FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Failure Tolerance for Edge Offloading based on Model Checking

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Eder, BSc
Matrikelnummer 01325852

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Univ. Prof. Dr. Ivona Brandić

Wien, 11. Dezember 2019

Michael Eder	Ivona Brandić

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Failure Tolerance for Edge Offloading based on Model Checking

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Eder, BSc
Registration Number 01325852

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Univ. Prof. Dr. Ivona Brandić

Vienna, 11th December, 2019

Michael Eder	Ivona Brandić

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Michael Eder, BSc
Altenmarktstraße 25, 3683 Yspertal

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

Wien, 11. Dezember 2019

Michael Eder

Danksagung

Zu Beginn bedanke ich mich herzlich bei meiner Betreuerin Dr. Ivona Brandic für ihre Unterstützung während der Literaturrecherche sowie bei der Entwicklung und Analyse des Prototypen. Die regelmäßigen Meetings, die technischen Diskussionen, sowie die fachlichen Anregungen waren für das Voranschreiten der Arbeit wesentlich. Zusätzlich bedanke ich mich bei Atakan Aral für die Bereitstellung der Datensatzsammlung des Institutes.

Ganz herzlich bedanke ich mich auch bei meiner Freundin Bettina, für ihre Unterstützung, ihre Hilfe und motivierenden Worte.

Abschließend gilt mein besonderer Dank meiner Familie und Freunden in Niederösterreich, im Speziellen meinen Eltern Maria und Franz Eder, die mir das Studium an der TU Wien ermöglichten und mich während der gesamten Studienzeit in allen Belangen unterstützten und immer an mich glaubten.

Vielen Dank.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Dr. Ivona Brandic who assisted me during the literature research, the development and the analysis of the prototype as well as the writing of the thesis. Our technical discussions during our regular meetings and her guidance to focus on the most relevant topics have been highly relevant for the writing progress. Additionally, I want to thank Atakan Aral for sharing the institute's dataset repository.

Moreover, I would also like to thank my girlfriend Bettina for listening and motivating as well as for her support and help.

Finally, I am especially grateful to my family and friends in Lower Austria who supported me from the beginning of my studies and always believed in me. Without the support of my parents Maria and Franz Eder, this thesis and the studies at Vienna University of Technology would not have been possible.

Thank you.

Kurzfassung

Die Anzahl der mobilen Endgeräte ist in den letzten Jahren erheblich gestiegen und folglich gibt es eine immer größer werdende Anzahl an komplexen Anwendungen, die mit dem Internet verbunden sind. Daher werden Batterielebensdauer und niedrige Latenzanforderungen immer wichtiger für die Nutzerinnen und Nutzer von mobilen Endgeräten.

In *Mobile Cloud Computing* werden Teile der Applikationen auf einen entfernten Cloud Server transferiert um eine Energieersparnis auf den mobilen Endgeräten zu erreichen. Da die Cloud Computing Datenzentren einerseits meist geographisch weit voneinander entfernt sind, und andererseits auch die Distanz zu der Endanwenderin oder dem Endanwender erheblich sein kann, steigt die Zeitspanne, die für die Übermittlung der Daten nötig ist, enorm. Eine Lösung zur Bewältigung dieses Problems ist *Mobile Edge Computing*, wodurch niedrigere Latenzzeiten und schnellere Antwortzeiten erreicht werden können. Die Entscheidung, ob eine Applikation oder ein Teil davon auf einem entfernten Server ausgeführt wird, hängt von einigen Faktoren ab, wie zum Beispiel einer bestehenden Netzwerkverbindung oder genügend freier Ressourcen am betreffenden Server.

In dieser Arbeit wird die formale Verifikationstechnik *Statistisches Model Checking* verwendet um die Entscheidung, ob eine Aufgabe auf einem Edge Server ausgeführt wird, zu unterstützen. Ebenso sollen dadurch falsche Entscheidungen minimiert werden. Um die Fehlertoleranz zu analysieren wird der bekannte Model Checker Uppaal verwendet. Das Proof of Concept beinhaltet einen *Edge Offloading Prototype*, welcher die Uppaal-Engine verwendet um das Edge Netzwerk zu analysieren. Zusätzlich unterstützt eine Webapplikation das Testen und ermöglicht eine bessere Interaktion mit dem Prototyp.

In der Praxis spielt auch die Mobilität der Benutzerinnen und Benutzer von mobilen Geräten eine entscheidende Rolle. Aus diesem Grund wurden zwei Datensätze aus der Praxis verwendet um den Prototypen zu evaluieren. Dabei zeigt eine worst-case und best-case Analyse in einem Mobile Edge Computing Szenario, dass der Prototyp in der Lage ist, die Entscheidung, ob eine Aufgabe auf einem Edge Server ausgeführt werden soll, präzise vorhersagen kann. Darüber hinaus zeigen die Ergebnisse der Evaluierung, dass es ein vielsprechender Ansatz sowohl für statische Szenarien, als auch Szenarien mit geringer Mobilität der Benutzerinnen und Benutzer ist. Das Proof of Concept ist daher imstande ISPs[1] oder Anbieter von Cloud Diensten bei der Planung ihrer Edge-Infrastruktur zu unterstützen.

[1] Internet Service Provider

Abstract

The number of mobile devices has been rising significantly and consequently a vast number of demanding applications are connected to the internet. Therefore, battery-lifetime and low latency requirements are gaining more and more importance for the end users' experience. In mobile cloud computing parts of applications are offloaded to a remote cloud server in order to save energy on the mobile devices. Concerning geographical distribution of cloud computing data centers and the resulting increase in offloading time, *mobile edge computing* is one solution to overcome these issues. It is proposed to handle this vast amount of devices and connections and to ensure low latency and fast response time. The decision whether an application is offloaded or not depends on several factors including network connectivity and free resources on the remote server.

In this thesis the formal verification technique statistical model checking is used to support the offloading decision and to minimize offloading errors. Based upon the well-known model checker Uppaal, failure tolerance for edge offloading will be analysed. The proof of concept includes the *Edge Offloading Prototype* which uses the Uppaal engine for analyzing the edge network and a state-of-the-art web application to enable convenient testing and interacting with the prototype.

In order to predict and handle the mobility of mobile device users in an edge environment, two real data traces are used for evaluation. The worst-case and best-case analysis show that the prototype is able to predict offloading activity in mobile edge computing accurately. Furthermore, the results show that it is a promising approach for static and low mobility scenarios in mobile edge computing. Therefore, this proof of concept is able to support ISPs[2] or cloud service providers in planning their edge infrastructure.

[2]Internet Service Provider

Contents

CHAPTER 1

Introduction

The number of mobile devices has rapidly increased and the Cisco Visual Networking Index report from February 2019 estimated 12.3 billion mobile-connected devices by 2022. Since the rise of 4G and 5G technologies in combination with edge and cloud computing, the computation offloading frameworks are gaining more and more interest and an efficient combination of these technologies will improve the user's experience a lot. In the Cisco report, previously referred to, more than 3% of the global mobile devices are estimated to be equipped with 5G by 2022 [31]. Furthermore, the smartphone is our daily companion and battery lifetime is an influencing aspect for a big amount of users. In an online survey 92% of the smartphone owners argued that battery lifetime is an important factor when purchasing a new smartphone [58]. Consequently, it is necessary to offer high-performance and energy-efficient services to meet the needs of the smartphone users.

1.1 Problem Definition

In general, smartphones are constrained by limited battery lifetime, limited computational power and limited storage capacity when executing a computational intensive application. In order to extend the battery-lifetime of the mobile device, a suitable solution is to offload the execution and workload to another system with better performance and resources. Therefore, Mobile Cloud Computing (MCC) extends cloud computing with the capability for mobile devices to use offloading techniques.

MCC is defined as a remote infrastructure where the data storage and the data processing of mobile devices is happens in the cloud infrastructure. Computation offloading describes the task to execute a computation intense part of an application on a remote infrastructure, which saves energy and extends the battery lifetime of the mobile device [18].

Nevertheless, computation offloading to cloud servers can cause significant execution delays because cloud servers are usually placed geographically far away and transferring data to the cloud and back is time consuming. This is unsuitable for many applications, such as real-time applications, and as a solution Mobile Edge Computing (MEC) has been introduced. In MEC the computational power and the storage resources are placed on the edge of the mobile network in order to reduce the execution delay of computation offloading.

Figure 1.1 shows an example of an edge network with different mobile devices connected that can offload their tasks to the respective edge node. In recent years different computation offloading technologies have arisen, mostly considering cloud infrastructure. The majority of the proposed frameworks are focusing on the offloading decision which is mostly done at runtime (e.g., MAUI [35]), but they aren't considering the available computational resources and storage capacity at the remote infrastructure and the overall cloud/edge system. Orchestration platforms offered by big cloud computing service providers (e.g., AWS, Google Cloud Platform, etc.) can benefit from such a system to provide their services more efficiently (e.g., to adapt the deployment plan).

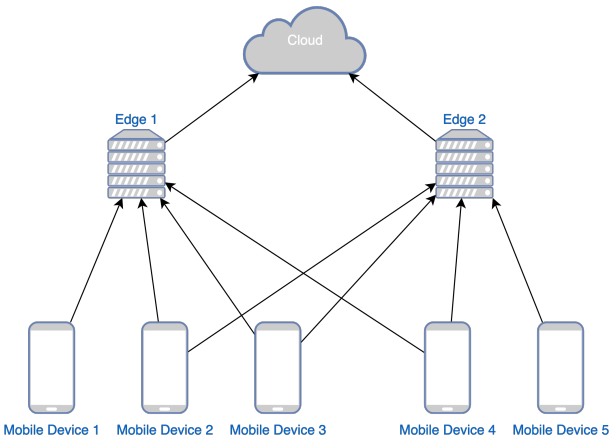


Figure 1.1: Example for an edge network with mobile devices connected

In a distributed environment with lots of mobile devices, edge networks are implemented to reduce bandwidth and execution time latency. An open problem in such an environment is how to use the individual edge components for computation offloading in order to optimize the use of available resources considering not only the decisions for offloading taken at runtime, but also the design of the overall edge network. It is important to note that an efficient utilization of the edge nodes is energy-efficient and cost-saving.

Due to the dynamic and volatile nature of Mobile Edge networks, it is challenging to optimize the effective computational resources needed on an edge node. If a task is offloaded to an edge node, which has currently no computational resources available, an offloading error will occur. In this case the task has to be offloaded to another edge

node or cloud server or the task has to be executed locally. The question how to support an offloading framework in order to minimize offloading errors will be evaluated in this thesis.

1.2 Approach and Contribution

In general, due to the mobility of the users and devices there is an inherent high degree of uncertainty in offloading activity and in the overall edge network. However, for providing resources in the cloud or edge in an efficient way, it is necessary to analyse and quantify the uncertainty. Moreover, current offloading optimization approaches are considering a single or multi-objective optimization problem regarding application run time and energy consumption[17][18]. Nevertheless, verifying the correctness of the offloading decision and an efficient resource allocation is not considered.

In order to address the above mentioned challenges, in this thesis a prototype based on the formal verification method, called model checking, is introduced. In practice, the existing edge infrastructure is large and due to the complexity of such systems, the state space explosion problem in model checking occurs. Additionally, verification with model checking is extremely time consuming and computational intense with such systems[22].

To overcome these issues, Statistical Model checking (SMC) is proposed to avoid the exhaustive exploration and to result a probability if a certain property is satisfied or not. For the formal verification including the SMC analysis, the well-know model checker Uppaal is used. This model checker is used because other projects in mobile offloading and edge computing are also utilizing Uppaal and the support by the community and the direct communication with the developers are major reasons for considering Uppaal in this thesis[16][12][13].

The methodological approach to reach the expected results includes three parts: *Mobile Edge Computing*, *Statistical Model Checking* and *Prototyping and Evaluation*. The given edge infrastructure is analyzed with statistical model checking in order to predict offloading activity. Therefore, a prototype is implemented for predicting future workloads and offloading errors in MEC. To show the weaknesses and strengths of the proof of concept, the prototype is evaluated with real data trace from the cloud and edge computing area including users' mobility. Furthermore, a sensitivity analysis and an execution time analysis are performed and a detailed analysis of the main three parts is following below.

Mobile Edge Computing

Computation offloading is an enabling technology in cloud and edge computing. It facilitates better performance and energy-efficiency for mobile device users. In the last few years a huge number of approaches for efficient computation offloading has been proposed. In this thesis a proof of concept shows the suitability of the formal verification technique model checking for computation offloading in MEC.

Therefore, the given or generated edge networks are converted to a timed automata which is a formal model. Such timed automata provide a formal framework for modeling and analyzing the behavior of real-time systems [68]. Model checking is used to formally analyze the given model with the help of temporal logic.

Statistical Model Checking (SMC)
The main idea of statistical model checking is to monitor simulations of the given system. With the help of monte carlo simulation and hypothesis testing, the results of the simulations can decide whether the system satisfies a property with certain degree of confidence. It is a compromise of testing and classical model checking and is used beside software engineering also in other research domains (e.g., automotive or avionics engineering) [54]. SMC is employed in this thesis to predict the offloading activity of the connected mobile devices.

Prototyping and Evaluation
The prototype is implemented with state-of-the-art technology and can handle typical workload scenarios commonly proposed in cloud computing environments [43], which can also be applied to edge computing scenarios as well. Therefore, synthetically generated static, periodic and unpredictable workload trace can be used with the prototype. Due to the novelty of this proof of concept, the well-known method rapid prototyping[44] is used. The iterative process in rapid prototyping is reasonable, because the design, modify and review procedure leads to the expected results and changing requirements can be better mastered. To test the feasibility of the proof of concept, an evaluation with real-data traces and synthetically generated data is performed. Additionally, the evaluation focuses on the question to what extent the prototype can predict offloading errors in a MEC environment.

This thesis introduces a novel approach to predict offloading activity in MEC with the support of (statistical) model checking. The primary contributions of this thesis can be summarized as follows:

- The prototype predicts offloading activity in MEC and minimizes offloading errors based on a statistical model checking approach with Uppaal SMC[37]. The edge infrastructure is converted in different steps to a network of stochastic timed automata (NSTA). Afterwards, a probability comparison is performed in order to analyze interesting properties, such as the average number of edge offloadings in a certain scenario. This NSTA probability comparison is a feature of Uppaal SMC and is integrated in the prototype.

- In designing process of the proof of concept the three main properties scalability, extensibility and simplicity are defined for the prototype. In order to fulfill these design principles the prototype is able to operate with three different workload trace strategies (static, periodic and unpredictable) and three level of computation intensity of the considered applications (low, medium, high).

- The input model is representing an edge infrastructure network and an interaction with the prototype through a graphical user interface is also possible. The prototype is composed of two components. First, the *Edge Offloading Prototype* including a REST-API and the model checker engine of UPPAAL. Second, the *Offloading-Client* consists of an Angular web frontend for a straightforward interaction with the Edge Offloading Prototype.

- The evaluation is done with real-data traces in order to show the efficiency of the proof of concept. Thus, real-data traces from the RICE university (LiveLab dataset[61]) and the EUA dataset from Lai et al.[50] are used. The users' mobility is also taken into account by implementing mobility prediction with a POI[1] based human approach[42]. The results indicate that the offloading errors will increase, if the mobility of the users rises. Nevertheless, the results show that the prototype works well in static scenarios and scenarios with low mobility.

1.3 Structure of the Thesis

This thesis is structured as follows. In Chapter 2 different concepts relevant for this thesis are discussed. The most important concepts are cloud and edge computing, Mobile Cloud Computing followed by a comprehensive analysis of Model Checking with a special focus on the tools PRISM and UPPAAL.

In Chapter 3 the current approaches in mobile cloud and edge computing are reviewed. The different offloading frameworks in cloud computing and the approaches focusing on offloading in mobile edge computing environments are analyzed. However, also offloading approaches with model checking support are compared and contrasted.

Chapter 4 focuses on the proof of concept and outlines the design and the implementation solution of the Edge Offloading Prototype. Additionally, also the details of the web-browser application for interacting with the prototype are described.

After analysing the solution, Chapter 5 contains the evaluation of the prototype including the experimental setup, the used mobility pattern and results of the evaluation.

Finally, in Chapter 6 the findings of this thesis are summarized and future work is outlined.

[1] Point of Interest

CHAPTER 2

Background

In this chapter the main technologies and techniques, which are most relevant for this thesis, are briefly described. Initially, the terms *Cloud Computing*, *Edge Computing* and *Mobile Cloud Computing* are defined and explained. Secondly, a detailed discussion of *Model Checking* and the answer to the question, how can it support *Mobile Edge Offloading*, is outlined. Moreover, the decision for using UPPAAL as the model checker for this thesis will be explained and justified in Section 2.4.

2.1 Cloud Computing

In the last 10 years the concept of *Cloud Computing* has become one of the most important areas in distributed systems. In 2011, the NIST[1] published a definition of cloud computing. They outlined that this technology is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. These computing resources can be rapidly provisioned and released with minimal management effort or service provider interaction [53].

The NIST cloud model consists of five essential characteristics, three service models and four deployment models. First, the five essential characteristics are described.

2.1.1 Cloud Characteristics

The five essential characteristics of cloud computing described in [53] are *on-demand self-service*, *broad network access*, *resource pooling*, *rapid elasticity* and *measured services*.

On-demand self-service

The needed computing capabilities (e.g., server time, network storage etc.) by the

[1] National Institute of Standards and Technology

customer are allocated and provisioned automatically without human interaction. The computing capabilities are allocated by the current needs of the customers.

Broad network access
The whole cloud capabilities are available over the network and can be accessed by standard and well-known mechanisms. Due to this fact various devices can access the cloud platform, for example, mobile phones, notebooks, workstations, IoT devices etc.).

Resource pooling
One of the key characteristics of cloud computing is the multi-tenant model serving multiple customers enabled by pooling the capabilities. The multi-tenant model has different physical and virtual resources that are assigned in accordance with customers' demand. Due to the higher level of abstraction, customers do not know exactly where the data is stored or processed. However, location constraint features exist by different cloud platform providers, for instance, AWS (Amazon Web Services) regions[2]).

Rapid elasticity
Another key characteristic is elasticity and scalability. From the customer's perspective, the available capabilities appear to be unlimited by elastic provisioning and releasing of cloud capabilities.

