoding objectives. As such, we model the placement rules as a first-order formula consisting of a set of constraints and an objective. Moreover, computational demands are kept low – we will discuss more on this in Section. 7.6. Specifically, we model the problem with the linear integer arithmetic and boolean theories, having three distinct targets; (i) placing a microservice on a node, (ii) placing a microservice replica on a node, and (iii) keeping track of the microservice's availability considering its host node.

Microservice Placement

A constraint that introduces an important rule to ensure the correct placement of a microservice on the edge system, i.e., a microservice can only be placed on a single edge node. Upon application deployment, a microservice $\mathsf{m_i}$ can be placed on a node $\mathsf{E_i}$ if the following conditions are true: (i) $\mathsf{E_i}$ has the available resources to host $\mathsf{m_i}$ and (ii) $\mathsf{m_i}$ does not exists in the system yet. Considering the two conditions, it is clear that the *microservice placement* constraint does not consider replicas of $\mathsf{m_i}$; the placement of the microservice replicas being handled by another constraint. In conclusion, with this constraint, we place all application microservices in the target edge system – if the system does not have the available resources to host at least all application microservices, then we cannot deploy the application on the system. For example, we have an edge system consisting of two edge nodes, i.e., $\mathsf{E_1}$ and $\mathsf{E_2}$, on which we want to map the *face recognition* microservice, i.e., $\mathsf{m_1}$, from our public-safety application. Under these conditions, there are two possible mappings of $\mathsf{m_1}$ – in this constraint, mapping $\mathsf{m_1}$ on both nodes is incorrect and must be avoided. This is captured with Formula 7.1, where $\mathsf{n_m}$ represents the total number of microservices found in the application model, while the *map()* function provides a mapping between $\mathsf{m_i}$ and one node E. For number of nodes n and microservices m, the formula construction exhibits complexity $\mathcal{O}(nm)$.

$$\mathsf{mrcFacts} : \bigwedge_{\mathsf{i=1}}^{\mathsf{n_m}} (\exists!\, \mathsf{E} : \mathsf{map}(\mathsf{m_i} = \mathsf{E})),\ \forall\, \mathsf{E} \in \mathsf{E_N}. \tag{7.1}$$

Microservice Replication

A constraint that provides rules for replica placement on the target system. In this case, the constraint considers for placement only the replica $\mathsf{rp_i}$ of a microservice $\mathsf{m_i}$, since $\mathsf{m_i}$ is already placed with the previous constraint. During placement, there is a set of rules that the *placement cycle* must abide to successfully place $\mathsf{rp_i}$ in the system. First, an edge node $\mathsf{E_i}$ can host only one replica $\mathsf{rp_i}$ – placing replicas on different nodes ensures higher application availability, offering the possibility to recover from node failure. Second, as in the case of *microservice placement* constraint, $\mathsf{E_i}$ must be capable to execute $\mathsf{rp_i}$. For example, consider we need one more replica of $\mathsf{m_1}$ to fulfill the application's availability requirement. In this case, we must deploy one $\mathsf{m_1}$ replica, i.e., $\mathsf{rp_1}$ on an edge system consisting of three nodes $\mathsf{E_1}$, $\mathsf{E_2}$, and $\mathsf{E_3}$. Following Formula 7.1, $\mathsf{m_1}$ is mapped on $\mathsf{E_2}$. As a result, to satisfy the first rule, we cannot map $\mathsf{rp_1}$ on the same node as $\mathsf{m_1}$. As a result, only two nodes can possibly host $\mathsf{rp_1}$ – $\mathsf{E_1}$ and $\mathsf{E_3}$. More concretely, Formula 7.2 ensures that if a replica exists on a certain node, then all other remaining replicas cannot be placed on that node, where $\mathsf{n_r}$ represents the total number of replicas for a certain microservice and $\mathsf{n_{rl}}$ the total number of replicas without considering the current $\mathsf{rp_i}$. Construction of Formula 7.2 exhibits complexity of $\mathcal{O}(nn_r)$, where n is the total number of possible locations where replicas of $\mathsf{m_i}$ can reside.

$$\mathsf{replDomain}: \bigwedge_{\mathsf{i=1}}^{\mathsf{n_r}}((\mathsf{map}(\mathsf{rp_i} = \mathsf{E})) \implies \bigvee_{\mathsf{j=i}}^{\mathsf{n_{rl}}}(!\mathsf{map}(\mathsf{rp_j} = \mathsf{E})), \forall\, \mathsf{E} \in \mathsf{E_N}. \tag{7.2}$$

Microservice Availability

A constraint aiming to provide for each microservice the associated availability factor. As we have defined in Section 7.2, a microservice $\mathsf{m_i}$ inherits its availability factor from the host node, i.e., the node where the microservice resides after the placement cycle. Knowing the availability of each microservice helps the *placement cycle* to determine the maximum number of replicas required for each microservice and decide if a certain placement configuration fulfills the requirements. For example, if microservice $\mathsf{m_1}$ is mapped to node $\mathsf{E_1}$, where $\mathsf{E_1}$ has a failure probability of *40%*, then the availability factor of $\mathsf{m_1}$ is $\mathsf{m_{avail_i}} = 1 - \mathsf{E_{failure}}$. Formula 7.3 encodes the availability factor of a microservice considering its deployment location; its construction complexity is analogous to the previous replication formula.

$$\mathsf{availDomain}: \bigwedge_{\mathsf{i=1}}^{\mathsf{n_m}}(\mathsf{map}(\mathsf{m_i} = \mathsf{E}) \implies \mathsf{m_{avail_i}} = 1 - \mathsf{E_{failure}}), \forall\, \mathsf{E} \in \mathsf{E_N}. \tag{7.3}$$

Placement Cycle Objective

The *placement cycle* aims to deploy the application's microservices on the target edge system considering the application's availability requirement and microservices' resource requirements. Finding the total number of replicas is dependent on the application's availability requirement. As a result, we define an *objective constraint* to ensure that the placement strategy satisfies the application's availability constraint. The microservice's availability is dependent on the number of replicas and their inherited availability factor, ensuring that each microservice has the required availability factor – the microservice's availability requirement is inherited from the application. Using the application's availability requirement for microservices ensures that we deploy enough resources on the target edge system to make the application more resilient to failures; the availability of a microservice helps to distribute multiple replicas of a microservice on different edge nodes. In Formula 7.4 we capture the placement cycle objective, where $\mathsf{m}_{\mathsf{maxAvail}}$ is the desired microservice availability; its construction amounts to complexity of $\mathcal{O}(n_r)$.

$$\mathsf{objConstraint} : 1 - \prod_{\mathsf{i}=1}^{\mathsf{n_r}} \mathsf{m}_{\mathsf{avail_i}} \geq m_{maxAvail}. \tag{7.4}$$

7.4.2 Placement Planning Activity

The *placement planning activity* aims at using the problem encoding of the previous step to find the minimum number of microservice replicas required to satisfy the application's availability and generate a deployment plan. By virtue of the design choice of employing SMT, we guarantee that the generated placement plan provably satisfies all the defined constraints and fulfills the placement objective. At the core of this activity stays an SMT solver capable of solving an SMT formula $\mathcal{F}$ and deciding if a placement strategy satisfies the cycle's objectives or not. $\mathcal{F}$ is a conjunction of Formulae (7.1 to 7.4) (see Formula 7.5).

$$\mathcal{F} : \mathsf{mrcFacts} \wedge \mathsf{replDomain} \wedge \mathsf{availDomain} \wedge \mathsf{objConstraint}. \tag{7.5}$$

To compute the number of replicas required and find a placement strategy, the *placement planning activity* uses $\mathcal{F}$ to place one microservice $\mathsf{m_i}$ on the system at a time. An iterative process, where at each step, the activity maps a new $\mathsf{m_i}$ replica in the system until the current deployment of $\mathsf{m_i}$ and its replicas satisfy the $\mathsf{m_i}$ availability requirement. Finally, when $\mathsf{m_i}$ is placed successfully in the system, the process continues with the next microservice $\mathsf{m_j}$. For example, consider the process of placing the face recognition microservice, i.e., $\mathsf{m_1}$. At this stage, the activity already found a suitable placement for $\mathsf{m_0}$ – resulting in fewer overall available resources when placing $\mathsf{m_1}$. The process starts by placing a $\mathsf{m_1}$ replica in the system. At this point, an evaluation occurs: if the current placement of $\mathsf{m_1}$ is enough to satisfy Formula 7.4, then we continue to map the next microservice, i.e., $\mathsf{m_2}$. However, if placing only one replica of $\mathsf{m_1}$ is not enough, then the number of replicas for $\mathsf{m_1}$ is increased by one and the activity attempts to solve $\mathcal{F}$ again. The process continues to increase the number of replicas until either a solution is found that satisfies Formula 7.4 or there are not enough available resources to accommodate all replicas. Finally, if we manage to find and place the minimum number of replicas for m1, the planning process continues with m2. However, if no solution is found for a placement strategy for m1, then the process stops without producing any placement strategy. The process terminates upon two conditions: (i) a successful deployment plan of all microservices is found or (ii) one or more microservices cannot be placed in the system. In the former case, since there are enough resources in the system to accommodate all application's microservices, the cycle can find a satisfiable placement strategy. In the latter case, since the target system lacks the required resources to host all microservices, the cycle cannot find a placement strategy to deploy the application on the current edge system because it does not exist. Figure 7.5 shows a placement strategy found for the public-safety application.

