VDI

REIHE 10
INFORMATIK/
KOMMUNIKATION

Fortschritt-Berichte VDI

M.Sc. Volkan Gezer,
Kaiserslautern

NR. 874

A Modular and Scalable Software Reference Architecture for Decentralized Real-Time Execution on Edge Computing

BAND
1|1

VOLUME
1|1

WSKL Werkzeugmaschinen und Steuerungen
TU KAISERSLAUTERN

A Modular and Scalable Software Reference Architecture for Decentralized Real-Time Execution on Edge Computing

Dem Fachbereich Maschinenbau und Verfahrenstechnik
der Technischen Universität Kaiserslautern
zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte
Dissertation

von
Herrn
M.Sc. Volkan Gezer
aus Istanbul

Tag der mündlichen Prüfung: 06.10.2021

Dekan:	Prof. Dr.-Ing. Tilmann Beck
Promotionskommission:	
Vorsitzende:	Prof. Dr.-Ing. Kristin de Payrebrune
1. Berichterstatter:	Prof. Dr.-Ing. Martin Ruskowski
2. Berichterstatter:	Prof. Dr. Paul Lukowicz

D 386

VDI

REIHE 10
INFORMATIK/
KOMMUNIKATION

Fortschritt-Berichte VDI

M.Sc. Volkan Gezer,
Kaiserslautern

NR. 874

A Modular and Scalable Software Reference Architecture for Decentralized Real-Time Execution on Edge Computing

BAND
1|1

VOLUME
1|1

WS KL Werkzeugmaschinen und Steuerungen
TU KAISERSLAUTERN

VDI verlag

Gezer, Volkan

A Modular and Scalable Software Reference Architecture for Decentralized Real-Time Execution on Edge Computing

Fortschritt-Berichte VDI, Reihe 10, Nr. 874. Düsseldorf: VDI Verlag 2021.
160 Seiten, 37 Bilder, 9 Tabellen.
ISBN 987-3-18-387410-1, ISSN 0178-9627,
57,00 EUR/VDI-Mitgliederpreis: 51,30 EUR

Für die Dokumentation: edge computing – echtzeit scheduling – fog computing – aufgabe übertragung – edge in produktion – edge server – software referenzarchitektur

Keywords: edge computing – real-time scheduling – fog computing – task offloading – edge in manufacturing – edge server – software reference architecture

Edge Computing is expected to solve the latency and best-effort delivery problems of the renowned Cloud Computing. However, Edge Computing must be supported with a vendor-independent, scalable, and decentralized software reference architecture to fully exploit its benefits. This dissertation explains the enablers, requirements, and conceptual approach to create such architecture, and validates it with a framework to show its possibilities.

Bibliographische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet unter www.dnb.de abrufbar.

Bibliographic information published by the Deutsche Bibliothek (German National Library)
The Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie (German National Bibliography); detailed bibliographic data is available via Internet at www.dnb.de.

D 386
Dissertation Technische Universität Kaiserslautern

ISBN 987-3-18-387410-1, ISSN 0178-9627

Foreword

The basis for this thesis originally stemmed from my position as a Researcher at German Research Center for Artificial Intelligence (DFKI). Firstly, I would like to thank Professor Martin Ruskowski, for giving me the chance to work on the related projects that allowed me to perform research on this area and deepen my knowledge.

Additional thanks go to Prof. Dr.-Ing. Kristin de Payrebrune for chairing the examination board and to Prof. Dr. Paul Lukowicz for accompanying my examination as a referee. I also would like to thank my colleagues at DFKI - especially Andre Hennecke, Christiane Plociennik, Max Birtel, Moritz Ohmer, Pascal Rübel - for their valuable inputs and feedbacks to my work, also colleagues in Team Shannon for the working atmosphere, team collaboration, openness for discussion, and the team activities within or outside the working hours, and Dr. Achim Wagner for supporting me during structuring my thesis with status meetings, and corrections.

Special thanks go to my family: my mother Billur, my father Sebahattin, and my wife, Elif Nur, who kept me motivated and patiently supported me during the stressful times.

Volkan Gezer

Contents

Abstract

Cloud Computing, or the Cloud, became one of the most used technologies in today's world, right after its possibilities had been figured out. It is a renowned technology that enables ubiquitous access to tasks that need collaboration or remote monitoring. It is widely used in daily lives as well as the industry. The paradigm uses Internet Technologies which rely on best-effort communication. Best-effort communication limits the applicability of the technology in the domains where the timing is critical. Edge Computing is a paradigm that is seen as a complementary technology to the Cloud. It is expected to solve the Quality of Service (QoS) and latency problems that are raised due to the increased count of connected devices, and the physical distance between the infrastructure and devices. The Edge Computing adds a new tier between Information Technology (IT) and Operational Technology (OT) and brings the computing power close to the source of the data. Computing power near devices reduces the dependency to the Internet; hence, in case of a network failure, the computation can still continue. Close proximity deployments also enable the application of Edge Computing in the areas where real-timeliness is necessary. Computation and communication in Edge Computing are performed via Edge Servers. This thesis suggests a standardized and hardware-independent software reference architecture for Edge Servers that can be realized as a framework on servers, to be used on domains where the timing is critical. The suggested architecture is scalable, extensible, modular, multi-user supported, and decentralized. In decentralized systems, several precautions must be taken into consideration, such as latencies, delays, and available resources of the neighbouring servers. The resulting architecture evaluates these factors and enables real-time execution. It also hides the complexity of low-level communication and automates the collaboration between Edge Servers to enable seamless offloading in case of a need due to lack of resources. The thesis also validates an exemplary instance of the architecture with at framework, called Real-Time Execution Framework (RTEF), with multiple scenarios. The tasks used are resource-demanding and requested to be executed on an Edge Server in an Edge Network comprising multiple Edge Servers. The servers can make decisions by evaluating their availabilities, and determine the optimal location to execute the task, without causing deadline misses. Even under a heavy load, the decisions made by the servers to execute the tasks on time were correct, and the concept is proven.

Kurzfassung

Cloud Computing, oder die Cloud, wurde zu einer der meistgenutzten Technologien in der heutigen Welt, gleich nachdem ihre Möglichkeiten entdeckt wurden. Es handelt sich um eine anerkannte Technologie, die einen ubiquitären Zugriff auf Aufgaben ermöglicht, die Zusammenarbeit oder Fernüberwachung erfordern. Sie ist sowohl im täglichen Leben als auch in der Industrie weit verbreitet. Das Paradigma nutzt Internet-Technologien, die auf Best-Effort-Kommunikation beruhen. Die Best-Effort-Kommunikation schränkt die Anwendbarkeit der Technologie in den Bereichen ein, in denen das Timing kritisch ist. Edge Computing ist ein Paradigma, das als eine ergänzende Technologie zur Cloud gesehen wird. Probleme mit der Dienstgüte (QoS) und Latenzzeiten, die durch die steigende Anzahl angeschlossener Geräte und der physischen Entfernung zwischen Infrastruktur und Geräten entstehen, sollen dadurch gelöst werden. Das Edge Computing fügt eine neue Ebene zwischen der Informationstechnologie (IT) und der Betriebstechnologie (OT) hinzu und bringt die Rechenleistung nahe an die Quelle der Daten. Das Näherbringen der Geräte reduziert die Abhängigkeit vom Internet und kann somit Berechnung auch bei einem Netzausfall sicherstellen. Ebenso kann dadurch das Einsatzgebiet des Edge Computing um Bereiche erweitern, in denen Echtzeitfähigkeit gefordert ist. Berechnung und Kommunikation im Edge Computing werden über Edge Server durchgeführt. Diese Dissertation schlägt eine standardisierte und hardwareunabhängige Software-Referenzarchitektur für Edge Server vor, die als Framework auf Servern realisiert werden kann, um sie in zeitkritischen Domänen einzusetzen. Die vorgeschlagene Architektur ist skalierbar, erweiterbar, modular, mehrbenutzerfähig und dezentralisiert. In dezentralen Systemen müssen verschiedene Maßnahmen berücksichtigt werden, wie z.B. Latenzen, Verzögerungen und verfügbare Ressourcen der benachbarten Server. Die resultierende Architektur wertet diese Faktoren aus und ermöglicht die Ausführung in Echtzeit. Sie kapselt auch die Komplexität der Low-Level-Kommunikation und automatisiert die Zusammenarbeit zwischen Edge-Servern, um ein reibungsloses Offloading zu ermöglichen, falls ein Bedarf aufgrund von Ressourcenmangel besteht. Die Dissertation validiert auch eine exemplarische Instanz der Architektur mit einem Framework, genannt Real-Time Execution Framework (RTEF), mit mehreren Szenarien. Die verwendeten Aufgaben sind ressourcenintensiv und sollen auf einem Edge-Server in einem Edge-Netzwerk mit mehreren Edge-Servern ausgeführt werden. Die Server können durch Auswertung ihrer Verfügbarkeiten Maßnahmen ergreifen und den optimalen Ort für die Ausführung der Aufgabe bestimmen, ohne dass es zu Terminüberschreitungen kommt.

Abbreviations

API	Application Programming Interface
CDN	Content Delivery Networks
CM	Cloud Manufacturing
CPS	Cyber-Physical Sytems
CPU	Central Processing Unit
EDF	Earliest Deadline First
ETD	Edge Topology Designer
GPOS	General-Purpose Operating System
HPC	High-Performance Computing
HTTP	HyperText Transfer Protocol
I/O	Input/Output
IoT	Internet of Things
ISO	International Organization for Standardization
IT	Information Technology
LAN	Local Area Network
LST	Least Slack Time
NAPATA	Non-resumable And Preemptible Aperiodic TAsk
OS	Operating System
OT	Operational Technology
POSIX	Portable Operating System Interface
PSC	program/software/command
QoS	Quality of Service
RTEF	Real-Time Edge Framework
RTOS	Real-Time Operating System
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
VM	Virtual Machine
VP	Virtual Processor
WCET	Worst-Case Execution Time
WCEU	Worst-Case Execution Utilization

1 Introduction

The Internet changed the way people live and how they have access to information. The way how data is being processed and accessed has changed rapidly. This includes everyday tasks, like reading emails, but also factory automation and device control. Thanks to the accessibility and benefits that it brings into the lives, new technologies and uses are emerging. One of the renowned technologies is the Internet of Things (IoT). IoT connects smart objects and builds a more intelligent, adaptive, and self-configurable system [APZ18]. The success of IoT leveraged connected devices to the Internet, as well as the data generated and transferred through the Internet, tremendously. However, this growth brings several issues which could degrade the Quality of Service (QoS) and introduce delays. Due to bandwidth limitations, even failed requests can occur. Cyber-Physical Systems (CPS) are systems that are composed of a computing platform, the physical world, sensors and actuators [Na21]. Hence, design decisions have to take several aspects into consideration: definition of an architecture, design and integration of systems and components, connectivity, interoperability, safety, security, reliability, computing, and storage. Smart grids, autonomous piloting systems in avionics or automotive, medical surgery or industrial control systems can be given as examples to the CPS [KM15]. The primary goal for CPS is an effective, reliable, accurate and real-time control. In IoT, however, the goal is better resource sharing and management, enabling interfacing among different networks, data storage, data mining, data aggregation, and information exchange with high QoS.

In 1992, the Internet-connected user device count was approximately one million, which went up to 500 million in 2003, thanks to the increased usage of personal computers. From 2003, IoT became even popular and reached three billions of connected devices. In 2012, wearable devices raised this number even further to 8.7 billion. In 2018, this number went up to 11.2 billion when home appliances are also became connected. The rapid growth in this number is due to the involvement of traffic lights and small personal objects, such as toothbrushes and digital watches. Finally, even door levers are expected to be part of the smart objects in 2020 [Ev11; NC14]. The Internet, by its nature, provides best-effort service. Therefore, Interned-based solutions also rely on best-effort communication. Thus, time-critical applications cannot benefit from the Internet. This limitation necessitates the introduction of an alternative approach for interacting with time-critical devices.

Integrating different technologies from multiple providers and merging them into a single infrastructure is complex. The maintenance of such infrastructure is hard and costly. In case of

a failure, relying on a single Information Technology (IT) infrastructure can also increase the downtime of the communication, which causes non-productive time. Cloud Computing can scale well when resources are not enough to complete a task. It also balances server resource usage by transferring (offloading) the tasks to other available servers. The load balancers also perform the same when a server is out of service.

After a marked tendency towards Industry 4.0, formerly centralized computing is becoming more decentralized by the involvement of CPS, IoT, and more intelligent components. Edge Computing is a recent paradigm that is believed to solve time-criticality and centralized computing problems, by providing enough computing power close to the source, namely the end devices, or the devices at the field level [GUR18]. It combines multiple state-of-the-art technologies, including CPS and IoT, and it is placed closer to the end devices. Computation close to the end devices also reduces the dependency of the systems to the Internet, making real-time computation and real-time control possible, even after a network outage. It also enhances system monitoring and gives more control over the data. Chemical industries, communication, energy, and food industries, defence and emergency services, nuclear reactors, and airborne vehicles can be given as examples to domains which are expected to benefit from Edge Computing due to their critical timing and QoS requirements. A software architecture and framework developed under the frame of Edge Computing will be beneficial if it is deployed in one of the scenarios of the domains mentioned above. However, first, an accurate description of the Edge Computing and its components is needed. Next, its requirements must be listed and concepts must be defined. Lastly, the concepts must be realized and validated.

Zuehlke [Zu10] stated that focusing too much on the technology to accomplish a market advantage and combining them into single devices limits the cost-efficient solution with sufficient quality measurement, due to shorter product cycles. Instead of eliminating the humans in the production, creating self-coordinating work teams and avoiding dreary job assignments helped the companies increase the productivity and the quality of the product. This methodology is called *lean production*. Lean means reducing complexity, unnecessary information, and technologies [Zu10]. Edge Computing solutions that abstract the complexity of lower-level technologies and enable customizable deployments can be considered a part of lean production. These kinds of solutions reduce the effort to start production and set the focus on the production quality instead.

IoT concept in daily life can be realized by combining several technologies: smart devices, networked systems, mobility of devices, and utilization of standards. However, daily life and industrial requirements vary. IoT in the factories requires – in addition to these technologies - reliability and safety under industrial conditions [Zu10].

Technologie-Initiative SmartFactory KL e.V[1] (short: *SmartFactory-KL*) is a non-profit initiative

[1]Website: `https://smartfactory.de`

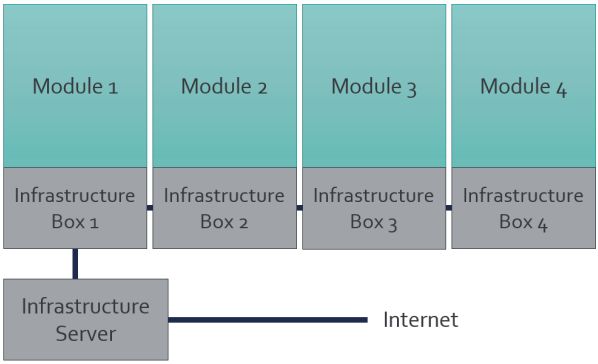

Figure 1.1: A high-level overview of the modular testbed demonstrator from *SmartFactory-KL*.

that was established in 2005, to implement joint Industry 4.0 projects for the factories of the future (FoF). Together with its consortium members, *SmartFactory-KL* creates manufacturer-independent demonstrators in a realistic, industrial production environment. The first Industry 4.0 demonstrator built by the initiative is a modular, distributed, and plug-and-produce production platform (See Fig 1.1). Each module is built by different industrial partners and performs one discrete step of the production. All modules work interoperably and produce a customizable and individualized product. However, there is no direct communication between the modules. The modules are connected to each other via infrastructure boxes. These infrastructure boxes seen in Fig. 1.1 are used only to supply power, pressured air, and network connection. The production information is transferred using the memory integrated into the product itself.

The thesis follows the vision of *SmartFactory-KL* by reusing the decentralized, scalable, interoperable, and modular system idea and reduces the complexity of the system by abstracting and encapsulating the low-level functionalities. First, it shortly explains the history of well-known Cloud Computing and its benefits, together with some related work. Then, it makes a transition to the Edge Computing paradigm, describing what it is and the problems it is believed to solve. Next, the work introduces a conceptual software reference architecture for Edge Servers to overcome these problems. According to this work, Edge Servers are physical computers in which the architecture is installed. They are the main components of Edge Computing and used to (pre-)process, compute, and possibly store data requested by End Devices. End Devices are resource-limited devices with low or no computing power. They request tasks from any of the connected Edge Servers. An End Device can be a smart sensor, machine, computer, mobile phone, or worker assistance glasses. Connecting the Edge Servers and End Devices creates Edge Networks. Separate Edge Networks can also be connected to each other. The overall system which one or multiple Edge Networks come together is called Edge Computing. Edge Computing can also have the ability to communicate with the Cloud.

The dissertation explains how Edge Servers and their software components are networked to-

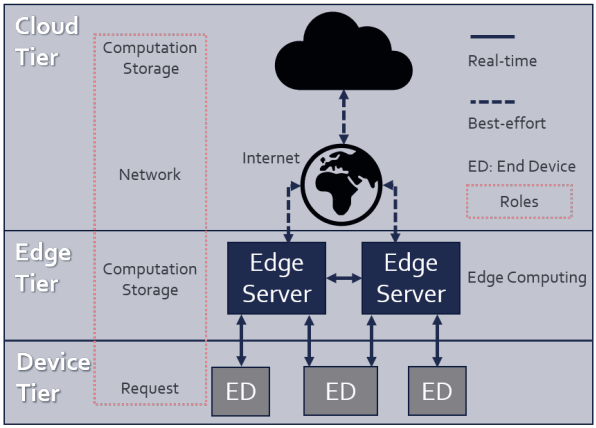

Figure 1.2: An example network consisting of three End Devices (ED) and two Edge Servers. Dashed arrows depict best-effort communication, whereas solid arrows are real-time capable connections. The aim is to execute tasks requested by EDs seamlessly and collaboratively, without causing them to miss their deadlines.

gether for a collaborative execution. There is not a limitation on the network topology. Hence, any of the available topologies or their combinations can be used to create Edge Networks. The architecture aims to guarantee the response time, especially for real-time applications such as motion control or automation. When an End Device requests a task from an Edge Server in an Edge Network, the execution is performed on any of the Edge Servers, seamlessly. If the Edge Server that initially receives the task does not have enough resources to execute the task, it can offload the request to another Edge Server in the network. If there are other tasks running in the Edge Server, their execution also needs to be planned to avoid deadline misses. This process is performed without a centralized load balancer or resource monitor. Each server is able to make independent decisions to choose a suitable server in case they are unable to perform the task.

The network participants (Edge Servers) introduce themselves to each other automatically, including their functions and hardware specifications. Then, Edge Servers decide on where to execute the task and return task output to the first requester. These decisions and the planning of the execution are complicated processes. Each Edge Server needs to know the specifications, available resources, and running tasks of the other neighbouring servers to yield a good result. To achieve that, it is also necessary to introduce standard communication patterns that can be understood and parsed by the Edge Servers. The dissertation also introduces decision making method for offloading, a novel scheduling algorithm to plan execution of specific tasks, and patterns for inter-communication of the Edge Servers. The precise positioning of the components and an arbitrary network are pictured in Fig. 1.2.