Measured services
The resource usage can be monitored, controlled and reported. Therefore, both, the customer and the cloud service provider have knowledge of the exact resource usage and a pay-per-use pricing model is possible.

2.1.2 Service Models

Beside these five cloud characteristics the NIST described in [53], there are three service models which are commonly used in research and industry. In recent years, a trend to offer *everything as a service (XaaS)* [40] can be observed in cloud computing. One prominent example of the serverless computing area is *function as a service* (FaaS)[29] offered by all major cloud computing platforms (e.g., AWS Lambda[1], Google Cloud Functions[8], Azure Functions[11])

Software as a Service (SaaS)
The applications are running on a cloud infrastructure and consumers can use various features of them. The accessibility of these applications can be different (e.g., web browser applications etc.). Nevertheless, the underlying cloud infrastructure can not be managed or controlled by the consumer. Normally, only specific application configuration settings can be managed by the consumer. Well-Known SaaS providers are Salesforce, G Suite or Microsoft Office 365 (see Figure 2.1).

Platform as a Service (PaaS)

If PaaS is used, it well be possible to deploy consumer-created or other compatible applications onto the cloud infrastructure. To the same extent as SaaS, it is not intended to manage or control the underlying cloud infrastructure, but the consumer is able to control the deployed applications and can modify or configure them. Prominent PaaS providers are AWS Elastic Beanstalk, Microsoft Azure App Services, Google App Engine and OpenShift[14].

Infrastructure as a Service (IaaS)

The consumer can use computing resources, for instance, processing, storage and network provisioned by the IaaS provider. On the provisioned resources, the consumer can install various software including operating systems and arbitrary applications. Nevertheless, even in this service model the consumer cannot manage or control the underlying cloud infrastructure, however, controlling the operating system, storage and applications is possible. Also the configuration of selected networking features is possible (e.g., firewalls). Furthermore, the three main players in IaaS are Amazon (EC2), Google (Compute Engine) and Microsoft (Azure Virtual Machine) [14].

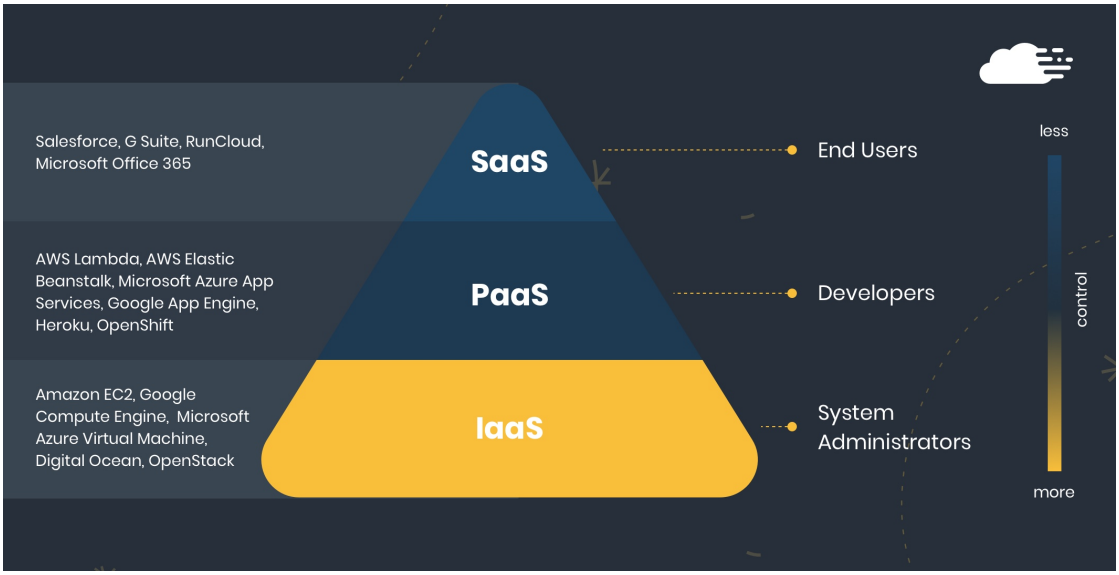


Figure 2.1: Overview and application of the three cloud service models [14]

In Figure 2.1 the three service models are summarized. I illustrates that in SaaS the control of the cloud infrastructure is the lowest and in IaaS highest, because configuration and management of operation system, storage and applications is possible. SaaS is mainly used by end users, whereas PaaS is used by developers. Finally, IaaS is normally used by system administrators [14].

2.1.3 Deployment Models

Furthermore, the NIST also described four deployment models in cloud computing which are briefly described in the next listing.

Private Cloud
The cloud is only available to a single organisation. The cloud can be operated by the organisation itself, but also a third party can manage and operate the private cloud. The private cloud can exist on- and off-premise.

Community Cloud
The access to community cloud is restricted to specific customers that have shared issues. The cloud can be operated by different providers (e.g., one or more organisations, third-party company) and it persists on- or off-premise.

Public Cloud
The cloud is generally public available and it exists on the premise of the cloud provider. The billing for a public cloud is usually pay-as-you-go [21].

Hybrid Cloud
It is a mixture of two or more other mentioned cloud infrastructures (private, community, public and hybrid. The individual clouds run independently but it is possible to share data and applications between the individual clouds.

2.2 Edge Computing

In the last Cisco Visual Networking Index report from February 2019, 28.5 billion networked devices by 2022 (up from 18 billion in 2017) are predicted. Especially, the number of mobile devices will grow from 8.6 billion to 12.3 billion mobile-connected devices by 2022. This number exceeds the world's population (8 billion are predicted for 2022) by more than one and a half times. The Internet of Things (IoT) heralds the post cloud era and the global M2M connections will grow from 6.1 billion in 2017 to 14.6 billion by 2022. This means there will be 1.8 M2M connections for every person on this planet by 2022 [31].

Due to this increasing number of connecting devices, edge computing is proposed to handle this vast amount of connections and to ensure low latency and fast response time. With edge computing it is possible to allow computation to be performed at the network edge. Therefore, the computation happens close to the device or data source. An edge device or edge server is any computing or networking resource which is placed between the data sources and the cloud. Examples for edge servers are cloudlets or micro data centers. As demonstrated in Figure 2.2, the edge components are located between the devices and the cloud [62][63].

One of the benefits of using edge computing is to offload computing tasks from the sensors or devices (e.g., wearable cognitive-assistance systems) to the edge components. This

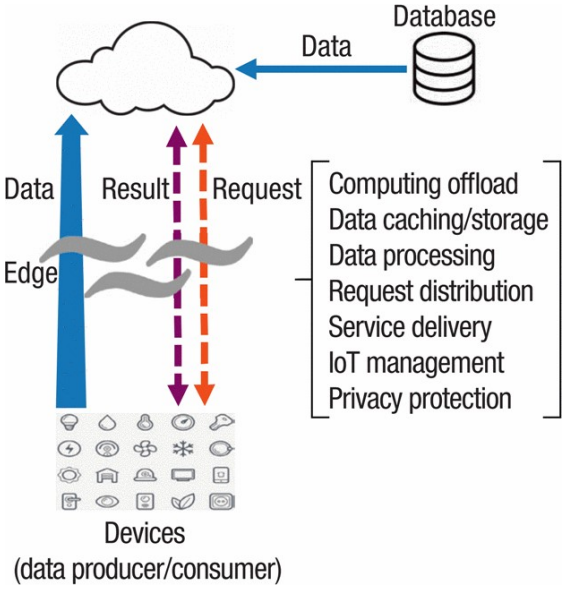


Figure 2.2: Edge computing overview [63]

improves the response times between 80 and 200 ms and also the energy consumption can be reduced from 30 to 40 percent compared to conventional cloud offloading [45].

Obviously, there are several points of concern when implementing edge computing. These are system reliability (e.g., wireless connection problems etc.), security and privacy concerns.

In order to succeed with edge computing, a killer application is needed (e.g., smart connected vehicles) and Shi and Dustdar[63] mentioned the following issues that need to be addressed in edge computing.

Programmability

It is important to offer applications which can partition (see Section 2.3.2) their features and functions, since only then offloading of certain parts to edge servers is possible. There are many prototypes and research frameworks available (see Chapter 3), but a well established programming framework and other useful tools do not exist.

Naming

Other important points to focus on are the heterogeneous devices and the proprietary network protocols. In order to implement an edge offloading framework, various network protocols and other heterogeneous elements have to be considered. Moreover, a uniform naming and communication model can handle the mobility of users and also improves scalability.

Privacy and Security

In edge computing also security and privacy concerns need to addressed. In a smart home environment a lot of sensor data are processed and a hacker can analyze the network traffic to determine whether a house is inhabited or not. Therefore, efficient tools are necessary to protect data and privacy in an edge network.

2.3 Mobile Cloud Computing

Mobile Cloud Computing (MCC) combines mobile computing and cloud computing. MCC is defined by the Mobile Cloud Computing Forum as "Mobile cloud computing at its simplest, refers to an infrastructure where both the data storage and data processing happen outside of the mobile device. Mobile cloud applications move the computing power and data storage away from mobile phones and into the cloud, bringing applications and mobile computing to not just smartphone users but a much broader range of mobile subscribers."[39].

As stated above, MCC integrates mobile computing into cloud computing and, therefore, helping to make mobile computing more efficient. Dinh et al. specifies the main advantages of mobile cloud computing that are given below [39].

Extending battery lifetime

In an online survey 92% of the smartphone owners said that battery lifetime is an important factor when purchasing a new smartphone [58]. It is, therefore, important to extend the battery lifetime and the computation offloading approach can help to save battery energy. The idea is to execute complex processing and large computations, not only on the resource-limited (mobile) device, but also on powerful machines elsewhere (e.g., servers in the cloud or on the edge). On the one hand, this can help to reduce the overall execution time on the mobile device and, on the other hand, the power consumption on the mobile devices is reduced.

Improving data storage capacity and processing power

Apart from computation offloading, it also possible to store and access large files on the cloud which are accessible through wireless networks. Due to the fact that these files are only stored and processed in the cloud, a considerable amount of energy and storage space can be saved. A further advantage of using MCC is that there is no storage capacity limit on the devices, because the data is stored remotely on the cloud.

Improving reliability

In MCC the data and applications are stored on multiple servers distributed across the cloud. Due to these backups, the risk of loosing data at the end device is minimized. Additionally, MCC can offer a broad data security model, because the cloud can protect digital content from being abused or unauthorized access (e.g., copyrighted digital contents) [73].

MCC can be implemented in lots of practical applications. The areas of applications range from mobile commerce, mobile learning, mobile healthcare to mobile gaming [39].

However, the integration of mobile computing and cloud computing also leads to several technical and organizational issues which need to be addressed. First of all, the low bandwidth in wireless networks can cause unforeseeable behaviour or unavailability of services. If applications or services use MCC technologies, a network connection will be required. Similar to the issues in edge computing, discussed in Section 2.2, also in MCC the heterogeneity of the different wireless network interfaces is difficult to handle. The efficient implementation of the computation offloading feature in MCC is important to reduce offloading errors and other problems. Especially dynamic offloading in a mobile changing environment has to be considered to offer efficient and battery lifetime extending services. Finally, the latency of computation offloading can be challenging because cloud servers are usually physically far away and long latency reduces the usefulness of MCC [39][18].

2.3.1 Mobile Edge Computing

To overcome the issue of long latency related problems in MCC, edge computing addresses these issues. Therefore, the servers are placed at the edge of the mobile network in order to be in the proximity of the end user's location.

To gain a better understanding of the differences between MCC and Mobile Edge Computing (MEC), Table 2.1 considers various technical aspects and compares and contrasts them. Firstly, the deployment in MCC is centralized, while in MEC it is distributed. Furthermore, the distance to the user equipment (UE), latency and jitter are higher in MCC than in MEC. Finally, the computational power and storage capacity is ample in MCC compared to MEC where it is limited [52].

Technical aspect	MCC	MEC
Deployment	Centralized	Distributed
Distance to UE	High	Low
Latency	High	Low
Jitter	High	Low
Computational power	Ample	Limited
Storage capacity	Ample	Limited

Table 2.1: Comparison of MCC and MEC [52]

In Mobile Edge Cloud Computing (MECO), offloading to both, cloud and edge servers, is possible, whereas edge servers are in geographic proximity to the user's mobile device. Such a hybrid approach between MCC and MEC can reduce the application runtime by up to 70.7% in comparison to MCC [38].

2.3.2 Mobile Offloading

In general, the mobile offloading process is characterized by the ability to offload a mobile application (or a part of it) to a remote infrastructure. The *offloading engine* is running on the mobile device. As illustrated in Figure 2.3, the offloading engine is responsible for deciding whether a task is offloaded or not. The *application profiler* is responsible for profiling the application and to extract the structure of the application, especially the parts which are offloadable. The *system monitoring* monitors and analyzes the data of the remote infrastructure and the *decision engine* receives the input from the application profiler, the system monitoring as well as the user preferences and performs the offloading decisions.

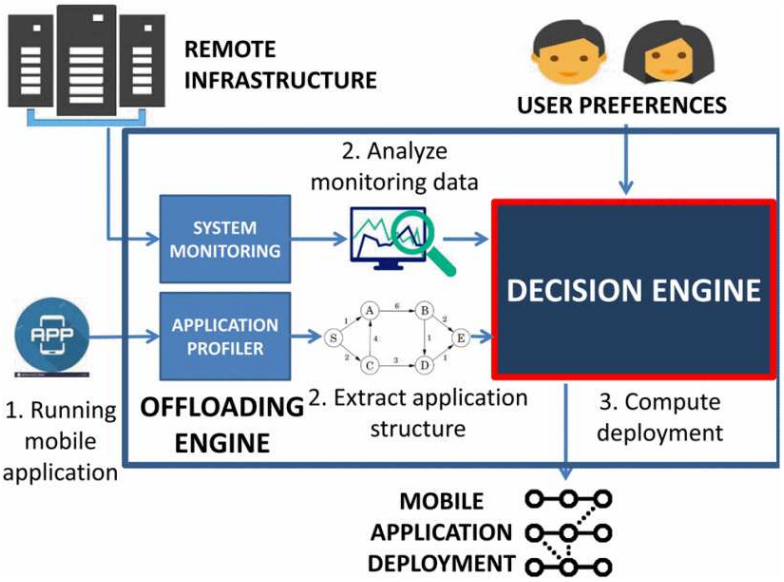


Figure 2.3: Offloading model [38]

Akherfi et al.[18] defines three important steps (*Application partitioning*, *Preparation* and *Offloading decision*) which are necessary to implement offloading. These steps are implemented differently in various frameworks and a detailed overview is described in Chapter 3.

Application partitioning

In a first step the application is analyzed regarding the parts which are offloadable or non-offloadable. This means that application has parts which cannot be executed remotely because of different reasons (e.g., GUI, camera etc.). Furthermore, it is possible that developers can annotate methods or components which are offloadable or not, if the offloading framework offers such a feature.

Preparation

In order to execute parts of the application on a remote server, the following aspects have to be taken into account:

- Selection of the remote servers
- Transfer and install the application's source code
- Proxy processes responsible for receiving and executing tasks
- Transfer other data which are necessary for the remote execution

Offloading decision

The offloading decision can be taken at runtime or it is predefined at design time of the application. If the decision is taken at runtime, various aspects have to be taken into consideration (e.g., wireless connections, energy consumption etc.). Obviously, if the situation changes, also the offloading decision must be reconsidered. In Figure 2.4 the different aspects affecting the offloading decisions are depicted including *User Preferences*, *Network Specifications*, *Mobile Specifications*, *Application Specifications* and *Server Specifications*.

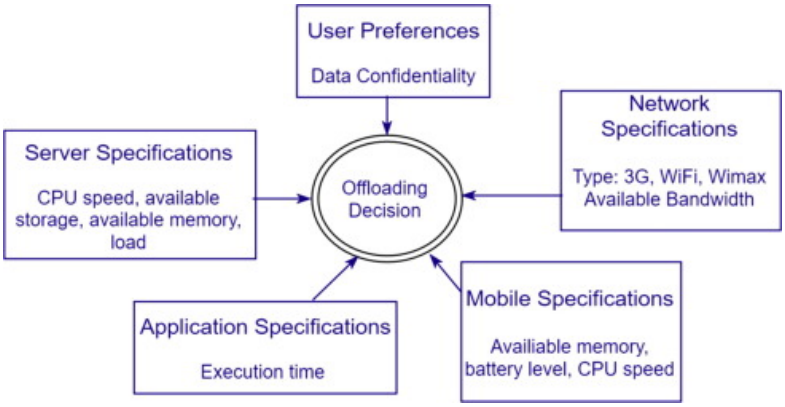


Figure 2.4: Aspects affecting the offloading decision[18]

There are two types of offloading frameworks, namely static offloading frameworks (offloading decision is taken at design time) and dynamic offloading frameworks (offloading decision is taken at runtime). Also the offloading mechanisms differ and it is suitable to distinguish between frameworks which are based on virtual machine cloning and code offloading frameworks [18].

In [52] it is defined that a computation offloading decision can result in three different actions.

Local execution - The execution of the application is done locally because offloading is not possible (e.g., no network connection etc.)

Full offloading - The whole execution of the application is offloaded and processed by the cloud or edge servers.

Partial offloading - A part of the application execution is offloaded and processed by the cloud or edge servers.

2.4 Model Checking

In this section the term *Model Checking* is analyzed, discussed and evaluated. The reason for this detailed overview is that Model Checking and in particular Statistical Model Checking is the key technology for predicting offloading activity by means of the proposed framework.

In general, Model Checking is a formal verification technique and it can be applied in many different fields of application. In recent history, bugs in software systems caused disastrous accidents with million of dollars damage costs. For example, the Ariane 5 rocket was unexpectedly destroyed only seconds after launching because of a data conversion error from a 64-bit floating point into a 16-bit integer value. This conversion error caused an overflow resulting in a hardware exception. This example shows the importance of formal verification techniques, such as, Model Checking in order to identify such problems and alleviate unexpected behaviour of software systems [27].