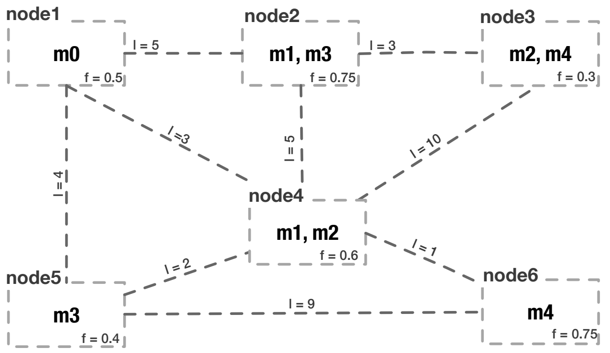


Figure 7.5: Overview of the placement strategy for the *initial placement* of our public-safety application example. The *placement planning activity* provides only the location of microservices and their replicas on the system.

7.5 Invocation path cycle

With the previous cycle, the framework deploys the application's microservices across available nodes comprising the system. After the application's microservices become available in the system, the framework can use the invocation path cycle to invoke the microservices according to the application's execution sequence. However, multiple replicas of a microservice may reside in the edge system – resulting in multiple possible invocation paths. The invocation path cycle is responsible for establishing (and maintaining) the call sequence across those replicated microservices such that it fulfills the application's latency requirements. Disruptions such as network changes or node failures may render an invocation path no longer usable; thereupon, the framework triggers the cycle again.

7.5.1 Invocation path encoding activity

As in the case of the placement cycle, SMT fits particularly well with the invocation path cycle; it provides guarantees that if an invocation path is found, then it satisfies the application requirements, i.e., e2e latency and availability. The *invocation path encoding* is responsible for transforming the information received from the monitoring activity into

SMT Formulae. Hence, this activity is an integral part of the cycle, providing the means to fulfill its objectives. Considering the cycle's objectives and information received from the monitoring activity, we can construct three different formulae – two formulae that reflect the cycle's constraints and one formula that considers the cycle's objectives. The invocation path encoding activity builds the formulae by using information gathered from the edge system; (i) the location of microservices on nodes and (ii) the communication latency between nodes. Next, we present the formulae building blocks as well as the cycle objectives.

Microservice Location

The microservice location represents one important knowledge, an asset to the invocation path cycle when finding a satisfiable invocation path for an application. A knowledge that is shared by sharing the domain for each microservice needs – a domain consists of a set of edge nodes where a microservice resides. First, by knowing the location of each $\mathsf{m_i}$, we lower the search space by considering a subgroup of nodes from the edge system – a group formed with nodes where the application's microservices reside. As we explained at the beginning of this chapter, knowing the microservice location lowers the execution time required to find a new invocation path. For example, there are three nodes $\mathsf{E_1}$, $\mathsf{E_2}$, and $\mathsf{E_3}$, where the *face recognition* microservice $\mathsf{m_1}$ exists. Under these conditions, the invocation path cycle considers one of the three nodes when building the application's invocation path. We illustrate the constraint in Formula 7.6, where $\mathsf{n_M}$ represents the total number of microservices and $\mathsf{n_E}$ is the total number of nodes where $\mathsf{m_i}$ is available; its construction exhibits complexity of $\mathcal{O}(n_E n_M)$.

$$\mathsf{microDomain} : \bigwedge_{\mathsf{i=1}}^{\mathsf{n_M}} \left(\bigvee_{\mathsf{j=1}}^{\mathsf{n_E}} (\mathsf{m_i} = \mathsf{E_j}) \right). \tag{7.6}$$

Microservice Latency

In the previous constraint, the invocation path cycle becomes aware of the microservices location. However, to satisfy the cycle objectives, i.e. e2e latency, we require a constraint to capture the communication latency between dependent microservices as well. Therefore, we create the *microservice latency* constraint. Similar to the previous constraint, the communication latency is dependent on the location of microservices. In our edge system, microservices are placed on nodes that have the actual latency, so the communication latency between two $\mathsf{m_i}$ and $\mathsf{m_j}$ microservices is inherited from the communication latency between their host nodes. Thus, a pair $\mathsf{m_i}$, $\mathsf{m_j}$ may have a different communication latency on different combinations of nodes. Given Formula 7.6, one can observe the latency between two microservices $\mathsf{l_{m_i,m_j}}$, considering their possible locations. For example, from the public-safety application (see Figure 7.3), we can observe that $\mathsf{m_1}$ has a communication dependency with $\mathsf{m_2}$; $\mathsf{m_1}$ and its replicas are mapped on $\mathsf{E_1}$, $\mathsf{E_2}$, and $\mathsf{E_3}$, while $\mathsf{m_2}$ is on $\mathsf{E_2}$. Under these conditions, the encoding consists of all possible combinations between the two microservices' domains. In the end, the constraint yields a connection between the location of a microservice and its associated node latency, i.e., $\mathsf{l}_{\mathsf{m_i},\mathsf{m_j}} = \mathsf{l}_{\mathsf{E_i},\mathsf{E_j}}$ – as illustrated in Formula 7.7. Its construction exhibits a complexity of $\mathcal{O}(n_{MD}n_{ES}n_{ED})$, where n_{MD} is the total number of dependent groups consisting of two microservices (a source and a destination), while n_{ES} and n_{ED} represent the total number of nodes where microservices part of a dependent group exist.

$$\begin{aligned} \mathsf{latencyDomain} : & \bigwedge_{\mathsf{D}}^{\mathsf{n_m}} (\mathsf{m_i} = \mathsf{E_i},\ \mathsf{m_j} = \mathsf{E_j}) \Rightarrow (\mathsf{l}_{\mathsf{m_i},\mathsf{m_j}} = \mathsf{l}_{\mathsf{E_i},\mathsf{E_j}}) \\ & \text{for } \mathsf{D} = \{\mathsf{i},\mathsf{j}\} \text{ where } \mathsf{i} \in [0,\ \mathsf{n_m}] \text{ and } \mathsf{j} \in [0,\ \mathsf{n_{EN}}]. \end{aligned} \tag{7.7}$$

Invocation Path Cycle Objectives

The invocation path cycle has two objectives that each new invocation path must abide, i.e., e2e latency and availability. On the one hand, we can obtain the latter using Formula 7.3 described in the placement cycle, where $\mathsf{n_r}$ represents all available microservices and their replicas found in the edge system. On the other hand, we obtain the former with the help of Formula 7.7 – the e2e delay of an application represents the sum of all communication latencies between every two dependent microservices found in the application's execution sequence. Recall that we assume that the maximum e2e latency and the desired availability are given by the developer at design time. The invocation path cycle must perform three important steps to capture the e2e latency of an invocation path: (i) choose an invocation path, (ii) get the communication latency between dependent microservices, and (iii) verify that the total e2e latency fulfills the objective. To perform the first two steps, the invocation path cycle can use the microservice location and latency constraints. For the third step, the cycle can use Formula 7.8 to validate the chosen invocation path. The formula ensures that the current invocation path's e2e latency is smaller or equal to the maximum e2e latency allowed (i.e., $\mathsf{e2e_{max}}$). Construction of Formula 7.8 exhibits complexity $\mathcal{O}(mn_{md})$, where m represents the number of available microservices and n_{md} the number of dependencies a microservice has.

$$\mathsf{objConstraint} : \sum_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{l_i} \leq \mathsf{e2e_{max}}. \tag{7.8}$$

7.5.2 Invocation path Planning Activity

Similar to placement planning, the invocation path planning activity creates a plan for defining a new invocation path that fulfills the application requirements. In Formula 7.9, we present the complete SMT formula $\mathcal{F}_{\mathcal{I}}$, which builds upon the formulae received from the previous activity. By using an SMT solver to solve $\mathcal{F}_{\mathcal{I}}$, we guarantee that every invocation path found fulfills the cycle's objectives. Recall that the placement planning activity uses an iterative process to devise a plan. In contrast, this activity considers all microservices and finds an invocation path in one single step. After obtaining

$\mathcal{F}_{\mathcal{I}}$, the invocation path planning activity takes the application's execution sequence and attempts to find a set of edge nodes (nodes that host microservices invoked in the execution sequence) such that it fulfills all cycle's objectives. As a result, with a single invocation of the solver, the activity can devise a plan for building an invocation path.

$$\mathcal{F}_{\mathcal{I}} : \mathsf{microDomain} \wedge \mathsf{latencyDomain} \wedge \mathsf{objConstraint}. \tag{7.9}$$

Recall that the framework uses this cycle to provide system stabilization after a node failure. Therefore, we must ensure that the invocation path cycle is capable of providing a plan in a short time. In comparison to the formula employed by the placement cycle, observe that $\mathcal{F}_{\mathcal{I}}$ has (i) a smaller formula size since it considers only a subgroup of nodes, (ii) awareness of microservice location, and (iii) information about microservices communication latency. As a result, $\mathcal{F}_{\mathcal{I}}$ is expected to scale well with an increase in network or application size – more on this topic will be discussed in Sec. 7.6. To this end, $\mathcal{F}_{\mathcal{I}}$ must find a combination of nodes that fulfill Formula 7.8.

In Figure 7.6, we can observe the conceived plan to build an invocation path for the public-safety application. As we can see, the invocation path planning activity devises an invocation path using as a blueprint the placement found by the placement cycle. With this knowledge, the activity tries to find a set of nodes where microservices reside to build an invocation path that follows the application's execution sequence. As expected, there may be multiple invocation paths in the system, but not all of them satisfy Formula 7.8. In our case, the activity returns a path, i.e., p_1, that meets all requirements.