The proposed architecture enables decentralized, scalable, and multi-user-supported (real-time) execution. It neither uses nor endorses using any proprietary standards that limit the interoperability. On the contrary, it is platform and hardware-independent. Nevertheless, the specifications of the hardware determine if the solution is real-time capable or not. The architecture also supports the integration of legacy software or programs that are developed to work only on a single computer. After analysis of existing literature, no reference architectures realizing these concepts were found. To validate the reference architecture, based on the concepts, a software framework, called Real-Time Edge Framework (RTEF), is realized. This validation was performed using multiple scenarios and setups for correctness. In these scenarios, multiple tasks with high load are requested from an Edge Server at different times. The tasks had strict timing requirements, which an Edge Server alone would not be able to execute them on time. The decisions made by the RTEF included execution on itself, throttling the load down, or offloading to another server. If a manual test of the tasks is feasible, the RTEF was also able to make the right decisions and execute the tasks on time, to meet their deadlines.

Chapter 2 will describe some of the existing concepts, technologies, and the related work. End of each technology will explain how it is related to the reference architecture, what its equivalent is and how it is going to be used. The following section will define the problems of Cloud Computing and explain why there is a need of another technology.

1.1 Problem Definition

Cloud Computing is a best-effort technology. Frotzcher et al. [Fr14] collected the requirements for wireless communications in industrial automation as well as its available solutions. They also showed the various cycle time requirements of different domains in their research. Depending on the automation level, the response times can vary between 1 microsecond (μs) and 1 second (Fig. 1.3). Even though response times less than one second are theoretically achievable with the Cloud, due to its nature, the Cloud cannot guarantee this timing at all times. Moreover, sending raw sensor data to the Cloud, without protection of the sensitive information is not desirable due to security reasons.

According to Satyanarayanan [Sa17], although direct fibre connections can overcome the latency and bandwidth problems, covering a large geographical area requires multiple access points. Each of them introduces queuing and adds a delay which is not fixed due to the multiple route possibility from the source. Virtual Reality (VR) applications can be given as an example of daily life as a latency-critical application. VR applications typically require less than 16 ms latency to achieve adequate performance [Sa17].

In the industrial domain, there usually exist several safety-critical computer systems or controllers to prevent real-time deadline misses, or directly designed for controlling purposes. The

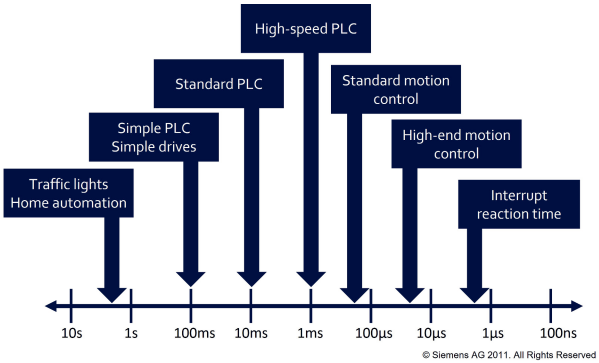

Figure 1.3: Depending on the application area, the timing requirements of automation applications vary from 1 microsecond (μs) to 1 second (Siemens, 2011).

most widely known controllers are Programmable Logic Controllers (PLC). They follow the IEC 61131 standard, which can be obtained for a fee [IE19]. PLCs provide real-time solutions for process controls where time-critical scenarios exist. Current PLCs also have more processing power than before, and they support networking. They also became multitasking, meaning they provide prioritization or scheduling to plan executions of the tasks. Control applications are usually vendor-dependent, far from flexibility and reusability. PLC programs are also often highly application-specific and neither flexible nor reusable for other applications. If systems need redundancy or fault-tolerant applications, programming PLCs becomes more complicated. Moreover, when considering scalability, advanced process controls require more engineering time and cost due to the complexity of the logic. Almost all of the PLC providers use proprietary toolchains. Although the standards define how PLCs should behave, extensions made to the standard by the companies obstruct the compatibility of a program written for a different manufacturer. Therefore, having PLCs from multiple providers and seamless execution of the same program is almost not possible. This limitation also prevents meeting the interoperability and scalability requirements, which are two of the principles of Industry 4.0 [HPO16; Ma16].

Unlike PLCs, there also exist several programming platforms for industrial controllers from different manufacturers. As an example, using CODESYS[2] and logi.CAD with its runtime system logi.RTS[3], multiple different hardware platforms can be programmed per the IEC 61131 standard. Although their programming toolkits are free to use, runtime systems require licence fees per device.

The benefits of Cloud Computing could overcome some problems of traditional industrial control technologies, but it comes with its own set of issues. That is why Edge Computing is an alternative to Cloud Computing, which provides similar benefits, without the shortcomings. If

[2]Website: `https://codesys.com`
[3]Website: `https://www.logicals.com/en/logi-cad/`

a problem requires a fault-tolerant, distributed, decentralized, scalable, or interoperable solution, then Edge Computing must be supported with a standardized and vendor-independent architecture. Currently, Cloud solutions provide offloading or load balancing in case more resources are needed. However, as of today, as Satyanarayanan [Sa17] mentioned, there are no software mechanisms nor algorithms that enable collective control in decentralized computing, hence, in Edge Computing.

This thesis proposes a software reference architecture for the Edge Servers in Edge Computing to deal with the problems that Cloud Computing has. While inheriting the capabilities of Cloud solutions such as scalability, decentralized computing, and resource sharing, Edge Computing helps reduce latency and dependency to the Internet. The work also tries to fill the gaps for an interoperable and collaborative computing in the industrial domain by enabling real-time computing at the edge, with minimum effort. The proposed architecture combines the benefits of other existing technologies and systems such as PLCs to meet the requirements which will be specified in Sec. 2.1.2. As of today, state-of-the-art lacks a reference architecture and methodology, which enables execution of (legacy) real-time tasks in decentralized environments. This architecture and the proposed framework allows converts regular computer into an Edge Server. These servers will then be able to share their resource information and work collaboratively to perform the requests from End Devices. Edge Servers enable seamless execution of such tasks within an Edge Network by abstracting low-level communication methods and help legacy software/programs work collaboratively on modern systems. From an End Device perspective, the Edge Network will be seen as a single system. When an End Device requests a task from an Edge Server, the task output will be returned to the requester, hiding the ongoing low-level activity. In the background, the Edge Servers will make decisions depending on the resource availabilities such as load and distance, to come to a consensus about the optimal location of execution. In case a server cannot execute the task on time, it will be offloaded to another server. If a task can be executed on a server, set of algorithms will plan its execution, depending on the type of the task. Unlike load balancing or offloading in Cloud Computing, this thesis does not introduce a centralized load balancer. Instead, it enables this feature on all Edge Servers. Since all available resources are known by the all participating servers, the server selection is dynamic, i.e., updated real-time based on the topology.

Calculation of the server to offload to and the time to execute the requested task depends on multiple parameters and is challenging. In a decentralized environment, for collaboration, all participants should be aware of the neighbouring resources and their current status. They should also have standard communication patterns to share this information correctly. The thesis, therefore, defines communication patterns for collaboration and presents decision mechanisms to use internally on an Edge Server as well as externally between Edge Servers in the Edge Network. The thesis also introduces a novel scheduling algorithm to execute legacy non-resumable and preemptible aperiodic tasks on time without missing their deadlines. Moreover,

the thesis will implement the proposed architecture as a framework to validate the concepts. Then, it will be tested using multiple scenarios and different setups by monitoring various task sets. These task sets have manually been proven to be schedulable, in advance. Finally, the framework behaviour will be analysed under heavy load and evaluated for correctness.

This work brings several technologies and concepts together, such as the Internet of Things (IoT), Cyber-Physical Systems (CPS), grid computing, load balancing, virtualization, and collaborative computing. These technologies will be detailed in Chapter 2. In the literature, there is not a vendor-independent and decentralized solution that provides enough flexibility to execute real-time tasks while monitoring system resources in a network for an optimal decision of execution location. A decentralized system can enable fair use of resources in a network if well-designed. Moreover, it also increases the network's fail-safe functionality. The work also partially uses the results of FAR-EDGE[4] and AUTOWARE[5] European projects. These projects dealt with the real industrial problems to enable Edge Computing in traditional factory environments.

The thesis also answers the following research questions:

Question 1: *How to structure an interoperable, hardware-agnostic, and operating system independent reference architecture for Edge Computing, to overcome latency problems of Cloud Computing?*

Question 2: *How to create a collaborative and decentralized Edge Network that allows task offloading between Edge Servers?*

The questions above also raise the following questions:

Question 3: *How to exchange information between End Devices and the Edge Servers to enable collaboration and resource-awareness in the network?*

Question 4: *What kind of decision mechanisms on the Edge Server side are necessary to execute End Device tasks on time, without missing their deadlines, including legacy software/programs? How can they be implemented?*

All questions are going to be answered with the conceptual design of the software architecture and validated in the implementation and validation section. The next section will explain the objectives and what is to be achieved at the end of the research.

1.2 Objectives

Edge Computing paradigm has a great potential to benefit from Cloud technologies and solve the limitations of Cloud Computing (Sec. 2.1). However, a deep understanding of the underly-

[4]Website: `http://faredge.eu/`
[5]Website: `https://autoware-eu.org/`

ing technologies is required to exploit the usage of the paradigm. This dissertation proposes a software reference architecture for Edge Servers in the Edge Computing domain, to solve decision problems for collaborative and decentralized computation given in Sec. 1.1. The proposed solutions will be mathematically formulated, and they will be validated. Before introducing the concepts, to ease the understanding, the objectives of the reference architecture are given below:

- Design a hardware-agnostic and OS-neutral software reference architecture for flexible, self-configurable, multi-user-enabled, interoperable, reliable, extensible, secure, and scalable Edge Servers; to respond to the requests from End Devices or other Edge Servers within the same network.
- Create an Edge Network with Edge Servers, to offload the tasks to the other Edge Servers in case a server lacks resources for an on-time execution.
- Create decision mechanisms to throttle the CPU usage of the tasks down to avoid preemption, or to schedule to avoid offloading — all to meet deadlines.
- Define standard introductory commands/patterns for automatically exchanging available services and resources of Edge Servers.
- Enable execution of legacy software/programs on the Edge Servers by introducing wrappers.
- Implement the software reference architecture as a framework to realize the concepts.

The implemented framework is also going to be validated against multiple complex scenarios to test its efficiency and success. The behaviours will be analysed under heavy load by testing task sets, which are manually proven in advance for their schedulability. The scenarios will use different setups.

Several aspects are out of scope of this thesis. For instance:

- This thesis does not cover task migration; moving a running task to another Edge Server and asking for resumption from where it is left off. This is proposed as possible future work in the outlook (Sec. 6.2).
- Combining periodic and aperiodic tasks requires using a schedule server. This work does not implement one, and it requires isolation of periodic and aperiodic tasks for optimal scheduling. Failing to do so may still schedule the tasks, but it does not guarantee the best scheduling.
- The algorithms assume that there is no jitter and they cause no additional overhead during the calculation.
- The hardware is considered to be ideal, has infinite disk space and memory, to reduce the

complexity of the problem.

- The execution durations of the processes are known a priori.

Another aspect that is not considered in this thesis is the network communication. It is assumed that the established communications are real-time.

1.3 Approach

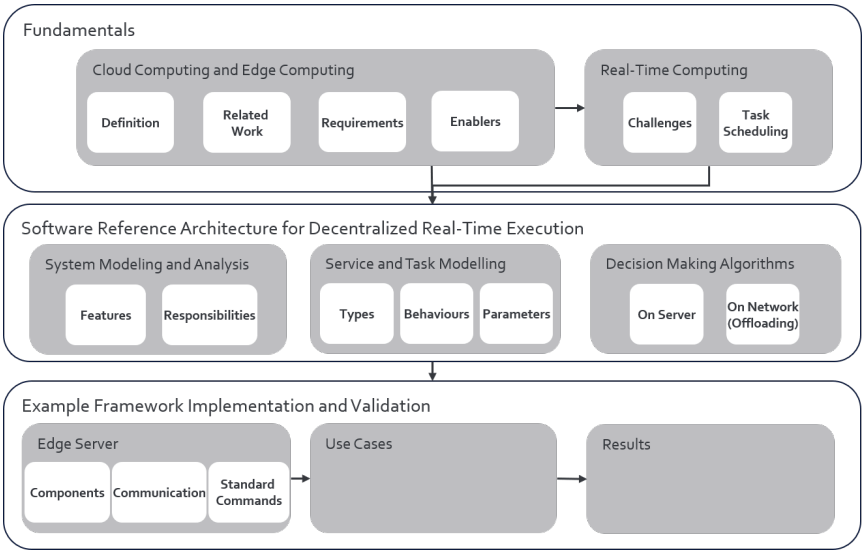

Figure 1.4: A summary of the approach to achieve the objectives.

To solve the problem explained in Sec. 1.1 and achieve the objectives mentioned in Sec. 1.2, an approach to narrow down the domain is planned. First, the history of Cloud Computing is explained, and some examples are given. Next, the issues on Cloud Computing are listed. Then, Edge Computing is defined, and it is explained how Edge Computing is expected to solve the issues of the Cloud Computing (Sec. 2.1). Combining Cloud Computing and Edge Computing, requirements (Sec. 2.1.2) and enablers (Sec. 2.1.3) are listed. Further, another related domain, Real-Time Computing, is explained together with its challenges and possible scheduling algorithms (Sec. 2.2). Next, a conceptual software reference architecture design is proposed (Chapter 4). Lastly, this architecture is used to realize the concept as a framework. The framework; and thus the architecture is validated with multiple use cases (Chapter 5). This approach to achieving the objectives is summarized in Fig 1.4.

2 State of the Art

This chapter is divided into three sections. Sec. 2.1 widens the definitions of Cloud Computing and Edge Computing, and lists the existing work in the same direction. Moreover, it explains the requirements (Sec. 2.1.2) and enablers (Sec. 2.1.3) of Edge Computing for the thesis to reach its goals. Sec. 2.2 moves into Real-Time Computing domain, which the thesis aims to connect with Edge Computing. It also explains its challenges (Sec. 2.2.1) and how the scheduling works on real-time operating systems (Sec. 2.2.2). The chapter is then finalized with a summary (Sec. 2.3).

2.1 Cloud Computing and Edge Computing

Computing has always switched between centralized and decentralized computing since the 1960s. In the 1960s, local terminals did not have enough computing power, and thus the computations were performed on hosts (servers). The optimizations were also focused on the limited resources of the hosts [Ba14]. Until the 1980s, the computation was centralized. Centralized systems tend to have lower administrative and operational costs, and their configurations are more straightforward compared to the decentralized systems [Sa17]. From the 2000s, the widespread use of personal computers and increased resources such as computing power on these systems, enabled computation directly on the client machines, eliminating the need of a centralized server [Ba14].

A decentralized system is a system in which each participant is controlled by itself, and the results are aggregated to create a universal system response [Ea17]. It can be other mechanisms that monitor decentralized participants, but the hierarchical structure is minimized. In case of a failure, automatic recovery is only possible with a decentralized system. Decentralized systems also increase flexibility, autonomy, and responsiveness [KDN17; Wa10]. However, the development, maintenance, and management of these systems are complex. This complexity pushed the computing and storage back to the server-side [Ba14]. The procedure of computation and storage in remote servers is called Cloud Computing. Sharing the unused computing power to balance the workload was one of the most significant advantages of Cloud Computing, or "the Cloud". Moreover, the Cloud reduced the marginal cost of the system administration and expenditure to create a data centre.

Cloud Computing [MG11], or only the Cloud, is an emerging technology and allows machines and people to access the data ubiquitously. It enables on-demand sharing of available computing and storage resources among its users which could be either human or machine, or even both. These resources are available in data centres. The data centres consist of one or more physical servers located on-site or in a distant physical location, in which they are accessed using Internet technologies. From its first initial concepts in the 1960s, the idea was brought to life as Remote Job Entry (RJE) [IB68]. Since then, different experimentations were made to exploit the usage of large-scale computing. The apparent success of the Cloud emerged new application areas in the last decade. The Cloud is well-used for daily tasks, such as emails or for collaborative work, file and data storage, finance, or remote monitoring.

Cloud Computing offers a Software-as-a-Service (SaaS) approach over the Internet. Services that the Cloud can also provide are Hardware-as-a-Service (HaaS), Platform-as-a-Service (PaaS), Infrastructure-as-a-Service (IaaS) [Ar10], and Function-as-a-Service (FaaS). Amazon Elastic Cloud Computing (EC2) and Microsoft Azure platforms can be given as examples to IaaS and FaaS. In 2006, Amazon released its elastic Cloud platform, EC2[1], and in 2008, Microsoft also entered in Cloud Computing with its Azure platform[2]. Both platforms present a wide range of tools to create and manage Cloud servers and also to deploy user software in the Cloud. Ubiquitousness, scalability, and accessibility are some of the evidential reasons that make Cloud so prevalent, which are also provided by these existing platforms. The available resources in the Cloud enabled low-powered or resource-limited end devices to perform complex tasks in the Cloud, saving exceptional computational time [HHG16]. The ubiquitousness of the Cloud allowed data to be accessed from anywhere and any time, as long as there is an active Internet connection.

Similar to Information Technology (IT) domain, the manufacturing domain also switched between centralized and decentralized production [KDN17]. It also proliferated its interest in the Cloud and looked for new possibilities for using it. One of the paradigms using Cloud terms in this domain is Cloud Manufacturing (CM). CM term was first introduced by Li et al. in 2010 [LS10]. Then, several authors [DS12; Xu12; Zh10a] have proposed their definitions of CM. However, the manufacturing-as-a-service concept was first seen in literature in 1990 by Goldhar and Jelinok [GJ90]. They discussed the transformation of factories from a mechanical focused operation to IT. They also discussed the possibility of mass-customization with computer integrated manufacturing (CIM).

The concept of CM is a combination of Cloud Computing, Internet of Things (IoT), Cyber-Physical Systems (CPS), service-oriented architecture (SOA), service-oriented manufacturing (SOM), virtual manufacturing, and the virtual enterprise [Re15]. One recent research on CM [SJ18] states that many works define CM as an emerging concept of virtualization of distributed manufactur-

[1]Website: `https://aws.amazon.com/ec2`
[2]Website: `https://azure.microsoft.com`

ing and congregating resources to provide a reliable and high-quality transaction of the manufacturing process.

The CM concept was also seen in additional research done by different groups. Rajagopalan et al. [Ra98] defined a system with clients, manufacturing services, and process brokers, which enables multiple users to access a design software by the introduction of plug-ins. Wu et al. [DS13] defined the CM using the work of [MG11] and [Sm09]. According to their work, CM is a customer-oriented manufacturing model that provides on-demand access to the shared collection of various and distributed manufacturing resources, to form temporary, reconfigurable production lines, enabling increased efficiency and reducing product life cycle costs. In another article, Wu et al. reported the vision and state-of-the-art of CM in the fields of automation, industrial control systems, service composition, flexibility, and proposed implementation models in 2013 [Wu13].

In the scope of CM, there have also been several initiatives, some of which target fully automated production. CloudFlow [HHG16] and CAxMan[3] are two of the projects funded by the European Commission that aim at task orchestration in the Cloud. In these projects, the software developers integrate their solutions in the Cloud using the provided tools. Then, users create workflows that specify the order of execution of the Cloud services stored in a common database. The workflows are started via the workflow manager (WFM) and executed seamlessly. The final results are finally displayed to the user. The Cloud services communicate with each other through WFM, which acts as a broker. These projects aimed at implementing a platform to connect services from different providers and execute them to address the user needs.