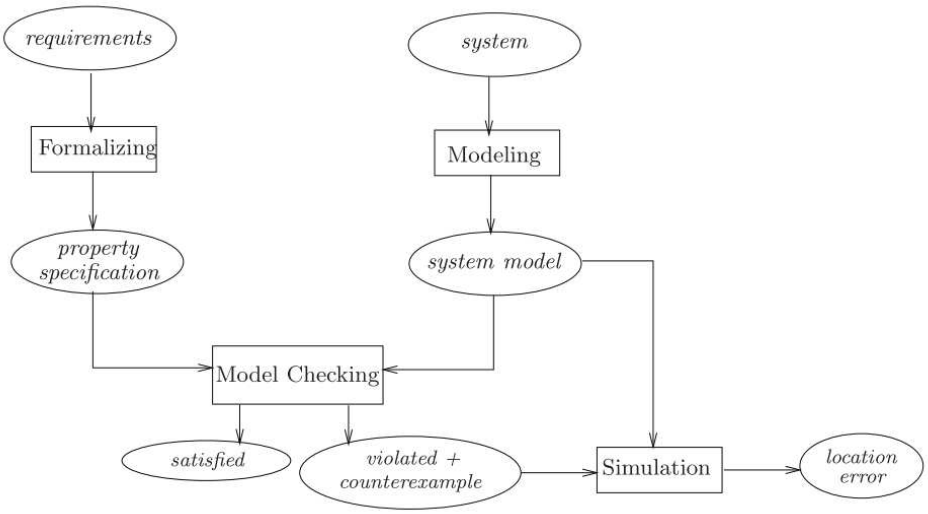


Figure 2.5: Schematic overview of the model checking approach [22]

For verifying finite-state reactive systems, model checking is an automatic technique to ensure that such systems satisfy the given specifications which are expressed in temporal logic [32]. The system is modeled as a state-transition graph, the states of the system can be visualized with a finite-state model. As this technology was developed in 1981 by

Clarke and Emerson [33], also other similar approaches existed. Nonetheless, the main advantage of model checking is that it is highly automatic. A model checker can either terminate with *true*, if the given model satisfies the specification or it terminates with a counterexample showing that the given specification is not satisfied.

As visualized in Figure 2.5, the *requirements* of a system have to be expressed in a formal way (e.g., temporal logic like LTL [2] formulas). The system model is generated by a model description. This model description can be written in certain software engineering languages like, C or Java, or hardware description languages like, Verilog or VHDL. The property specification describes *what* the system should do and the system model characterizes *how* the system behaves. Basically, the model checker examines all the possible system states in order to check possible transitions or moves. Due to the brute-force behavior of the model checker, it is evident that state space becomes very large with respect to the number of states in a system. After examining all system states, it is possible to check whether a property is satisfied or not. In the case that a property does not satisfy the model checker, it provides a counterexample which describes the path to the states violating the property. Another feature of a model checker is to simulate the scenario for the purpose of identifying the error in the model. Afterwards, the model or the property can be adapted to meet the predefined specifications.

In practice, the models to be verified are very large and due to the complexity of such systems, the state space explosion problem in model checking makes it difficult to handle and often these problems are undecidable. The number of states can be huge because the needed space is exponential regarding the number of processes and the number of components per process. In order to prevent, respectively, to mitigate the state explosion problem, different approaches exist (e.g., bounded model checking) and it is an ongoing and highly active research area [22].

2.4.1 Temporal Logic

The specifications or the properties which define a software system can be expressed in temporal logic. One possible question or specification is, if we start at an initial state s_1, will it be possible to reach an error state e_1? To formally describe this kind of question, *temporal operators* are used. The temporal logic is divided into several different logics [34]. This is merely a fraction of all proposed logics, but these are the most relevant ones for this work:

Linear Temporal Logic (LTL)
The formulas in LTL specify the events along a single computation path (linear-time logic).

Computation Tree Logic (CTL)
The formulas in CTL specify all possible paths from a given state (branching-time logic).

[2] Linear Temporal Logic

CTL*

CTL* subsumes both LTL and CTL. This logic combines branching-time and linear-time operators.

Timed Computation Tree Logic (TCTL)

TCTL is a sublogic of CTL and in contrast to CTL, time constraints are obeyed to verify timed automata models. TCTL is important for this thesis because the model checker Uppaal (see Section 2.4.4) uses a restricted version of TCTL as query language [24].

2.4.2 Statistical Model Checking

The main idea of Statistical Model Checking (SMC) is to monitor several runs of the system and with the help of different statistical algorithms to estimate the correctness of the overall design. As mentioned above, in practice these models are large, complex cyber physical systems with a lot of states and this will cause in state(-space) explosion problem in common Model Checking. To overcome this issue, SMC is proposed to avoid the exhaustive exploration and to result a probability if a certain property is satisfied or not.

SMC is used in different areas of application and it is divided into numerical and statistical approaches. If the numerical approach is applied, the exact measure of paths satisfying subformulas is repetitively computed. This variant of a numeric approach is often called probabilistic model checking and there are well-known tools that support this approach (e.g, PRISM) [48]. A comprehensive analysis of the model-checker PRISM is outlined in Section 2.4.3.

Beside the numerical approach, the statistical approach simulates the system with finitely many executions and captures the results accordingly. With the help of hypothesis testing, the approach provides statistical evidence, if the specification is satisfied or violated.

The statistical approach has several advantages compared to the numerical approach. The SMC approach can be applied to larger systems, because it is necessary to execute sample runs of the systems. Another advantage is that the process is easy to parallelize because the sample runs can be executed independently. One of the main disadvantages of this approach is that the sample size will be enormous, if the answer of the model checker has to be highly accurate. Moreover, the result of the SMC analysis will only provide probabilistic guarantee correctness in contrast to numerical model checking. For the application of fault tolerance and supporting an offloading decision engine, it is not absolutely necessary to provide exactly correct results for the given model. In practice, the models are changing often and the results are only a forecast and approximation [51].

The guiding principle of SMC is to observe and monitor executions of the systems and with the help of hypothesis testing and classic Monte Carlo simulations, a statistical evidence for satisfaction or violation of the specification is calculated [37]. A prominent

representative of this approach is UPPAAL with the SMC extension which has been available since version 4.1.4 [3].

In order to get a proficient overview between the different approaches, the above mentioned tools PRISM and UPPAAL are analysed in detail and the advantages and disadvantages are outlined. Both model checkers are formal verification tools for timed automata.

UPPAAL and PRISM are developed with a combination of C++ and Java. Both tools are useable for free and open source under the GNU GPL. In [55] there is also a comprehensive summary and comparison of UPPAAL and PRISM available.

2.4.3 PRISM

The model checker PRISM[4] was developed in 1999 by a joint project of the University of Birmingham and the University of Oxford. The latest version of PRISM is 4.5 published on the 19th April 2019. PRISM is a probabilistic model checker and includes following probabilistic models:

- Discrete-time Markov chains (DTMCs)
- Continuous-time Markov chains (CTMCs)
- Markov decision processes (MDPs)
- Probabilistic extension of timed automata (PTAs)

As input language PRISM uses its own language called PEPA and as query language for verification PRISM supports the following temporal logics:

- **CSL** (Continuous Stochastic Logic)
- **LTL** (Linear Temporal Logic)
- **PLTL** (Probabilistic Linear Temporal Logic)
- **PCTL** (Probabilistic Computation Tree Logic)

With PRISM 4, the implementation and verification of *probabilistic timed automata* (PTA) are introduced. PTA are finite-state automata with real-valued clocks and discrete probabilistic choice. Furthermore, PTA can be modeled using a textual language based on guarded command notation. PRISM offers also PTA verification techniques where two main classes of properties can be anlayzed. Firstly, the minimum/maximum probability of reaching a target (e.g., the maximum probability of offloading to a certain

[3] http://www.uppaal.org/
[4] https://www.prismmodelchecker.org/

edge server within 2 seconds). Secondly, the minimum/maximum expected reward accumulated until a target is reached (e.g., the minimum expected time for an edge server to reach its resource limits). There are two verification methods implemented, whereas *quantitative abstraction refinement* is the default engine in PRISM [49].

2.4.4 Uppaal

Uppaal was first presented as a model checker for real time systems in 1995 which had been 4 years before PRISM was published. It was developed as a cooperation between the universities of **Upp**sala and **Aal**borg. It is constantly evolving and the program has an active community that is supporting users with handling problems or fixing bugs relatively fast [12]. The Uppaal toolkit has also useful extensions such as Uppaal CORA (Cost Reachability Analysis), Uppaal Port (Component based development) and Uppaal TIGA (Timed Gamed Automata) and Uppaal SMC (Statistical Model Checking). This program uses timed automata for modeling and verifying with the model checker engine. As input language it uses XTA and XML and as query language for verification a subset of Computation Tree Logic (CTL) called TCTL is used. A more detailed evaluation of temporal logic is provided in Section 2.4.1.

In this thesis one major aspect is the Statistical Model Checking approach of this model checker which is implemented within Uppaal SMC.

As explained in Section 2.4.2, statistical model checking basically monitors several executions and with the help of hypothesis testing, a probability for the given specifications is calculated.

In Figure 2.6 three timed automata, in particular, stochastic timed automata (STA) A_1, A_2 and A_3 are modeled in Uppaal. At first, weight annotations on edges and locations are ignored. The END-locations of the three timed automata are reachable within $[6, 12]$, $[4, 12]$ and $[0, +\infty)$. The calculation of the intervals is calculated as follows[13]:

A_1: The amount of time that can be spent inside the first three states is lower-bounded by the guard $x >= 2$ and upper-bounded by $x <= 4$. Therefore, the time interval for reaching the END-location is $[2 + 2 + 2, 4 + 4 + 4] = [6, 12]$.

A_2: In this timed-automata are two different paths to the END-location. The upper path is the same as in A_1. The time interval for the lower path is $[4, 8]$. Hence, the time interval for A_2 is $[min(6, 4), max(12, 8)] = [4, 12]$.

A_3: On account that there are no time limits in this timed-automata the resulting time interval is $[0, +\infty]$

The reachability time for the timed automata A_1 is the sum of the three uniform distributions. In the timed automata A_2 there are two paths to the end location, one with 3 intermediate states and one with 2 intermediate states. In this case, it is the combination of the sums of uniform distributions whereby the probability of the upper

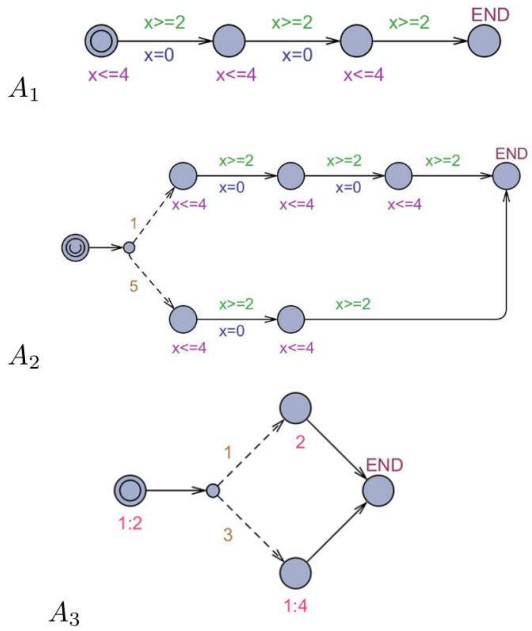


Figure 2.6: Stochastic Timed Automata A_1, A_2 and A_3 [37]

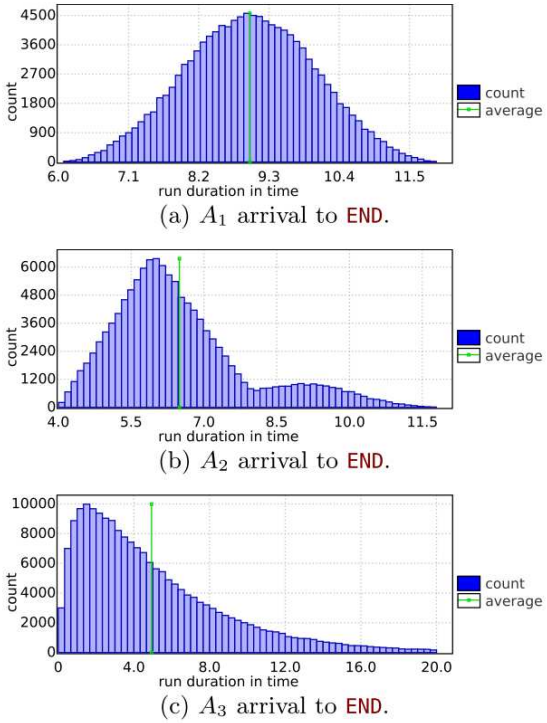
path is $\frac{1}{6}$ and the probability of the lower path is $\frac{5}{6}$. At last, the timed automata A_3 has exponential rates added to the states ($\frac{1}{2}$, 2, $\frac{1}{4}$). Additionally, there is a probability choice of the upper path with $\frac{1}{4}$ and the lower path with $\frac{3}{4}$. The resulting reachability time distribution diagram for all three timed automata are shown in Figure 2.7.

A model in Uppaal and in particular Uppaal SMC consists of a network of different interacting stochastic timed automata. Communication between these automata is possible with broadcast channels or shared variables. The resulting network of STA is called Network of Stochastic Timed Automata (NSTA) [54].

Since the statistical model checking feature has been added to the toolkit, also new queries are possible. In addition to the LCTL formulas, new queries for the stochastic interpretation of timed automata are available.

In order to describe a selection of these queries, the following short example aims to support the explanations. In a Mobile Edge Computing (MEC) environment the devices can offload their tasks to one or more specific edge servers. In this simple example there is one (mobile) *Device* and 2 edge servers. The device can offload its tasks to both edge servers. The probability to offload the task to *Edge1* is $\frac{1}{4}$ and for *Edge2* the probability is $\frac{3}{4}$. In Figure 2.8 a stochastic timed automata model of this scenario can be seen.

As a first query, the values generated from the simulated runs are displayed in a diagram. The following query is applied to the above explained system:



(a) A_1 arrival to END.

(b) A_2 arrival to END.

(c) A_3 arrival to END.

Figure 2.7: Reachability time distributions[37]

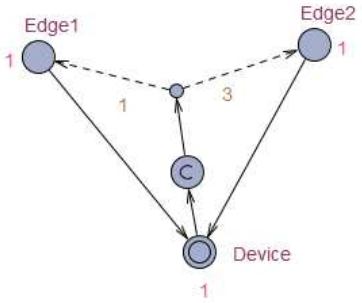


Figure 2.8: Simple stochastic timed automata model for edge offloading

```
simulate[<=10] {Device(0).Edge1, Device(0).Edge2}
```

The result of this query brought expected results because as illustrated in Figure 2.9 offloading tasks to *Edge2* are more likely than to *Edge1*.

In Uppaal SMC, a weighted extension of the temporal logic MITL (Metric Interval Temporal Logic)[19] is used and the properties of NSTA are defined by the following

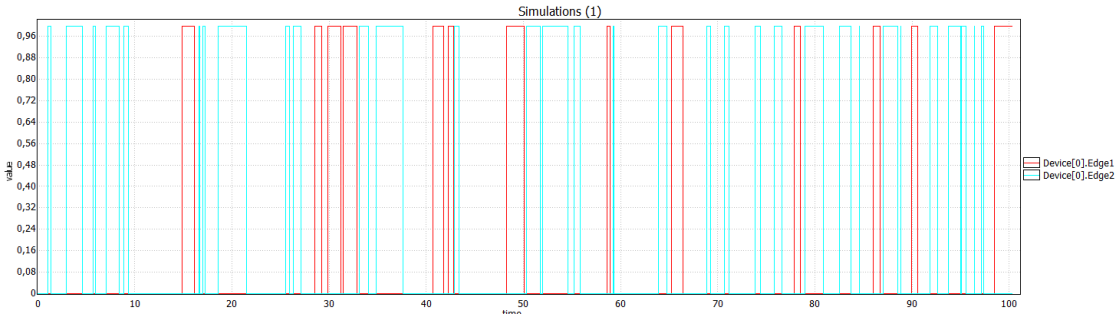


Figure 2.9: Visualization of offloading to *Edge1* and *Edge2* on one random run

grammar:

$$\varphi ::= \ ap \mid \neg\varphi \mid \varphi_1 \ \wedge \ \varphi_2 \mid O\varphi \mid \varphi_1 \ \mathrm{U}^{x}_{\leq d} \ \varphi_2 \tag{2.1}$$

The term ap is the conjunction of predicates over the state of a NSTA, d is a natural number and x is the clock. The logical operations are normally interpreted and the O is an operator indicating the next state.

Next, the weighted MITL formula $\varphi_1 \ \mathrm{U}^{x}_{\leq d} \ \varphi_2$ is analysed. The formula is satisfied by a run if φ_1 is satisfied until φ_2 is satisfied. Additionally, this must happen before the value of the clock x is greater than d. After applying standard MITL abbreviations[19] we get $\mathrm{tt} = \varphi \vee \varphi$, $\Diamond_{x\leq d}\varphi = \mathrm{tt} \ \mathrm{U}^{x}\varphi$ and $\Box_{x\leq d}\varphi = \neg\Diamond_{x\leq d}\neg\varphi$.

Based upon a NSTA M, following is defined: $P_M(\varphi)$ is the probability of a random run M statisfying φ. The cost bounded reachability problem is defined as follows: $\mathbb{P}_M(a_{x\leq C}ap) \geq p$, whereby x is a clock and C is a bound. Due to the fact that $\mathbb{P}_M(y) \geq p(p \in [0,1])$ is undecidable in general, UPPAAL SMC approximates the result with statistical model checking [37]. As a basis, the simulation runs are encoded as a Bernoulli random variable which returns true if the property is satisfied or otherwise false if it is not. UPPAAL offers the following three types of questions which can be answered using SMC[54].

Probability Estimation - Given a NSTA M, what is the probability $\mathbb{P}_M(\Diamond_{x\leq C} \ ap)$?

A probability estimation algorithm calcuates a probability within a given confidence interval. This algorithm is based on a classical Monte Carlo simulation.