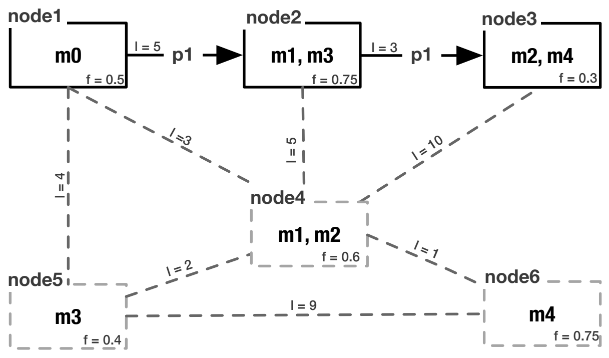


Figure 7.6: The invocation path found for the public-safety application example, considering the application's requirements as well as the microservices' location.

7.6 Evaluation

In this section, we evaluate the performance of our proposed solution considering different synthesized scenarios. For each, we first assess the performance of the placement cycle which yields a microservice placement strategy. Then, given placement, we employ the invocation path cycle to find valid invocation paths. To concretely support evaluation and investigate the feasibility of the proposed framework, we realized a proof-of-concept implementation which is available as open-source software reflecting the placement and invocation path cycles of Sections 7.4 and 7.5, along with auxiliary functionalities supporting monitoring and execution. Thereupon, we assess the performance and scalability of the critical placement and invocation path cycles. We additionally investigate the framework's capability to reach stabilization when nodes fail and conclude with a discussion.

7.6.1 Framework realization

To investigate feasibility, we have realized the proposed framework as a prototype implementation; its main architectural components are illustrated in Figure 7.7. In our edge system, the coordinator node resides on a machine with an Intel i5 2.3GHz processor and 16 GB of RAM and hosts the placement and invocation cycles – the two cycles construct the required formulae and through interaction with a solver build the corresponding plans. In contrast, the participant nodes are resource-constrained devices, i.e., a Raspberry Pi with 1GB of RAM and 32 GB of storage, where the deployed application resides. Furthermore, to host microservices, each participant edge node runs a common operating system, i.e., Raspberry Pi OS Lite [4], and Docker CE [5] (or alternatively Kubernetes/OpenFaaS). Note that in our framework all interactions are performed through REST APIs. Therefore, on each participant node, there is an API that facilitates the communication with the coordinator node as well as the other edge nodes found in the system. To easily set up all participating edge nodes, we use Ansible [6] to provide our infrastructure as code. As a result, all nodes can have the same setup, enabling the user to easily build our framework and test it on their infrastructure.

Our prototype has monitoring functionality responsible for (i) detecting if a node is available in the system, (ii) finding nodes' available resources, and (iii) obtaining the communication latencies between participating nodes. First, to provide overall monitoring of the system, the coordinator starts monitoring the participant node status as soon as it joins the system. To achieve this, we continuously check if a node is available every 100 ms and we trigger the invocation path cycle if one of the nodes participating in the application's current invocation path has failed. Second, the coordinator can request the currently available resources of each node using the API at any time – usually, this happens during the placement cycle when we need to know the available resources to correctly define a microservice placement strategy. Finally, for the invocation path cycle, the coordinator must know the current communication latency between nodes. Finding the communication latency between all nodes is not a trivial task, since the coordinator can only measure the communication latency between it and all supervised nodes. Therefore, we distribute the latency monitoring between all participant nodes. As a result, when a node receives, from the coordinator, a request for its communication

[4]https://www.raspberrypi.org/
[5]https://www.docker.com/
[6]https://www.ansible.com/

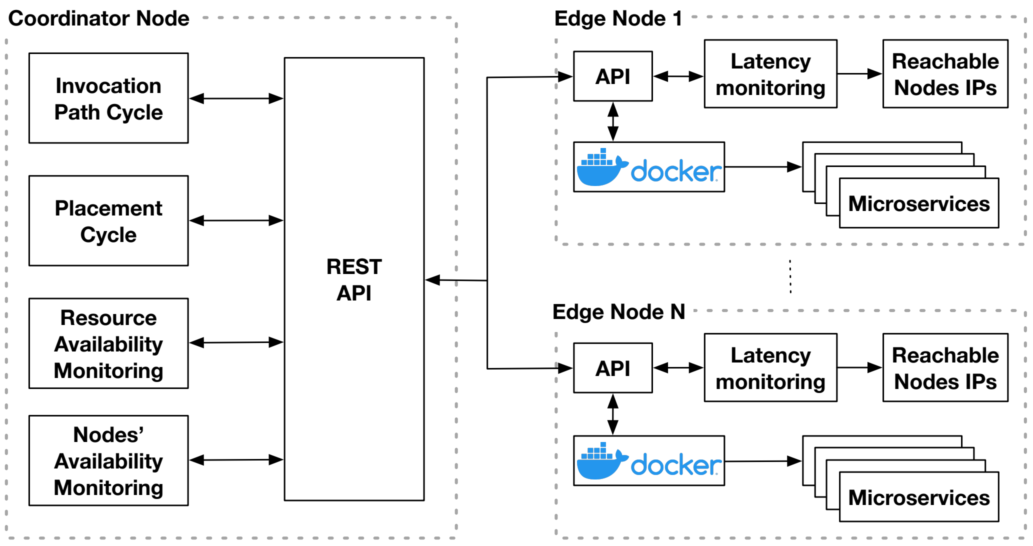


Figure 7.7: Combined deployment and dataflow diagram for a framework realization.

latency, the node reports its communication latency measured to all reachable nodes. To measure the latency between two nodes, we use the ping command to measure latency between two nodes. Each node sends a set amount of pings to a list of reachable nodes' IPs and takes the average response time as the latency. Finally, execution entails deployment of containerized microservices on nodes and setting up communication between them. The prototype, built in Python using Z3 [DMB08] as the underlying SMT solver and an example microservice application are available as open source software[7].

7.6.2 Experiments Setup

We create a total of five scenarios, each having different characteristics such as the number of microservices, application requirements, number of edge nodes, and node's failure probabilities. We create these scenarios to evaluate the performance in terms of the time required to find a valid solution. Therefore, in every scenario, we gradually increase the number of nodes found in the target system – Table 7.1 shows the five scenarios along with their specifications.

Each scenario in Table 7.1 has a different number of microservices allowing us to examine the impact of the system size as the total number of nodes on execution time. Throughout our performance evaluation, we do not change the e2e latency and availability because we want to understand the impact of the number of nodes on the placement execution time. Choosing more stringent constraints, i.e., having a lower e2e latency or a higher availability impacts the execution time; however, this does not alter how the execution time rises with an increase in system or application size. We model the resource requirements of

[7]github.com/cavasalcai/Adaptive-Volatile-Edge-Systems.

scenario	micro–services	e2e latency	availability	failure probability	nodes
1	10	350	0.75	[0.1, 0.5]	10 - 500
2	20				
3	30				
4	40				
5	50				

Table 7.1: Specifications of the synthesized scenarios adopted for performance evaluation.

each microservice as a set of computational resources, i.e., a tuple *(RAM, CPU, HDD)*; for each we randomly choose a value between *[5, 18]* units. Recall that the edge system is characterized by (i) the number of nodes, (ii) the failure probabilities of each node, and (iii) the communication latency associated with each communication link. We set the failure probability of each node and the communication latency by choosing a random value between *[0.1, 0.5]* and *[1, 10]* ms respectively. Regarding the node's available resources, we assign a random value chosen between *[15, 30]* units. Finally, for each scenario, we increase the number of nodes by 10, up to a maximum of 500 nodes. Further, when we increase the system size, we extend the last system with the addition of new nodes. As a consequence, a larger system size always contain the previous nodes; this intends to simulate a real setting where new nodes connect to the edge system.

There is only one constraint left to adopt, i.e., the execution sequence. For each application size, we choose an execution sequence that starts with a source and ends with a sink. To have a diverse selection of application models, we propose a procedure to create a sequence that involves all application microservices and leads to a different execution sequence for each application size. For example, let us build the execution sequence for an application with 10 microservices, $\mathsf{M_N} = \{\mathsf{m_1}, \mathsf{m_2}, \mathsf{m_3}, \dots, \mathsf{m_{10}}\}$. The procedure starts by choosing $\mathsf{m_1}$ as the initial source – therefore, we remove $\mathsf{m_1}$ from $\mathsf{M_N}$ and randomly choose a destination microservice from the remaining elements found in $\mathsf{M_N}$. Let us assume that we choose $\mathsf{m_3}$ as the destination microservice of $\mathsf{m_1}$ – we now have a dependency between $\mathsf{m_1}$ and $\mathsf{m_3}$. Notice that we do not remove $\mathsf{m_3}$ from $\mathsf{M_N}$; we remove $\mathsf{m_3}$ only when it becomes the source microservice. Now that we found a destination for $\mathsf{m_1}$, we can restart the procedure by taking the next microservice from $\mathsf{M_N}$, i.e., $\mathsf{m_2}$. Finally, the procedure stops when there are no more elements in $\mathsf{M_N}$.

7.6.3 Placement and Invocation Path Performance

In our framework, the placement cycle consists of two phases, i.e., (i) finding a placement strategy and (ii) starting the microservices on the host nodes. We evaluate the placement cycle's performance by measuring the execution time required to find a satisfiable placement strategy according to the application's objectives. We perform a total of 50 placements per scenario, one placement for each new system size – once we find a placement strategy for the current edge system, we increase the number of available nodes by 10 and trigger the placement cycle again. We mention that when the placement cycle is triggered, the target edge system does not contain any microservices.