As may be understood from the definition and the related work, CM does not target controlling the factories remotely or performing real-time computations from distant servers. Instead, it provides access to a service pool, where participants find and choose the requested services. It is defined as a parallel distributed system where all kinds of involved users throughout the manufacturing life cycle are serviced, on-demand [Zh10b]. Similar to many Cloud solutions that can be deployed in a public, private, or hybrid cloud, CM can be deployed as a private cloud as well.

Whether it is nearby or lies in a far distance, using a single infrastructure to keep a system reliable may seem reasonable. However, in case of a failure in the infrastructure, the downtime may be costly and hard to recover. Scaling this kind of systems may also be hard. Companies and research institutes continuously seek solutions to avoid low Quality of Service (QoS) due to insufficient hardware and network resources. Cloud solutions are supportive of catching the competition on a global scale to create upgradeable solutions and eliminate these hardware limitations swiftly [MUK00]. Data centres for Cloud solutions usually provide scalable hardware to respond to the demands of their users. Usually, tasks in scalable systems are distributed via

[3] Website: `https://caxman.eu`

load balancers to distribute the load equitably among servers (See Sec. 2.1.3). Nevertheless, a failure in centralized load balancers also hinders productivity. Moreover, the physical distance from the data source to the Cloud is one of the primary reasons for high latency and low QoS. Edge Computing is an alternative approach to address these issues.

The rapid and continuous shift from mass to individualized production changes production conditions. These are mainly due to market demands and shorter product cycles [Fe09]. Existing paradigms in the industrial domain do not satisfy the increasing demand for adapting production systems to new product variants [Pa18]. Increasing digitalization and demand due to short product-cycles emerge new technologies. These new technologies focus on the flexibility and scalability of the production systems [MUK00]. With the idea of flexible and scalable servicing, Edge Computing also targets simplifying the execution of complex tasks seamlessly, without limiting their usability. Similarly, the Cloud Computing provides on-demand resources to its users to perform requested tasks on remote servers. However, assigning real-time tasks to the Cloud is infeasible since the Cloud follows a best-effort approach through the Internet, rather than an on-time reaction. Moreover, in the case of a network failure, the computational power is lost. To perform (near) real-time computations and continue functioning even after a network outage, the computation power needs to be close to the field or device tier. The paradigm, which adds a tier between the Cloud and device tier, and moves the computational power near the user as much as possible is called *Edge Computing*.

A layer is a logical organisation of a set of services, devices, or software with the same/similar specific functionality, mainly defined for the abstraction of tasks. A tier is, however, a physical deployment of layers for scalability, security and to balance performance [Lh05]. The tier created between the client and Cloud servers, where the computation is performed, and possibly the data is stored, is called Edge Tier. Edge Computing enables decentralized computation in this tier.

The roots of Edge Computing go back to the 1990s where Akamai Technologies introduced Content Delivery Networks (CDN) to increase web performance [Di02]. They cached contents at the Internet's edge, aiming to reduce requests on the site's own infrastructure and faster response times for the users, by responding to their requests using nearby servers. Noble et al. [No97] first demonstrated the potential of Edge Computing by realizing a speech recognition scenario on resource-limited devices. They offloaded the computation to a nearby server, and the results delivered an adequate performance. In 2012, Bonomi et al. [Bo12] introduced a new paradigm called *Fog Computing*. Unlike Edge Computing, its participants are distributed over a broader network, similar to CDN. They also explain the need for a unifying platform to create a distributed intelligence. In 2014, Chang et al. [Ch14] proposed a new model for Cloud Computing, with the name *Edge Cloud*. Then, they tested the performance of their architecture with indoor localization application to evaluate latency, and with video monitoring application to measure the bandwidth. Their results showed a better performance compared to the existing

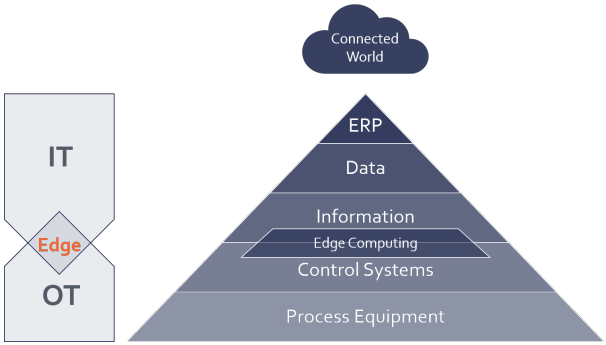

Figure 2.1: Location of Edge Computing in an Automation Pyramid. Edge Computing is placed at the intersection between OT and IT.

Cloud solutions.

Edge Computing is one recent technology that can help decentralize the control of the systems, decrease the dependency on the Internet and a single central IT system, and increase flexibility and scalability. It can be inserted where the IT intersects with Operation Technology (OT) in the automation pyramid, as seen in Fig. 2.1. Below OT, all tasks are expected to give the expected results at the expected time. In IT, however, the timing requirements are less critical; most of the time, failures on timing reduce the QoS. Delivering the expected results at expected times is the primary topic of real-time computing. More on the real-time computing will be explained in Sec. 2.2.

Edge Computing can perform computations close to the devices, and it can also play a role as a buffer to filter out the raw data and reduce the network traffic between the data source and the Internet. It can also reduce privacy and security risks of the confidential data that is being exposed to the Internet because data can be processed locally [Sh16]. There are several available computers or gateways, which have pre-installed operating systems (OS), platforms and generic Inputs/Outputs (I/Os) to allow migration from legacy systems to this new technology [Be21; CO21; De20; Kl21; TT16]. However, only a limited amount of the existing solutions support (near) real-time execution. Besides, almost all of them are proprietary and not optimized for working with solutions from other providers. Automation infrastructures from different vendors often limit the interaction of solutions from different companies. Providers usually administer their management interfaces, and solutions can establish networks only with the solutions from the same provider. This may be the case to enforce the usage of their proprietary standards and products. Alternatively, this enforcement can be for safety or security reasons, or both. A standard and unifying architecture to diminish the cumbersome installation and configuration steps of a network consisting of multi-vendor solutions is missing. As there is no ”fit-for-all” guide for the setup, the preparation phase requires different solutions to be analysed and well-studied. The setup and learning phase also needs time and cost investment.

Most of the time, field tier devices (or end devices) are resource-limited, and not all operations can be performed in the device tier. Especially, computationally heavy tasks such as Artificial Intelligence (AI) or image processing jobs are often offloaded to more powerful computers for completion. These can be other servers in the vicinity, or the Cloud.

Edge Computing targets reduced latencies and high QoS that are hindered by the Cloud solutions, mainly due to relying on the wide-area network (WAN) connection and the physical distance between the participants. While overcoming these problems, it also benefits from Cloud Computing advantages. Different researches define Edge Computing as Edge Cloud, Fog Computing or Cloudlet [Bo12; Ch14]. Dolui and Datta [DD17] define Edge Computing as a superset of Fog Computing, Cloudlet Computing, and Mobile-Edge Computing. However, these terms are interchangeable, and throughout this work, only Edge Computing will be used.

A lot of latency-sensitive domains such as wearable cognitive assistance systems [Ha14] may benefit from Edge Computing. Choi et al. [Ch17] lists the challenges that Fog Computing has and how the research could be directed in the domain. Several research activities clearly indicate that there is a reasonable performance gain in latency and bandwidth-intensive applications thanks to Edge Computing [CDO17; Wa17; Yi17; Zh17]. McChesney et al. [MWT19] proposed a benchmarking suite that can evaluate the performance of Cloud-only, Edge-Cloud, and Edge-only systems. They execute several applications using these three setups and compare the performance of communication latency, computation latency, and impact of concurrent users on the latency. The improvements for using only Edge or Edge-Cloud is also reported in their work.

This thesis proposes a software reference architecture for Edge Servers to enable collaborative and real-time task executions using an Edge Computing approach, together with its requirements and enablers. Edge Servers are physical hardware that are responsible for finding the optimal server and executing the tasks requested by the End Devices, without missing their deadlines. The architecture is designed to be hardware and operating system agnostic. It defines standards, decision, and execution mechanisms for Edge Servers to enable collaboration and offloading in an Edge Network, including each other. As mentioned in the introduction (Chapter 1), End Devices are resource-limited devices with low or no computing power. They can be sensors, end-user devices, or smart modules. They request tasks from any of the connected Edge Servers. Then, the tasks are executed on the most available Edge Server, decided by a common agreement of all Edge Server in the network. To enable collaborative execution, one or more Edge Servers and End Devices establish a network together to create their resource-aware Edge Networks. An Edge Network needs at least an Edge Server and End Device to operate. However, there is no limitation on the network topology. More details of the architecture will be elaborated in Chapter 4.

The thesis also introduces a new preemptive and online scheduling algorithm called Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling, which is integrated into the

architecture to decide on the execution order with negligible overhead and low complexity. This algorithm can also plan execution of non-resumable legacy tasks for optimal scheduling.

Based on this conceptual architecture, the thesis realizes a framework, called Real-Time Edge Framework (RTEF). The framework is developed using the Java programming language. Finally, the decision mechanisms are validated mathematically, and the framework is evaluated using multiple scenarios for correctness. After literature research, no reference architectures nor frameworks were found that feature these specifications. The next section will list set of existing work that provide complete solutions for generic use, rather than application-specific solutions.

2.1.1 Related Work

This section will list existing activities that follow Edge Computing approach to execute tasks in the proximity of field devices with or without the help of the Cloud.

There have been several novel architecture proposals on the Cloud for big data analytics in the process control industry [Go17] and also in the direction of Edge Computing. The constant rise in the global competition of the manufacturing domain created new paradigms. CM is one of the paradigms which was first introduced in 1990. It assumes that the modern manufacturing industry is being transformed into global manufacturing networks whose systems and resources can commonly be used. It is considered one of the main directions in the development of the manufacturing industry [YDC17].

The research activities mostly combine Cloud and Edge to deal with time-sensitive tasks. Mohamed et al. [Mo17] proposed a service-oriented middleware for CPS. It provides a service-based infrastructure to develop and operate CPS applications. The approach also enables the integration of CPS with Cloud and Edge Computing.

Pallasch et al. [Pa18] introduced a concept to utilize Cloud and Edge Computing for industrial control. They refer to the devices connected to field tier devices as Edge Devices (Edge Server in this thesis). Their setup uses Amazon Web Services (AWS) Cloud services for non-real-time, but computing-intensive tasks. One Edge Device has a direct access to the AWS. This device has several sensors serially connected to it. In the setup, two robots and two IoT devices (that also act as Edge Devices) are connected. Instead of the robots, the IoT devices are connected to the robot controllers and the data travels through these devices for robot controlling. The research shows that industrial control using Edge Computing approach is a feasible solution, in terms of aggregating and processing the collected data, and feedback control loops in the shop floor. However, even though the setup has a network of Edge Devices, the IoT devices closest to the robots can work only with the Edge Devices attached to them, and Edge Devices do not provide task offloading. They act as gateways to forward the task to the Cloud and return the response back to the original requester.

Vicks et al. [Vi15] introduced a virtualized Robot Controller and a virtualized Programmable Logic Controller (vPLC), to enable outsourcing control functions of an industrial robot. They created Virtual Machines (VMs) to realize Programmable Logic Controllers (PLCs) and a VM in the Cloud to perform control operations that demand lower real-time requirements. Horn and Krüger from the same group then tested the feasibility of this novel architecture [HK16]. They performed this test using three different experiment setups. First, they changed the program of existing hardware PLCs to use them only as connectors to the hardware. Second, they removed the hardware PLCs and used microcontrollers as connectors. Lastly, they directly connected the solution with the hardware. The results showed that the direct connection had the lowest latency. However, the work was specific to this use case, and the offloading was not dynamic, i.e. it did not automatically decide where to execute the task.

Elbamby et al. [EBS17] investigated the problems of Edge Computing and a cache-enabled Edge Network. They proposed a clustering method to group end-users with the same interests on specific tasks. The idea was to track end-users and the popularity of the tasks that they requested and to compute the results in advance. The solution was simulated, and the results gave 91% better latency results with guaranteed reliable computations. It allowed network users to offload their tasks to any Edge Server in their vicinity. However, the servers were neither allowed to offload the tasks to each other nor the Cloud. Furthermore, their intended to achieve less latency on average, rather than real-time computations.

Sonmez et al. evaluated the performance of three different possible generic Edge Computing architectures: single-tier, two-tier, and two-tier with a load balancer [SOE17b]. The evaluation analysed the performance of each architecture on wireless communication, and it was performed using a simulator based on CloudSim [CL19]. The results showed that the two-tier approach with load balancer had given the best results. However, this architecture needs a centralized load balancer, which the task is first directed to. The load balancer is responsible for transferring the task from the pool of Edge Servers to one of the available Edge Servers. In the case of a load balancer failure, no recovery mechanism can balance the load or share the tasks within the network. The scenario tested the feasibility of a latency-intolerant application but did not realize a real-time use case.

Mayer et al. [Ma17] introduced an emulator to test Fog Computing applications without deploying them on large network topologies. The emulator is able to test the efficacy of different network topologies and allows the creation of topologies from scratch. The applications in their emulator can run on Docker containers [Do21]. Containers make it easier for the developers to evaluate their Fog applications in different setups, without actually deploying them on real-world environments. Another emulator that uses the Docker containers is contributed by [Co18]. Coutinho et al. proposed a framework for Fog Computing that enables the deployment of the Fog nodes as software containers. Their work allows the testing of components using third-party systems through standard interfaces. They claim that the framework can be used

to test real-world scenarios with minimal changes. Both of these activities [Co18; Ma17] follow the same direction as this thesis. However, they do not focus on the real-timeliness of the tasks nor decision mechanisms for a collaborative and seamless execution.

Yi et al. [Yi15] analysed the goals and challenges in Fog Computing and implemented a platform prototype for Fog Computing. They define Fog Computing as a geographically distributed computing architecture with a resource pool that contains one or more connected heterogeneous devices at the edge of a network, and not exclusively backed by Cloud services. Their work supports mobility, has offloading capability and location-awareness. However, the implementation determines the server to offload based on the user location, rather than a collaborative decision of all available servers. When users leave the area covered by the current Fog system, VMs that contain user-related data must also be migrated to the active server. Bruneo et al. [Br16] designed a Fog platform based on OpenStack. Their platform focuses on smart city applications but goes in the direction of data mobility. Their framework enables code injection at runtime through the Cloud. Nevertheless, neither it deals with resource-intensive tasks nor offers offloading between servers.

Cozzolino et al. [CDO17] introduced an Edge offloading architecture to run tasks at the edge of the network. They used MirageOS unikernels[4] to isolate and embed application logic in Xen[5]-bootable images. Cozzolino et al. then discussed the effect of local data on computation time on different hardware. They also justified the limitations of the existing IoT hardware and virtualization platforms.

In addition to research work, some companies also proposed architectures and platforms in the Edge Computing domain. The architecture proposed by IBM considers the requirements for autonomy and self-sufficiency of production sites. Their architecture is three-layered to balance the workload between the Edge (named as Proximity Network), Plant, and the Enterprise. The challenges of the architecture are listed as productivity gains for high throughput, failure prevention for a reliable system and high product quality, and flexibility while hiding the complexity and allowing reconfiguration without much effort [IB17]. OpenFog Consortium proposes another reference architecture [Op17]. This architecture names the core principles as pillars. Pillars group requirements and functionalities. These pillars are Security, Scalability, Openness, Autonomy, Agility, and Programmability. OpenFog Reference Architecture is proposed after covering requirements collected from industrial use cases. In 2018, it was accepted as an official standard for Fog Computing as IEEE 1934 [IE18a]. It lists several recommended aspects to create a full-featured Fog Infrastructure with use cases that have no real-time requirements.

One of the commercial frameworks ready for the enterprise is called Everyware Software Framework[6]. It adds provisioning, advanced security, remote access, and diagnostics monitoring to

[4]Website: `https://mirage.io`
[5]Website: `https://xenproject.org`
[6]Website: `https://esf.eurotech.com/`

IoT gateways. The framework supports field protocols to collect data, process it at the edge, and publish it to the several IoT Cloud platforms. Nevertheless, the framework does not provide offloading between the servers. If the Internet connection is lost, the servicing stops.

Another recent initiative to build a common platform for Industrial IoT Edge Computing is EdgeX Foundry [Ed20]. It was launched by Linux Foundation and the initial contribution made by Dell. EdgeX Foundry is a vendor-neutral open-source software platform that interacts at the Edge of a network. It defines its requirements in architectural tenets. It is platform agnostic in terms of hardware and operating system, flexible in terms of replaceability, augmentability, or scalability up and down. It is also capable of storing or forwarding data, intelligent to deal with latency, bandwidth, and storage issues, secure, and easily manageable. A similar framework is called Liota. Firstly, it aims to be easy to use, install, and modify. Secondly, it targets a general, modular and enterprise-level quality. This framework is governed by VMware and is also open source [VM17].

There are also several works done for computation and control in the Cloud, combining hardware and software. A research project called ”pICASSO” focuses on the control of a robot using a Cloud-based control platform. The project implemented a platform and Cloud controller that can perform motion planning and control for industrial robots [Kr16]. A work by Givehchi et al. [Gi14] studied several industrial use cases for using virtual control service in a private Cloud. Instead of using hardware PLC on the site, they used a computer containing multiple cores and dedicated each core as a virtual PLC to control sensors and actuators. The solution suggests a slightly lower performance software PLC, compared to the hardware PLCs, but it expands the variety of useable software and improves the flexibility.

Goldschmidt et al. did another study on Cloud-based control [GMS15]. Their work introduces a new architecture for scalable and multi-tenant Cloud-based control, utilizing virtual PLCs. It also considers and evaluates the architecture concerning its scheduling policies and time-sensitiveness. The Cloud architecture is located in a different physical location than the industrial site where the actual control is done, and the communication is performed through the Internet. The results showed over 99% success rate for tasks requiring a response within one second. They suggest that architecture is feasible for soft or firm real-time applications. However, as mentioned earlier, relying on an Internet connection is not a robust solution where the timing is critical, especially for hard-real time tasks.

Realizing an unproven concept in real environments without testing and validating is costly in terms of engineering time and monetary expenses. Failure in the design may also be disastrous. Nevertheless, virtual environments can simulate several hours of real environment tasks in a couple of minutes and save much time.

CloudSim is a framework to model and simulate Cloud Computing infrastructures and their services. It supports modelling and simulation of large scale Cloud data centers, their application

containers, costs as well as power consumption [CL19]. Another simulation tool to evaluate the reliability of the system is called iFogSim, and it is implemented by Gupta et al. [Ha16]. It is based on CloudSim and allows the addition of Fog or Edge devices, creation of topologies and evaluation of resource management policies focusing on latencies [Ha16]. Sonmez et al. [SOE17a] introduced another simulator on top of CloudSim, which is called EdgeCloudSim. It adds a mobility model and non-fixed delays into the network which is fixed in iFogSim. The simulator also gives detailed information on resource usage as well as task success rates. In both simulators, the data is passed to the Cloud in case there are no resources available in the Edge/Fog Server. However, in this thesis, the Edge Servers can also offload the tasks to other Edge Servers in the network, by considering their available resources, connection statuses, and computation delays. Nevertheless, in the thesis, the End Devices do not have mobility; only the data has.