The following example calculates the probability confidence interval of the question with which probability the *Device* offloads the task to *Edge1* within the first 10 time-units:

```
Pr[<=10](<> Device(0).Edge1)
```

The result for this query for one sample execution of the above mentioned edge offloading example is [0.685153,0.785019] with a confidence interval of 95 percent (316 runs). This means that there is approximately a two-third probability that the *Device* offloads a task to *Edge1* within the first 10 time-units.

Hypothesis Testing - Given a NSTA M, is the probability $\mathbb{P}_M(\Diamond_{x \leq C} ap$ greater or equal to a certain threshold ($p \in [0,1]$)?

As in hypothesis testing usual, the null-hypothesis ($H : p = \mathbb{P}_M(\phi) \geq \theta$)) is tested against the alternative hypothesis ($K : p = \mathbb{P}_M(\theta) \leq \theta$). As in [37] explained, strength parameters are used and the test is solved by Wald's sequential hypothesis testing [66]. Returning to the edge offloading example, hypothesis testing is used to answer the question if offloading a task to Edge1 within the first 5 time-units has a greater or equal probability of 0.5:

```
Pr[<=5](<> Device(0).Edge1) >= 0.5
```

The sequential hypothesis testing confirms this with 1674 runs and a 5% level of significance.

Probability Comparison - Given a NSTA M is the probability $\mathbb{P}_M(\Diamond_{x \leq C} ap_1)$ greater than the probability $\mathbb{P}_M(\Diamond_{y \leq D} ap_2)$?

With an extended form of Wald testing, it is possible to perform a probability comparison in order to analyze interesting properties, such as the average number of Edge Offloadings bounded by predefined time-units. The algorithm is explained in [36].

To explain the full power of probability comparison, an expansion of the edge offloading example is necessary. Now 5 (mobile) *Devices* can offload their tasks (with the same probability) to the two edge servers (*Edge1, Edge2*). With Uppaal SMC it is possible to query the average number of the maximum offloading tasks of all Devices to *Edge1* or to *Edge2* resprectively. The query bounds the runs, which are performed (30000 in this case), and the considered time-units for this query (10 time-units).

```
E[<=10; 30000](max: sum(i:id_t) Device(i).Edge1)
```

The result of the query is $E(max) = 2.1829$, which can be interpreted that in average approximately two tasks are offloaded to Edge1.

```
E[<=10; 30000](max: sum(i:id_t) Device(i).Edge2)
```

In contrast to the results above, for this query is $E(max) = 4.06123$.

Due to the fact that *Edge1* is used for offloading tasks with a probability of $\frac{1}{4}$ and *Edge2* is used for offloading with a probability of $\frac{3}{4}$, the results of the query above can be used to forecast the needed resources on a specific edge server. A detailed analysis of how to apply statistical model checking, in general these kind of queries to edge offloading scenarios in particular, is outlined in Section 4.2.

2.4.5 Discussion

PRISM and Uppaal are both state-of-the art model checking tools. While PRISM implements probabilistic model checking with probabilistic timed automata (PTA), Uppaal uses a network of stochastic timed automata (NSTA) for the statistical model checking approach by means of classical Monte Carlo simulations and hypothesis testing.

In this thesis Uppaal is utilized because it is used in many other projects in the area of mobile offloading and edge computing where timed automata are a suitable representation of distributed systems [16]. Furthermore, the support by the community and direct communication with the Uppaal SMC extension developers are also a major reason for considering Uppaal as the used model checker in this thesis [12]. Additionally, the statistical approach can be applied to more larger systems/networks than the numerical approach used in PRISM [51]. Nevertheless, PRISM was also used in the context of edge computing in the literature [72].

CHAPTER 3

Related Work

First, this chapter outlines the existing solutions for MCC offloading analysing four different MCC offloading frameworks: MAUI[35], Cuckoo[47], CloudClone[30] and Jade[59]. In recent years, a vast number of solutions has been proposed, nevertheless, different publications or framework concepts are based on diverse assumptions. Akherfi et al.[18] surveyed the existing literature in MCC and the most relevant MCC offloading frameworks are compared and contrasted.

Secondly, the highly active field of research of MEC offloading is outlined. In the research area a large amount of publications exist mainly focusing on energy-aware and energy-efficient MEC offloading. Different solutions for effective MEC offloadings will be analysed in this section.

Finally, the literature which is most relevant for this thesis is analyzed and evaluated. Therefore, it is useful to distinguish publications focusing on offloading with UPPAAL and PRISM support.

3.1 MCC Offloading Frameworks

During the last few years different mobile cloud computing offloading frameworks have been proposed. The following frameworks are considered for a detailed analysis as they are the most relevant ones for this thesis. They appear in chronological order according to the time they have been proposed.

- MAUI - Making Smartphones Last Longer with Code Offload[35]
- Cuckoo - A Computation Offloading Framework for Smartphones[47]
- CloneCloud - Elastic Execution between Mobile Device and Cloud[30]

- Jade - An Efficient Energy-aware Computation Offloading System with Heterogeneous Network Interface Bonding for Ad-hoc Net- worked Mobile Devices[59]

The similarities of the frameworks above are that three offloading steps introduced in Section 2.3.2 can be identified. According to the offloading steps and other important properties, a more detailed description and comparison can be found in the paper of Akherfi et al. [18].

In addition to aforementioned frameworks, many other exist such as Cloudlet[60], Mirror Server[69] or Phone2Cloud[67], but they are not part of the analysis in this thesis as this would go beyond the scope of the discussion.

3.1.1 MAUI

Cuervo et al. proposed MAUI[35] in 2010 as an energy-aware offloading framework. Due to the continuously executed profiling process, it is a dynamic offloading framework. The end users do not note anything about the offloadings because MAUI hides the complexity of the remote execution. A special feature is that the partitioning is done by annotations directly in the source code of the application. Therefore, developers can decide whether a part of the application can be offloaded or not. For the preparation of the offloading, two facts need to be considered. On the one hand, the application binaries must be available on the mobile device and the server and on the other hand, proxy, profiler and solver have to be installed on both sides. The MAUI profiler is responsible for the offloading decision and it keeps monitoring the application and network characteristics. Figure 3.1 depicts the overall architecture of MAUI with the two main parts, the *smartphone* and the *server*.

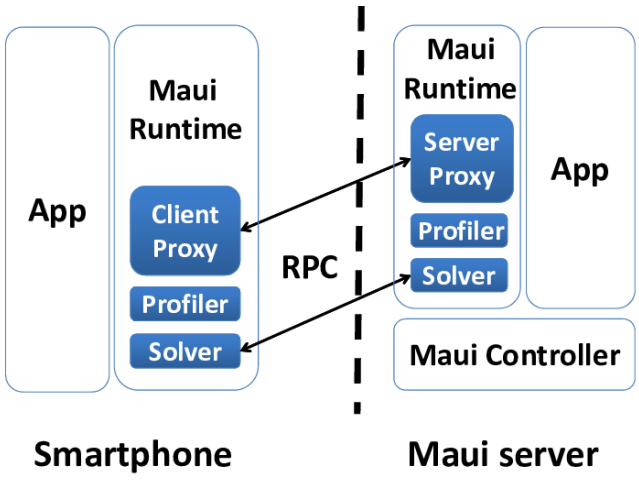


Figure 3.1: Overall architecture of MAUI [35]

In the MAUI runtime of the smartphone there are a *proxy*, a *profiler* and a *solver*. Considering an offloading procedure, the *profiler* decides if the task is offloaded or not and, therefore, MAUI's offloading decision is taken at runtime. Afterwards, the *solver* determines where the task will be executed. Finally, the *proxy* manages the data transfer between smartphone and server. Similarly, the three components on the MAUI server side handle the requests from the smartphones and perform the execution of the offloaded tasks. The MAUI coordinator/controller is in charge of handling authentication and resource allocation for incoming requests.

The energy savings which can be attained with MAUI are significant. For example, for a video game application 27% energy saving can be achieved and for a chess application the possible savings are with 45% even higher [35].

3.1.2 Cuckoo

The offloading framework Cuckoo[47] proposes a practical implementation for Android mobile devices. Similar to the other offloading framework, also Cuckoo focuses on enhancing the performance of the application and to reduce battery usage.

Developers using this framework can use the well-know IDE Eclipse, because the programming model is integrated into the Eclipse build system. The developers of Cuckoo utilize the existing activity/service model in Android to make a separation of compute intensive parts (services) and interactive parts of the application (activities). The service implementations have to be implemented on both sides, for the local and the remote executions. Cuckoo decides at runtime where the computation will take place and is, therefore, a dynamic offloading framework. If the remote execution is not possible (e.g., no network connection), the task can be executed on the local device as well. The remote execution is performed with the Ibis communication middleware[65]. It is a Java open source scientific software package for high performance distributed computing.

In the evaluation process of the Cuckoo framework, two real world smartphone applications are analyzed. It has been shown that the computation performance can be improved significantly. For example, the computation performance can be sped up by the factor of 60, if Cuckoo offloading framework is used [47].

3.1.3 CloneCloud

In 2011 Byung-Gon et al. introduced the offloading framework CloneCloud[30]. This framework is capable of offloading portions of unmodified mobile applications to make them faster and energy-efficient. According to Figure 3.2, the CloneCloud system automatically transforms an application which is designed for single-machine execution *(a)* into a distributed execution *(b)*.

In the partitioning step, a combination of a static program analysis and a dynamic profiler selects a set of offloadable components. Non-offloadable tasks, such as reading sensor data, are considered in this process to be executed locally. The different parts

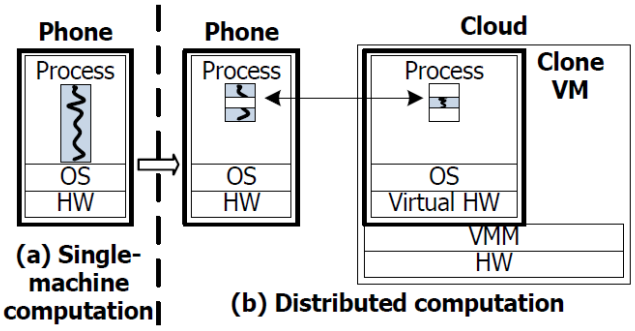


Figure 3.2: CloneCloud system model[30]

of the application can be portioned at a thread level because CloneCloud operates with thread level granularity.

Figure 3.2 shows that a duplicate of the software is deployed and runs within the cloud server. CloneCloud is a dynamic offloading framework taking decisions at runtime and then migrates the threads from the mobile device to the clone in the cloud. The distributed execution relies on a database with pre-computed partitions for current execution conditions (e.g., available network bandwidth).

Therefore, the affected threads must be suspended and all states must be transferred to the synchronized clone in the cloud server. Afterwards the thread can be resumed on the remote server and finally, after the execution of the offloadable part, the clone thread is reintegrated into the application of the mobile device. To show the effectiveness of CloneCloud, Byung-Gon et al. evaluate the framework with three applications (virus scanner, image search and privacy-preserving targeted advertising). Because of the fact that the migrations of the threads to the clones are resource and time intensive, larger workloads benefit more from offloading [30].

3.1.4 Jade

A more recent proposed offloading framework is Jade[59] introduced in 2014. The main goal is to implement energy-aware computation offloading for mobile applications and to minimize the development effort to build a Jade-ready application. One special feature is the sophisticated programming model which is offered by the framework. The partitioning of the application can be controlled by the developer and is performed on class level. Similar to the other frameworks, also Jade decides at runtime, if the code is offloaded or executed locally. Due to the dynamic decision process, the energy consumption on the mobile device can be reduced.

Figure 3.3 illustrates an architectural overview of the Jade framework. On the left side there is the mobile device running the application called *client*. On the right side the *server* is responsible for executing the offloaded code. As stated before, the Jade runtime automatically decides where to execute remote tasks. Additionally, the Jade

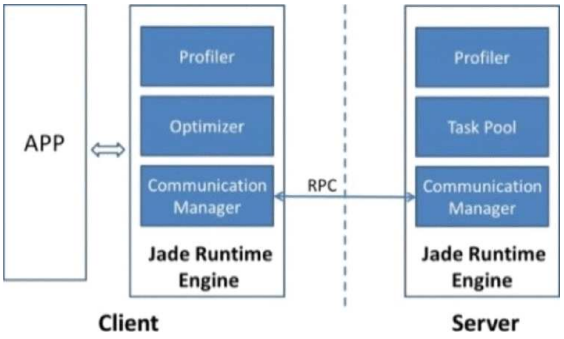


Figure 3.3: Jade's runtime engine overview[59]

runtime is responsible for starting the distributed execution. The *Jade Runtime Engine* is composed of the *Profiler*, the *Optimizer* and the *Communication Manager*.

The Jade server can run on Android servers as well as Non-Android servers working on a different operation system (e.g., Windows, Linux). One condition for Non-Android is that Java is installed in order to run Jade's runtime engine.

The framework is evaluated by two mobile applications (FaceDetection, TextSearch) containing heavy computation. The evaluation shows that there are major savings in energy consumption (74% for FaceDetection and 86% for TextSearch). An execution time speedup will only exist if more than one server is used due to the less powerful server in the evaluation and the transfer overhead [59].

3.2 MEC Offloading

A highly active field of research is MEC offloading. In recent years different frameworks and approaches have been appearing, mostly focusing on the offloading decision engine. In this section different solutions for energy-aware and effective MEC offloadings are analyzed. In the survey study of Mach and Becvar there is a comprehensive overview of the many different approaches in this field [52].

Deng et al. propose a cross-edge computation offloading framework for partitionable applications. Cross-edge collaboration is possible if an application is partitionable. This means that it can be presented as a directed acyclic graph (DAG). Among other aspects the transmission, execution time, coordination cost and offloading failures are considered in this approach. A special focus is on the heterogeneity and computation resource limitation of edge sites as well as user mobility[70].

In [38] a comparison between MCC's and MECO's (see Section 2.3.1) performance with regard to application's runtime, mobile device battery lifetime and cost for the user is outlined. Additionally, a new approach called Edge Cloud Heuristic Offloading (ECHO) is introduced. ECHO is an heuristic based offloading approach characterized by a shorter

computation time in order to make it more suitable for offloading latency sensitive applications. The results of the evaluation show that MECO outperforms MCC regarding application runtime and cost [38].

Brandic et al.[23] focus on the optimal edge node selection for offloading using fuzzy logic. Furthermore, bandwidth, processor speed and latency concerns are take into account for the optimal edge node for offloading. Additionally, a handoff controller is proposed in order to handle the users' mobility.

Since mobility is particularly interesting, in [64] the "Follow-me-Edge" principle is implemented in order to ensure that applications as well as services follow the mobility of the users. Moreover, the solution contains ultra-short latency through a smart MEC architecture which is especially important for the upcoming 5G mobile systems.

Also in the automotive research area (e.g., smart connected cars), mobile edge offloading can be used. Due to the fact that low latency is the core requirement in 5G-enabled vehicular communications, mobile edge offloading can meet such requirements [57].

3.3 Offloading with model checking support

In this thesis model checking is utilized to support the proposed framework. Therefore, the current research in this area is analyzed and divided into frameworks utilizing Uppaal, or respectively, PRISM as the model checker.

3.3.1 Uppaal

Aceto et al.[16] used Uppaal for supporting the decision engine and for generating the Uppaal model, they introduced a new language called MobiCa[1] implemented in an Eclipse[4] plugin. MobiCa is a domain specific language where the developer can define system and application structure involving devices, computational power, code partitioning and required memory.

A system in MobiCa consists of mobile devices and cloud machines. Furthermore, applications, fragments, structure and operators exist in the syntax of MobiCa. Especially interesting are the three operators available in MobiCa connecting fragments with each other.

Non-deterministic choice (→): The execution will progress from the source fragment to one of the target fragments. The actual target fragment is selected non-deterministically.

Sequential progress (⇢): The execution will progress from the source fragment to the target fragments in sequential order.

[1] Mobile Cloud Computing Language

Parallel execution (↠): The execution will progress from the source fragment to all target fragments parallel.

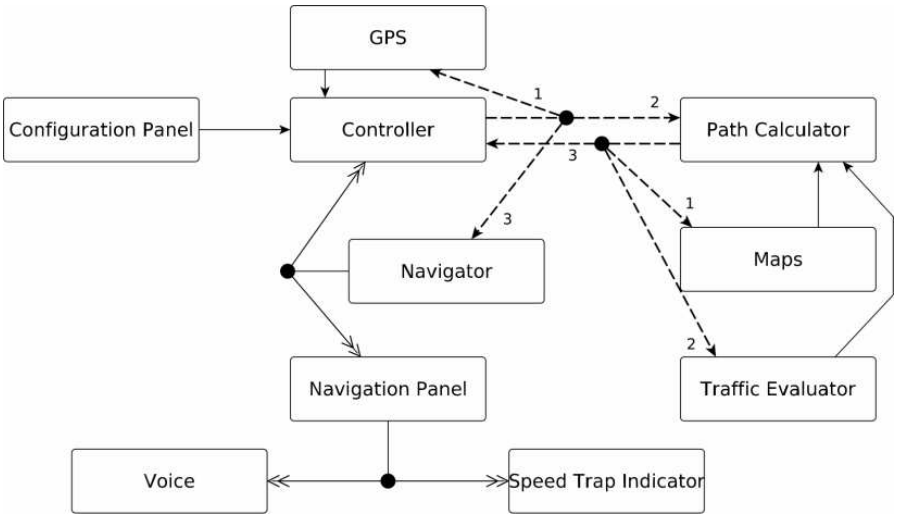


Figure 3.4: Navigator application: graphical representation of the structure defined in MobiCa[16]

As visualized in Figure 3.4, there is an example of a navigator application designed with MobiCa. For example, the *Controller* fragment executes the *GPS*, *Path Calculator* and *Navigator* in sequential order. Otherwise, the *Navigation Panel* can call the *Voice* or the *Speed Trap Indication* fragment in a non-deterministic way.

In order to provide the decision support for the applications, the MobiCa defined applications are translated to Uppaal timed automata. Afterwards, the resulting timed automata can be analyzed with different TCTL formulas which can be handled by Uppaal. Nevertheless, statistical model checking is not possible and this solution is only suitable for design-time offloading [16].