Figure 7.8 presents the execution time required to find a placement strategy for all scenarios, where *x-axis* shows the number of nodes and *y-axis* presents the execution time in seconds. Subsequently, the invocation path cycle uses as a basis the edge system populated with microservices and their replicas by the placement cycle. Similar to the placement cycle, we measure the invocation path cycle's performance as the time required to derive a feasible path considering the application's latency and availability requirements. We evaluate the cycle's performance on obtaining satisfiable invocation paths on 250 different edge systems of different sizes – 50 for each scenario. Figure 7.9 illustrates the time required to find a satisfiable invocation path among all replicas.

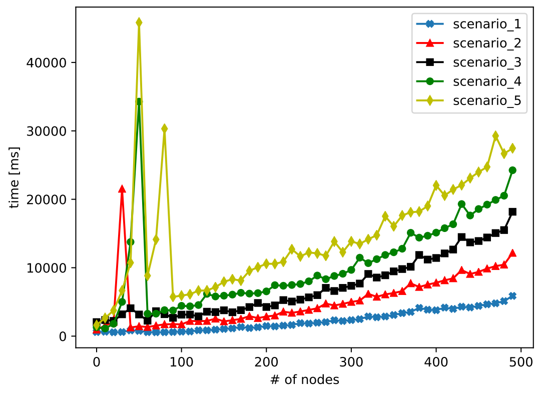


Figure 7.8: Execution time of the placement cycle for different scenarios across different network sizes.

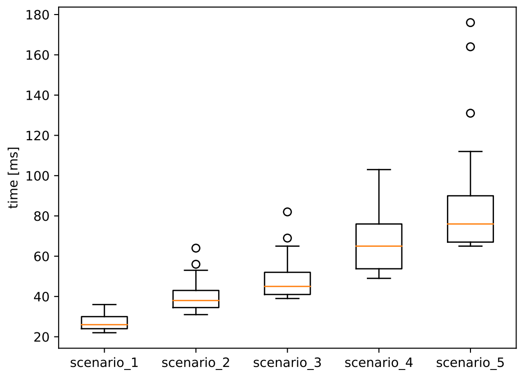


Figure 7.9: Average execution time required by the invocation path cycle across the five scenarios.

7.6.4 Stabilization Performance

This section outlines the adaptive framework's capabilities to reach a system stabilization when one or more nodes have failed. More concretely, we focus on evaluating the performance of the *invocation path cycle* in providing satisfiable invocation paths after the system has lost some nodes. For this purpose, we present a realistic scenario, where after the application is operational on the edge system, some nodes fail and disrupt the application's functionality. This occurs when the system recovers from node failure, by employing the *invocation path cycle*.

We consider an application with a realistic number of microservices that we must deploy on an edge system with a total number of 50 edge nodes. The application model is a modified version of the complex application model presented in [RCP20] and illustrated in Figure 7.10. In our case, the microservice-based application consists of 7 microservices, but we alter the execution sequence by adding a new dependency between m_4 and m_5. As such

Micro–services	Nodes	Initial path
m0	E_6, E_{16}, E_{49}, E_{50}, E_1, E_{43}	E_6
m1	E_6, E_{16}, E_{49}, E_{50}, E_1, E_{43}	E_6
m2	E_6, E_{16}, E_{49}, E_{50}, E_1, E_{43}	E_6
m3	E_{39}, E_{17}, E_{33}, E_{48}, E_{32}, E_4	E_{33}
m4	E_4, E_{32}, E_{44}, E_6, E_2, E_{39}	E_4
m5	E_{25}, E_{45}, E_2, E_4, E_{33}, E_{50}	E_{25}
m6	E_{17}, E_2, E_4, E_{32}, E_{33}, E_{43}	E_{17}

Table 7.2: Initial placement and invocation path.

the application has one source (i.e., m_0) and one sink (i.e., m_6). We choose a maximum e2e latency of *100 units* and required availability of *0.96*. All other characteristics, like microservices' resource requirements and node's communication latency, remain the same as described for the other scenarios. However, to evaluate stabilization, we adopt a different set of characteristics for edge nodes, aiming for a higher number of replicas in the system: we randomly select node failure probabilities between *[0.1, 0.5]* and set available resources in the range of *[50, 80]* units respectively.

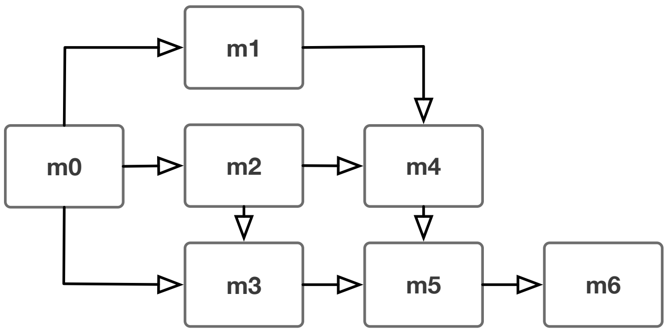


Figure 7.10: Overview of a microservice-based application model and its execution sequence.

Initially, the application's microservices and its invocation path are missing from the target system. Therefore, we must trigger the placement and invocation path cycles once, in sequence, to make the application operational in the edge system. In Table 7.2, we can observe the placement configuration and the chosen invocation path. Each row shows all nodes where a microservice resides and the node that participates in the derived invocation path. During the application's lifespan, some nodes may fail, disrupting the application's invocation path – a disruption that triggers the stabilization process. However, not all failed nodes from the system disrupt the application's functionality: only ones that are part of the invocation path. To produce a disruption, we incrementally fail a node from the current path and use the invocation path cycle to recover. In Table 7.3, we present each path found during stabilization as well as its associated e2e latency. We can observe, in the first row, that the current invocation path (i.e., the path from Table 7.2) has a total e2e latency of *32* units. Let us consider that after the initial deployment the edge node E_4 has failed, disrupting the initial invocation path. As a consequence, the application does not function properly, requiring the stabilization process to start and find a new stable path – we present the new invocation path on the next row. The process continues until there is no possibility to achieve system stabilization with the remaining available microservices, resulting in the need of employing the placement cycle again to re-populate the system. In Figure 7.11 we present the performance results.

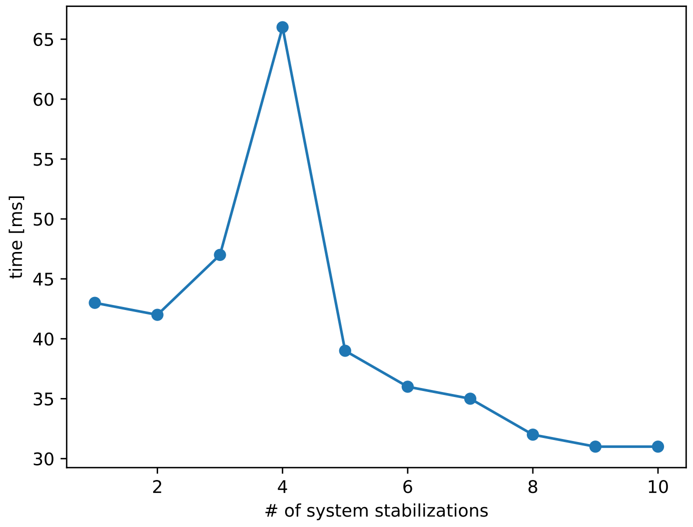


Figure 7.11: Execution time required by the invocation path cycle to reach stabilization after node failures.

7.6.5 Discussion

We have demonstrated that by using our adaptive framework and its two cycles, we can perform application placement and its recovery from an unstable state resulting from edge system volatility promptly. We can observe from the results presented in Figure 7.8 that the placement cycle performance increases with (i) the total number of available nodes found in the system and (ii) the application size. Among the two, the application size has a greater impact on the execution time, since we deploy one microservice and its replicas at a time; a step that requires employing the SMT solver for each microservice at a time. Other factors influence the execution time required to find a placement strategy, i.e., applications' requirements (i.e., e2e latency, microservice's resource requirements, and availability) and network characteristics (i.e., failure probabilities and available resources). However, these factors only impact the time required to find a solution for specific scenarios; it can increase or decrease the execution time depending on how stringent the requirements are. We can observe the impact of these factors by looking at the spikes

#	Microservice invocation path	e2e latency	Node failure
1	E_{17}, E_4, E_{25}, E_{33}, E_6	32	E_4
2	E_{32}, E_{17}, E_{25}, E_{33}, E_6	41	E_{25}
3	E_{32}, E_{17}, E_{45}, E_{33}, E_6	39	E_6
4	E_{32}, E_{17}, E_{45}, E_{16}, E_{33}	31	E_{32}
5	E_{17}, E_{44}, E_{45}, E_{16}, E_{33}	42	E_{44}
6	E_{45}, E_{17}, E_2, E_{16}, E_{33}	42	E_{17}
7	E_2, E_{45}, E_{39}, E_{16}, E_{33}	46	E_{33}
8	E_{45}, E_{39}, E_{16}, E_2	45	E_{45}
9	E_{49}, E_{39}, E_{16}, E_2	46	E_{16}
10	E_{49}, E_{39}, E_2, E_{50}	41	E_{50}
11	E_{49}, E_2, E_{48}	30	E_2

Table 7.3: Invocation paths and their e2e latency found when incrementally failing nodes.

found at lower system sizes. In these particular cases, the edge system, the node's failure probabilities, and the node's available resources play an important role. The reasons for the spikes in execution time are the following: (i) there is no feasible solution due to the lack of available resources found in the target system or (ii) the system's available resources are equal to the application's requirements. In both situations, the SMT solver must traverse the entire search space to try and find a placement strategy. We can observe that with the increase in the available nodes, there are more available resources, offering the underlying SMT solver more possibilities to find a satisfiable placement strategy, also lowering the execution time.