Lera, Guerrero, and Juiz introduced another simulator for IoT scenarios in Fog Computing, called YAFS [LGJ16]. They evaluated its performance compared to iFogSim using three different complex scenarios. Their work performed slightly better results than iFogSim. Similar to other simulations, it did not consider real-time tasks.

One of the biggest problems of Edge Computing is the non-existence of common and widely-accepted standards for Edge Computing [Sh16]. Although there exist numerous researches on the topic, there are no available simulators nor architectures in the literature that deal with offloading the tasks of immobile End Devices between the Edge Servers nor a standard Edge Server architecture which is capable of performing real-time calculations. The aim of this research is not only to propose another architecture but also to analyse the existing architectures and consider industrial requirements to make up a generic software reference architecture. The architecture must be decentralized, vendor-independent, multi-user-supported, collaborative, modular, extensible, and real-time capable. This work also implements a framework based on this novel architecture, providing a simulator to validate the correctness of the results.

Some of the notable features of existing work are also summarized in Table 2.1. Resource-awareness column shows whether a decision to offload is made considering the current server or the network resource availability. User mobility is whether the data source moves physically. Local server offloading means if the servers in the same network are able to offload the tasks between each other, rather than the Cloud. Remote maintenance column shows whether monitoring or configuration can be changed remotely. Latency modelling indicates the way how the latency between the devices is calculated. This column is valid only for the solutions that provide simulation functionality. Parallel execution presents if a task can be executed in parallel using multiple servers. Load balancing states whether a load balancing mechanism is used for fair task distribution. If yes, it defines whether it is a centralized or decentralized one. Simulation functionality shows whether the solution can be used without deploying on hardware. Finally, hardware-ready column denotes whether the solution is ready to be deployed on hardware.

Table 2.1: Comparison of state-of-the-art solutions following Edge Computing approach (✓: Supported, -: Not supported).

Work	Resource-awareness	User Mobility	Local Server Offloading	Remote Maintenance	Latency Modelling	Parallel Execution	Real-timeliness	Load Balancing	Simulation Functionality	Hardware-Ready
Chang et. al. [Ch14]	-	-	-	-	random latency	✓	-	-	-	✓
Pallasch et. al. [Pa18]	-	-	-	-	n/a	-	✓	-	-	✓
Vicks et. al. [Vi15]	-	-	-	-	n/a	-	✓	-	-	✓
Elbamby et. al. [EBS17]	✓	✓	-	-	probabilistic latency	-	-	-	✓	-
EdgeCloudSim [SOE17a]	✓	✓	-	-	probabilistic latency	-	-	centralized	✓	-
Yi et. al. [Yi15]	✓	✓	✓	-	n/a	-	-	-	-	✓
Bruneo et. al. [Br16]	-	✓	-	-	n/a	-	-	-	-	✓
Cozzolino et. al. [CDO17]	✓	-	-	-	n/a	-	-	centralized	-	✓
IBM [IB17]	-	-	-	✓	n/a	-	-	centralized	-	✓
OpenFog [Op17]	-	✓	✓	✓	n/a	-	-	centralized/ decentralized	-	✓
Everyware [Eu21]	-	-	-	✓	n/a	-	-	centralized	-	✓
EdgeXFoundry [Ed20]	✓	-	✓	✓	n/a	-	-	centralized	-	✓
Liota [VM17]	-	-	-	✓	n/a	-	-	-	-	✓
pICASSO [Kr16]	-	-	-	-	n/a	-	✓	-	-	✓
YAFS [LGJ16]	✓	✓	✓	-	parameter-based	-	-	decentralized	✓	-
This thesis	✓	-	✓	✓	parameter-based	-	✓	decentralized	✓	✓

The next section will list the requirements to meet, to enable the aforementioned features. The requirements are collected from literature and Edge Computing projects such as FAR-EDGE and AUTOWARE that have directly worked with industrial partners.

2.1.2 Requirements

Requirements are set of necessary or desired functionalities or characteristics of a system, product, or service. The Cloud Computing and thus, the Edge Computing also introduce some requirements that are necessary for a flawless task execution. This section will recommend high-level requirements identified for the architectural design of an Edge Computing solution in industrial contexts. The requirements were collected from the existing literature [Ge19], as well as from the industrial companies that were part of two projects related to Edge Computing, namely FAR-EDGE[7] and AUTOWARE[8]. These requirements were considered in this thesis, as the resulting architecture aims to achieve a close performance to the industrial standards.

As mentioned in Chapter 1, solutions in IT cannot be directly applied to the industry. The industrial domain has stricter requirements than the ones in the IT domain. The central part of the thesis focuses on the real-time capability of the solution with offloading functionality. In this section, identified industrial requirements, some of which are common with the IT domain, will be briefly explained. It may be the case that some specific industrial scenarios do not need all of the listed requirements. However, the resulting architecture fulfils these requirements as well.

Interoperability

Edge Servers communicate with other Edge Servers and various devices. The solution should support widely-used communication protocols and standards. Moreover, the solution should also be hardware-agnostic as much as possible. This flexibility will remove the technology barrier and avoid vendor lock-in problems that may occur.

Scalability

Scalability is the physical expansion of the system. Unlike legacy systems, the solution should be ready to scale in case more Edge Servers or End Devices are connected to the Edge Network. It should always respond to the increasing demand. Newly added devices must be introduced to the other components in the network.

[7] Website: `http://faredge.eu/#/partners`
[8] Website: `https://autoware-eu.org/#partners`

Extensibility

Computing technology is developing rapidly. The solution should allow easy deployment of new software and devices, with minimal (re-)configuration. The reference architecture should also support extensibility.

Time sensitiveness

As explained in the introduction (Chapter 1), below OT, the activities must be (near) real-time. Beginning of this chapter also stated that Cloud Computing provides best-effort service. To enable applicability on OT, an Edge Computing solution is expected to guarantee an on-time and predicted response.

Reliability

Similar to time sensitiveness, reliability is also a critical requirement for real-time tasks. Depending on the level of real-timeliness, a failure may be fatal. Therefore, it is vital to have a reliable system that reacts when it is needed and how it is needed. Fault-tolerant systems also fall into this category. If a controller of a running system fails to function, then, the backup system should continue from where the task is left off. The physical reliability requirements for Edge Servers providing services are similar to the ones in Cloud Computing: Harsh environments, such as factories and construction yards, require water-proof ceiling, fanless computers, and dust-proof systems. In power plants, a magnetic shield is also necessary.

Security and Privacy

Typical Cloud Computing solutions store sensitive data from enterprises (high-technology or manufacturing companies) or people from all over the world. However, leakage of these data may cause significant financial loss for the companies, or personal data may be abused. The usual practice is to send data to the Cloud after encryption. Edge Computing is at the factory level; thus, the companies have more control over the data. However, if Edge Servers can offload the tasks to the Cloud, then the security considerations of Cloud Computing must also be taken into account. Additionally, Edge Computing solution should prevent unauthorized users from accessing the data.

Intelligence

Edge Servers can have the ability to preprocess the data before sending it to the Cloud. The preprocessing will save some bandwidth and reduce network traffic. The Edge Servers can also complete calculations without sending them to the Cloud. Computation at the edge will also keep the computing power alive when the Internet connection is lost. Close proximity also means reduced latency. The integration of artificial intelligence (AI) or machine learning (ML) methods also fall into this category.

Abstraction

Abstraction hides the complexity of the low-level systems. When a solution is available, abstracting it via Application Programming Interfaces (APIs) ensures that the users can only access the provided functions, without breaking the integrity of the system. APIs also increase backward compatibility in case the architecture sees a major change.

2.1.3 Enablers

This section will explain the existing technologies that have been in use for Cloud Computing and adapted or used for Edge Computing.

Communication Protocols

Distributed systems require communication with multiple devices. To enable communication of many devices with each other as much as possible, a choice of widely-used communication protocols must be given. Depending on where they are used, not all protocols may be suitable for each application. Some of them are only good for low transfer rates (e.g. Universal asynchronous receiver-transmitter, UART), and some of them only provide high speed in close distances (e.g. Serial to Peripheral Interface, SPI [Bu18]). The eXtensible Messaging and Presence Protocol (XMPP) is one of the protocols that enable the exchange of structured data between two or more devices. However, it lacks a QoS mechanism, which makes it not suitable for real-time applications [Co15]. Constrained Application Protocol (CoAP) is a one-to-one communication protocol, aimed for IoT devices to communicate over the Internet. Although it is designed to work on microcontrollers with a meagre memory size, it does not guarantee a real-time execution by its original implementation. Nevertheless, by using a distributed time server, it is possible to extend the protocol to enable real-time behaviour [Ko16]. Message Queue Telemetry Transport (MQTT) is a lightweight machine to machine (M2M) messaging protocol, but similar to CoAP; it does not guarantee real-time execution due to overheads.

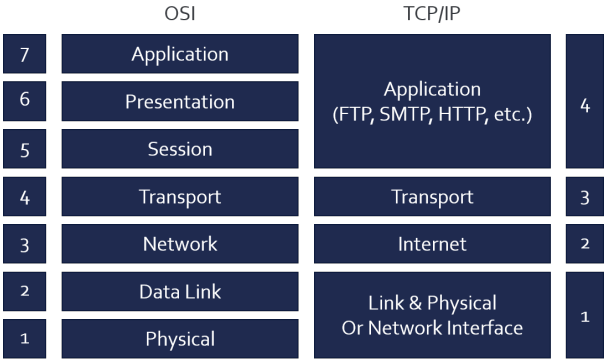

Figure 2.2: Comparison of seven layers of the OSI model and four layers of the TCP/IP model [Br89].

Open Systems Interconnection (OSI) Model is a reference model that standardizes the communication functions independent from the underlying technologies. The model is standardized by the International Telecommunication Union (ITU) in 1983 as the standard ITU-T X.200 and by the International Organization for Standardization (ISO) in 1984 as standard ISO 7498 [IS94]. It is considered as a reference model for hardware and software vendors to create interoperable solutions by grouping a set of networking functions. The model is criticized for its inherent implementation complexity that renders networking operations as inefficient and slow [Ru13]. The OSI model is seldom used in practice, but it is considered as a reference point in discussions of other protocols.

Transmission Control Protocol/Internet Protocol (TCP/IP) suite was developed within the Department of Defense's (DoD) Advanced Research Projects Agency (ARPA) Internet Program and published in 1981, by Internet Engineering Task Force (IETF) as RFC-793 [Po81]. The effort aimed to build a nationwide packet data network. TCP is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks and interconnected systems of such networks [Po81]. TCP/IP is one of the most widely-used protocols for transmitting and receiving information over one or more networks. The comparison of layers of the OSI model between the TCP/IP model is depicted in Fig. 2.2. Application, Presentation, and Session layers of the ISO OSI Model are combined into one layer in the TCP/IP Application layer. Similarly, Data Link and Physical layers of the OSI model are also grouped as Link & Physical layer or Network Interface Layer in TCP/ICP.

TCP/IP provides end-to-end connectivity specifying how data should be formatted, addressed, transmitted, routed, and received at the other end. Its name is received from its third and second layers. It can be used through the Internet and within the private networks. It is one of the key enablers of Cloud Computing.

Although TCP/IP is mostly designed for the transmission over WAN, there have been several works [Ge09; PY98; ZT01] to reduce the maximum latency in local area networks (LAN). There are even open-source projects such as Xenomai[9] and RTNet[10] or commercial libraries such as CoralReactor[11] which can bring the maximum latency below 10 microseconds (μs).

TCP/IP enables a connection-oriented delivery for which the communication requires an acknowledgement from the receiver. Moreover, it uses segment numbering, and the segments are passed to the application in the same order. By introducing an error detection code, it can check whether the data is corrupted, and verify its receipt by sending an acknowledge message. It can also request retransmission of the data, in case they are lost during the transport due to network congestion or errors.

Intercommunication protocols between the Edge Servers are out of the scope of this thesis. However, the realization of the conceptual architecture in the framework is performed using TCP/IP communication protocols to get the benefits required for reliable communication.

Sockets

A *socket* is a bi-directional communication and data transfer mechanism. Sockets are used to transfer data with minimum overhead between two processes. On a UNIX system, these two processes can be running on the same system, by using Unix Domain Sockets (UDS) or TCP/IP loopback, or on different systems.

For socket programming, programs can be written either using User Datagram Protocol (UDP) or TCP. In TCP connection, for the communication to take place, a connection between the clients and server is necessary. However, the UDP connection is not connection-oriented, meaning there is not an active session between the client and the server. UDP is a lightweight protocol, faster than TCP, but it does not guarantee the delivery of the package. UDP is usually used for high throughput, whereas TCP is used for guaranteed and ordered delivery.

Edge Servers of the framework based on this thesis use TCP socket communication to prevent data loss. The communication is performed in the Transport layer, without additional overhead. However, other communication protocols can also be utilized to have even better performance. Real-time communication between the Edge Servers is out of the scope of this thesis. Communication between two Edge Servers and End Devices are further elaborated in Sec. 5.2.

[9] Website: `https://xenomai.org`
[10] Website: `http://rtnet.org`
[11] Website: `http://www.coralblocks.com/index.php/category/coralreactor/`

Web Services

A service is a set of related software functionalities that can be reused for different purposes, together with the policies that should control its usage (See: Definitions). Web Service is a service offered via the World Wide Web (WWW) using Web technologies such as HyperText Transfer Protocol (HTTP). According to W3C Consortium [HB04], it is a software system designed to support interoperable machine-to-machine interaction over a network. They provide their functionality via methods available with service description languages. They generally do not guarantee nor provide real-time specifications. This thesis offers services for End Devices, and they can be called by issuing commands. The services, then, call programs, software, or commands to complete the task requests. Services will be detailed in Sec. 4.2.

APIs

Application Programming Interfaces (APIs) are used to abstract the functionalities of lower-level operations.

APIs can be used for communication with the I/Os, end devices, or the Cloud. An API is necessary to abstract the functionalities of other components. It enables more straightforward lower-level modifications in a system if a new software component is added, without requiring a complete change in the system. It also guarantees that the requests cannot interfere with the internal components since direct access to the individual modules or components is prohibited. One goal of the proposed architecture is to keep the migration efforts at the minimum and improve backward compatibility. Therefore, it is essential to have a reliable and standard API to make it compatible with as many devices and software as possible. Having a lightweight yet stable API helps achieve low latency requirements. Last but not least, the API should make sure that the requests are always authorized.

The proposed architecture endorses the utilization of APIs for easier integration and interoperability. The recommended methods endowed with the API will be listed in Sec. 5.3.

Virtualization Technologies

The history of virtualization technologies (VT) goes back to the mid-1960s, where IBM M44/44X experiment introduced the virtual machine term [Ho08]. Later, in January 1966, the first OS that can run multiple OSes on IBM System/360 Model 67, IBM CP-40, was released [Co82; IB71]. Since then, due to the limited resources and intensive usage of these within control systems, advancements and acceptance of the VT were slow [MSV14].

Virtual Machine Monitor (VMM), or hypervisor, is a software, firmware, or hardware that creates and runs the Virtual Machines (VMs). Popek and Goldberg [PG74] grouped VMMs in two

types:

Type-1 hypervisors run directly on the host's hardware.

Type-2 hypervisors depend on an OS to run, similar to other programs.

The advancements in the multi-core technology improved the VMM technologies as well. Central Processing Unit (CPU) VT (e.g., Intel-VT and AMD-V) and network VT (e.g., VMware NSX and openvSwitch) boosted virtualization performances, tremendously [AA06].

In the telecommunication domain, there has been a rise in the interest in employing virtualization technologies because of the cost and power consumption reduction aspects of the technology [Pa09]. Patnaik et al. [Pa09] analysed the performance implications of hosting IP telephony infrastructure in virtualized environments. They use Xen[12] virtualization technology for the experiments. Then, they discuss the challenges after deployment of the infrastructure in multi-core systems. Menon et al. [Me05] present a debugging tool to evaluate the performance of Xen on uni- and multi-processor systems to increase its performance. Mahmud et al. [MSV14] evaluated the industrial applicability of virtualization on a distributed multi-processor platform. Mahmud et al. use various open-source solutions and check the feasibility of application in the industrial control systems.

ESXi, Workstation, and Server products of VMware[13], and VM VirtualBox[14] from Oracle can be given examples to the Type-2 hypervisors. In addition to hypervisors, in recent years, a container-based virtualization software, Docker became known. Unlike hypervisors, container-based virtualization tools use the same hardware and OS to isolate processes from each other at fewer resource costs. They provide applications with separate run spaces but share the same hardware resources [Do21]. Another recent technology that uses containers is Kubernetes [Li20b]. Containers are useful to distribute and run applications on several platforms. However, in a production environment, when a container fails, a backup solution must be started to avoid downtime. Kubernetes overcomes these issues and can run distributed systems reliably. It provides services discovery and load balancing features. LinuX Containers (LXC) are another OS-level virtualization technology that creates isolated environments on a single host. It uses control groups (cgroups) functionalities to create containers and to execute the applications [Ca20a].

The architecture in this thesis does not use multiple OSes on an Edge Server for virtualization nor container-based virtualization. Instead, it uses Virtual Processors (VPs) to isolate tasks and manipulate their resource usage, such as CPU utilization. For example, if the server consists of multiprocessors, the tasks can be assigned to different CPUs. VPs are similar to cgroups functionality on Linux systems. More on cgroups are explained in Sec. 2.2.2 and VPs in Sec. 5.1.7.

[12] Website: `https://xenproject.org`
[13] Website: `https://vmware.com`
[14] Website: `https://virtualbox.org`

Grid Computing

Grid computing is a computer network in which each computer's resource is shared with other computers in the same network. The metaphor as *utility computing* was first mentioned in 1961 by John McCarthy [Ga99]. McCarthy foresaw computing as a public utility, similar to a phone system.

In grid computing, resources are made available in a resource pool in which all participating computers can access. Grid computing can also be called distributed computing. Unlike High-Performance Computing (HPC), grid computing does not only contain locally connected processors. It also does not guarantee low latency. Ideal grid computing shares all resources to speed up the computation. Although any computer can be upgraded in terms of CPU, memory, or storage to achieve a performance increase, grid computer scales it even better, allowing this upgrade to be used by other participants as well.

Today, many systems that utilize grid computing rely on proprietary software and tools. This limitation makes different systems using different protocols hard or even impossible to collaborate. Coordinating tasks within the grid is a cumbersome task. Grid computing usually requires a central server, which is also called a control node. This node is responsible for administrative tasks. Moreover, it also requires a grid computing software to be installed on all participating computers. The participating computers can run the same or different operating systems. One of the concerns of grid computing is data privacy and security.

Microsoft, IBM, The Organization for the Advancement of Structured Information Standards (OASIS), and The Globus Alliance created an open forum for grid computing, called The Open Grid Forum (OGF)[15]. The OGF created a set of standards called Open Grid Services Architecture (OGSA).

There are several projects taking advantage of unused computer processing power. SETI institute analyses the gathered radio communication data to search and explain the origins of life [SE19]. Folding@Home project by the Pande Group in Stanford University's chemistry department studies proteins combining idle resources of thousands of personal computers [Fo19]. Additionally, BOINC[16] and Einstein@Home[17] can also be given as examples to grid computing, which are still active. Grid computing is an enabler for Cloud Computing, Edge Computing, as well as the architecture for this research. The aim of grid computing is, however, to achieve higher performance, rather than providing low latency. Therefore, it cannot be used for real-time computing as in its current state. Nevertheless, the grid computing idea is considered in the proposed architecture. The unused computing power of available Edge Servers is broadcast in the Edge Network and taken into consideration in decision mechanisms for offloading.