3.3.2 PRISM

Besides Uppaal, PRISM is also a well-known model checker and Aral, Brandic and Zilic[72] utilize the probabilistic model checker feature of PRISM for maximizing the ratio of successful offloadings in an edge computing scenario. In this work *stochastic* and *non-stochachstic workload-aware offloading decision engines* (ODE) are introduced. Non-stochastic ODE perform the offloading decision based on the application task size and the current CPU workload. Stochastic ODE work similar to non-stocastic ODE, but stochastic ODE take the predicted CPU workload instead of the current CPU workload into account. For stochastic ODE two different CPU prediction models (Linear Model and Near Over-Consumption (NOC) Model) are introduced and used.

In the scenario used for the evaluation, two applications (Facerecognizer, Chess) are introduced which can be divided into parts that are offloadable and non-offloadable. As demonstrated in Figure 3.6, these mobile applications can be modeled as Directed Acyclic Graphs (DAG). The blue nodes cannot be offloaded, but instead the nodes with a white background are offloadable to an edge server. In Figure 3.6 two successful offloading scenarios are shown (green and blue) and one potential offloading error, if both mobile devices (MD1, MD2) offload their task to the same edge node E2 (red).

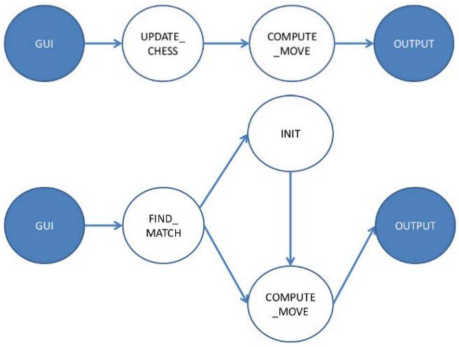


Figure 3.5: DAG of Facerecognizer and Chess applications[72]

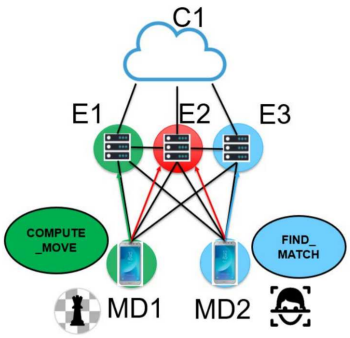


Figure 3.6: Task offloading scenario[72]

Probabilistic model checking with PRISM is used to verify edge offloading failures and prediction accuracy. For the evaluation a single edge node infrastructure is used and results show that non-stochastic ODE has the best offloading performance in the applied workload scenarios (periodic, static, once in a lifetime, unpredictable, continuously changing). A real world field of application is a network operator which can use the results of the simulations to determine edge offloading failures and system performance issues [72].

Formal verification techniques are also applied in the EFPO[2] framework. It is based on the model checking solution Value Iteration Algorithm (VIA). The algorithm supports the offloading decision by taking offloading failure probability into account. Additionally, the properties energy consumption and time are taken into account. To capture the stochastic and non-deterministic behavior of edge offloading, the Markov Decision Process (MDP) is used. Therefore, the system performance is increased because the framework determines the feasible offloading decision policy [71].

Apart from edge computing, probabilistic model checking is utilized in cloud computing contexts. In [41] probabilistic model checking in combination with performance modeling is used to guarantee rule-based auto-scaling policies. The evaluation with Amazon EC2 cloud shows that this approach can help to minimise Quality of Service (QoS) violations. Finally, Naskos et al.[56] used probabilistic model checking to support the elasticity decisions (on-demand resource provisioning) in cloud computing.

[2]Energy Efficient and Failure Predictive Edge Offloading

CHAPTER 4

Edge Offloading with Model Checking Support

The prototype for predicting future workloads and offloading errors in edge computing is designed as a state-of-the-art software with a web user interface for analyzing different edge computing scenarios. The main purpose of this proof of concept is to minimize offloading errors in MEC. Due to the novelty of this proof of concept, the well-known method rapid prototyping[44] is used. The iterative process in rapid prototyping consists of following three steps which are applied during the implementation of the prototype: design, modify and review. Through this prototyping method changing requirements can be better managed and the iterative enhancement process leads to the expected results.

This chapter is divided into two main sections, first, the design of the prototype and second, the details of the technical implementation for the proposed solution with in-depth information and analysis of problems and pitfalls during the implementation process.

4.1 Design of the Prototype

In this section the design of the prototype is discussed and analysed. The prototype is composed of two components. First, the Edge Offloading Prototype including a REST-API and the model checker engine of Uppaal. Second, the Offloading-Client consists of a web-browser frontend for a straightforward interaction with the Edge Offloading Prototype. In the beginning, the design principles for the prototype are outlined and then the modeling approach and the architecture is evaluated.

4.1.1 Design Principles

In the designing process of the prototype the following three main properties are considered.

Scalability

In practice, large-scale edge networks consist of an enormously high number of servers and connected devices. Therefore, it is from utmost importance that the application is scalable and able to handle input models including lots of states. Due to the application of statistical model checking, resource-optimized Uppaal models and the possibility to configure the simulation for the current needs, the scalability of the prototype is ensured.

Extensibility

The introduced prototype is designed to use state-of-the-art technologies in order to be extended for different fields of application in the next development iterations. The Service API, the multitier architecture and the extensible input model guarantee the extensibility throughout the whole prototype.

Simplicity

The program offers an interface for providing the input model, which is defined by the interface description. Additionally, a frontend for user interaction is provided for an easy and user-friendly interaction with the prototype (see Section 4.2.2).

4.1.2 Modeling and Architecture

In order to fulfill the design principles introduced in the previous section, the subsequent flow chart of the prototype is proposed in Figure 4.1.

The application is designed to incorporate following modules:

Service API

This module offers a uniform interface to receive requests from different other systems. As demonstrated in Figure 4.1, access through a web interface or a direct REST-API call with the input model is possible. In this context it is important to notice that access from other platforms is also possible (e.g., network operator applications etc.).

Convert Input Model to Uppaal Model

The input model for the framework consists of following components:

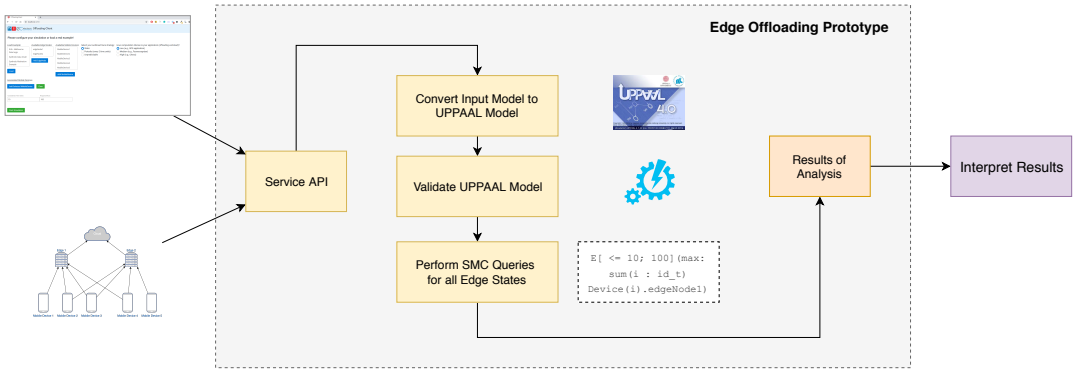


Figure 4.1: Flow chart of the prototype

- A list of all available *mobile devices* belonging to this simulation
- A list of *edge nodes* containing a list of the connected *mobile devices*
- The *computation intensity* of the considered application (LOW, MEDIUM, HIGH)
- The number of *considered time units*
- The number of *required runs*
- A *strategy* (static, periodic or unpredictable)

A minimal example of an input model JSON[1] object suitable for converting to an UPPAAL model can be seen in Listing 4.1. This minimal example consists of one mobile device and one edge node.

Listing 4.1: Input Model JSON object

```
{
    "allAvailableDevices": [
        {
            "id": 0,
            "name": "device0"
        }
    ],
    "edgeNodeList": [
        {
            "id": 0,
            "message": "string",
            "mobileDeviceList": [
                {
```

[1] JavaScript Object Notation

```
                    "id": 0,
                "name": "string"
                }
            ],
            "name": "mobileDevice0",
            "query": "string"
        }
    ],
    "computationIntensity": "LOW",
    "consideredTimeUnits": 10,
    "requiredRuns": 100,
    "strategy": "STATIC"
}
```

As stated in the preceding paragraphs, three well-known workload trace strategies are implemented. Therefore, typical workload scenarios commonly proposed in cloud computing environments[43] are implemented and defined as follows:

Static Workload The mobile devices are offloading tasks statically and the resource utilization is equal over time.

Periodic Workload The mobile devices are offloading tasks in a periodic time interval (every two time units).

Unpredictable Workload The mobile devices are offloading tasks in a random and unforeseeable way.

This model has to be converted to a valid Uppaal model where all edge nodes are modeled as own states. More information about the precise implementation details are discussed in Section 4.2.

Validate Uppaal Model

Uppaal offers a Java API for interacting with the Uppaal ecosystem. This API offers methods for manipulating the model, import and export traces. In order to validate the model, it also offers methods to interact with the model-checking APIs of the engine. Besides the Java API, also a C++ API and also interaction through a command line tool is possible [15].

Perform SMC Queries for all Edge Node States

One of the most important parts of the tool is the SMC analysis. The queries are performed for all edge node states in the model and the input parameters can be configured from the calling system. The syntax of the SMC query for predicting the workloads is illustrated below.

```
E[bound; N](max :expr)
```

The variables are interpreted as follows:

- *bound* is the number for considered time-units for the simulation
- *N* is the number of runs which should be performed by the engine
- *expr* is the expression to evaluate, in this context, the number of edge offloading activities from the different devices

Result of Analysis

The analysis results are converted to a response object containing the results of the SMC queries grouped by the edge node objects.

After the simulation with the *Edge Offloading Prototype*, the results must be interpreted in a proper way (*Interpret Results*). Possible actions after evaluating the results could be rescaling the edge resources of certain edge nodes or changing the deployment plan.

4.2 Solution

The concrete implementation details are discussed in this section and the features of the proposed prototype are presented. This project consists of two main parts. First, the *Edge Offloading Prototype* and second, the *Offloading-Client* which offers a web-based user interface for configuring the simulations and interacting with the Edge Offloading Prototype. Both software modules are build with open-source products and the whole sources, including this thesis, are available on two different GitHub repositories[6][7]. The overall architecture of the software can be seen in Figure 4.2. The Angular frontend is explained in detail in Section 4.2.2 and the implementation details of the Edge Offloading Prototype are discussed in the next section.

4.2.1 Edge Offloading Prototype

The software for interacting with the UPPAAL engine is implemented with the state-of-the-art application framework *Spring*[2]. In order to create a standalone spring application with the embedded application server *Tomcat 9*, the *Spring Boot*(Spring Boot Version 2.1.7) framework is used. As a build tool, *Maven 3*[3] is used and for an ease testing of the REST-API, the *Swagger*[4] toolset is utilized.

To further enable the multi-tier architecture, the Spring feature Dependency Injection (Inversion of Control) is applied to enable the communication between the layers. As described in [15] the interaction with the UPPAAL engine is performed with the available Java API. It is important to notice that the current and latest version of the UPPAAL engine is used because older versions of the engine contain software bugs concerning the

[2]https://spring.io/
[3]https://maven.apache.org/
[4]https://swagger.io/

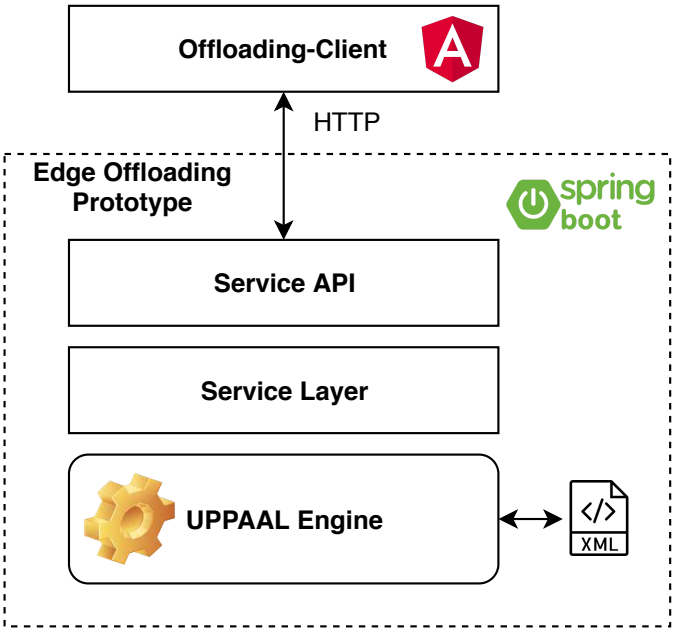


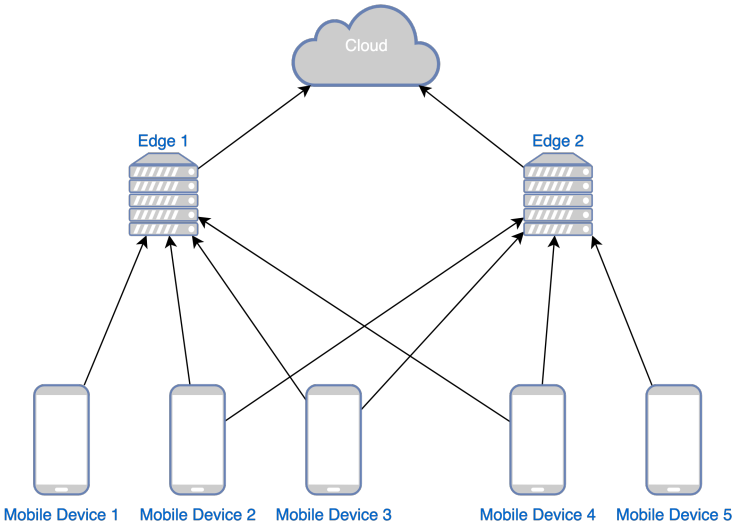
Figure 4.2: Overall architecture of the Offloading-Client and the Edge Offloading Prototype

fint() method which is utilized in the prototype[12]. The latest version of Uppaal is 4.1.23 released on the September 11th, 2019[5].

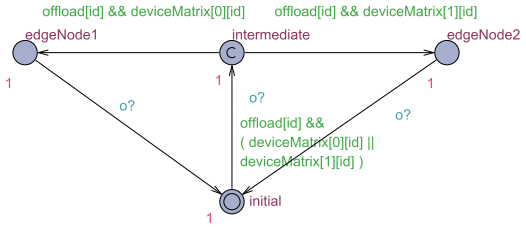
First of all, the given input model is converted to an Uppaal model as outlined in Section 4.1.2. To understand the behavior of the converting method from a Java Object to an Uppaal *Document*, the following motivating example explains the process.

In the motivating example (see Figure 4.3a) there are two edge nodes (*Edge 1*, *Edge 2*) and five mobile devices (*Mobile Device 1-5*). It is important to mention that this prototype focuses on the first hop offloading to the edge nodes, communication and transferring data from and to the cloud is not considered in this work. As demonstrated in Figure 4.3a, the mobile devices 2-4 can offload their tasks to both edge nodes and the remaining two devices can only offload to one edge node. Before the prototype takes action, the input model is built following the rules defined in Section 4.1.2. Afterwards the actual converting process starts with defining two templates in Uppaal, namely **Device** and **Generator**. The resulting system in this case contains five device processes and one generator process.

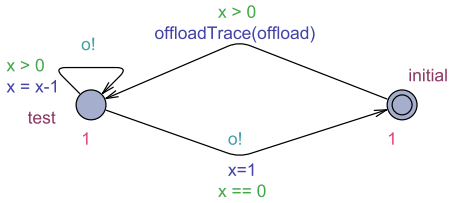
[5]http://www.uppaal.org/



(a) Input Model



(b) Device Template (parameters: *const id_t id*)



(c) Generator Template

Figure 4.3: Overview of the Conversion from the input model to the device and the generator template

Device

This template (Figure 4.3b) represents one mobile device and consists of 4 states including *initial*, *edgeNode1* and *edgeNode2*. It is obvious that for all input model edge nodes there is a corresponding edge node on the device template. UPPAAL SMC models must exhibit *Input Determinism* which is ensured when for all locations l in the model is receiving any input synchronization event, the subsequent location l' is the same for all edges originating in l [46].

The example model in Figure 4.4 violates the required input determinism. If there is another process which activates the broadcast channel o at the same time, the model will have two options to choose whether to switch to *edge1* or to *edge2*. The model can choose one of the two possible options which is violating the above definition of input determinism. To overcome this issue, the model in Figure 4.5 introduces a new committed intermediate location. This helps to fix the synchro-

nization problem because if a synchronization event occurs, only a transition to the new added location will be possible and input determinism will be assured. Worth mentioning is that it is necessary to validate the whole model including all templates in order to ensure that this workaround with the new committed intermediate location works.

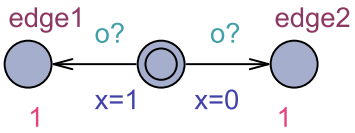


Figure 4.4: Model violating input determinism

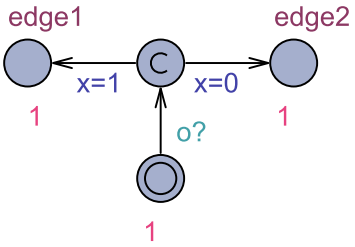


Figure 4.5: Model fulfills input determinism

Generator

As visualized in Figure 4.3c, the *Generator* template is responsible for calling the *offloadTrace()*-method. After this transition, the system is remaining in the location *test* until the variable x gets zero and the broadcast channel o is activated for synchronization with the device processes running in the same system. The *offloadTrace* method is called in a regular interval and the containing functionality including the *deviceMatrix* array is displayed in Listing 4.2. The *offloadTrace()* method is responsible for generating a boolean array with the size of the number of devices. Depending on which workload trace strategy is used, this method is implemented differently.

The following code snippet in Listing 4.2 shows the global declarations of the overall Uppaal system. Since in this case the workload trace strategy has been *unpredictable*, the Uppaal standard *random()* function combined with the *fint()* function is used. To mitigate memory problems (the maximum memory space is 4GB), a one dimensional array with the size of the number of devices is used. In the previous versions of this prototype, the *select*-statements are utilized for random number generation, but this leads to memory space explosion even with a small number of mobile devices and edge nodes [12].