Figure 7.9 shows the execution time required to find an invocation path for different use cases. We can see that compared to the placement cycle, the invocation path cycle can find an invocation path in a short time. Furthermore, the invocation path cycle does not have any big spikes in execution times throughout a single scenario since the SMT solver has more knowledge about the microservices and their replicas location on the edge system. Hence, independent of the number of available nodes, the solver only attempts to find an invocation path between the nodes where the application's microservices reside. To conclude, only the (i) application's size, (ii) number of microservice replicas, and (iii) their location influence the invocation path cycle performance.

Finally, in Figure 7.11, we can observe the invocation path cycle capabilities to restore the application's functionality after node failures. In this scenario, we make use of specific requirements – like the application's availability and node's failure probability – to increase the number of microservice replicas found in the target edge system. With this purpose, we choose the failure probabilities of all nodes under *0.5* and high availability of *0.96*. Furthermore, we increase the nodes' available resources and limit the number of available nodes to 50. In a system with many available nodes (e.g., 500) there are more chances to find nodes with available resources and low failure probability to satisfy the microservices resource requirements and availability. However, when the number of available nodes in the system is limited and very stringent requirements are set, we force the placement cycle to find placement strategies that distribute more replicas in the system. As a result of this scenario, the placement cycle requires more time to find a placement solution. In contrast, we can observe that the invocation path cycle is capable of recovering an application from an unstable state under *70 ms*. Furthermore, recall that the invocation path cycle only considers the edge nodes where microservices are placed (see Table 7.3). Therefore, with every failed node, the required time to find a satisfiable path decreases – the formula size decreases with the number of available microservices found in the system.

Regarding the overall process, observe that we perform adaptation by changing the invocation path between replicated microservices, instead of changing the microservice location using migration techniques. We make this decision because a migration typically requires higher communication overhead as a microservice must be moved from one node to another. For example, a 25 MB container would take 25 seconds to migrate from one location to another, assuming a 1 MB link. In comparison, our approach achieves the same result, i.e., to recover an application, with small communication overhead, since no migrations are involved in the process. Of course, we have to make a trade-off between communication overhead and the increased redundancy due to replication. Depending on the situation, the advocated technique may require significantly more available resources to host replicas. However, we can mitigate this by considering microservice boot/cold starts. On the one hand, if we desire very fast stabilization, then the replicas can be kept warm – a practice that requires more available resources. On the other hand, if we desire to keep more resources available, then we can keep the replicas cold and the latency introduced by cold starts can be considered in the e2e latency computation by the invocation path cycle.

Multiple factors influence the framework's capability to perform stabilization using only the invocation path cycle. The distribution of available microservices on nodes plays the most crucial role since it is not possible to recover an application from an unstable state if there are not enough available microservices in the edge system. A second factor that influences the stabilization process is the application's requirements, i.e., e2e latency and availability. Consider that in the edge system there are still some available microservices required by the application, but the invocation path cycle is unable to find a new satisfiable path due to stringent application requirements and their placement on nodes. For example, the remaining microservices can be placed on nodes that have a very high failure probability making it impossible to satisfy the application's availability or the communication latency between two microservices is too high. As a result, we can observe that with every application disruption we lower the number of possible stabilizations; losing more nodes will eventually lead to one of the two factors previously discussed. In conclusion, if there is at least one available replica for each microservice and both the e2e latency and availability are fulfilled, system stabilization is guaranteed. Finally, we note that the proposed framework is not bound to a specific edge system topology or density and assumes that all nodes are reachable. However, density may hinder the *invocation path cycle's* ability to find invocation paths if edge nodes are scattered in a large area, increasing their communication latency.

We acknowledge the high computational demands of the placement cycle in some scenarios (see the spike of Figure 7.8). However, since placement is performed only at the beginning of the application's lifespan and sporadically during its execution, it does not introduce any extra delays during the application's execution. Of course, this depends if we consider the initial placement as part of the application execution. Moreover, as mentioned above, the placement cycle performance depends on the target edge system and how stringent the application requirements are. The placement cycle is not used for recovery from an unstable state, since for such cases the invocation path cycle is used for adapting to volatility, which can be employed at runtime and is capable of recovering an application timely (under 100 ms for sizes considered in Sec. 7.6).

7.7 Conclusion

In this chapter, we consider that applications execute on distributed hosts where node failure is a prime concern; devices may leave the system or fail without prior notice. As such, the application's stable state needs to be maintained throughout its execution. We proposed an adaptive framework consisting of two MAPE cycles – the placement and invocation path cycles. The former aims at devising a placement to provide the required resources for a microservice-based application. Furthermore, the placement cycle facilitates availability by replicating microservices throughout the system. The latter cycle provides fast recovery from an unstable state by building a satisfying invocation path across deployed microservices and their replicas. The proposed adaptive framework builds upon and improves the resource management techniques presented in Chapter 4 and Chapter 5. The adaptive framework improves the scalability problems that the decentralized resource management technique had, being capable of considering edge systems with hundreds of nodes. Furthermore, it introduces monitoring capabilities, ensuring that the current node's available resources are considered when deploying an application. Finally, as described above, in this chapter, we consider a new application requirement, i.e., availability – a requirement very important in edge computing settings.

CHAPTER 8

Conclusion and Future Work

In this chapter, we present the contributions of this thesis. In Section 8.1, we summarize the main contributions made to the research literature during the course of this thesis. Section 8.2 provides an overview of the research questions introduced in Chapter 1, while Section 8.3 provides an outlook to future work.

8.1 Summary of Contributions

In this thesis, we have tackled the issues related to the migrations of applications from a central location, i.e., cloud, to a decentralized system where resources are shared between multiple nodes. More concretely, we develop novel techniques and methodologies to advance the current state of the art from the perspective of three areas, i.e., (i) application development, (ii) application deployment, and (iii) application management. These three areas play an important role in successfully deploying and managing applications in an edge computing system.

First, we focus on aiding the developer in developing emergent IoT applications to be deployed on the target edge system. For this purpose, we introduce a new methodology to develop and deploy IoT applications at design time. Using our framework, the developer can correctly define and validate application requirements as well as creating its communication flow. To facilitate an application development environment, that allows the developer to create IoT applications without requiring a technical background in computer science, we have extended the FBP paradigm with new timing and resource requirements. The FBP paradigm fits rather well with our purpose since it allows us to combine existing components to achieve the goal of the developed application. As a result, FBP provides a separation between components and application development – some developers may only create components that are shared in the FBP library and used by the other developers in their application. To prove our methodology and present the application development experience, we develop a prototype based on *drawFBP*. The purpose of our methodology is not only to develop an application, but it fulfills a much more important role in the overall application deployment, i.e., with our framework we collect and store as much information as possible regarding the application – information that enables the successful deployment and management of IoT applications in an edge system. In Chapter 6, we further improve on the application model, by providing a new robust IoT application model enabling the efficient utilization of available resources found at the edge of the network. In the research literature, a typical IoT application model is represented with a DAG and consists of atomic components that cannot be further broken during the deployment stage – a model that limits the deployment stage to fully utilize the available resources of edge nodes. In contrast, with our new IoT application model, we offer more component granularity by modeling applications as a collection of composite components. In this case, a composite component can have one or more microservices – we call such a group an aspect. Based on the two IoT application models and the information collected during the application development process, in this thesis, we provide novel resource management techniques and adaptive techniques to deploy and manage an application in an edge system.

Second, we propose novel resource management techniques to deploy IoT applications on the target edge system according to their requirements. In this thesis, we introduce two different resource management techniques, i.e., (i) a resource management technique capable to find feasible or optimal deployment strategies at design time and (ii) a decentralized resource management technique performing application deployment at runtime. With the former technique, we provide the means to validate the defined application's timing and resource requirements according to the target edge system. This provides the developer with valuable information that helps him/her to decide if (i) the application requirements are too stringent or (ii) the target edge system is not suitable for the developed application. Moreover, in this technique, we consider applications with multiple communication flows – flows that the developer can define and assign to them a maximum e2e delay. During the deployment stage, we find a deployment strategy that satisfies all the defined communication flows constraints. With the latter technique, we solve the limitation that the previous resource management technique faced at design time, i.e., it cannot provide deployment strategies for volatile edge systems since the target edge system may change during the deployment process. To overcome this issue, we have proposed a decentralized resource management technique capable to deploy applications at the edge of the network, guaranteeing adherence to (i) defined application's e2e latency and (ii) resource preferences of participating nodes. We focus solely on microservice placement performed by a resource-constrained device, aiming to fully utilizing the available resource found at the edge of the network. In this case, we consider that an application has only one communication flow and the e2e latency considers only the communication latency measured between dependent microservices. By performing the application deployment at runtime, we manage to consider the currently available resources of a node, helping us to cope with changes found in the system. One important contribution is that we account for the node preferences and we thrive to satisfy them. As a result, to capture the node preferences we propose a set of four indicative tactics divided into two decision strategies. However, in our evaluation, we observed that (i) the successful deployment of applications exclusively on resource-constrained devices is still highly dependent on the resource requirements of the smallest entity found in the application model, which in this case is a microservice (or an atomic component if we follow the FBP notation) and (ii) the provided tactics face challenges in ensuring coverage of microservices between individual participant nodes. As a result, we have extended the decentralized resource management technique to be able to deploy our robust IoT application. Furthermore, we propose a new and improved decision strategy that is capable to provide 100% component coverage. As our evaluation shows, with the extended version we manage to deploy applications exclusively on resource-constrained devices, without resorting to the cloud.