[15]Website: `http://ogf.org`
[16]Website: `https://boinc.berkeley.edu/`
[17]Website: `https://einsteinathome.org/`

Load Balancing

Efficient usage of parallel computing systems requires that the tasks are optimally partitioned over these systems [ISB86]. The distribution of these tasks is called *load balancing*.

Load balancing usually works with computers that have the same functionality. It optimizes the workloads of the participating computers in a network, by distributing tasks across multiple computers, efficiently. It also provides flexibility to add to or remove servers from the network. Round-robin, client-side random load balancing, server-side load balancing, and Domain Name Service (DNS) delegation are some of the load balancing methods.

Round-robin does not need any special software or hardware. It merely assigns the client one of the available IP addresses in round-robin fashion. In client-side random load balancing, each client receives a list of available IP addresses and the client randomly chooses one. In DNS delegation, one address is pointed to multiple IP addresses. Depending on the criteria (e.g., proximity of the computer), one of them is chosen and sent to the client. However, if the chosen server is down, the DNS will not respond, failing to continue servicing.

The load balancer itself can also be dedicated hardware to perform its task. In this case, the hardware usually has software in it, with access to the back-end servers. These load balancers receive a response from the back-ends and deliver it to the clients. The clients do not notice the difference since their only contact is with the load balancer. This kind of load balancers has several methods to distribute the load across all back-ends. Round-robin, random choice, or least connections are some of them [Ne17]. Additionally, they may have more sophisticated methods that make decisions based on the load, uptime, distance, and so on. Once a request is diverted to a server, the load balancer keeps track of the session, sending the responses back to the originating client. The session information is stored in a file or storage until it expires.

Although this hardware solution has many advantages such as control of the data, attack prevention, health checking, and firewall, when the load balancer is down, the session information and the communication between clients and the back-ends are lost. This thesis uses the load balancing idea and gives each participating Edge Server the ability to balance the load, in a decentralized manner. Each server becomes aware of the resources of its neighbouring Edge Servers, and they can make a decision based on these resources, including the distance to the servers and the end user. Since there is no limitation on the topology, if a server is down, another server can receive the request and complete it. The response is also sent back to the originating caller after the request is handled in the optimal location. This approach will be explained in Sec. 4.3.

2.2 Real-Time Computing

A real-time system is a system that has to respond to a request within a finite and predictable time. The system behaviour depends on the logical results of the computations and real-world time when these results are produced [We04].

Real-time systems are often part of larger (embedded) systems and have to model parallelism (concurrency) that exists in the real-world objects they are controlling or monitoring. They have significant roles in the process control, manufacturing support, command and control areas. They are also often required in the automotive, aerospace industry or production domains. Moreover, they interact tightly with physical environments such as sensors and actuators. Since the timing is critical, the resource efficiency must be at maximum, and the importance of safety is even higher than other approaches.

The real-time term has several misconceptions, including its usage. In computer time, the speed of the tasks depends on the computer itself. Unlike computer time, in real-time, the environment and physical objects control the speed. Therefore, the system must adapt itself to its environment since the speed of the objects or physics cannot be changed. Simply, using the term ”fast” does not reflect real-timeliness. Speed is a relative term which is environment-dependent. If the environment changes, the correct reaction may require an even ”faster” response. In real-time systems, average values are also not used. Even though one peak value may not affect the average value too much, it could be fatal to the system. This peak value must be considered during design and implementation. Finally, performance testing is even not enough. The system must be proven and formally verified to guarantee real-time behaviour.

Figure 2.3: The usefulness of the task results after deadline miss for soft, firm, and hard real-time systems.

Depending on the criticality of the failure, real-time systems can be divided into three categories. They are explained below, and an example graph to reflect the usefulness of their results when they miss their deadlines is given in Fig. 2.3.

1. *Soft Real-Time*: This category groups the real-time applications which are less critical and have wider deadline intervals for acceptance. For example, interruptions in voice calls or video streams are tolerated in case some data packages are lost. Even though a higher loss rate degrades the performance, it is not irreversible as in a firm real-time.
2. *Firm Real-Time*: A real-time category between hard and soft real-time. It tolerates some deadline misses, but an increase in the misses degrades the service, in the end causing unacceptable results [KLB06]. Missorting the colours of the parts can be given as an example [GMS15].
3. *Hard Real-Time*: Failure in the system is mostly fatal. For example, if an airbag in a car deflates before or after the specified timeframe (between 100 ms and 300 ms, within 10 ms), it loses its protective impact [OD08]. The results after deadline misses are fatal.

OSes can be for general purposes or real-time. General-purpose operating systems (GPOS) are focused on high throughput. They tend to execute numerous tasks, rather than executing one high priority task. They usually contain non-preemptible kernels, i.e. calls from kernel often override processes and threads. GPOS usually provide adequate performance for general use.

However, real-time operating systems (RTOS) are focused on priorities. They preempt a low priority task if a higher priority task comes, provided that the tasks are also preemptible. Their kernel is also preemptible, meaning kernel processes and threads are also considered as external user processes. RTOS always perform the predicted behaviour. Solutions such as VXWorks[18], QNX[19], and Windows Embedded Compact[20] (former Windows CE) can be given as examples to RTOS. Additionally, RTOS such as FreeRTOS[21] or RTOS-UH[22] can also be directly programmed in microprocessors. RTOS aim to be precise at timing, rather than yielding a high throughput.

Linux is one of the open-source kernels for operating systems [Li19a]. Vanilla Linux is a GPOS by its design. However, there have been several initiatives to make the kernel real-time capable. One of the well-known initiatives to convert vanilla kernel into an RTOS is RT-Linux [Gl17]. There are also other development activities that focus on improving real-time support on Linux. Xenomai[23] supports the Linux kernel with a co-kernel running together with the Linux. This co-kernel deals with all time-critical activities such as scheduling and interrupt handling. It is the first extension that supports Real-Time Drive Model (RTDM) [Ki05]. Real-Time Application Interface (RTAI)[24] is another open-source project, and it brings real-time capabilities into the Linux kernel by extending it. It introduces a hardware abstraction layer (HAL) and is used to acquire data or

[18] Website: `https://windriver.com/products/vxworks/`
[19] Website: `https://blackberry.qnx.com/en`
[20] Website: `https://docs.microsoft.com/en-us/previous-versions/windows/embedded/gg154201(v=winembedded.80)`
[21] Website: `https://freertos.org/`
[22] Website: `http://www.rtos.iep.de/indexe.htm`
[23] Website: `https://xenomai.org`
[24] Website: `https://www.rtai.org/`

control hardware supported by Comedi [Co19]. RTAI allows a set of specific hardware to work in real-time.

Many real-time systems play a significant role in critical systems, such as chemical plants, aerospace industry, and many more. The criticality of the roles in these areas defines the requirements of the systems. A real-time system will often communicate with the real-world. For example, a program that requires high interaction with the hardware and environment needs to continue operation reliably, even under a heavy load. It must attempt to tolerate faults and make the environment safe before shutting the system down. Due to the time-critical nature of the devices, direct access to the device instead of through a layer of an OS function can give more control on the device behaviour.

Two of the essential characteristics of a real-time system are the reliability and response time. It is quite challenging to design and implement a system that guarantees the expected output at the expected time in all conditions. To achieve this problem, real-time systems contain a ”spare” capacity which ensures that ”worst-case behaviour” does not delay the critical operations. Supplying sufficient power and run-time support, the following parameters must be provided [We04]:

- times that the action must be performed and completed.
- respond to situations when timing requirements cannot be met, and when the timing requirements are changed.

These parameters ensure that system behaviour is predictable.

The architecture proposed in this dissertation is OS- and programming language-neutral. However, the realization of the framework based on this architecture is performed employing the Java programming language. The framework is then tested on Linux-based RTOS to meet the timing requirements. The communication between Edge Servers uses event-based synchronization. During the design and testing, the hardware is considered to be ideal and real-time capable.

The sections of this chapter will explain the challenges and characteristics of the real-time systems. They will also focus on scheduling algorithms for real-time systems, which is one of the challenges.

2.2.1 Challenges in Real-Time Systems

Criticality, in general, brings its challenges, almost in any domain. The number of challenges in real-time systems increases as the criticality of a task raises. This section will explain the challenges in real-time systems and how they can be overcome.

If a real-time system is going to be designed with the assistance of an OS and if the hardware is not a custom-design, then real-time proven hardware must be chosen. The OS must also be real-time capable of supporting the hardware. The hardware selection for the real-time environment is a critical part, as not all hardware is suitable for real-time operations. Interrupt handling, caches, memory allocation/access, Central Processing Unit (CPU) power management, hardware timers, and latencies vary from hardware to hardware. Besides, the number of CPUs must also be considered as having multiple cores may cause additional problems such as cache incoherency. Hence, the chosen hardware impacts the way that the system operates.

If the solution is based on distributed or decentralized systems, then synchronization between participants may also be necessary. Synchronization can be either event-based and time-based. Typically, when a task is split across participants, if the tasks are supposed to merge, both should arrive at the expected moment to further continue or give the correct results. For instance, in microcontrollers, multiplication always requires more cycles than an addition. To prevent wrong results, before continuing the computation, the addition result must wait for the calculation of the multiplication. This waiting continues until the multiplication event is completed. Contrarily, if the participants are working individually and performing the same task, then they should be aware of the statuses of each other. For example, a power plant with multiple generators, which supply power to the grid, require that their Alternating Current (AC) phases are aligned. To achieve that, they need a strict time synchronization between each other. Not doing so would decrease efficiency.

Poorly written programs can result in endless loops affecting the whole system; making it unresponsive. Additionally, expecting a real-time performance from devices that are not formally verified may cause problems depending on the criticality of the task. If the hardware to be used for real-time is not formally verified, even random or long tests giving 100% success rates do not guarantee a lifetime real-time response. If the hardware is real-time dedicated, or formally verified for real-time, then, the software within or the installed OS is also required to support real-time. For example, a Linux-based OS with RT-patch installed can preempt running tasks if a higher priority task arrives. It comprises only a few kernel calls that are not preemptible. The rest can be preempted, even the kernel itself [Li18b]. All other calls are treated as external processes.

Time-driven communication in distributed control systems requires high accuracy of clock synchronization. Due to the nature of communication, the message delay is inevitable. The deviation of time in distributed systems causes synchronization error. There are several algorithms found to overcome these problems [Fl89; KO87; Ma04; WL88]. In distributed systems, using a global time to synchronize participants is sub-optimal. Due to the propagation delays, it takes a finite amount of time to deliver a message from one end to another. The farther the distance is, the later the message is delivered. A protocol that can calculate or estimate the variance could partially solve this problem. Network Time Protocol (NTP) is one of the protocols us-

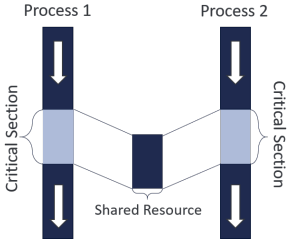

Figure 2.4: A critical section is the location where access to a shared resource during execution occurs.

ing this approach. IEEE 1588 Precision Time Protocol (PTP) is another protocol used for clock synchronization [IE08]. Both of these protocols have, however, error bounds. Corbet et al. [Co12] achieved minimizing the time offset across distributed database servers by using Global Positioning System (GPS) receivers and atomic clocks, which is called TrueTime. Tavakoli et al. [Ta15] introduced a quantum algorithm that can achieve clock synchronization using only a single quantum system, without propagation delay.

Two of the encountered problems in real-time systems are race condition and deadlock, which may also exist in non-real-time software development. However, in real-time systems, they are more critical to be solved. Understanding these problems requires explanation of what a critical section is. As can be seen in Fig. 2.4, a critical section is where a shared resource or variable is accessed. A scheduling algorithm can swap between threads at any time. Since it is not known which thread attempts to access the shared resource at any moment, the value read for that thread may be different than desired, or even not accessible. Therefore, the threads *race* to access or change the value/resource. This situation is called a *race condition*.

Ideally, a process works with shared resources using three steps: (1) it requests, (2) uses, and then (3) releases it. During its use, the process also blocks access to the same resource until the process completes using it. If this resource can only be accessed by one process, other processes cannot use it when it is in use. As an example, two processes, each using a shared resource, can be considered. Assuming that these tasks require each other's shared resources in the next cycle, if these shared resources are never released, a *deadlock* can arise. Deadlock is a situation where a set of processes wait for each other due to one or more of them holding (or using) a shared resource.

There are several ways to overcome each of the problems mentioned earlier. One simple solution is called *busy waiting*. This is accomplished by creating an empty loop, cycling until a condition holds, such as until the resource is available. *Mutual exclusion (mutex)* is another solution for multi-threaded software. They protect the critical regions and thus prevent race conditions of the threads that belong to a process. A further solution is introducing *semaphores* inside the software. Semaphores were designed by E. W. Dijkstra in 1965 [Ma08]. On the OS-level,

they are set by two atomic functions that limit the usage of shared resources. Under Linux, these functions are namely `wait()` and `signal()`. These atomic functions are provided by the OS and cannot be interrupted or modified at the same time. They are used for signalling between threads and processes. Other kernels may have different names that are used to represent these functions. Unlike mutexes, semaphore values can be changed by any process that acquires or releases the resource. However, mutexes can only be released by the process or its threads that lock the mutex. Semaphores can also be used for *conditional synchronization*. For example, if a part of a function in one process needs to be executed until the second individual process is executed, two atomic functions simply make this possible. An additional solution to solve deadlock problems includes adding a timeout to the process or thread that is using the shared resource, or *preemption*.

Preemption requires that the scheduling algorithm, the kernel, and the process to be preemptible. Kernel preemption under Linux can be enabled by compiling the kernel with support for kernel preemption. This allows not only user applications to be preempted, but also the kernel if a higher-priority process comes into play. There are additional concepts to care about while dealing with real-time systems. Memory management with profiling and debugging are also two of them.

Depending on the functions they provide, Input/Output (I/O) devices may be significantly slower than the processors or internal hardware components. While a device is busy with performing a task, the CPU may wait for many cycles before the device becomes ready. The status of the I/O device must be monitored to detect whether it has finished the task and ready for another transaction. The CPU monitors the status of the device repeatedly. However, it does not perform any other tasks during this process. This monitoring process is also called busy-waiting or polling.

The interrupts can also implement preemption. While a slow transaction is being performed on an I/O device, the CPU can work on other tasks. When the transaction of the I/O device is completed, it signals the CPU using interrupts to notify its availability. Whenever an interrupt is signalled, the running program on the CPU is suspended, and the interrupt handler is called. After the operation is completed, the CPU switches back to the suspended program.

In a typical system, there will be more than one I/O device. The interrupt handler is expected to distinguish the source of the interrupts and treat them according to their urgency. There are two interrupt mechanisms for multiple I/O devices: *interrupt priorities* that specify which interrupt is more critical than others and *interrupt vectors* that tell the CPU which service routine must be called to handle the request. Interrupt mechanism brings some overheads to the CPU time, e.g. context switching, hardware-level program counter jump, interrupt acknowledgement.

Typical memory operations are performed by the CPU. When a buffered I/O is used, repeatedly performing this operation brings a significant overhead. Since CPU cannot perform another

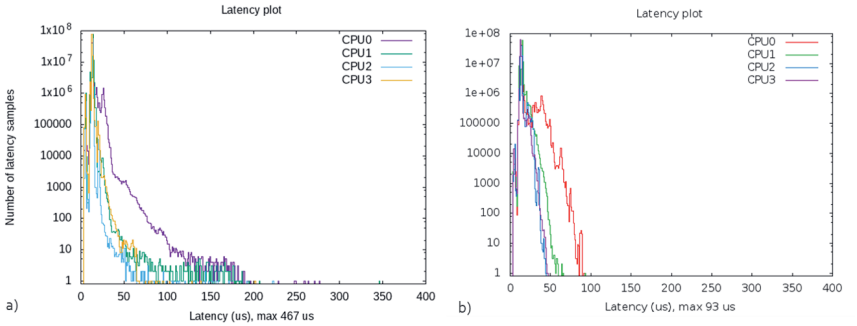

Figure 2.5: Results of latency tests ran for 8 hours on (a) stock kernel and (b) real-time patched kernel under Raspberry Pi 3B+.

operation during the data transfer, its efficiency is reduced. This problem can be solved using Direct Memory Access (DMA). When a memory operation is to be performed, the CPU initiates the DMA transfer via the DMA controller. During the transfer process, the CPU can perform other tasks. When the transfer is completed, the controller signals the CPU with an interrupt.

Real-time systems are usually built using dedicated custom hardware. However, custom hardware is expensive and prone to errors. They are usually designed for custom tasks, and they have reduced reusability. Besides, upgrades are also hard to perform and expensive. Another way of building a real-time system is by using a mix of hardware and an RTOS. With this approach, the development and maintenance times, therefore costs, are also reduced. The approach also reduces the burden on upgrades. Wolter and Albert [AWG03; WAG03] analyse some of the existing real-time systems and buses by introducing a method based on Walsh correlation. Custom hardware for dedicated tasks is formally verified to ensure the functionality in all situations. However, it is quite hard to verify sophisticated hardware designed for multi-purposes formally. In these circumstances, long tests are performed to increase the detection of problems. One of the test sets that can be performed under Linux is called Worst-Case Latency Test Scenarios [Li18a]. They generate workloads and heavy loads to test a system with real-time patch enabled. Fig. 2.5 shows the latencies of (a) stock and (b) real-time patched Linux 4.14 kernel performed under Raspberry Pi 3B+[25]. Eight hours of running the latency tests showed a maximum latency of 467 μs on the stock kernel and 93 μs on the real-time patched kernel. The maximum latencies were seen only in one sample of 100 million samples, but these values are taken as a basis while calculating latencies for the worst-case scenario. Repeating tests multiple times yielded the same results. However, as mentioned before, the tests do not guarantee a life-time real-time response. This should be considered while using hardware on hard real-time applications.

[25]Website: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/`

During the validation of the architecture, hence the framework, the tasks are implemented in such a way that they avoid the problems explained before. Moreover, while designing the architecture, it is assumed that hardware to be ideal. To achieve a hassle-free real-time execution, software developers should address these issues during development. The next section will explain how scheduling for real-time processing can be planned to avoid deadline misses.

2.2.2 Scheduling for Real-Time Processing

A modern computer allows performing several tasks at the same time. Modern OSes support multitasking. For instance, a piece of music can play in the background while surfing on the Web. Windows started supporting multitasking since its 386 version in May 1988 [In18; Mi14]. Linux kernel added support for multitasking from its first release version 0.1, in September 1991 [To91]. Multitasking, however, does not necessarily mean parallelism. A single CPU can execute one instruction per clock cycle. In a single core (or CPU) system, having multiple task running means that the tasks are switched in a way that the user thinks that they work in parallel. Unless the system comprises more than one CPU, true parallelism is not possible.

In practice, hardware resources are not unlimited. Running processes compete with each other to get more resources. Misbehaving processes may affect the whole system, causing other processes to fail as well. To prevent this, the processes must be given enough resources to execute, and they must be isolated from each other to avoid interference [Ma08].