Additionally, the *deviceMatrix* is implemented in an efficient way, whereas the *true* values in the two dimensional array will indicate, if offloading to a specific edge node is possible or not. Accordingly, in the device template both arrays are used for offloading decisions as *guard*-statements in the edges to the nodes which represent the edge nodes (see Figure 4.3b).

Listing 4.2: UPPAAL methods and global declarations

```
broadcast chan o;
int x = 1;
const int NODE_SIZE = 2;
const int DEVICE_SIZE = 5;
typedef int[0, DEVICE_SIZE-1] id_t;
bool offload[id_t];
clock z;

const bool deviceMatrix[NODE_SIZE][id_t] = {
{1,1,1,1,0},
{0,1,1,1,1}
};
typedef int[0,(1<<DEVICE_SIZE)-1] uint32_t;

void offloadTrace(bool& test[DEVICE_SIZE])
{
    uint32_t number = fint(random(1<<DEVICE_SIZE));
    for (i: id_t)
       test[i] = (number >> i) & 1;
}
```

In the code snippet 4.2 the global declarations and methods of the motivating example are shown. The implemented software renders dynamically the global declarations and methods according to the amount of edge nodes, mobile devices and the configured workload strategy. This is done by the service layer of the application.

In Figure 4.6 a sequence diagram shows the usual interaction of a user with the application. The user can either call the *analyzeModel* method through the REST API, or the Offloading-Client is used. The *UppaalRestController* calls the the responsible service method in *UppaalService*. In the service method the UPPAAL method is generated and validated. Afterwards, the *UppaalEngine* is called in order to initialize the system. The main part takes place in the loop where statistical model checking queries are performed for all edge nodes and the results are collected by the service method running in *UppaalService*. Finally, the results containing all edge node query results are returned to the user.

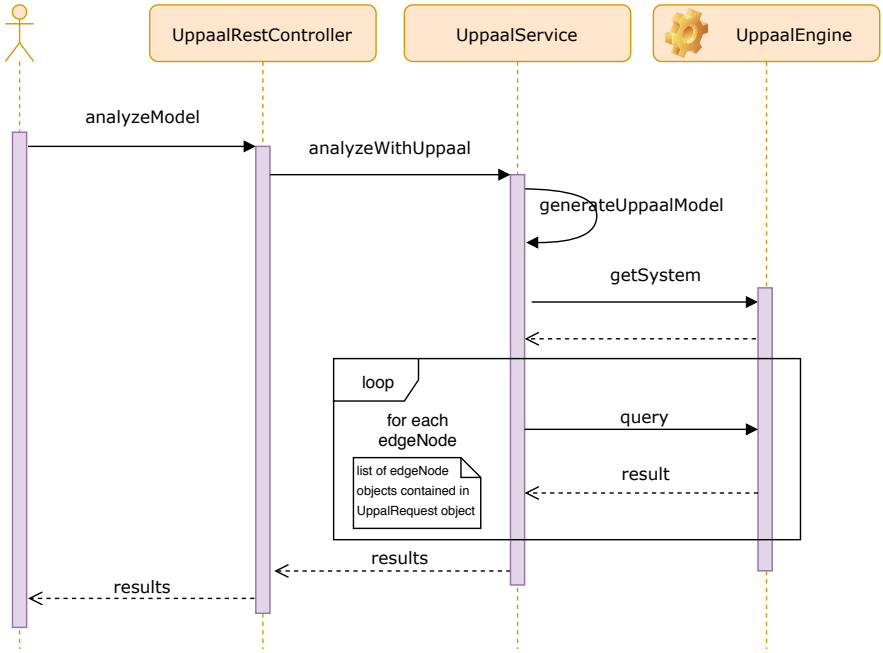


Figure 4.6: Sequence diagram of the Edge Offloading Prototype

4.2.2 Edge Offloading Client

The web frontend is implemented with a state-of-the art Typescript[25] frontend framework Angular[3]. In this thesis the current version 8 is used. The main feature of this application is to configure a simulation within a user-friendly interface. Additionally, it is also possible to load different real-world or synthetically created examples.

The top left side of the window in Figure 4.7 shows that it is possible to load examples from a list. Right next, the available edge nodes and mobile devices are displayed, whereby additional edges nodes and mobile devices can be added directly with clicking *Add EdgeNode* and *Add MobileDevice*, respectively. By clicking on one of the different available edge node elements, the associated mobile devices are displayed in the underlying *Associated Mobile devices* list box. If it is needed to add another mobile device to the selected edge node, it will be necessary to select a new mobile device from the *Available Mobile Devices* list box and submit this choice with the button *Add Selected MobileDevice*. After finishing the simulation configuration, on the right top side of the browser window the workload trace strategy and the computation intensity can be chosen by selecting the appropriate radio buttons.

Before starting the simulation, the two values for *Considered Time Units* and *Required Runs* can be adapted. Detailed information about these properties and how they affect the analysis can be found in Section 2.4.4 and Section 5.3.3.

Finally, the analysis with the Edge Offloading Prototype can be started with the *Start*

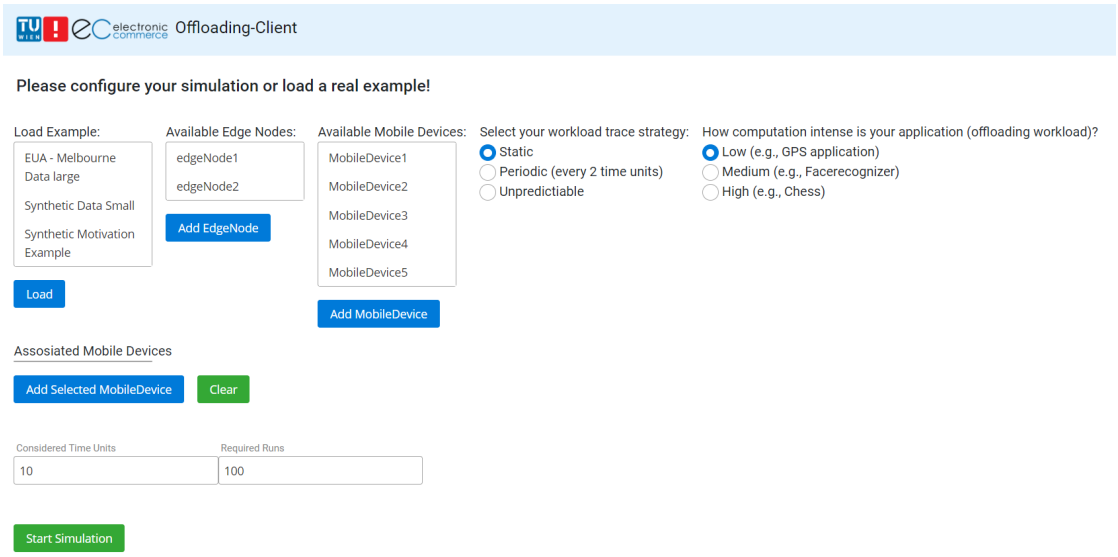


Figure 4.7: Offloading-Client web frontend

Simulation button. The just composed simulation is converted to a suitable input object (see Section 4.1) for sending it to the Edge Offloading Prototype REST API. After processing the request and performing the analysis, the response is displayed in the web frontend.

Id	Edge Node	POU[a]	UPPAAL Query
0	edgeNode1	3	$E[<= 10; 100](max : sum(i : id_t)Device(i).edgeNode1)$
1	edgeNode2	3	$E[<= 10; 100](max : sum(i : id_t)Device(i).edgeNode2)$

[a] Predicted Offloading Units

Table 4.1: Exemplary Response of the Edge Offloading Prototype

As illustrated in Table 4.1, the response contains detailed information about every edge node. The input model for this response is the motivating example showed in Figure 4.3a, the computation intensity is set to LOW and the workload trace strategy is *unpredictable*. In the column *Predicted Offloading Units* the maximum predicted number of offloadings for the respective edge node is displayed. Furthermore, the exact UPPAAL query, which was used during the analysis process, can be obtained from the table above.

CHAPTER 5

Evaluation

One of the main focus of this thesis is to evaluate the proof of concept with real data traces. Therefore, two data traces from the RICE university[10] and the EUA dataset from Lai et al.[50] are used. Since the EUA dataset does not contain mobility data, mobility prediction is performed with a POI[1] based human mobility approach[42].

The prototype can deal with typical workload scenarios commonly proposed in cloud computing environments[43], which can also be applied to edge computing scenarios as well. The configurable workload traces are static, *periodic* and *unpredictable* (see Section 4.1.2).

The evaluation of the proof of concept consists of two different scenario types:

- static scenarios (best-case/worst-case analysis and execution time analysis)
- mobility scenarios (offloading error analysis)

The results of the various analyses show the strengths and the weaknesses of the approach and the detailed results are discussed in Section 5.3.

5.1 Experimental Setup

For a proper evaluation of the introduced prototype, a comprehensive testing and evaluation is necessary. Therefore, two real data traces are used for analyzing the different scenarios. First, a dataset from business district area of Melbourne (Australia) containing 125 edge nodes and 816 mobile device users[5] is used. Secondly, a dataset from the RICE university in Houston, Texas is utilized. In this dataset iPhone user data and

[1] Point-of-Interest

the connected cell towers have been recorded for one year [61]. In this thesis only a selected part of the dataset is used, the full datasets are published on the university's webpage[10].

5.1.1 Test Environment

All test scenarios are implemented as integration tests with Spring Boot Tests. One advantage of utilizing this test method is that the interaction with the Spring components and also the h2 in-memory database[9] is possible by default. The analysis results are exported in a first step to a Microsoft Excel file to visualize the data fast and easily with diagrams and statistics. In order to plot the results of the tests, the data is converted to a CSV-File and the diagrams are plotted directly with LaTeX.

The simulations and tests are performed with a Lenovo notebook (Intel Core i7-6600U CPU with 2-cores running at 2.60 GHz each and 16 GB RAM). The Uppaal engine is running with version 4.1.22.

5.1.2 EUA Dataset

As previously mentioned, the EUA dataset contains real data traces from base stations and end-users. The data was collected in the central business district area of Melbourne, Australia. The total area of this region is $6.2km^2$. The Australian Communications and Media Authority (ACMA) make the radio-comms license dataset publicly available including the geographical location of all celluar base stations in Australia. This base stations are used as edge server locations [50]. The complete dataset can be downloaded from GitHub[5]. A visualisation of the dataset within Google Maps is illustrated in Figure 5.1.

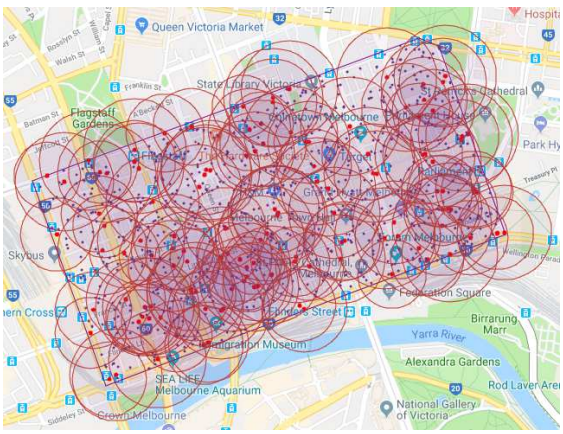


Figure 5.1: Visualisation of EUA dataset showing the edge servers including their coverage (red circles) and the devices (red dots) in the Melbourne metropolitan area

Before analyzing this dataset with the prototype, the given data structure has to be converted to a valid *UppaalRequest* object which is the input for the *Edge Offloading Prototype*. Due to the fact that in the dataset are 125 edge nodes and 816 users, in a first step smaller parts of this dataset are taken into account. For this purpose, in the test environment there is the possibility to configure the number of edge nodes and users for a defined test case. Additionally, the maximal range of one edge node can be defined and afterwards, the test environment identifies all edge nodes which are reachable by a mobile device in this scenario.

Listing 5.1: Method to find all EuaServers within a maximum distance

```
1  private void findAllEuaServersWithinDistance(Location loc, List<
       EuaEdgeServer> allServers, double maxRange, EuaUser temp) {
2
3      for(EuaEdgeServer s : allServers)
4      {
5          s.setDistance(distance(loc.getLatitude(), s.getLocation().
               getLatitude(), loc.getLongitude(), s.getLocation().
               getLongitude(),0.0, 0.0));
6          temp.setDistance(s.getDistance());
7      }
8      Collections.sort(allServers, Comparator.comparing(EuaEdgeServer::
           getDistance));
9
10     allServers.stream().filter(a -> a.getDistance() <= maxRange).
           forEach(i -> temp.setDistance(i.getDistance()));
11     allServers.stream().filter(a -> a.getDistance() <= maxRange).
           forEach(i -> i.addEuaUserList(temp));
12 }
```

Listing 5.1 illustrates the algorithm finding of all edge nodes reachable by an *EuaUser* (represents a mobile device). Therefore, the distance to all egde nodes (*EuaEdgeServer*) is calculated and after that the list of edge nodes (*allServers*) is sorted by the before calculated distance. Finally, all edge nodes which are not in the predefined maximal range are removed from the list. The complete source code including all test scenarios can be found in the respective GitHub repository[6].

5.1.3 LiveLab Dataset

In the evaluation process of the prototype also a dataset provided by the RICE University in Houston, Texas is used[61]. In the LiveLab project 25 users of iPhone 3GS were tracked for one year, exactly from 30st March 2010 to 1st April 2011. Beside the installed apps, the number of phone calls, web browser history, accelerometer data and other metadata as well as the connected cell towers were periodically recorded. For the evaluation of the prototype, the celltower data is interesting because the celltowers are interpreted as edge nodes and the iPhone users are the mobile devices in this evaluation. The celltower dataset contains the following information:

- uid: (user ID)
- time: time and date (POSIX)
- geoid: the md5 hash of the geoid
- towerid: the md5 hash of the towerid

The *uid* contains the user id, which represents one of the 25 users. In the evaluation the *towerid* is converted to more easily legible names than md5 hashes. Therefore, the distinct towerid values are converted to ids consisting of the prefix *"edge"* following by an ascending number for all distinct *towerids*. In this dataset 7357 towerids, representing edge nodes, are recorded.

The data is obtained with a SQL-File which is imported to an in-memory h2 database running within the Spring Boot application described in Section 4.2.1. One test case can cover a whole day and the system is configured to analyze the network with the *Edge Offloading Prototype* every hour. It is clear that offloading activity and the network itself is not static and, therefore, the current network and offloading situation is obtained every 10 minutes. As the *Edge Offloading Prototype* performs the analysis hourly, the obtained offloading errors are counted every 10 minutes. The implementation of a test case in this scenario follows the following steps:

1. First, all edge nodes are selected which are relevant for that day (at least one connected device).
2. Every hour of the day, the current network is obtained by selecting the edge nodes and the connected devices to them. This model is the input for the *Edge Offloading Prototype* and the results are the predicted maximum offloading units per edge node.
3. Every 10 minutes the current offloadings are analyzed and the offloading errors are counted.

This procedure is done for the three workload trace strategies, namely *static*, *periodic* and *unpredictable*.

5.2 Mobility

In the given dataset from the business district of Melbourne (EUA dataset) only a snapshot of edge nodes and devices is given and also Lai et al. mentioned that the users' mobility has to be taken into account [50]. For this purpose a mobility pattern is applied to synthetically generate mobility data where users are moving within a given location (see Figure 5.1). In literature there are different approaches concerning human mobility. In [20] a POI (Point of Interest) based approach is used for simulating mobility in an

edge environment. This work is based on an approach introduced by Fan et al. [42]. As a basis for their work they used a dataset of over 37 million check-in records generated by more than 350 thousand users during the second half of 2013. Basically, this approach states that people usually have three choices with respect to mobility:

1. Exploring a new POI
2. Returning to a previous visited POI
3. Staying at the current one

Therefore, the probabilities P_{new}, P_{return} and P_{stay} have to be defined. Fan et al. define P_{stay} with an average value of 0.445 because it is independent of time and follows a bell-shaped normal distribution. P_{new} is defined as follows:

$$P_{new} = a * t^{-0.3} \tag{5.1}$$

The variable a is the rate of exploration which provides information to what extend a user explores new places. If a is set to 0, no new places will be explored by the user, otherwise if a is set to 1, the user will explore a new POI in every case. Additionally, t is the current number of time steps and the exponent -0.3 is the decaying rate of P_{new}. Finally, P_{return} is defined as follows:

$$P_{return} = 1 - (P_{new} + P_{stay}) \tag{5.2}$$

Since it is possible in some cases that $P_{new} + P_{stay} > 1$, the adapted $P_{new} = 1 - P_{stay}$ is used instead of $P_{new} = a * t^{-0.3}$. In this evaluation the POIs are the edge nodes distributed in the given area. All POIs have the same attraction level and are selected randomly with the restriction that only the 3 nearest edge nodes can be chosen. Also the new POI must be different to the previous one of the current user. The factor a is used to vary the different mobility scenarios.

For the evaluation with the above explained mobility model, the EUA dataset with 100 devices and 15 edge nodes is used.

In the evaluation process the following mobility scenarios are used:

- Scenario 1: low mobility (e.g., stationary services)
  - Use of EUA dataset with low mobility ($a = 0.01$)
- Scenario 2: moderate mobility (e.g., commuters)
  - Use of EUA dataset with moderate mobility ($a = 0.2$)
- Scenario 3: high mobility (e.g., vehicular services)
  - Use of EUA dataset with high mobility ($a = 0.5$)

5.3 Results and Discussion

The prototype is evaluated with static and mobility scenarios from real data traces and synthetically generated mobility data. Moreover, the configuration properties *considered time units* and *required runs* are analysed and a sensitivity analysis is performed. In addition, an execution time analysis demonstrates the performance issues with lots of mobile devices in a single scenario.

5.3.1 Static Scenarios

In a first step, a static scenario with the EUA dataset is analyzed. Therefore, 15 edge nodes are considered and the number of devices is increased in every simulation step from 10 devices to 310. To evaluate the usefulness and the correctness of the prototype, the offloading units predicted by the prototype are compared with a best-case and worst-case analysis. The best-case and the worst-case analysis in this scenario is defined in the following.