Finally, we provide an adaptive technique capable to manage IoT applications in volatile edge systems. We consider that the application's microservices reside on nodes that may fail or leave the system without prior notice. As a result, we require a novel technique to deploy and manage the application, in an edge system, throughout its entire lifespan. A task that we cannot achieve with the previous resource management techniques. Furthermore, with this technique, we manage to ensure the desired application's availability. We proposed an adaptive framework consisting of two MAPE cycles – the placement and invocation path cycles. The former aims at devising a placement to provide the required resources for a microservice-based application. Upon microservice deployment, the placement cycle considers only one application objective, i.e., the availability. Furthermore, the placement cycle finds the minimum number of microservices replicas required to provide application availability considering the current target edge system. With the latter cycle, i.e., the invocation path cycle, we provide fast recovery from an unstable state by finding a satisfying invocation path across deployed microservices and their replicas. This cycle uses as its objectives both the application's e2e latency and availability. Note that by not considering latency as an objective in the placement cycle, we manage to tremendously increase the scalability of our deployment stage – compared to the decentralized resource management where above 20 edge nodes the deployment requires a high execution time to find a deployment strategy, in the placement cycle we can consider more than 500 nodes during the deployment stage. Finally, to prove the feasibility of our adaptive framework, we have built a prototype containing additional important components, such as monitoring of node's status, available resources, and their communication latency.

Notice that, in this thesis, we do not propose solutions for one very important aspect that must be considered when adopting edge computing systems, i.e., security and privacy. By migrating applications closer to the edge of the network, can have a very positive impact on people day to day lives. However, with such benefits arise a set of privacy and security issues that must be solved before we start adopting edge systems. If these issues are not solved, we may end up providing new possibilities to harm the users. One example of a security issue can be seen in smart homes, where one can easily study the behavior of a family by accessing the sensor data. Typically, to evaluate the security and privacy of an edge system we can employ the confidentiality, integrity, and availability

(CIA) trial model [FWKM15] – while a confidentiality and integrity breach represents a data privacy issue, the most important security issues are authentication, access control, and intrusion attacks [YLL15]. Therefore, we can adapt the current security solutions proposed for cloud computing to account for threats that do not exist in its controlled environment [OCY^{+}17]. One solution to securely authenticate edge devices is presented in [PON^{+}18]. A comprehensive study of security threats for edge systems was presented in [RLM18], where the authors motivate the importance of security by looking at the overall system and each component. Regarding privacy, one open challenge that must be solved is to raise privacy awareness among users – currently, almost 80% of WiFi users still use their default passwords for their routers [SCZ^{+}16]. As you can see, this topic alone can be the basis for one PhD thesis. As a result, we consider them as out of scope, but see them as an interesting avenue for future work that must exist in our proposed resource management techniques.

8.2 Revisiting Research Questions

Three main research questions (see Section 1.2) have guided the work presented in this thesis. As a result, to conclude our thesis, we revisit these questions and summarize how these have been answered. Furthermore, we present the possible limitations of our work.

Q1: *What is a suitable programming model and methodology to develop novel microservice-based applications efficiently, providing sufficient information to enable its deployment?*

We have addressed this question in Chapter 4 by introducing EdgeFlow – a methodology for IoT application development and deployment. During the emergent IoT application development process, it is important to collect as much information as possible – information that assists the deployment techniques in finding satisfiable solutions. With EdgeFLow, we extend the FBP paradigm with new timing and resource requirements and offer the possibility to extract all application characteristics into JSON files. Furthermore, we provide a resource management technique capable to validate the application requirements considering the target edge system capabilities. In this chapter, we have highlighted a set of limitations that the current version of EdgeFlow has. First, during the application development stage, we assume that the developer provides the component's resource requirements and their WCET. If the components' resources can be provided by the component developer, finding the WCET is not a trivial task. To find the component's WCET at design time, we must know the current edge node's internal status, i.e., the currently hosted components and the CPU load – information that may be known when the target edge system is in a controlled environment, but we cannot know this information in volatile edge systems. As a result, we can overcome this limitation by developing a decision strategy for our decentralized resource management technique to find the WCET at runtime. Because we do not compute the WCET, a second limitation is that we cannot deploy and develop hard real-time IoT applications, i.e., applications that have strict deadlines that must be satisfied. A third limitation is that we do not have the means to map the component's input and output virtual ports on its host ports. Finally, we perform the application deployment at design time, which may make the technique unsuitable for volatile edge systems.

Q2: *How to efficiently deploy an application on resource-constrained edge nodes, particularly in the absence of cloud resources?*

To address this question, we have proposed a decentralized resource management technical framework in Chapter 5 and Chapter 6. With the proposed framework, we solve the limitation of the previous deployment technique presented above – we perform the deployment stage at runtime to account for possible changes in the edge system. Efficiently using the available resources distributed among multiple resource-constrained nodes, asks for a technique capable to take into account the preferences of each node. Therefore, to capture their preferences, we need to deploy the application at runtime using decentralization. With our technique, we empower nodes to provide their preferences concerning what microservices to host. In Chapter 5, we have observed that our technique has several limitations, i.e., (i) the proposed indicative decision strategies do not provide a high enough microservice coverage and (ii) the DAG application model may impose limitations when deploying the application entirely at the edge of the network – limitations that were overcome in the extension presented in Chapter 6. In the extension, we provide an optimal decision strategy capable of finding nodes' preferences that have the maximum coverage and propose a new robust IoT application model. Finally, there is still a limitation that the SMT and the e2e delay objective impose on the technique. Currently, our decentralized resource management technical framework is suitable for the deployment of applications in edge systems with a limited number of available nodes. From our evaluation, we can observe that after 20 nodes the technique performance suffers, becoming very computational demanding for a resource-constraint node.

Q3: *How to deploy and manage an application in a volatile edge system?*

We have addressed this question by providing the adaptive framework presented in Chapter 7. All the previous resource management techniques cannot provide management of the deployed application throughout its entire execution. Therefore, we propose an adaptive technique, that consists of two MAPE cycles, capable of deploying and managing IoT applications in edge systems with high uncertainty. In this case, our framework consists of a coordinator node that is in charge of the entire system – therefore, we do not perform deployment in a decentralized manner and do not consider the node's preferences. Furthermore, we solve the performance issues faced by our decentralized resource management framework – we have proved in our evaluation that the adaptive framework is capable to consider systems with 500 available nodes. The framework requires novel monitoring components to find the communication latency, the node's status, and the node's available resources among others – latency monitoring is common among all our resource management techniques. We have provided these monitoring components as an example in our prototype, however, we only used basic techniques to implement them since these techniques are out of scope. These areas require future research and more efficient techniques to provide an advantage in our frameworks. To conclude, we will mention one last challenge that must be overcome with future research, i.e., to provide a context-aware technique to be deployed on each participant node in the system. To conclude, to manage the application with no downtime, we will need to be able to react before the actual node has failed. Different context factors could make a node fail or leave the system – monitoring and understanding the context in which nodes operate will allow us to adapt to changes in a predictive manner. In our current framework, we adapt after a node has failed, i.e., we perform a reactive adaptation.

8.3 Future Work

In this thesis, we present a methodology and several frameworks to develop, deploy, and manage IoT applications in a volatile edge system. We address several crucial challenges associated with migrating applications from the cloud closer to the edge of the network. Despite this, there are still several open challenges that were out of the scope of this thesis – we intend to address them in future work. Therefore, in this section, we outline the identified challenges and discuss possibilities for future research.

Context-aware techniques

One promising research direction is to develop a context-aware decision technique that decides when to migrate an application from its current host edge nodes to preserve its functionality. Such a technique will extend the adaptive framework by providing valuable information to the coordinator node regarding the node's internal status and context. The context-aware module will reside on each participant edge node – a module that computes the node's probability to become unsuitable based on the contextual information gathered from its environment. As described in Section 8.2, our adaptive framework is capable of recovering an application from an unstable state in a reactive manner – the framework starts the stabilization process only after one or more nodes failed. An approach that may introduce downtime in the application if the invocation path cycle does not have the required resources to recover the application. In contrast, with our new context-aware module, we can predict when the current microservice host becomes unavailable. At this point, the coordinator finds a new invocation path cycle for the application – preventing any application downtime. Note that an edge node does not have to fail or leave the system to become unsuitable for its local microservices – the context may change, making the host node lose access to some of the resources required by a microservice. For example, consider a machine learning microservice that requires special hardware to run properly. During the deployment stage, we find a location for our microservice that has the microservice required resources. However, during the microservice's lifespan, the node becomes unsuitable to host the machine learning microservice. Under these conditions, the context-aware module informs the coordinator node, which migrates the microservice to another node.

Incentive mechanisms

Recall that an edge system consists of interconnected edge nodes – nodes that must collaborate and share resources to host an application closer to the edge of the network. In a smart city scenario, most of these nodes may be owned by different administrative entities that will not allow the nodes to share resources without being properly rewarded. Furthermore, defining an incentive mechanism capable to reward edge node users that share data and resources will have tremendous effects on the adoption of IoT devices and edge nodes between common users. An effort in this direction is made by the I3: the intelligent IoT integrator, developed by USC [1], having the purpose of creating a marketplace where users can share their private data with application developers and receive incentives for it. We intend to incorporate an incentive system to reward participant nodes for sharing their resources, perhaps by enticing them to use particularly efficient system strategies. In the research literature, we can see some examples of frameworks that consider the collaborations between nodes when deploying an IoT application [ZMZ+18, KKV+17]. However, providing incentives according to edge node participation remains an open challenge – a challenge that may play an important role in the frameworks presented in this thesis.