The kernel of an OS is responsible for giving CPU time to a thread, for execution. Switching, or giving thread their turns are performed by the *scheduler* component in the kernel. The process of changing turns is called *scheduling*. By considering several properties of a thread, the scheduler chooses the next task and maintains the order of other tasks. Under Linux, the processes can be grouped to be affected by the same scheduler policy. This is performed via control groups (cgroups) which will be explained in Sec. 2.2.2.

All Portable Operating System Interface (POSIX)-oriented OSes implement priority-based scheduling [IE18b]. Each task with higher priority runs earlier than lower ones. Voluntarily giving up the CPU resources for other processes is called *cooperative multitasking*. Involuntarily suspending a running process or thread after a specific time is called *preemption*. The scheduler policy determines preemption and CPU time allocation [Ma08].

According to their arrival pattern, tasks, in general, can be classified into three categories [Au91; SSL89]. *Periodic* tasks are the tasks that arrive at a constant rate. The time between activations is called a period. They have an infinite sequence of identical activities. *Aperiodic* tasks are usually event-driven. They can run once or infinite times, but their inter-arrival time is not bound to any value. During design time, their arrival times are not known a priori. *Sporadic* tasks are aperiodic tasks, but the inter-arrival times are bounded by a minimal inter-arrival time.

According to Garey and Johnson [GJ79], general scheduling problems are NP-hard. In practice, it means that, if the number of tasks rises, the time for finding a feasible schedule grows exponentially. In this thesis, it is assumed that the computations for finding a scheduling graph have no or negligible overheads, regardless of the task count. For real-time applications, the scheduling is more critical, especially for hard real-time tasks. GPOS do not provide real-time support. Linux vanilla kernel is also not shipped with real-time support. Therefore, OSes using Linux vanilla kernel are also GPOS and need to be patched to support real-time computation.

The scheduler is activated whenever a timer interrupt is activated. In POSIX systems, this value is set to 1 microsecond. This value is tied to the high-resolution hardware timer value. Hardware without a high-resolution timer cannot guarantee this timing [IE18b]. Thus, hardware support is also necessary to enhance the schedulability.

The schedulers defined in the following sections require several properties of the processes known in advance. One of them is the worst-case execution time (WCET). Determining this value is quite intricate and requires in-depth analysis. Structural analysis is one of the analyses that can be performed on the source code, object code, or the assembly code. Each of these has both advantages and disadvantages. Source code analysis requires access to the source code of the process. It is the purest form and easier to understand compared to the other two structures. However, the analysis may not give concrete results, as the program depends on the preprocessors, linkers, macros, and the compiler. Object code analysis is done on the code after compiling optimizations are done. However, it is harder to analyse as much information is lost. Assembly code has the same issues listed for object code. This makes the analysis even harder due to assembly being lower-level. The WCET analysis also requires a deep understanding of the hardware that the software is running on. The interaction between the software and hardware and the hardware properties of the processors are two of them. In this thesis, it is assumed that the WCET values of the processes are known a priori.

There are many algorithms to find an optimal schedule. Rate Monotonic (RM), Deadline Monotonic (DM), Least Slack Time (LST), and Earliest Deadline First (EDF) are among the known scheduling algorithms used in real-time computing [KK12; SB17]. There exist also other algorithms such as Earliest Deadline Critical Laxity (EDCL) [KY08], Latest Release Time (LRT), and Earliest Due Date First (EDD) [Ja55] that yield optimal scheduling diagrams. The algorithms are initially applied for uniprocessor systems. With the increased usage of multi-core systems, some of these algorithms were optimized to be applied to multi-core systems or new ones became available. Rate Monotonic Next Fit Scheduling (RMNFS) and Rate Monotonic First Fit Scheduling (RMFFS) by [DL78] and Least Slack Time Rate First (LSTRF) by [HCK11] can be given as examples to the scheduling algorithms for multi-core environments. Upcoming sections will explain two of the widely-used scheduling algorithms in real-time computing as well as in this thesis: LST and EDF.

The schedulers can be classified according to several categories. A scheduler may be placed in

more than one group. They are grouped by:

The time that the scheduling decision is made: *Offline schedulers* organize the scheduling table before the system is activated and store the results in a table. At run-time, the execution is performed by referring to this table. They are suitable for deterministic systems. As they have full knowledge about when the tasks are going to be executed, the optimizations can be performed in advance. The overhead during runtime is quite low and requires a few instructions to implement. However, it does not apply to event-based systems or dynamically changing systems. *Online schedulers* decide which task to execute at runtime. It also increases flexibility during design time. Compared to offline schedulers, they add additional overhead during runtime, depending on the complexity of the algorithm. They are useful for event-based systems.

Decision mode: *Preemptive schedulers* can preempt tasks at any time for several parameters. They reduce the response time for higher priority tasks and enable higher CPU utilization. However, they may switch task context more than necessary, causing preemption delay. This con can be expensive with modern processors due to the total number of cores. *Non-preemptive schedulers* select a task to execute and execute it until the task finishes execution.

Priority assignment: *Fixed task priority schedulers* assign a fixed priority to all jobs of a task, and this priority never changes at run-time (Fixed Task Priority Scheduling, FPS). Although they are simpler to implement, from the analysis point of view, they are suboptimal. *Dynamic task priority schedulers* can assign different, but fixed-job priorities for all jobs of a task, or each priority may change at run-time, even during its execution (Dynamic Job Priority Scheduling, DPS). These schedulers are harder to implement but provide optimal scheduling.

Conservatism: This kind of schedulers ensure that a processor never runs idle when there are existing jobs ready to run.

Optimality: *Optimal* schedulers find the minimal cost function based on the scheduling criteria. *Heuristic* schedulers satisfy the criteria; however, they do not provide the best schedule.

The next sections will detail the challenges in scheduling and explain two of the significant scheduling algorithms. It will also explain why there is a need for another novel scheduling algorithm in the proposed architecture. This scheduling algorithm is going to be detailed in Sec. 3.1.

Challenges in Scheduling

Scheduling to plan the execution order of real-time tasks is challenging. Due to stochastic execution times of the tasks, schedule plans cannot use average values. Moreover, the WCET of tasks can be significantly longer than their average execution times. Even in a high load of CPU, the scheduling algorithm is expected to be stable, to prevent deadline misses for real-time

tasks. However, the algorithms usually have overloads which may cause deadline misses. To avoid that, real-time systems often introduce a priority task dispatcher. Two of the conventional approaches to solving the stability problem are assigning priorities according to tasks' importances or creating a set of time-division multiplexing (TDM) slots and categorizing the tasks into these slots [SLR87].

Typically, when task priorities are assigned for real-time operation, the load is tested using the WCET of tasks. If a deadline is missed, or the CPU load is not optimal, the priority adjustments are made until the targeted utilization is achieved [SLR87]. Depending on the characteristics of a scheduling algorithm, there exist several scenarios where the priority model is violated. One of these scenarios is *priority inversion*. One of the publications on priority inversion was published in 1980 by Lampson and Redell [LR80]. It happens when lower priority tasks block a high priority task due to unreleased resources.

Consider two tasks with a high priority (*H*) and low priority (*L*) and a resource (*R*) which only one task can access it simultaneously. If *L* is using *R*, *H* can only access it once *L* releases. In a good design, this can be avoided by relinquishing *R* when *H* needs it and preempting *L*. However, if a third medium priority task (M), which does not require *R* runs before *H* is executed, it can preempt *L*, causing *H* to be unable to run until it completes its execution. One of the real-world examples of this issue was seen in 1997 by the Mars Pathfinder lander project [JP98]. Similar to the example above, low priority Atmospheric Structure Instrument/Meteorology (ASI/MET) task of Pathfinder project was preempted during its use of a shared resource, by independent medium priority tasks, and the higher priority task could not be executed. This was later solved by uploading a small program to activate the priority inversion.

It is possible that priority inversion does not cause a critical failure. The blocked process of high priority tasks can still miss their deadlines, even with an unnoticeable delay. However, resource starvation must be taken care of during the design time, by using pre-defined corrective measures such as resetting the (part of the) system using watchdogs. A similar issue can be seen in EDF scheduling, called deadline interchange, where deadlines of the tasks set the priorities.

There are many existing solutions to prevent or solve priority inversion. For instance, disallowing process preemption in critical sections prevents the issue. However, if the duration of the critical section is long, the high priority task may miss its deadline. If a maximum duration (timeout) to access and stay in the critical section is defined, the task scheduling can be calculated considering this duration as well. This solution works both with fixed and dynamic priorities.

Another solution is using a *priority inheritance* method. This method elevates the assigned priority of a lower task to the highest one that waits the shared resource until the task goes out of the critical section. Using the same example above, with priority inheritance, *L* will get the same priority as *H* until it releases *R*. Later, it will awaken *H* and return to its original priority. This solution is also applicable to systems both with fixed and dynamic priorities. In EDF, it can

be named as *deadline inheritance*.

Priority ceiling protocol is another method to avoid priority inversion, and also the deadlock problem. This method assigns a priority to each shared resource, which is equal to the highest priority of a task that may lock the resource. In certain circumstances, it may require raising the priorities of tasks temporarily. Therefore, it needs a scheduler that supports dynamic priority scheduling [Re14]. Priority ceiling can, however, deny the execution of tasks even when the resource is available.

Random boosting is another method used by the scheduler in Microsoft Windows to avoid deadlocks due to priority inversion. Priorities of the threads holding shared resource locks are randomly increased to allow them to exit the critical section [CW97].

Usually, I/O devices are slower than internal calculations. This creates another problem of shared I/O usage of tasks. Most of the time, First In First Out (FIFO) approach is used to schedule data concerning I/Os. Sha et al. [SLR87] test several other approaches to increase the utilization concerning the I/O usage. Moreover, they also introduce a refined version of the EDF algorithm, calling it *propagated deadline* scheduling algorithm. Assuming that all tasks are periodic, their deadlines are equal to their periods, and the data related to the tasks are brought by DMA and sent back to DMA after calculation, their novel algorithm gave the best results reaching up to 90% utilization.

The scheduling algorithms mentioned in this chapter are theoretical. Each algorithm is highly dependent on the time; they need a hardware clock with a precise resolution that generates interrupts at a constant frequency. The interrupts can be used at the start of each task's period, or for preemption. Additionally, context switching, interrupt handling, algorithm overhead, and jitter are threats for ideal scheduling. When the bus between the processor and memory is shared with the hardware devices which use DMA, the calculation of worst-case execution times becomes difficult.

Choice of whether a fixed or dynamic priority preemptive scheduling depends on the application use. It is usually true that the implementation of the former is more straightforward than the latter. However, theoretical maximum utilization that can be achieved with EDF is 100% whereas it is 69% in RM scheduling. This thesis uses online, dynamic priority preemptive schedulers for scheduling. It also assumes that the software and programs used are free of issues such as *priority inversion*. Moreover, scheduling overheads and jitters are also assumed to be negligible.

The following sections will explain two of the scheduling algorithms mentioned above that are taken into consideration in this thesis.

Least Slack Time (LST) Scheduling

Slack time means the remaining spare time of a task at a time t. In the Least Slack Time (LST) or Least Laxity First (LLF) scheduling, the shorter slack time receives higher priority and executed. A task is run until its slack time reaches zero (0) to avoid context switching. If a task T_i has an arrival time of a_i, the worst-case execution time of x_i, relative deadline of d_i and the remaining time of m_i, the remaining execution time is then $m_i = x_i - (t - a_i)$. The slack time then becomes $t_{slack,i} = d_i - t - m_i$. LST can run online; however, it can only work with preemptible tasks, and the implementation is not easy.

An example of LST is given in Table 2.2.

Table 2.2: An example list of tasks for the Least Slack Time (LST) Scheduling.

Task	a	x	d
T_1	0	4	20
T_2	8	3	15
T_3	9	4	15

Task T_1 arrives at $t = 0$ and ends at $t = 4$. Since there are no task arrivals, it is not necessary to compute the slack time. Then at time 8, T_2 is executed until time 9. At $t = 9$, both T_2 and T_3 slack times are calculated to decide the next task.

At $t = 9$

For T_2:

$$t_{slack,2} = 15 - 9 - (3 - (9 - 8)) \tag{2.1}$$
$$t_{slack,2} = 4$$

For T_3:

$$t_{slack,3} = 15 - 9 - (4 - (9 - 9)) \tag{2.2}$$
$$t_{slack,3} = 2 \tag{2.3}$$

Since Eq. 2.2 is less than Eq. 2.1, T_3 is executed. At $t = 13$, the waiting task is rechecked for its slack time.

At $t = 13$

For T_2:

$$t_{slack,2} = 15 - 13 - (2) \tag{2.4}$$

$$t_{slack,2} = 0 \tag{2.5}$$

As Eq. 2.4 shows, the slack time is zero, meaning the task has no spare time. Therefore, T_2 is executed.

Although this algorithm provides optimal scheduling diagram for aperiodic tasks, as mentioned at the beginning of the section, it cannot work with non-resumable tasks. Another scheduling algorithm is needed for scheduling both preemptible and non-resumable aperiodic tasks together. Therefore, the thesis introduces a novel online scheduling algorithm, called Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling, to overcome this problem. This algorithm will be explained in Sec. 3.1.

Earliest Deadline First (EDF) Scheduling

Liu and Layland introduced Rate Monotonic (RM) [LL73] algorithm in 1973. The RM algorithm assigns the priorities based on the activation frequencies of the tasks. A task with higher frequency than existing ones are given a higher priority. Therefore, a task received a priority inversely proportional to its period.

RM algorithm is based on the task model, introduced by its inventors:

1. Tasks run periodic, and their deadlines are equal to their periods.
2. Tasks are released at the beginning of their period.
3. Tasks are independent of each other.
4. Tasks do not suspend/terminate themselves.
5. Tasks have known execution times.
6. The scheduling overhead is negligible.

Earliest Deadline First (EDF) Scheduling is a *dynamic priority* real-time scheduling algorithm. As its name suggests, the earliest deadline gets the highest priority. EDF is also based on Liu and Layland task model, which is defined for RM algorithm, thus, based on the same assumptions.

EDF is an optimal scheduling algorithm on preemptive uniprocessors. That means, if there ex-

ists any scheduling plan by any algorithm that ensures all the jobs in a collection of tasks are executed without missing their deadlines, the EDF can schedule them as well. However, if jobs are non-preemptible, then the EDF is not an optimal algorithm.

In Deadline Monotonic (DM) scheduling, priorities are static, meaning that they never change on the tasks when assigned. Considering two tasks — T_1, with a period of 4 and deadline of 4 and T_2, with a period of 10 and deadline of 10 — T_1 would always get the highest priority. However, in EDF, after T_1's third period, the absolute of the deadline for T_1 is 12 whereas it is 10 for T_2. This creates problems with the feasibility test of DM; thus, EDF requires a different feasibility test.

The basic idea of the EDF feasibility test is to check if the system has enough CPU time for each absolute deadline of a task set.

Let $T = \{(x, P) \mid x, P \in \mathbb{R}^+\}$ be a task where P is the period, and x is the worst-case execution time. Assuming tasks having their periods equal to their deadlines as required by the model, a necessary and sufficient schedulability test U can be calculated for the task set $S = \{T_i \mid i \in N^+\}$ by the Eq. 2.6, where P_i is the period, and x_i is the worst-case execution time of task i.

$$U = \sum_{i=1}^{n} \frac{x_i}{P_i} \leq 1 \tag{2.6}$$

The task set repeats itself after the current time reaches the *hyperperiod*. Hyperperiod H for the task set S can be calculated by finding the least common multiple (LCM) of periods of tasks, as given by Eq. 2.7.

$$H = LCM(P_j, ..., P_k), \text{ where } \forall P \in S \tag{2.7}$$

As an example, assume that there are three tasks, T_1, T_2, and T_3 in a task set S_1. Also, assume that each task executes 1 time unit in the worst-case and their periods are 2, 3, and 4 time units, respectively. Then, the S_1 can be written as in Eq. 2.8.

$$S_1 = \{(1, 2), (1, 3), (1, 4)\} \tag{2.8}$$

By using Eq. 2.6, the schedulability test U_1 yields:

$$\begin{aligned} \frac{x_1}{P_1} + \frac{x_2}{P_2} + \frac{x_3}{P_3} &\leq 1 \\ \frac{1}{2} + \frac{1}{3} + \frac{1}{4} &\leq 1 \\ \frac{11}{12} &\leq 1 \end{aligned} \tag{2.9}$$

Since the inequality 2.9 holds, the task is schedulable by the EDF.

The hyperperiod H_1 of S_1 is then:

$$H_1 = LCM(2, 3, 4) = 12 \tag{2.10}$$

Table 2.3: An example list of tasks for the feasibility test of the Earliest Deadline First (EDF) scheduling.

Task	x	P	d
T_1	1	3	2
T_2	2	4	3
T_3	2	2	11

The feasibility test shown as Eq. 2.6 works only if deadlines of the tasks are equal to their periods. However, the tasks may have their deadlines earlier than their periods. In this case, another feasibility test is used.

This test uses two known mathematical functions: ceil and floor. Ceil function $\lceil x \rceil$ gives the smallest integer that is greater than or equal to x. Floor function $\lfloor x \rfloor$ gives the largest integer less than or equal to x.

Let R be the current system time. The result of $\left\lceil \frac{R}{P_i} \right\rceil$ tells how many times that a task's period is started, whereas the result of $\left\lfloor \frac{R}{P_i} \right\rfloor$ shows how many complete periods have occurred. Then, the floor function $\left\lfloor \frac{R}{P_i} \right\rfloor$ gives the completed period count. If L is any time between zero and the hyperperiod, then, the number of repetitions (periods) of a task T_i that have deadline before and at L is equal to $\left\lfloor \frac{L-d_i}{P_i} \right\rfloor + 1$, where d_i is the relative deadline of task i.

Defining L to be the total execution time required by all n tasks with deadlines before or at L, the CPU demand E can be calculated as in Eq. 2.11.

$$E = \sum_{i=1}^{n} \left(\left\lfloor \frac{L - d_i}{P_i} \right\rfloor + 1 \right) x_i \tag{2.11}$$

Since the floor function Eq. 2.11 changes only when L is a multiple of d_i, the calculation can be computed only for the absolute deadlines of the task T_i. As the schedule repeats after the hyperperiod, the absolute deadlines are only considered up to the hyperperiod.

Let A be set of different absolute deadlines up to the hyperperiod H. Then,

$$A = \{d_j, ..., d_k\}, \text{ where } d_j \neq d_k \text{ and } \forall d \leq H \tag{2.12}$$

Provided that $\forall L \in A$, the feasibility test will be as seen in inequality 2.13:

$$L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - d_i}{P_i} \right\rfloor + 1 \right) x_i \tag{2.13}$$

Test performed using inequality 2.13 is necessary and sufficient for schedulability of task set using EDF. An example set of tasks to this test is given in Table 2.3.

The hyperperiod of the tasks is 12 ($LCM(3, 4, 2)$). Absolute deadlines for a task is calculated by multiplying the number of iteration of the period by period and adding the relative deadline. The iteration of the period is zero-based. For T_1, they are 2 ($0 \cdot 3 + 2$), 5 ($1 \cdot 3 + 2$), 8 ($2 \cdot 3 + 2$),

and 11 ($2 \cdot 3 + 2$). Similarly, for T_2, they are 3, 7, and 11. Lastly, for T_3, it is only 11. This yields $A = \{2, 3, 5, 7, 8, 11\}$. For each absolute deadline in A, we perform the feasibility test.