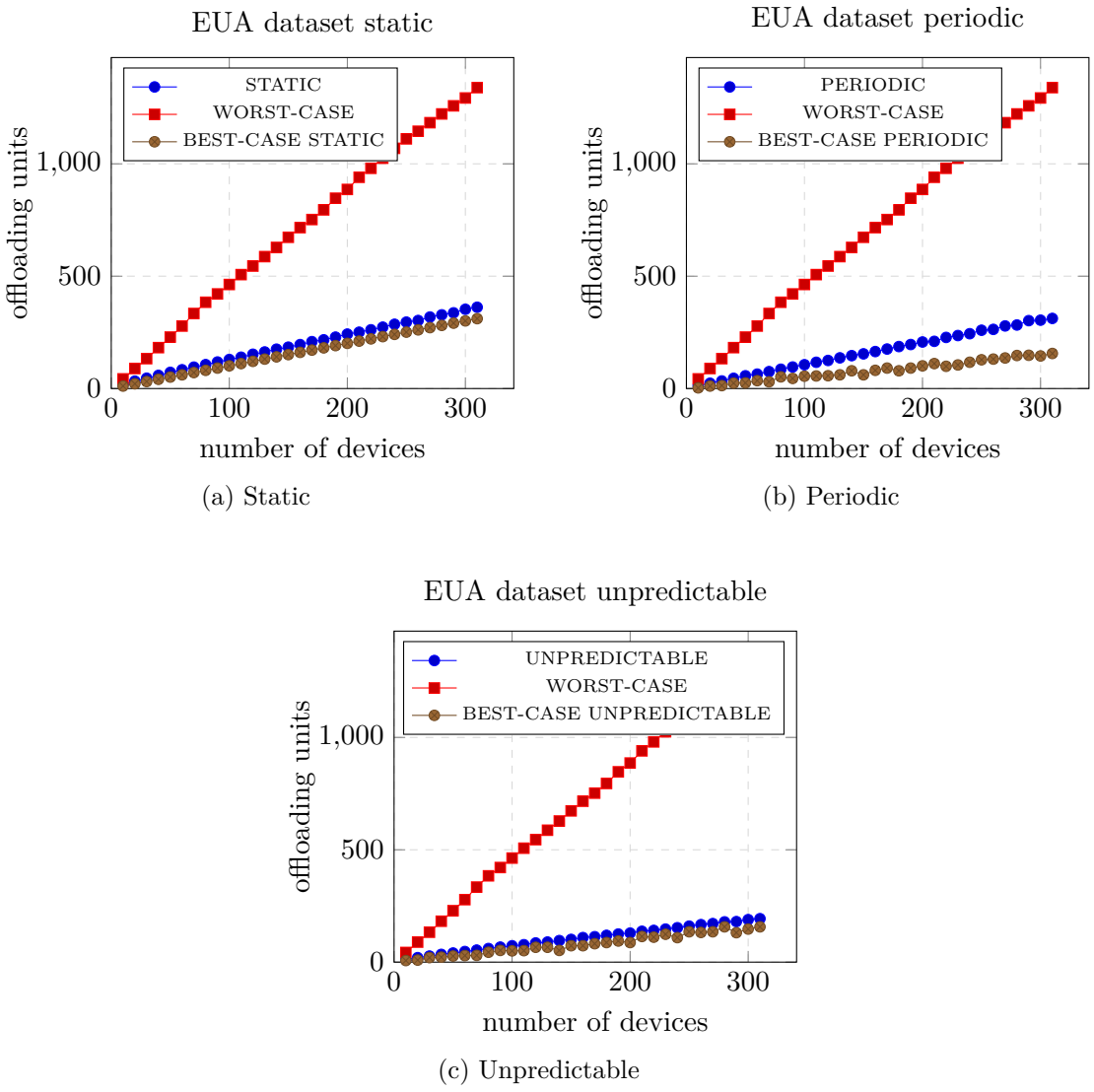
Best-Case
In order to analyze the best case for a given scenario, an example run is performed and the actual offloading units per edge node are accumulated.

Worst-Case
The worst-case is given by the connected mobile devices per edge node and in this worst-case analysis each edge node has not any information about other edge nodes in the scenario. Furthermore, the mobile devices have not any information whether they are connected to other edge nodes or not.

In Figure 5.2 the results for all three workload traces are visualized with diagrams. The predicted offloading units by the prototype are in all cases between the best-case and the worst-case. This means that the prototype can predict the offloading activities accurately. Nevertheless, in practice offloading errors can happen due to unforeseen offloading activities (e.g., users' mobility). Scenarios with user mobility are discussed in the next section. In Table 5.1 the configurations for this analysis are displayed. On the x-axes of the diagrams the number of devices is displayed and on the y-axes the offloading units are illustrated.

considered time units	10
required runs	50
computation intensity	LOW

Table 5.1: Settings for static scenario analysis



(a) Static

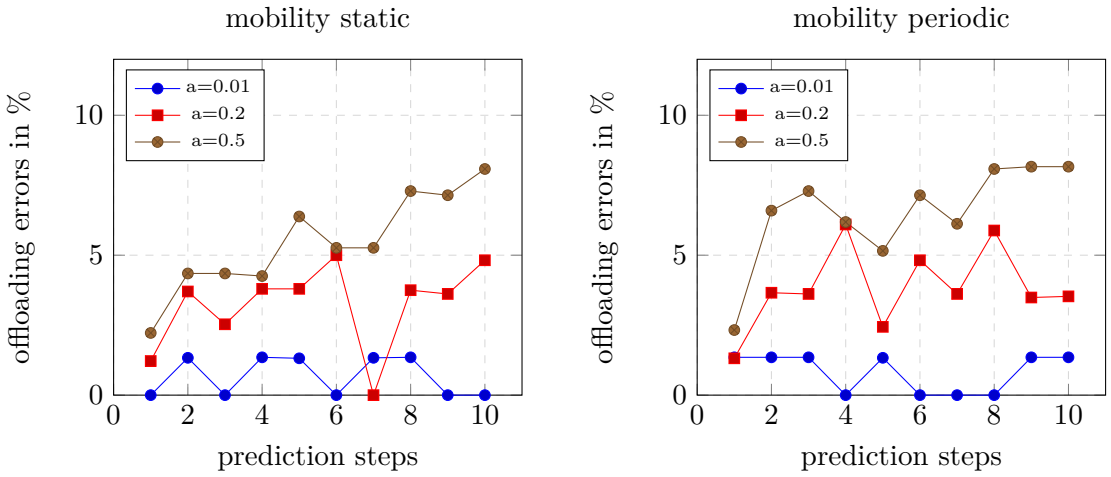
(b) Periodic

(c) Unpredictable

Figure 5.2: Predicted offloading units with static, periodic and unpredictable workload traces compared to with worst-case and best-case analysis

5.3.2 Mobility Scenarios and Offloading Errors

The users' mobility is an important factor in mobile edge computing. In this evaluation two different scenarios are analyzed. First, the introduced mobility pattern in Section 5.2 is applied to the EUA dataset. Second, real data traces from the LiveLab dataset are used to evaluate the prototype.



(a) Mobility static (b) Mobility periodic

Figure 5.3: Comparison of the different mobility scenarios and the workload traces static and periodic

In Figure 5.3 the mobility analysis of the EUA dataset is illustrated, whereas the workload traces static and periodic are analyzed. The mobility analysis of an unpredictable workload trace is not expedient because results show that offloading errors do not follow a trend or pattern in this case.

In this analysis three values for the rate of exploration are used. The significance of the different rates of exploration is explained in Section 5.2. The analysis of the edge network with the prototype is once done at time 0. Then the user's mobility is predicted for ten steps and after every step the offloading errors are counted. The prediction steps are displayed on the x-axes and the offloading errors in percent to the total number of offloadings are illustrated on the y-axes. Table 5.2 shows the settings for the mobility analysis of the EUA dataset.

The results indicate that the offloading errors will increase, if the mobility of the users rises. Moreover, the offloading errors in a low mobility scenario ($a = 0.01$) are on a low level (<2% in both scenarios) even after more steps of mobility prediction. In general, the offloading errors are on a similar level in both cases. It is particular interesting that offloading errors are not increasing faster with each additional step. One reason for this is that the mobility prediction allows to return to a previous POI.

number of devices	100
considered time units	100
required runs	50
computation intensity	LOW

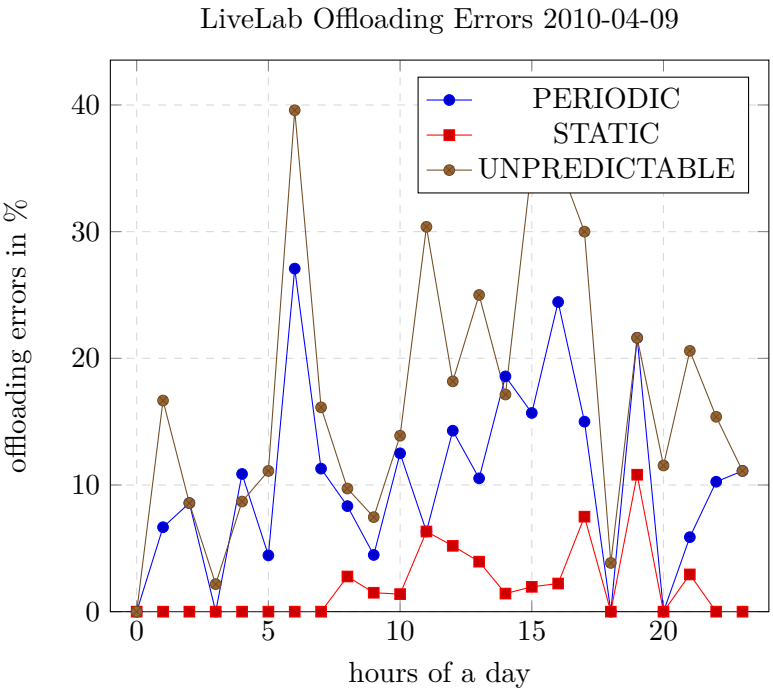
Table 5.2: Settings for mobility analysis of the EUA dataset

Beside the synthetically generated mobility of the EUA dataset, also real data from the LiveLab dataset is analyzed. Therefore, two days within the time window from 30st March 2010 to 1st April 2011, potentially many offloading activities are analyzed. In Figure 5.4 the offloading errors for the 9th April and the 16th September 2010 are outlined. As mentioned in Section 5.1.3, the edge network is analyzed every 10 minutes and every hour the *Edge Offloading Prototype* analyzes the current edge network at that time. As illustrated in Figure 5.4a (9th April 2010), there are only a few offloading errors for predicting a static workload trace. Only 27 out of 1197 offloadings failed in this scenario. The periodic and unpredictable workload produces more offloading errors. To summarise, 2,26% of the offloadings result in an offloading error in the static scenario, 10,86% in the periodic scenario and 17,79% in the unpredictable scenario. This leads to the conclusion that in this case the dataset indicates that the offloading behavior is static which can be easily confirmed because the mobile phones registered with a celltower is static in nature. Furthermore, it is true that in the static scenario 17,43 % more offloading units are predicited than in the unpredictable scenario. Similar observation can also be made in Figure 5.4b (16th September 2010). Table 5.3 illustrates the settings for the mobility analysis of the LiveLab dataset.

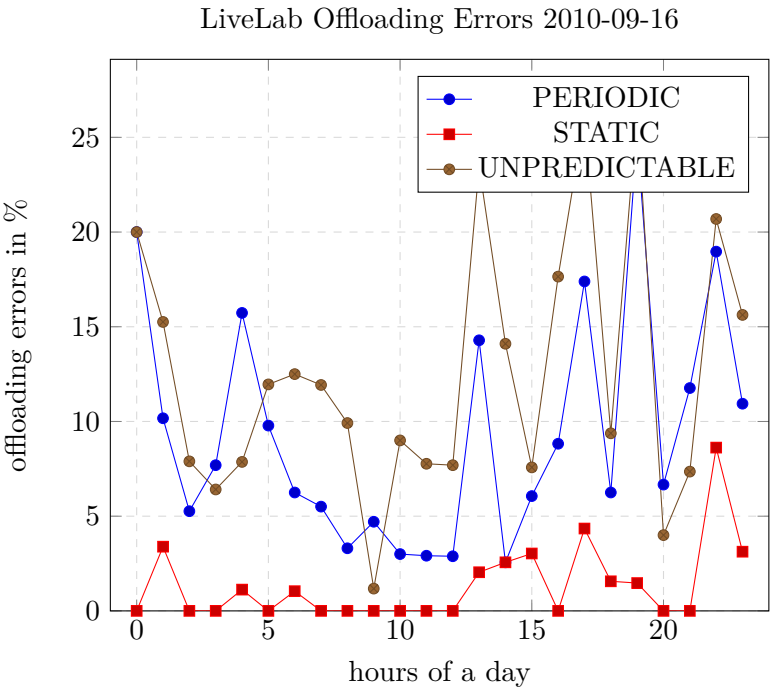
considered time units	current device size + 10
required runs	10
computation intensity	LOW

Table 5.3: Settings for mobility analysis of the LiveLab dataset

This practically leads to more resource consumption on the edge node on the hand, and to fewer offloading errors on the other hand. It should also be noted that the prototype is not able to identify the type of workload in the current scenario. However, such additional features are possible improvements for further developments of the prototype.



(a)



(b)

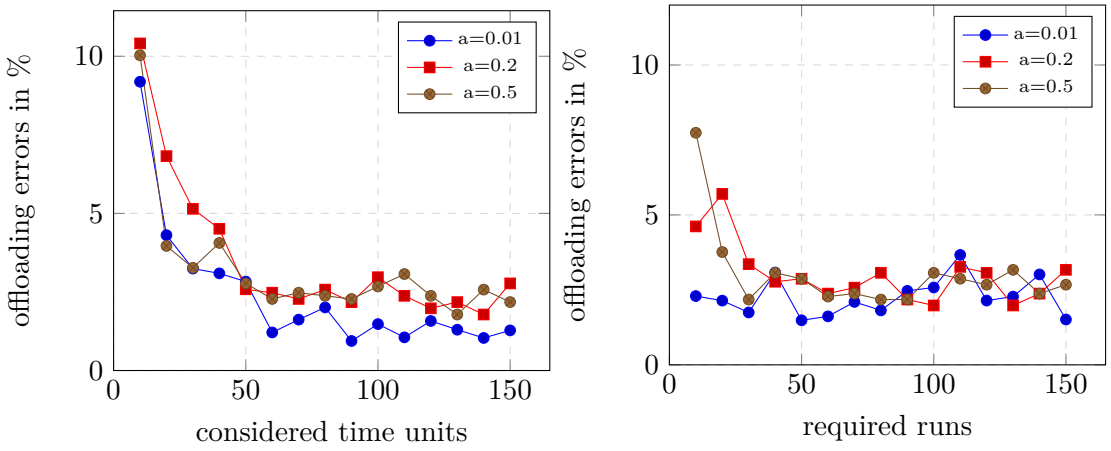
Figure 5.4: Offloading errors in percent for the LiveLab dataset for the days 9th April 2010 and 16th September 2010

5.3.3 Sensitivity Analysis

The two main properties *considered time units* and *required runs* are especially interesting for a sensitivity analysis. In such an analysis all parameters apart from one are fixed in order to show the impact on changing a significantly considerable parameter. The aim is to provide a configuration of these two properties which is minimizing the resulting offloading errors. Therefore, it is particular interesting how these two properties are influencing the offloading errors because they are required for the probability comparison introduced in Section 2.4.4. First, *considered time units* is the number of time-units considered in this analysis. Second, *required runs* is the number of independent test executions (see Section 2.4.2) which are performed for this analysis.

For the purpose of this analysis the EUA dataset with mobility prediction is used. Similarly to the mobility analysis in the previous section, the offloading errors are counted and accumulated. Furthermore, three mobility scenarios ($a = 0.01$, $a = 0.2$, $a = 0.5$) are used and analysed analogous to the simulations above. Figure 5.5 consists of two diagrams (Figure 5.5a and Figure 5.5b) and two tables (Table 5.5c and Table 5.5d), whereby on the left side the *considered time units* sensitivity analysis is displayed. On the right side the sensitivity analysis for the *required runs* property is illustrated. On the x-axes of the diagrams the numbers of *considered time units* respectively *required runs* are shown. The offloading errors in percent are visualized on the y-axes of the diagrams.

The optimal configuration for an analysis with the *Edge Offloading Prototype* is not trivial and results show that offloading errors are more likely with unsuitable configuration. As demonstrated in Figure 5.5a the offloading errors are on a high level if *considered time units* smaller than 20 are used. Similar observations are made in Figure 5.5b, if the mobility scenarios are $a = 0.2$ or $a = 0.5$. In these cases more offloading errors occur when using *required runs* smaller than 20. However, these results are scenario specific and it is quite possible that the most appropriate configuration is completely different in other scenarios. Additionally, the used settings for the two sensitivity analyses are illustrated in Table 5.5c and Table 5.5d.



(a) Considered time units

(b) Required Runs

number of devices	100
number of edge nodes	15
considered time units	10 - 150
required runs	100
strategy	PERIODIC
computation intensity	LOW

(c) Settings for sensitivity analysis of *considered time units*

number of devices	100
number of edge nodes	15
considered time units	50
required runs	10 - 150
strategy	PERIODIC
computation intensity	LOW

(d) Settings for sensitivity analysis of *required runs*

Figure 5.5: Sensitivity analysis of the properties *considered time units* and *required runs*

5.3.4 Execution Time Analysis

In order to determine the suitability of the prototype for large datasets, the needed execution time for the analyzing process is recorded. This experiment is done with the EUA dataset with 15 edge nodes starting with 10 devices and in every iteration the number of devices is increased by 10 further devices until 340 devices are reached. In Figure 5.6 the result of this execution time analysis is shown. On the x-axis the number of devices from 10 to 340 devices is displayed and the needed time in seconds necessary for the analysis process is shown on the y-axis. The configuration for the analysis is depicted in Table 5.4.