Extension of our work

In the last part of our future work section, we present the extensions that we intend to do for each presented framework. First, for our EdgeFlow, we intend to provide further extensions to the FBP paradigm, i.e., add the possibility to (i) define QoS requirements and (ii) add privacy and security requirements for each component. Security and privacy play an important role in any edge system. As a result, in our resource management frameworks, we want to enforce security and privacy mechanisms similar to the notion of considering resource preferences of edge nodes. Regarding the decision strategy presented in Chapter 6, we plan to investigate the possibility of finding the required optimal number of groups sent by a node in its preferences list based on the (i) node's available resources, (ii) aspect resource requirements, and (iii) application size. Furthermore, we intend to develop a component ranking, at the node level, to help a collaborator choose what component functionality should be upgraded first in the case when multiple components are mapped on the same node and the available resources have increased. For our adaptive framework, we aim to develop a new module capable of finding nodes' failure probability at runtime; the failure probability of devices evolves in time. Finally, integration with

[1] https://i3.usc.edu/

microservice management frameworks such as OpenFaas or Kubernetes is another aspect that should be further tackled.

List of Figures

List of Tables

Bibliography

[AAY$^+$17] E. Ahmed, A. Ahmed, I. Yaqoob, J. Shuja, A. Gani, M. Imran, and M. Shoaib. Bringing computation closer toward the user network: Is edge computing the solution? *IEEE Communications Magazine*, 55(11):138–144, 2017.

[ADAP19] Marios Avgeris, Dimitrios Dechouniotis, Nikolaos Athanasopoulos, and Symeon Papavassiliou. Adaptive resource allocation for computation offloading: A control-theoretic approach. *ACM Trans. Internet Technol.*, 19(2), April 2019.

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AH15] M. Aazam and E. N. Huh. Dynamic resource provisioning through fog micro datacenter. In *IEEE Intl. Conf. on Pervasive Computing and Communication Workshops*, pages 105–110, March 2015.

[AMD20] Cosmin Avasalcai, Ilir Murturi, and Schahram Dustdar. *Edge and Fog: A Survey, Use Cases, and Future Challenges*, chapter 2, pages 43–65. John Wiley & Sons, Ltd, 2020.

[ASDL19a] Farah AIT SALAHT, Frédéric Desprez, and Adrien Lebre. An overview of service placement problem in Fog and Edge Computing. Research Report RR-9295, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, LYON, France, October 2019.

[ASDL$^+$19b] Farah AIT SALAHT, Frédéric Desprez, Adrien Lebre, Charles Prud'Homme, and Mohamed Abderrahim. Service Placement in Fog Computing Using Constraint Programming. In *SCC 2019 - IEEE International Conference on Services Computing*, pages 19–27, Milan, Italy, July 2019. IEEE.

[ATD19a] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Decentralized resource auctioning for latency-sensitive edge computing. In *IEEE International Conference on Edge Computing (EDGE)*, 2019.

[ATD19b] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Decentralized resource auctioning for latency-sensitive edge computing. In *IEEE International Conference on Edge Computing (EDGE)*, 2019.

[ATD20a] C. Avasalcai, C. Tsigkanos, and S. Dustdar. Adaptive volatile edge systems management at runtime with satisfiability. *ACM Transactions on Internet Technology (under review)*, 2020.

[ATD20b] C. Avasalcai, C. Tsigkanos, and S. Dustdar. Resource management for edge computing services. *IEEE Transactions on Services Computing (under review)*, 2020.

[AZ10] Danilo Ardagna and Li Zhang. *Run-time Models for Self-managing Systems and Applications.* Springer Science & Business Media, 2010.

[AZD20] C. Avasalcai, B. Zarrin, and S. Dustdar. Edgeflow - developing and deploying latency-sensitive iot edge applications. *IEEE Internet of Things Journal (under review)*, 2020.

[AZPD20] C. Avasalcai, B. Zarrin, P. Pop, and S. Dustdar. Efficient hosting of robust iot applications on edge computing platform. In *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, 2020.

[BBG18] T. Bahreini, H. Badri, and D. Grosu. An envy-free auction mechanism for resource allocation in edge computing systems. In *IEEE/ACM Symposium on Edge Computing*, pages 313–322, Oct 2018.

[BCD$^+$11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BF17] A. Brogi and S. Forti. Qos-aware deployment of iot applications through the fog. *IEEE Internet of Things Journal*, 4(5):1185–1192, 2017.

[BKA$^+$20] M. Barzegaran, V. Karagiannis, C. Avasalcai, P. Pop, S. Schulte, and S. Dustdar. Towards extensibility-aware scheduling of industrial applications on fog nodes. In *2020 IEEE International Conference on Edge Computing (EDGE)*, 2020.

[BMNZ14] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. *Fog Computing: A Platform for Internet of Things and Analytics*, pages 169–186. Springer International Publishing, Cham, 2014.

[BSEB19] Martin Breitbach, Dominik Schäfer, Janick Edinger, and Christian Becker. Context-aware data and task placement in edge computing environments.

In *Proceedings of the International Conference on Pervasive Computing and Communications (PerCom). IEEE*, 2019.

[BSM10] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *18th Euromicro Conf. on Parallel, Distributed and Network-based Processing*, pages 27–34, Feb 2010.

[BSPE18] A. Belsa, D. Sarabia-Jacome, C. E. Palau, and M. Esteve. Flow-based programming interoperability solution for iot platform applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 304–309, 2018.

[BT18] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[BWFS14] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. Mobile edge computing: A taxonomy. In *Proc. of the Sixth International Conference on Advances in Future Internet*, pages 48–55. Citeseer, 2014.

[CCP19] Claudio Cicconetti, Marco Conti, and Andrea Passarella. Low-latency distributed computation offloading for pervasive environments. In *Pervasive Computing and Communications (PerCom), 2019 IEEE International Conference on. IEEE*, 2019.

[CCW$^+$19] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu. iraf: A deep reinforcement learning approach for collaborative mobile edge computing iot networks. *IEEE Internet of Things Journal*, 6(4):7011–7024, 2019.

[CZ16] M. Chiang and T. Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, Dec 2016.

[CZP18] X. Cao, J. Zhang, and H. V. Poor. An optimal auction mechanism for mobile edge caching. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 388–399, July 2018.

[DB16] A. V. Dastjerdi and R. Buyya. Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8):112–116, Aug 2016.

[DLC17] Z. Duan, W. Li, and Z. Cai. Distributed auctions for task assignment and scheduling in mobile crowdsensing systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 635–644, June 2017.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Intl. conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DMB19] Vincenzo De Maio and Ivona Brandic. Multi-objective mobile edge provisioning in small cell clouds. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 127–138, New York, NY, USA, 2019. Association for Computing Machinery.

[DPPA18] Nader Daneshfar, Nikolaos Pappas, Valentin Polishchuk, and Vangelis Angelakis. Service allocation in a mobile fog infrastructure under availability and qos constraints. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2018.

[DTF16] Maofei Deng, Hui Tian, and Bo Fan. Fine-granularity based application offloading policy in cloud-enhanced small cell networks. In *IEEE Intl. Conf. on Communications Workshops*, pages 638–643, May 2016.

[EPR20] Raphael Eidenbenz, Yvonne-Anne Pignolet, and Alain Ryser. Latency-aware industrial fog application orchestration with kubernetes. In *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 164–171, 2020.

[FWKM15] Muhammad Umar Farooq, Muhammad Waseem, Anjum Khairi, and Sadia Mazhar. A critical analysis on the security concerns of internet of things (iot). *International Journal of Computer Applications*, 111(7), 2015.

[GBLL15] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung. Developing iot applications in the fog: A distributed dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 155–162, 2015.

[GD18] Marjan Gusev and Schahram Dustdar. Going back to the roots—the evolution of edge computing, an iot perspective. *IEEE Internet Computing*, 22(2):5–15, 2018.

[GJAK19] Keerthana Govindaraj, Jibin P. John, Alexander Artemenko, and Andreas Kirstaedter. Smart resource planning for live migration in edge computing for industrial scenario. In *2019 7th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 30–37, 2019.

[GVC+18] Diogo Gonçalves, Karima Velasquez, Marilia Curado, Luiz Bittencourt, and Edmundo Madeira. Proactive virtual machine migration in fog environments. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00742–00745, 2018.

[GVGB17] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[HLW$^+$20] M. Huang, W. Liu, T. Wang, A. Liu, and S. Zhang. A cloud–mec collaborative task offloading scheme with service orchestration. *IEEE Internet of Things Journal*, 7(7):5792–5805, 2020.

[HV18] Cheol-Ho Hong and Blesson Varghese. Resource management in fog/edge computing: A survey, 2018.

[JT17] R. Jain and S. Tata. Cloud to edge: Distributed deployment of process-aware iot applications. In *IEEE International Conference on Edge Computing*, pages 182–189, June 2017.

[KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KdVW$^+$20] Saadallah Kassir, Gustavo de Veciana, Nannan Wang, Xi Wang, and Paparao Palacharla. Service placement for real-time applications: Rate-adaptation and load-balancing at the network edge. In *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 207–215, 2020.

[KKSF10] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton. Smart objects as building blocks for the internet of things. *IEEE Internet Computing*, 14(1):44–51, 2010.