For $L = 2$,

$$2 \geq \sum_{i=1}^{3} \left(\left\lfloor \frac{2 - d_i}{P_i} \right\rfloor + 1 \right) x_i \tag{2.14}$$

$$2 \geq \left(\left\lfloor \frac{2-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{2-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{2-11}{12} \right\rfloor + 1 \right) 2 \tag{2.15}$$

$$2 \geq 1 + 0 + 0 \tag{2.16}$$

Since inequality 2.16 holds, the task set is feasible at time unit 2. Similarly, the same test is performed for $L = 3$.

$$3 \geq \left(\left\lfloor \frac{3-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{3-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{3-11}{12} \right\rfloor + 1 \right) 2 \tag{2.17}$$

$$3 \geq 1 + 2 + 0 \tag{2.18}$$

The inequality 2.18 holds, therefore, the task set at time unit 3 is also feasible.

Moving on to $L = 5$,

$$5 \geq \left(\left\lfloor \frac{5-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{5-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{5-11}{12} \right\rfloor + 1 \right) 2 \tag{2.19}$$

$$5 \geq 2 + 2 + 0 \tag{2.20}$$

Since the inequality 2.20 also holds, the procedure continues with $L = 7$.

$$7 \geq \left(\left\lfloor \frac{7-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{7-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{7-11}{12} \right\rfloor + 1 \right) 2 \tag{2.21}$$

$$7 \geq 2 + 4 + 0 \tag{2.22}$$

This member of A also holds as seen in inequality 2.20. Iterating further on the next member, at $L = 8$,

$$7 \geq \left(\left\lfloor \frac{8-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{8-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{8-11}{12} \right\rfloor + 1 \right) 2 \tag{2.23}$$

$$7 \geq 2 + 4 + 0 \tag{2.24}$$

the inequality 2.24 also holds. Finally, the last member $L = 11$ is also tested.

$$11 \geq \left(\left\lfloor \frac{11-2}{3} \right\rfloor + 1 \right) 1 + \left(\left\lfloor \frac{11-3}{4} \right\rfloor + 1 \right) 2 + \left(\left\lfloor \frac{11-11}{12} \right\rfloor + 1 \right) 2 \tag{2.25}$$

$$11 \geq 4 + 6 + 2 \tag{2.26}$$

The inequality 2.26 does not hold. Hence, the task set is not feasible. The result means that the deadline at time 11, for T_i will be missed. The left side of the inequality shows how many time units are available for the current time. The right side of the inequality shows how many time

units are needed for the tasks running at the current time.

EDF scheduler has to update the schedule each time a new job is activated. Compared to FPS algorithms, this seems to have a more significant overhead. However, if context switches are also considered, compared to FPS such as RM, EDF has a smaller run-time overhead.

In EDF scheduler, the exact schedulability analysis can be performed with a complexity of $O(n)$ whereas the analysis on RM is pseudo-polynomial. Besides, EDF can fully exploit the processor, while RM can only achieve up to 69% utilization [LDG04].

Several kernels implement EDF algorithm. Some examples of open-source implementations can be given as, S.Ha.R.K. [SH21], Erika Enterprise[26], AQuoSA [Aq10], Xen [DSS21], Plan 9 OS [Ja21], Linux [Li19b], MaRTE OS [Ko09], RTEMS [RT21], Litmus-RT (no planned updates as of 2018) [Bj20], and The Everyman Kernel [Wa14].

EDF is going to be used in the proposed architecture for preemptible periodic tasks.

Scheduling Servers

Mainly, server algorithms were invented to enhance the handling of sporadic and aperiodic tasks when periodic tasks also run at the same time. Scheduling servers are basically periodic tasks running at a specified rate. They improve the responsiveness and guarantee a particular bandwidth for each task without harming schedulabilities of themselves and other tasks. The server algorithms can be based on FPS or DPS algorithms. When the execution of an aperiodic task is not critical, usually they are treated as background tasks. If they are also critical, a common approach is to create a periodic server to service these tasks. This approach is called the *polling approach* [SLR87]. Aperiodic tasks create incompatibilities with periodic servers due to the polling approach. Aperiodic tasks arrive at bursts, rather than in a periodic way. The algorithms that overcome this problem are called *bandwidth preserving* algorithms [Ca84].

Polling Server (PS), Deferrable Server (DS), Priority Exchange Server (PES), and Sporadic Server (SS) can be given as examples to servers that use FPS algorithms. Adapted fixed-priority servers, Total Bandwidth Server (TBS), and Constant Bandwidth Server (CBS) are examples to the servers that utilize DPS algorithms.

In addition to the scheduling servers above, there have been extended versions of the algorithms. Spuri and Buttazo [SB96] introduced five new online algorithms for servicing soft aperiodic tasks in real-time systems, where a set of hard periodic tasks is scheduled using the EDF algorithm. Their algorithms achieve full processor utilization, but they assume that the aperiodic tasks have no deadlines. Dynamic Priority Exchange from Spuri et al. is the extended version of PES [LSS87], which trades its run-time with the run-time of the lower priority peri-

[26] Website: `http://erika.tuxfamily.org/`

odic tasks in case no aperiodic task requests are pending. It eliminates the wasted time and improves processor utilization.

The proposed reference architecture does not optimally schedule the combination of aperiodic and periodic tasks, when they are running together. However, this is considered as possible future work. For an optimal scheduling, the arrival times of the different type of tasks must be planned in advance or CPUs must be isolated. The architecture supports changing CPU affinity before or during runtime. Process affinities will be explained further below.

Process Life Cycle

A process is a runnable instance of a program (See Definitions). Occasionally, the process has to wait for events from external sources, such as keyboard input or input from a peripheral device.

To switch the processes, the scheduler must know the status of each process. A process can be in one of the following states at a time [Ma08]:

- **Running**: The process is being executed at the moment.
- **Waiting**: The process is able to run, but the CPU does not allow execution as it is allocated to another process. The scheduler can change the status when this process is the next to run.
- **Sleeping**: The process is waiting for an external event to run. The scheduler cannot select this process at the next task switch. Once the external event is triggered, it can only go to the ”waiting” state.

All processes are saved in a process table. Once the execution is completed, the process state goes to ”stopped.” Then, the allocated resources are released (memory, connection to peripherals, and CPU) and process entries are removed from the process table. However, even if the resources are released, they may be leftovers in the process table for a process. This process is then called a ”zombie” process.

The architecture in this thesis assigns an internal status to its active tasks during their life cycles, namely, running, paused, and stopped. These are assigned after retrieving the thread and process status. Waiting and sleeping tasks are considered as paused tasks. The tasks are assumed never to become a zombie.

Process and Interrupt Affinity

Each thread and interrupt in a computer system has a processor affinity. Together with policy and priority settings, affinities can help achieve maximum performance. Running processes

always race with other processes and interrupts for resources, principally for CPU time. Multitasking systems are more exposed to be indeterministic. As happened in the Mars Pathfinder Lander project [Sy13], a high priority task may be forced to wait while another low priority task to leave the critical section. Moreover, on multiprocessor (or multi-core) systems, migration of processes or threads from one CPU to another can be expensive due to cache invalidation.

Multi-threaded applications tend to run related threads on the same core. With affinities, all threads can be assigned to one core, as well. Affinity value is used by the scheduler of the OS to determine which threads and interrupts run on which CPU. If not specified, the threads and interrupts use the maximum affinity number, which means they have access to all CPUs. Linux kernel and many OSes, e.g., Solaris [Or11], OS X [Ap07], Windows [Mi18] support affinity settings. On Linux, affinity and bandwidth can be combined by using `cgroups`.

Under UNIX-compliant systems and Windows, affinity is defined as a bitmask, having values between 1 and $2^N - 1$, where N is the number of available cores. The bitmask is converted to the available CPUs by writing the number in the base-2 system and checking the bit locations. A one (1) in the bit location means that this CPU is available for use, whereas a zero (0) means that the CPU is not available. CPU numbering is a zero-based numbering. The first CPU's ID is 0. For example, if there exists hardware with one processor and four cores, then the bitmask can have values between 1 and 15. Assuming that a thread has an affinity of 13, the available CPUs for the thread can be calculated as seen in Eq. 2.27.

$$(13)_{10} = (1011)_2 \tag{2.27}$$

As seen in the Eq. 2.27, bit locations 0, 1, and 3 are 1. This means that CPU 0, CPU 1, and CPU 3 can be utilized for the execution.

In multi-core systems, one of the typical affinity settings is assigning one core for all system processes and allow applications to run on the other cores, with one core per application thread. However, the affinity settings must be designed in conjunction with the program/software/-command and its related settings. The usual practice to set affinity on a real-time system is first to determine how many cores are needed to run an application, then isolate those cores. Under Linux, setting affinities can be done by using `taskset` command. Entries in `/proc` file system must be modified to change the affinities of interrupts.

The proposed architecture in this thesis supports assigning and limiting threads/processes to use dedicated CPUs, to improve scheduler optimization. This process can be done during design time, or while the tasks are running. As the architecture does not support optimal scheduling for running a combination of periodic and aperiodic tasks on the same core (See Scheduling Servers), affinities can be used to isolate them. More on this topic will be explained in Sec. 4.2.

Control Groups (cgroups)

Real-time means determinism, meaning whenever a process or group of processes start, they must rely on a set of determined parameters, such as CPU time, which is also called CPU bandwidth. Although these parameters can be assigned individually for each process or a thread, the choices can also be left to the other kernel components. Under Linux, control groups (`cgroups`) enable setting these parameters to specialize the behaviour of threads, processes, and their children tasks, by grouping [Li20d]. `cgroups` are beneficial if resource planning needs hierarchy. For example, if low priority tasks should be allowed to use only 10% of the CPU at any time, a group with CPU utilization of 10% can be created. The sum of CPU utilization of all tasks assigned to this group will not exceed 10%, equally sharing the usage among the number of active tasks and giving more CPU time to higher importance tasks. CPU affinities of the groups can also be changed with `cgroups`. `cgroups` are called as credit scheduler in Xen [Ze13].

Linux kernel also features a scheduling group for real-time tasks called *rt-group-sched* based on `cgroups`. The bandwidth in *rt-group-sched* has a lower limit — the minimum guarantee — and an upper limit for execution. It defines how much time can be spent running a task in a given period. For each period, a process is run for the allocated runtime. The rest of the time is given for the processes outside of this group. These groups lack the complete implementation of EDF scheduling. Similar to priority inheritance (PI) for priority-based schedulers, EDF needs deadline inheritance to be implemented to prevent blocking problems. However, Linux PI mechanism is one of the most complex pieces of code in the kernel source [Li20a]. Therefore, it is a challenge to integrate this approach into the Linux kernel.

The proposed architecture in this thesis allows the creation of Virtual Processors (VPs) to determine the execution capacity of a task and change affinities by using the approach of `cgroups`. Multiple tasks can also be assigned to the same VP to create a hierarchy. Under Linux machines, VPs can directly work with `cgroups`. VPs will be explained in Sec. 5.1.7.

Scheduling on Multiprocessor Systems

Until the year 2002, there was an increase in the processor clock frequency for faster calculation [PH13]. However, the clock frequency had a hardware limit due to heat dissipation and power consumption. Computing power was increased without increasing the processor clock by the introduction of multiple cores. Multiprocessor or multi-core systems contain more than one CPU. Generally, they are considered in two groups: *Symmetric multiprocessing (SMP)* in which all processors are equal and the communication is via shared memory, and *asymmetric multiprocessing (AMP)* in which the processors are separate and used for specific tasks. AMP communication is usually carried out through message-passing. An example to the AMP is an audio player device. It is an embedded device that has different computing units, e.g., Digital

Signal Processing (DSP) unit and a hardware encoder/decoder for audio.

Using multicore brings additional challenges and requirements:

- **Load balancing**: All processors should be used with the same amount of work.
- **Scheduling**: Scheduling must be done so that all CPUs can work efficiently.
- **Synchronization**: Synchronization is required for the CPUs to schedule and prevent incoherent data sharing.
- **Communication**: Processes running in different CPUs must communicate. Additionally, the software/program must be designed to exploit multicore programming.

Moreover, multiprocessor systems can have two types of access to memory, namely uniform memory access (UMA) and non-uniform memory access (NUMA). In UMA, the access time to the memory is roughly the same, whereas in NUMA, the CPU physically closest to the memory has the shortest access time.

In shared memory processors, the performance is increased by the introduction of local caches. The shared data is migrated and replicated in these caches to increase the access speed. However, caching the shared memory may cause *cache incoherency*. Cache coherence protocols maintain the coherency of the caches. In these protocols, cache controllers keep track of the cached data and update them whenever the values are changed.

The scheduling issues on multiprocessor systems were being addressed since the early 1960s [Gr69; He61; Ri60]. Muntz and Coffman [MC70] introduced a preemptive and efficient scheduling algorithm for tree-structured computations. They assume all CPUs are identical, and all computations are specified as a set of tasks. They also accept that the task computations are acyclic, directed graphs, and they have known execution time. Tasks can also depend on each other. The research gives an efficient algorithm for tree-structured computations. In the late 1990s and early 2000s, several algorithms to schedule on multiprocessor systems are also proposed [BK01; Bu95; ELA94; MM98; MMR98; O-97].

In 2003, Lee et al. [LHK03] introduced an algorithm to schedule real-time tasks online, on multiprocessor systems. They evaluated their algorithm through simulation, and the results showed better values compared to the conventional fixed number of processor algorithms. Their algorithm uses an extensive search within the tasks list in the system. To reduce complexity, they introduced a heuristic approach. Lee et al. assumed that all processors are identical, the real-time tasks are non-preemptible, independent from each other, aperiodic, and scalable. A task is scalable if it is executed faster when its execution is distributed into multiple processors. They assumed the tasks to be non-preemptible due to the cost of implementing preemptive scheduling algorithms when I/O scheduling comes into the play. Additionally, the amount of extra overhead of this kind of algorithms was bigger, which is neglected in the theoretical phase [JSM91].

Unlike it might be expected, increasing the core count does not linearly decrease the execution time of scalable tasks. Contrarily, the *execution efficiency* is reduced as parallel execution brings more execution overhead due to communication, load distribution, and contention [LHK03].

As explained in the beginning of the chapter, the thesis introduces a novel scheduling algorithm, called Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling. This algorithm does not automatically manipulate CPU affinity of the tasks on multiprocessor systems during runtime. Similarly, the EDF scheduling will also be used as it is. CPU allocation must be performed during the design or execution time, by the user. The NAPATA scheduling will be explained in detail in Sec. 3.1.

2.3 Summary

This chapter explained the terms Cloud Computing and Edge Computing (Sec. 2.1), related work (Sec. 2.1.1) done in both areas with explanations of their differences from this thesis. Later, the chapter specified the requirements (Sec. 2.1.2) considered in the thesis and the enablers (Sec. 2.1.3) of which the thesis is inspired. The thesis aims to create a software reference architecture for Edge Servers, satisfying interoperability, extensibility, time sensitiveness, software reliability, security, privacy, and abstraction requirements. It is also influenced by several technologies such as TCP, APIs, virtualization, grid computing, and load balancing. The proposed architecture does not solely use these enablers, but also adapt them to the needs. TCP allows communication between Edge Computing components; APIs abstract low-level complexity; virtualization isolates tasks while keeping their performance at the desired level; grid computing establishes resource sharing among network, and load balancing offloads tasks depending on resource availability in the Edge Network.

Furthermore, this chapter continued with Real-Time Computing (Sec. 2.2) and its challenges (Sec. 2.2.1) while working with tasks or systems that require real-time execution. It elaborated how scheduling is performed under real-time computing, and explained two of the available schedulers (Sec. 2.2.2). The proposed architecture uses the Earliest Deadline First (EDF) for periodic task scheduling. For aperiodic tasks, Least Slack Time (LST) cannot be used, as it cannot schedule non-resumable aperiodic tasks optimally. To overcome this problem, the thesis introduces a novel scheduling algorithm which is going to be explained in Sec. 3.1 in detail. Optimal scheduling for a combination of periodic and aperiodic tasks is only possible by using scheduling servers. The proposed architecture, however, does not utilize any of them. Instead, it introduces Virtual Processors (VPs) to isolate cores of different type of tasks for optimal scheduling. This isolation is based on process affinities and follows the approach of control groups (`cgroups`).

3 Scheduling and Decision Making Methodology

In the previous chapters, existing technologies on Cloud Computing, Edge Computing and Real-Time Computing were clarified and the state-of-the-art technologies were listed. This chapter will introduce a novel scheduling algorithm (Sec. 3.1), explaining why there is a need for it. Moreover, based on the problem defined in Sec. 1.1, problem during selection of the server to offload will be formulated (Sec. 3.2), and the procedural approach (Sec. 1.3) to solve the problems will be shown.

3.1 NAPATA Scheduling

Limited hardware resources require a fair distribution of resources among all tasks on an Edge Server. The orchestrator that prevents resource starvation, enables fair resource usage, and switches the turn of the tasks is called *scheduler* (See Sec. 2.2.2). In the thesis scheduling is used as a fallback solution and performed if the tasks cannot be offloaded to alternative servers, or they are likely to miss their deadlines.

As defined in Sec. 2.2.2, there are three types of tasks in the computing domain: periodic, aperiodic, and sporadic. For periodic tasks, one of the scheduling algorithms developed for periodic tasks can be used. To schedule tasks in Periodic type, the thesis will use the Earliest Deadline First (EDF) scheduling (Sec. 2.2.2). For aperiodic tasks, Least Slack Time (LST) (Sec. 2.2.2) can be used. However, LST cannot optimally schedule non-resumable tasks. A non-resumable task cannot be resumed if paused by another higher priority task. Instead, it is restarted. This thesis also aims to work with legacy tasks and execute them without missing their deadlines. It also aims to improve the efficiency by enabling a combination of non-resumable and preemptible tasks to run together. This requires another scheduling algorithm.

The dissertation introduces a novel simple scheduling algorithm, called Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling. As its name suggests, its used to schedule non-resumable (legacy) and preemptible aperiodic tasks. NAPATA scheduling provides an online, dynamic priority, and preemptive scheduler with negligible overhead, and it uses counting sort with the complexity of $O(n+k)$ [Mo20]. NAPATA scheduling can work with non-resumable and preemptible tasks together. Instead of slack time, NAPATA scheduling uses only the remain-

ing times of the active tasks. If a non-resumable task needs to be preempted, the algorithm can terminate it if it can still be completed on time and start it from the beginning to complete execution.

Let T_i be the only task running on an Edge Server. Also, let x_i be the worst-case execution time and d_i be the relative deadline of this task. On an idle server N with enough computing power, if $x_i \leq d_i$ holds, this task can be executed on this server before its deadline. If the task starts at the time a_i, then, the absolute deadline of the T_i becomes $d_i + a_i$. Nevertheless, the inequality for feasibility does not change as a_i on both sides cancel themselves out ($a_i + x_i \leq d_i + a_i$).

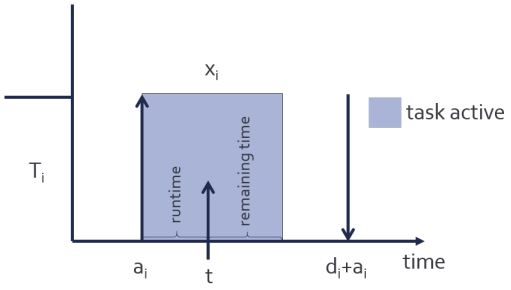

Figure 3.1: Calculation of the remaining execution time for a task T_i at time t.