As outlined in Figure 5.6, the execution time increases exponentially (especially in a static workload scenario) with the number of devices which are considered for the analysis. As an example, the analysis for 300 mobile devices takes approximately five minutes for a static workload trace and more than one minute for a periodic or unpredictable workload trace. In a real world mobile edge computing scenario with hundreds of devices

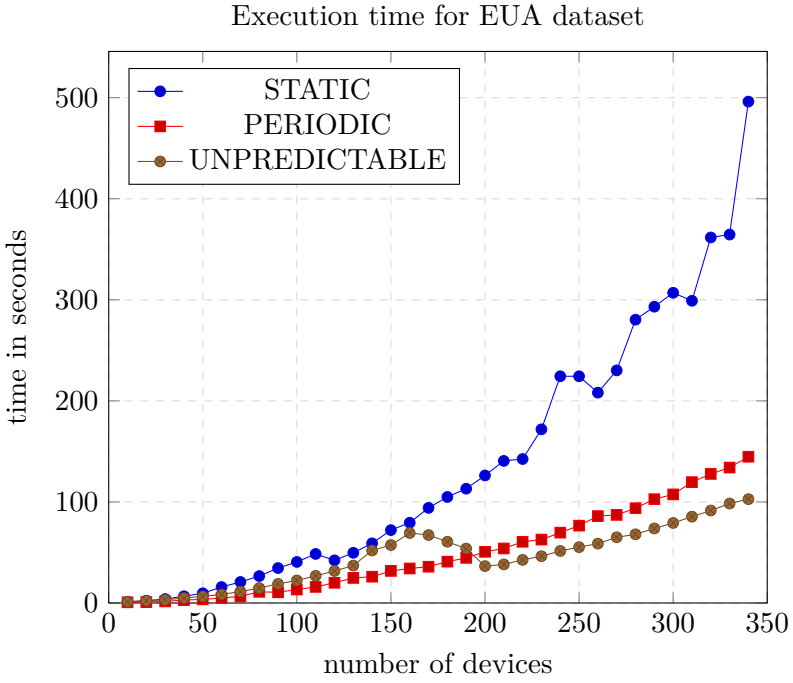


Figure 5.6: Execution time for EUA dataset

number of devices	10-340
considered time units	10
required runs	50
computation intensity	LOW

Table 5.4: Settings for execution time analysis of the EUA dataset

and edge nodes, this extremely long execution time is normally not acceptable. However, different fields of application for this prototype exist (e.g., capacity planning of an ISP (Internet Service Provider).

CHAPTER 6

Conclusion

Offloading errors in MEC are time consuming and lead to offloading the tasks to the cloud or to execute the tasks locally on the device. In any case these errors decrease the performance of the application and usability is influenced as well. One the one hand, this prototype is designed to reduce offloading errors and on the other hand, the needed computational resources on the edge nodes should also be minimized. The results show that the prototype works well in static scenarios and in scenarios with low mobility. Due to the continuous change of the edge network in a real world setting, the prediction of offloading units is difficult and the best time for a reanalysis of the network is hardly foreseeable. Nevertheless, the findings demonstrate that this prototype with model checking support can predict offloading activities and is able to provide services in a MEC more efficiently.

First, the results of the prototype can be used by offloading decision engines and second, the findings can be used to rescale the edge resources based upon the analysis results or for using the available edge infrastructure in an optimal way. Therefore, this approach is able support ISPs[1] or cloud service providers in the process of designing or monitoring their edge infrastructure. Additionally, orchestration platforms and deployment planning will benefit from the results of this approach.

However, the execution time increases with the number of devices and edge nodes significantly. To prevent this, the properties *considered time units* and *required runs* can be adapted and decreased, but this will lead to imprecise results. The importance of a suitable configuration of *considered time units* and *required runs* has been shown by a sensitivity analysis with the EUA dataset including mobility prediction. The used data traces for evaluation show the strengths and weaknesses of the proof of concept. Furthermore, the interpretation of the offloading behaviour in the LiveLab dataset (cell-towers and users) leads to static workload in most scenarios. Therefore, the results can

[1] Internet Service Provider

be interpreted that for a meaningful analysis the workload trace of the applications must be identified beforehand.

The proof of concept is a first step to utilize statistical model checking with Uppaal in a computation offloading scenario especially in MEC and further research in this area is necessary to demonstrate the benefits of this approach.

6.1 Future Work

This thesis is an initial approach to use statistical model checking in a mobile edge computing scenario. The evaluation demonstrates that the results are suitable for scenarios with low mobility. Nevertheless, there are various features and further improvements which have to be added in order to meet the current requirements in edge offloading:

Performance improvements
It has been discovered that the execution time increases significantly if a larger number of mobile devices is in the edge network. In order to address these issues, it is suggested that only parts of the overall edge network should be analysed to reduce the execution time. Additionally, also other statistical model checking tools must be taken into account with respect to their performance. Another approach is distributed statistical model checking which can increase the performance of different scenarios[26][28].

Implement ODE [2]
At present, the proof of concept is focusing on the analysis of the overall edge network. Therefore, a complete implementation of an ODE is part of the future work as this will help to better benchmark the approach. Moreover, a detailed evaluation with other state-of-the-art computational offloading approaches should be taken into account for further examination.

Mixture of different workload traces
In practice, applications with different offloading workload traces exist. In addition, offloading behavior is subject to change and does usually not follow a pattern. Thus, for the evaluation of the proof of concept different workloads, such as static, periodic and unpredictable, must be applied for one scenario. As mentioned before, also a workload recognition feature based on the current offloading activity needs to be considered for the current approach.

Auto-configuration
Finally, an additional feature for the current proof of concept will be the automatic configuration of different properties. Basically, the prototype requires 4 properties to specify (computation intensity, strategy, considered time units, required runs, strategy). In the current evaluation these properties are assigned manually whereas an automatic and appropriate configuration would have been useful.

[2] Offloading Decision Engine

List of Figures

List of Tables

Listings

Bibliography

[1] Amazon Web Service - AWS Lambda. https://aws.amazon.com/de/lambda/features/, 2019. [Online; accessed 09-Nov-2019].

[2] Amazon Web Services - Region Table. https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/, 2019. [Online; accessed 10-September-2019].

[3] Angular. https://angular.io/, 2019. [Online; accessed 19-May-2019].

[4] Eclipse IDE. https://www.eclipse.org, 2019. [Online; accessed 04-May-2019].

[5] Github - Edge server, user dataset for Edge Computing research. https://github.com/swinedge/eua-dataset, 2019. [Online; accessed 14-August-2019].

[6] GitHub Repository - EdgeOffloading. https://github.com/edamike/edgeOffloading, 2019. [Online; accessed 5-Nov-2019].

[7] GitHub Repository - EdgeOffloadingClient. https://github.com/edamike/edgeOffloadingClient, 2019. [Online; accessed 5-Nov-2019].

[8] Google Cloud Functions. https://cloud.google.com/functions/, 2019. [Online; accessed 09-Nov-2019].

[9] H2 Database Engine. https://www.h2database.com/html/main.html, 2019. [Online; accessed 19-September-2019].

[10] LiveLab: Measuring wireless networks and smartphone users in the field. http://livelab.recg.rice.edu/traces.html, 2019. [Online; accessed 12-September-2019].

[11] Microsoft Azure - Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2019. [Online; accessed 5-Nov-2019].

[12] StackOverflow - UPPAAL SMC avoid State Space Explosion. https://stackoverflow.com/questions/55951475/uppaal-smc-avoid-state-space-explosion, 2019. [Online; accessed 19-May-2019].

[13] StackOverflow - UPPAAL SMC motivating example. https://stackoverflow.com/questions/57605497/uppaal-smc-motivating-example, 2019. [Online; accessed 8-Nov-2019].

[14] Understanding Cloud Service Models – A Look At IaaS, PaaS And SaaS. https://blog.runcloud.io/understanding-cloud-service-models/, 2019. [Online; accessed 5-Nov-2019].

[15] UPPAAL - Frequently Asked Questions. http://www.it.uu.se/research/group/darts/uppaal/documentation.shtml, 2019. [Online; accessed 04-September-2019].

[16] Aceto, L., Morichetta, A., and Tiezzi, F. Decision support for mobile cloud computing applications via model checking. In *Proceedings - 2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2015* (March 2015), IEEE, pp. 199–204.

[17] Ahmed, A., and Ahmed, E. A survey on mobile edge computing. In *2016 10th International Conference on Intelligent Systems and Control (ISCO)* (Jan 2016), pp. 1–8.

[18] Akherfi, K., Gerndt, M., and Harroud, H. Mobile cloud computing for computation offloading: Issues and challenges. *Applied Computing and Informatics 14*, 1 (January 2018), 1–16.

[19] Alur, R., Feder, T., and Henzinger, T. A. The benefits of relaxing punctuality. Tech. rep., Cornell University, 1994.

[20] Aral, A., Brandic, I., Uriarte, R. B., De Nicola, R., and Scoca, V. Addressing application latency requirements through edge scheduling. *Journal of Grid Computing* (2019), 1–22.

[21] Armbrust, A. Above the clouds: A Berkeley view of cloud computing. *University of California, Berkeley, Tech. Rep. UCB*, January (2009), 07–013.

[22] Baier, C., and Katoen, J.-P. *Principles of model checking*. MIT press, 2008.

[23] Basic, F., Aral, A., and Brandic, I. Fuzzy handoff control in edge offloading. In *2019 IEEE International Conference on Fog Computing (ICFC)* (June 2019), pp. 87–96.

[24] Behrmann, G., David, A., and Larsen, K. G. A tutorial on uppaal 4.0, 2006.

[25] Bierman, G., Abadi, M., and Torgersen, M. Understanding typescript. In *ECOOP 2014 – Object-Oriented Programming* (Berlin, Heidelberg, 2014), R. Jones, Ed., Springer Berlin Heidelberg, pp. 257–281.

[26] Boyer, B., Corre, K., Legay, A., and Sedwards, S. Plasma-lab: A flexible, distributable statistical model checking library. In *Quantitative Evaluation of Systems* (Berlin, Heidelberg, 2013), K. Joshi, M. Siegle, M. Stoelinga, and P. R. D'Argenio, Eds., Springer Berlin Heidelberg, pp. 160–164.

[27] Bozga, M., Mounier, L., and Lesens, D. Model checking ariane-5 flight program, 2001.

[28] Bulychev, P., David, A., Guldstrand Larsen, K., Legay, A., Mikučionis, M., and Bøgsted Poulsen, D. Checking and distributing statistical model checking. In *NASA Formal Methods* (Berlin, Heidelberg, 2012), A. E. Goodloe and S. Person, Eds., Springer Berlin Heidelberg, pp. 449–463.

[29] Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 2658–2659.

[30] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. CloneCloud. *Proceedings of the sixth conference on Computer systems - EuroSys '11* (2011), 301.

[31] Cisco, and Cisco Systems, I. Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper. Tech. rep., Cisco, 2019.

[32] Clarke, E. M. Model Checking, 1980.

[33] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs* (1981), Springer, pp. 52–71.

[34] Clarke, E. M., and Klieber, W. Model Checking and the State Explosion Problem, 2012.

[35] Cuervo, Eduardo and Balasubramanian, Aruna and Cho, Dae-ki and Wolman, Alec and Saroiu, Stefan and Chandra, Ranveer and Bahl, P. MAUI-Making Smartphones Last Longer with Code Ofoad-mobisys-2010.pdf. *8th international conference on Mobile systems, applications, and services, MobiSys'10* (2010), 49–62.

[36] David, A., Larsen, K. G., Legay, A., Mikučionis, M., Poulsen, D. B., van Vliet, J., and Wang, Z. Statistical model checking for networks of priced timed automata. In *Formal Modeling and Analysis of Timed Systems* (Berlin, Heidelberg, 2011), U. Fahrenberg and S. Tripakis, Eds., Springer Berlin Heidelberg, pp. 80–96.

[37] David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Wang, Z. Time for statistical model checking of real-time systems. In *Computer Aided Verification* (Berlin, Heidelberg, 2011), G. Gopalakrishnan and S. Qadeer, Eds., Springer Berlin Heidelberg, pp. 349–355.

[38] De Maio, V., and Brandic, I. First hop mobile offloading of DAG computations. *Proceedings - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018* (2018), 83–92.

[39] Dinh, H. T., Lee, C., Niyato, D., and Wang, P. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing 13*, 18 (2013), 1587–1611.

[40] Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., and Hu, B. Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In *2015 IEEE 8th International Conference on Cloud Computing* (2015), pp. 621–628.

[41] Evangelidis, A., Parker, D., and Bahsoon, R. Performance modelling and verification of cloud-based auto-scaling policies. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (May 2017), pp. 355–364.

[42] Fan, C., Huang, J., Yang, D., and Rong, Z. Modeling POI transition network of human mobility. *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* (2016), 364–367.

[43] Fehling, C., Leymann, F., Retter, R., Schupeck, W., and Arbitter, P. *Cloud Computing Patterns*. Springer Vienna, Vienna, 2014.

[44] Gordon, V. S., and Bieman, J. M. Rapid prototyping: lessons learned. *IEEE software 12*, 1 (1995), 85–95.

[45] Ha, K., Chen, Z., Hu, W., Richter, W., Pillai, P., and Satyanarayanan, M. Towards wearable cognitive assistance. *MobiSys 2014 - Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (2014), 68–81.

[46] Jørgensen, K. H., Christoffersen, N. B., Petersen, R. D., and Gjøderum, T. H. *VisuAAL-an application for visualizing realistic mesh network protocol behavior through Uppaal simulations*. PhD thesis, Master Thesis. Aalborg University, Department of Computer Science, 2017.

[47] Kemp, R., Palmer, N., Kielmann, T., and Bal, H. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services* (2010), Springer, pp. 59–79.

[48] Kwiatkowska, M., Norman, G., and Parker, D. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (2002), Springer, pp. 200–204.

[49] Kwiatkowska, M., Norman, G., and Parker, D. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification* (Berlin, Heidelberg, 2011), G. Gopalakrishnan and S. Qadeer, Eds., Springer Berlin Heidelberg, pp. 585–591.

[50] Lai, P., He, Q., Abdelrazek, M., Chen, F., Hosking, J., Grundy, J., and Yang, Y. Optimal edge user allocation in edge computing with variable sized vector bin

packing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 11236 LNCS* (2018), 230–245.

[51] Legay, A., Delahaye, B., and Bensalem, S. Statistical model checking: An overview. In *Runtime Verification* (Berlin, Heidelberg, 2010), H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds., Springer Berlin Heidelberg, pp. 122–135.

[52] Mach, P., and Becvar, Z. Mobile Edge Computing: A Survey on Architecture and Computation Offloading. *IEEE Communications Surveys and Tutorials 19*, 3 (2017), 1628–1656.

[53] Mell, P., Grance, T., et al. The nist definition of cloud computing, 2011.

[54] Mikučionis, M., Legay, A., Poulsen, D. B., David, A., and Larsen, K. G. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer 17*, 4 (2015), 397–415.

[55] Naeem, A., Azam, F., Amjad, A., and Anwar, M. W. Comparison of model checking tools using timed automata - prism and uppaal. *2018 IEEE International Conference on Computer and Communication Engineering Technology (CCET)* (2018), 248–253.

[56] Naskos, A., Stachtiari, E., Gounaris, A., Katsaros, P., Tsoumakos, D., Konstantinou, I., and Sioutas, S. Cloud elasticity using probabilistic model checking, 2014.

[57] Ning, Z., Wang, X., and Huang, J. Mobile edge computing-enabled 5g vehicular networks: Toward the integration of communication and computing. *IEEE Vehicular Technology Magazine PP* (12 2018), 1–1.

[58] Pilon, A. Smartphone Battery Survey: Battery Life Considered Important. `https://aytm.com/blog/smartphone-battery-survey/`, 2016. [Online; accessed 04-April-2019].

[59] Qian, H., and Andresen, D. Jade: An efficient energy-aware computation offloading system with heterogeneous network interface bonding for ad-hoc networked mobile devices. *2014 IEEE/ACIS 15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2014 - Proceedings* (2014), 1–8.

[60] Satyanarayanan, M., Bahl, V., Caceres, R., and Davies, N. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* (2009).

[61] Shepard, C., Rahmati, A., Tossell, C., Zhong, L., and Kortum, P. Livelab: measuring wireless networks and smartphone users in the field. *ACM SIGMETRICS Performance Evaluation Review 38*, 3 (2011), 15–20.

[62] Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal 3*, 5 (2016), 637–646.

[63] Shi, W., and Dustdar, S. The Promise of Edge Computing. *Computer 49*, 5 (2016), 78–81.

[64] Taleb, T., Dutta, S., Ksentini, A., Iqbal, M., and Flinck, H. Mobile edge computing potential in making cities smarter. *IEEE Communications Magazine 55*, 3 (March 2017), 38–43.

[65] Van Nieuwpoort, R., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., and Bal, H. Ibis: A flexible and efficient java-based grid programming environment. *Concurrency and Computation: Practice and Experience 17* (02 2005), 1079 – 1107.

[66] Wald, A. *Sequential Tests of Statistical Hypotheses*. Springer New York, New York, NY, 1992, pp. 256–298.

[67] Xia, F., Ding, F., Li, J., Kong, X., Yang, L. T., and Ma, J. Phone2cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing. *Information Systems Frontiers 16*, 1 (2014), 95–111.

[68] Yovine, S. Model checking timed automata. In *School organized by the European Educational Forum* (1996), Springer, pp. 114–152.

[69] Zhao, B., Xu, Z., Chi, C., Zhu, S., and Cao, G. Mirroring smartphones for good: A feasibility study. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services* (2010), Springer, pp. 26–38.

[70] Zhao, H., Deng, S., Zhang, C., Du, W., He, Q., and Yin, J. A mobility-aware cross-edge computation offloading framework for partitionable applications. In *2019 IEEE International Conference on Web Services (ICWS)* (July 2019), pp. 193–200.

[71] Zilic, J., Aral, A., and Brandic, I. Efpo: Energy efficient and failure predictive edge offloading. In *12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)* (December 2019).

[72] Zilic, J., Aral, A., and Brandic, I. Towards Verification of Edge Offloadings. Tech. rep., 2019.

[73] Zou, P., Wang, C., Liu, Z., and Bao, D. Phosphor: A cloud based drm scheme with sim card. In *2010 12th International Asia-Pacific Web Conference* (April 2010), pp. 459–463.