[KKV$^+$17] A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, and C. Z. Patrikakis. A cooperative fog approach for effective workload balancing. *IEEE Cloud Computing*, 4(2):36–45, March 2017.

[KVRF16] A. M. Khan, X. Vilaça, L. Rodrigues, and F. Freitag. A distributed auctioneer for resource allocation in decentralized systems. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 201–210, June 2016.

[LBDP19] C. Liu, M. Bennis, M. Debbah, and H. V. Poor. Dynamic task offloading and resource allocation for ultra-reliable low-latency edge computing. *IEEE Transactions on Communications*, 67(6):4132–4150, 2019.

[Ley09] Frank Leymann. Cloud computing: The next revolution in it. In *Photogrammetric Week '09*, page 3–12. Wichmann Verlag, 2009.

[LF] J. Lewis and M. Fowler. Microservices. `https://martinfowler.com/articles/microservices.html`. Accessed: 2021-01-26.

[LGJ19] Isaac Lera, Carlos Guerrero, and Carlos Juiz. Availability-aware service placement policy in fog computing based on graph partitions. *IEEE Internet of Things Journal*, 6(2):3641–3651, 2019.

[LYWG20] X. Liu, J. Yu, J. Wang, and Y. Gao. Resource allocation with edge computing in iot networks via machine learning. *IEEE Internet of Things Journal*, 7(4):3415–3426, 2020.

[MATD19] Ilir Murturi, Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Edge-to-edge resource discovery using metadata replication. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–6, May 2019.

[MB18] V. De Maio and I. Brandic. First hop mobile offloading of dag computations. In *18th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*, pages 83–92, May 2018.

[MG+11] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National ..., 2011.

[MJEA19] Amina Mseddi, Wael Jaafar, Halima Elbiaze, and Wessam Ajib. Intelligent resource allocation in dynamic fog computing environments. In *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, pages 1–7, 2019.

[Mor10] J Paul Morrison. *Flow-Based Programming: A new approach to application development.* CreateSpace, 2010.

[MRB18a] Redowan Mahmud, Kotagiri Ramamohanarao, and Rajkumar Buyya. Latency-aware application module management for fog computing environments. *ACM Trans. Internet Technol.*, 19(1), November 2018.

[MRB18b] Redowan Mahmud, Kotagiri Ramamohanarao, and Rajkumar Buyya. Latency-aware application module management for fog computing environments. *ACM Trans. Internet Technol.*, 19(1):9:1–9:21, November 2018.

[New15] Sam Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[NTBG15] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs. Frasad: A framework for model-driven iot application development. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 387–392, 2015.

[OCY+17] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K. R. Choo, and M. Dlodlo. From cloud to fog computing: A review and a conceptual live vm migration framework. *IEEE Access*, 5:8284–8300, 2017.

[PON+18] D. Puthal, M. S. Obaidat, P. Nanda, M. Prasad, S. P. Mohanty, and A. Y. Zomaya. Secure and sustainable load balancing of edge data centers in fog computing. *IEEE Communications Magazine*, 56(5):60–65, May 2018.

[PRZR19] I. Petri, O. Rana, A. R. Zamani, and Y. Rezgui. Edge-cloud orchestration: Strategies for service placement and enactment. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 67–75, June 2019.

[RB19] F. Rao and E. Bertino. Privacy techniques for edge computing systems. *Proceedings of the IEEE*, 107(8):1632–1654, 2019.

[RCP20] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Self-adaptive threshold-based policy for microservices elasticity. In *2020 IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)(to appear)*, 2020.

[RGXZ17] J. Ren, H. Guo, C. Xu, and Y. Zhang. Serving at the edge: A scalable iot architecture based on transparent computing. *IEEE Network*, 31(5):96–105, 2017.

[RLM18] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78:680 – 698, 2018.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming.* Elsevier, 2006.

[RZY$^+$20] W. Rafique, X. Zhao, S. Yu, I. Yaqoob, M. Imran, and W. Dou. An application development framework for internet-of-things service orchestration. *IEEE Internet of Things Journal*, 7(5):4543–4556, 2020.

[SA17] M. M. Shurman and M. K. Aljarah. Collaborative execution of distributed mobile and iot applications running at the edge. In *Intl. Conf. on Electrical and Computing Technologies and Applications*, pages 1–5, Nov 2017.

[SAB$^+$18] Vincenzo Scoca, Atakan Aral, Ivona Brandic, Rocco De Nicola, and Rafael Brundo Uriarte. Scheduling latency-sensitive applications in edge computing. In *Closer*, pages 158–168, 2018.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(June):30–39, Jan 2017.

[SBS$^+$17] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady. Flow-based programming for iot leveraging fog computing. In *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 74–79, 2017.

[SCM18] S. Sarkar, S. Chatterjee, and S. Misra. Assessment of the suitability of fog computing in the context of internet of things. *IEEE Transactions on Cloud Computing*, 6(1):46–59, Jan 2018.

[SCZ$^+$16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SD16] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, May 2016.

[SDW$^+$15] Y. Shi, G. Ding, H. Wang, H. E. Roman, and S. Lu. The fog computing service for healthcare. In *2015 2nd International Symposium on Future Information and Communication Technologies for Ubiquitous HealthCare (Ubi-HealthTech)*, pages 1–5, May 2015.

[SLYZ18] W. Sun, J. Liu, Y. Yue, and H. Zhang. Double auction-based resource allocation for mobile edge computing in industrial internet of things. *IEEE Transactions on Industrial Informatics*, 14(10):4692–4701, Oct 2018.

[SNS$^+$17] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Optimized iot service placement in the fog. *Service Oriented Computing and Applications*, 11(4):427–443, Dec 2017.

[SRP$^+$19] Deepa R. Sangolli, Nagthej M. Ravindrarao, Priyanka C. Patil, Thrishna Palissery, and Kaikai Liu. Enabling high availability edge computing platform. In *2019 7th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 85–92, 2019.

[STD15] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Pringl – a domain-specific language for incentive management in crowdsourcing. *Computer Networks*, 90:14 – 33, 2015. Crowdsourcing.

[TAD19] Christos. Tsigkanos, Cosmin. Avasalcai, and Schahram. Dustdar. Architectural considerations for privacy on the edge. *IEEE Internet Computing*, 23(4):76–83, 2019.

[TGBG20] Christos Tsigkanos, Martin Garriga, Luciano Baresi, and Carlo Ghezzi. Cloud deployment tradeoffs for the analysis of spatially distributed internet of things systems. *ACM Trans. Internet Techn.*, 20(2):17:1–17:23, 2020.

[TN18a] Klervie Toczé and Simin Nadjm-Tehrani. A taxonomy for management and optimization of multiple resources in edge computing. *CoRR*, abs/1801.05610, 2018.

[TN18b] Klervie Toczé and Simin Nadjm-Tehrani. A taxonomy for management and optimization of multiple resources in edge computing. *CoRR*, abs/1801.05610, 2018.

[TND19] Christos Tsigkanos, Stefan Nastic, and Schahram Dustdar. Towards resilient internet of things: Vision, challenges, and research roadmap. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, Texas, July 7-10, 2019*, 2019.

[WD19] P. Wang and X. Du. Qos-aware service selection using an incentive mechanism. *IEEE Transactions on Services Computing*, 12(2):262–275, 2019.

[WIH16] F. Wagner, F. Ishikawa, and S. Honiden. Robust service compositions with functional and location diversity. *IEEE Transactions on Services Computing*, 9(2):277–290, 2016.

[WSMB18] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Fogernetes: Deployment and management of fog computing applications. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2018.

[WZH$^+$20] X. Wang, Z. Zhou, P. Han, T. Meng, G. Sun, and J. Zhai. Edge-stream: a stream processing approach for distributed applications on a hierarchical edge-computing system. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020.

[WZZ$^+$17] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang. A survey on mobile edge networks: Convergence of computing, caching and communications. *IEEE Access*, 5:6757–6779, 2017.

[XJL$^+$19] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv. Edge computing security: State of the art and challenges. *Proceedings of the IEEE*, 107(8):1608–1631, 2019.

[YHQL15] S. Yi, Z. Hao, Z. Qin, and Q. Li. Fog computing: Platform and applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)(HOTWEB)*, volume 00, pages 73–78, Nov. 2015.

[YHZ$^+$17] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[YLL15] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, pages 37–42. ACM, 2015.

[YZW21] Kuang Yuejuan, Luo Zhuojun, and Ouyang Weihao. Task scheduling algorithm based on reliability perception in cloud computing. *Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)*, 14(1):52–58, 2021.

[ZB15] Bahram Zarrin and Hubert Baumeister. Towards separation of concerns in flow-based programming. In *Companion Proceedings of the 14th International Conference on Modularity*, MODULARITY Companion 2015, pages 58–63, New York, NY, USA, 2015. ACM.

[ZBS18] Bahram Zarrin, Hubert Baumeister, and Hessam Sarjoughian. An integrated framework to develop domain-specific languages: Extended case study. In *International Conference on Model-Driven Engineering and Software Development*, pages 159–184. Springer, 2018.

[ZH18] He Zhu and Changcheng Huang. Edgeplace: Availability-aware placement for chained mobile edge applications. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3504, 2018. e3504 ett.3504.

[ZMZ+18] D. Zhang, Y. Ma, C. Zheng, Y. Zhang, X. S. Hu, and D. Wang. Cooperative-competitive task allocation in edge computing for delay-sensitive social sensing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 243–259, Oct 2018.