At any time t, the feasibility may be rechecked, regardless of the need. As seen in Fig. 3.1, if the feasibility at the time t is to be calculated, the remaining execution time of the task can be used, which should be between t and $d_i + a_i$.

If r_i is the runtime since a_i and until t, and m_i the remaining execution time of the task until completion, Eq. 3.1 and 3.2 can be used to find them out.

$$r_i = t - a_i \tag{3.1}$$

$$m_i = x_i - r_i \tag{3.2}$$

$x_i \leq d_i$ can also be written in terms of t as seen in equations 3.3 to 3.8.

$$t = r_i + a_i \tag{3.3}$$

$$r_i = x_i - m_i \tag{3.4}$$

$$\tag{3.5}$$

inserting r_i in Eq. 3.3

$$t = x_i - m_i + a_i \tag{3.6}$$

and solving for x_i

$$x_i = t + m_i - a_i \tag{3.7}$$

yields

$$t + m_i \leq d_i + a_i \tag{3.8}$$
$$\tag{3.9}$$

However, if tasks are preemptible, equations 3.1 and 3.2 may not reflect the actual runtime or remaining time, as the task may be preempted at any time ($k_{i,j}$) between a_i and $d_i + a_i$. This case is illustrated in Fig. 3.2.

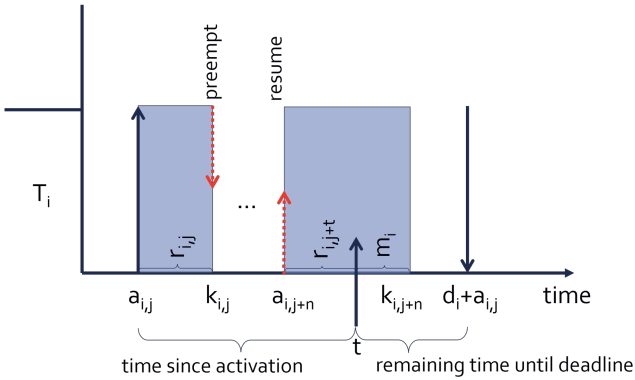

Figure 3.2: Calculation of the remaining execution time of a preemptible task.

In this case, actual runtime R_i of the task T_i until time instance t from the first arrival time a_i can be calculated as seen in Eq. 3.10.

$$R_i = \sum_{j=0}^{t} r_{i,j} = \sum_{j=0}^{t} (k_{i,j} - a_{i,j}) \tag{3.10}$$

Then, the remaining execution time M_i for task T_i becomes:

$$M_i = x_i - R_i \tag{3.11}$$

If there is more than one task request on an Edge Server, at each time instance t where a task request is made, it is necessary to check if any of the tasks miss their deadlines. In this scenario, the algorithm first sorts the active tasks at t by their absolute deadlines ($d_i + a_i$). Similar to EDF scheduling, the earliest deadline receives the highest priority. Then, starting from the highest priority task, for each task, the algorithm sums up the remaining times of the tasks with the same or higher priority, adds the current measurement time t, and compares the sum with the absolute deadline of the current task. If the sum is less than or equal to the deadline, the task is schedulable at time instance t. If the sum is greater than the deadline, the algorithm stops as

the task is not schedulable.

Based on inequality 3.8, the NAPATA feasibility algorithm F_i for task T_i is summarised in Eq. 3.12. The algorithm calculates the schedulability of task T_i with N running tasks at time t, whose priorities are equal to or higher than task T_i. The left side of the inequality represents the minimal time required to execute the task T_i.

$$F_i = t + \sum_{n=1}^{N} M_n \leq d_i + a_i \tag{3.12}$$

Let task $T_i = \{a_i, x_i, d_i\}$ have an arrival time of a_i, an execution time of x_i, and a relative deadline of d_i. Also, assume two tasks T_1 and T_2 as seen in Eq. 3.13 arrive at $t = 0$.

$$\begin{aligned} T_1 &= \{0, 2, 5\} \\ T_2 &= \{0, 3, 4\} \end{aligned} \tag{3.13}$$

If they were periodic tasks whose deadlines are equal to their periods, using the feasibility equation for EDF given given in Eq. 2.6 would yield:

$$\frac{x_1}{d_1} + \frac{x_2}{d_2} \leq 1 \tag{3.14}$$

$$\frac{2}{5} + \frac{3}{4} \leq 1$$

$$\frac{23}{20} \leq 1 \tag{3.15}$$

As inequality 3.15 does not hold, according to EDF Scheduling, the tasks are not schedulable. However, if these tasks are not periodic, Eq. 3.12 can be applied to test the feasibility. If the tasks are in aperiodic type, they run only once, and they will not be requested again in a predictable time. For T_2 having a higher priority than T_1, first, T_2 will be calculated. Both tasks request execution at time zero ($t = 0$). The calculation of feasibility F_2 for T_2 is shown in Eq. 3.16.

$$\begin{aligned} F_2 &= t + (x_2 - R_2) \leq d_2 + a_2 \\ F_2 &= 0 + (3 - 0) \leq 4 \end{aligned} \tag{3.16}$$

Since the inequality 3.16 holds, the calculation is repeated for other tasks (in this case T_1) that are requested at $t = 0$. Calculation of F_1 for T_1 will also require T_2 values as T_1 has a higher priority than T_1.

$$F_1 = (0 + (2 - 0)) + (0 + (3 - 0)) \leq 5 \tag{3.17}$$

Inequality 3.17 also holds. This means that these tasks can be scheduled if they are known as aperiodic.

One important remark here is that, as mentioned, preemptible tasks can continue from where they left off, keeping their remaining times as they are preempted. However, preempting non-

resumable tasks means that they are terminated. Hence, their running time resets — also their remaining times. The following example will demonstrate another scenario with different task types.

Assume that the tasks arrive at an Edge Server as shown in Table 3.1.

Table 3.1: An example set of tasks with mixed types for the Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling.

Task	a	x	d	$d+a$	Type
T_1	0	2	8	8	Non-resumable aperiodic
T_2	1	1	3	4	Non-resumable aperiodic
T_3	2	2	3	5	Preemptible aperiodic
T_4	1	1	3	4	Preemptible aperiodic

Unlike the first example, this example cannot directly use Eq. 2.6, due to arrival times of tasks being different. Moreover, the tasks are also not periodic. However, NAPATA scheduling can be used to check whether they meet their deadlines.

There are three arrival times: 0, 1, and 2. The algorithm will be repeated at each arrival for each task that is active in these times, whether it is in running or preempted state.

At $t = 0$

$$F_1 = 0 + (2 - 0) \leq 8 + 0 \tag{3.18}$$
(3.19)

At $t = 0$ only T_1 is active, and the inequality 3.18 holds. Then, T_1 is executed until $t = 1$, since there is another arrival at that time point. At $t = 1$, T_1, T_2, and T_4 are active. Sorting them by their absolute deadlines gives the following order: T_2, T_4, T_1. Until this point, T_1 ran for one unit, and one unit execution remains, however, there are higher priority tasks which require T_1 to be preempted. T_1 has the non-resumable type, meaning, when preempted, at each resumption, the remaining time resets to its original execution time. Other tasks have just arrived; hence they have remaining times equal to their execution times. T_2 and T_4 have equal priorities. Any of them can be picked first, but their remaining times must be added when checking each of them. Starting from the highest priority:

At $t = 1$

$$F_2 = 1 + (1) + 1 \leq 3 + 1 \tag{3.20}$$
$$3 \leq 4$$
$$F_4 = 1 + (1) + 1 \leq 3 + 1 \tag{3.21}$$
$$3 \leq 4$$
$$F_1 = 1 + (2) + 1 + 1 \leq 8 + 0 \tag{3.22}$$
$$5 \leq 8$$

Inequalities from 3.20 to 3.22 hold. In this example, T_2 is chosen to be executed until $t = 2$.

At $t = 2$, T_3 also arrives. At this time point, T_2 completes execution, and the server contains three active tasks. Similarly, sorting the tasks by their absolute deadlines gives: T_4, T_3, T_1. Following the algorithm:

At $t = 2$

$$F_4 = 2 + (1) \leq 3 + 1 \tag{3.23}$$
$$3 \leq 4$$
$$F_3 = 2 + (2) + 1 \leq 3 + 2 \tag{3.24}$$
$$5 \leq 5$$
$$F_1 = 2 + (2) + 2 + 1 \leq 8 + 0 \tag{3.25}$$
$$7 \leq 8$$

Inequalities from 3.23 to 3.25 also hold. Since there are no more task arrivals, using the algorithm, it can be ensured that the tasks will be scheduled on time. The resulting scheduling diagram is shown in Fig. 3.3.

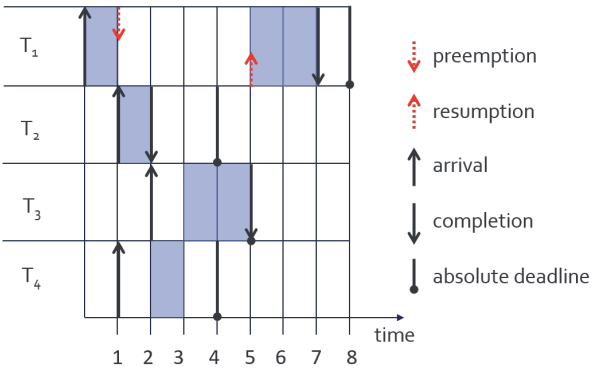

Figure 3.3: Resulting scheduling diagram of the NAPATA scheduling example.

If T_1 were to be a preemptible type instead of non-resumable, inequality 3.25 would be written as shown in inequality 3.26 and T_1 would be finished at $t = 6$ instead of $t = 7$.

$$F_1 = 2 + (1) + 2 + 1 \leq 8 \tag{3.26}$$

$$6 \leq 8 \tag{3.27}$$

Preemptible tasks always satisfy $M_i \leq x_i$ condition. As seen in Eq. 3.26, the time required to execute T_1 is less than the value in Eq. 3.25.

Next section will address server selection problem and how the thesis plans to solve it.

3.2 Problem Formulation on Server Selection

Almost all operating systems (OS) introduce schedulers to plan the execution optimally. In a decentralized environment, an offloading requires further precautions. Problem definition (Sec. 1.1) briefly mentioned decisions that should be made to execute tasks in a decentralized environment collaboratively. Each Edge Server needs to know the resources of other Edge Servers in the Edge Network. Moreover, this information should always be kept up-to-date to avoid wrong decisions. This section will exemplify the execution of a periodic task using the Earliest Deadline First (EDF) scheduling on an Edge Server to answer the fourth question from the problem definition (Sec. 1.1). It will show how an Edge Server is different than an ordinary computer by formulating some additional parameters. It will also show how an Edge Server calculates the feasibility of multiple task executions based on the task parameters. Finally, it will summarize the formulations in a table, leaving the solutions to the upcoming chapters.

Let C be a set of computers (Eq. 3.28) and R be a set of resources in an Edge Network N (Eq. 3.29). Also let S be a set of services, which are wrappers to define behaviours of programs, software or commands, and let Y be a pre-defined set of service types defined as in Eq. 3.30.

$$C = \{C_i \mid i \in \mathbb{N}^+\} \tag{3.28}$$

$$R = \{(p, m, c, s) \mid p, m, c, s \in \mathbb{N}^+\} \tag{3.29}$$

$$Y = \{LEGACY, SIMPLE, SIMPLE\_PERIODIC\} \tag{3.30}$$

where

- p: maximum execution speed in millions of instructions per second (MIPS)
- m: total available memory
- c: total core count
- s: total disk space

and

$$S = \{(a, x, d, o, w, l, f, Y_y) \mid a \in \mathbb{N}, x, d, o, w \in \mathbb{N}^+, \forall y \in Y, x \leq d, l \in \mathbb{R}_{>0}, l \leq 1\} \quad (3.31)$$

where

- a: arrival time instance of this request
- x: worst-case execution time in terms of millions of instructions (MI) for this service
- d: relative deadline of the service (in seconds)
- o: required memory throughout the execution
- w: required disk space throughout the execution
- l: allowed load w.r.t. CPU usage ($0 < l \leq 1$)
- f: list of allowed CPUs that this task can use ($|f| \leq |c|$)

Using the definitions and the equations 3.30 through 3.31, the Edge Server E set can be defined as:

$$E = \{(R_i, Q_i) \mid i \in C, Q_i \subseteq S\} \quad (3.32)$$

To define the Edge Network N as seen in Eq. 3.34, the connection set B is denoted as shown in Eq. 3.33.

$$B = \{O_i \mid i \in E, O_i \subseteq E\} \quad (3.33)$$

$$N = \{(K_i, X_i, D_i) \mid i \in E, K_i \subseteq E, D \in \mathbb{R}_{>0}\} \quad (3.34)$$

$$T = \{(S_z, E_i) \mid z \in S \text{ and } S_z \in Q_z \text{ and } i \in E\} \quad (3.35)$$

where D is the delay between two E that are connected to each other. Each available S in E is executed as task T (See Eq. 3.35). To check whether a task T can be executed in E, a feasibility check must be performed. Depending on the service type (hence the task type) Y, one of the scheduling algorithms explained in Sec. 2.2.2 can be chosen. To reduce the complexity of the problem, it is assumed that memory (m) and disk space (s) are unlimited. Moreover, the scheduling algorithms are considered to have no overhead and hardware executing the task is assumed to be ideal and real-time proven.

Assuming that the service types are periodic and that they satisfy the requirements by Liu and Layland task model [LL73], if EDF Scheduling is chosen (Sec. 2.2.2), the schedulability test F for

n tasks can be performed as shown in Eq. 3.36. Let

$$F = \sum_{j=1}^{n} \frac{x_j}{p_{E_i} d_j} \leq 1 \tag{3.36}$$

where x_j and d_j are worst-case execution time and deadline, respectively, of task T_j, where T is a running task set on an Edge Server E_i with one core, and p_{E_i} is the maximum execution speed of E_i.

With the assumptions mentioned earlier, if the inequality in Eq. 3.36 holds, then the tasks can be scheduled using EDF. However, if the Edge Server has multiple cores, then the tasks can have different CPU affinities. In this case, the utilization $U_{c_{i,m}}$ for the m^{th} core of E_i — $c_{i,m}$ — can be calculated as:

$$U_{c_{i,m}} = \sum_{T} \frac{x_j k_i}{p_{E_i} d_j} \text{ where } k_i = \begin{cases} 1 & \text{if } c_{i,m} \in f_j \\ 0 & \text{otherwise} \end{cases} \tag{3.37}$$

Then, the sum of all utilizations of all c cores of E_i (Eq. 3.38),

$$F_{E_i} = \sum_{m=1}^{c} U_{c_{i,m}} \leq c \tag{3.38}$$

decides if a task set can be scheduled with the current CPU affinities. If the CPU load l_j of each task is to be considered and T tasks have been running at a time, then Eq. 3.37 becomes as in Eq. 3.39.

$$U_{c_{i,m}} = \sum_{T} \frac{x_j k_i l_j}{p_{E_i} d_j} \text{ where } k_i = \begin{cases} 1 & \text{if } c_{i,m} \in f_j \\ 0 & \text{otherwise} \end{cases} \tag{3.39}$$

If $F_{E_i} \leq c$, then the tasks can be scheduled on that server, but only if their CPU affinity tests pass. The affinities to the CPUs are tested using Eq. 3.38 for each CPU in one Edge Server. If $\forall U_{c_{i,m}} \leq 1$, then the tasks can be scheduled on this Edge Server. If $\exists U_{c_{i,m}} > 1$, then the tasks can be scheduled only if their affinities are changed. However, if Eq. 3.38 is not satisfied, another alternative server must be searched for the execution. The decisions based on results are summarized in Table 3.2.

Table 3.2: The feasibility table to determine the schedulability of the tasks within an Edge Server.

		if $F_{E_i} \leq c$	
		Satisfied	**Not Satisfied**
if $\forall U_{c_{i,m}} \leq 1$	**Satisfied**	Schedulable	
	Not Satisfied	Change affinity	Not schedulable on this server

Eq. 3.38 and 3.39 can be used on a single Edge Server. If a task is not feasible for scheduling on an Edge Server, it may still be possible to execute on another server within the Edge Network N. However, it is necessary to know that server's current resource information. If there are multiple servers, a decision must also be made to pick the location for the execution of this task. First, the load (l_j) of task j can be throttled down if its absolute deadline allows it. Besides, each service type requires different handling for the correct scheduling. Then, a trade-off stems between the execution time and CPU utilization. During the calculation, delays D between Edge Servers must also be considered. The decision making to choose an adequate E in N, and the execution utilization of task set T are two of the problems that are to be solved in this thesis. Moreover, planning the execution order of the non-resumable (legacy) and preemptible tasks is another contribution of the work. The proposed software reference architecture deals with these problems and defines the methodology on how to achieve optimal performance. The solutions are given in Chapter 4 and validated in Chapter 5.

3.3 Summary

This chapter explained why there is a need of another scheduling algorithm for non-resumable and preemptible aperiodic tasks and introduced an online and preemptive novel scheduling algorithm (Sec. 3.1). Moreover, Sec. 3.2 represented an Edge Server as mathematical formula and then formulated an example of offloading problem of multiple periodic tasks. To address this problem, the following chapter will explain how a software reference architecture can be built based on the previous knowledge.

4 Software Reference Architecture

This chapter describes a software reference architecture for Edge Servers to put Edge Computing benefits such as real-time execution, self-configuration, extensibility, and resource-awareness as defined in Chapter 2.1 into practice.

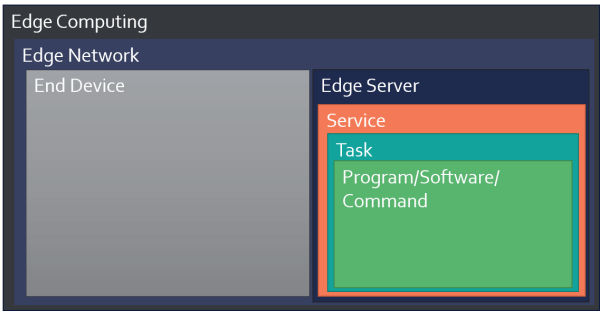

Figure 4.1: Subsets and supersets of Edge Computing, as defined in this thesis.

Edge Computing components are illustrated as subsets and supersets in Fig. 4.1. In this thesis, Edge Computing is seen as the superset of Edge Network, which consists of End Devices and Edge Servers. Services define how tasks behave when run and tasks are instances of services, linked with the programs/software/commands to execute them.

Terms used throughout the chapters are further explained in the chapter and also summarized in the Appendix A.3 to avoid ambiguities. These terms and a high-level overview of an example Edge Network are shown in Fig. 4.2. As seen from the figure, Edge Servers have direct access to the End Devices or each other using real-time communication methods. An End Device is a resource-limited device, which can be a smart sensor, machine, computer, mobile phone, smart glasses, or more. In this thesis, its role is to request tasks from Edge Servers. Edge Servers are physical computers, which can perform computations and store data. They need to implement the defined software reference architecture in this thesis to exploit the benefits of Edge Computing. It is also possible to connect Edge Servers with the Cloud. However, in this case, best-effort communication will be established due to the nature of the Internet.

The architecture or thesis does not recommend a specific network topology. A single topology or combination of topologies are possible as long as the servers and End Devices have enough

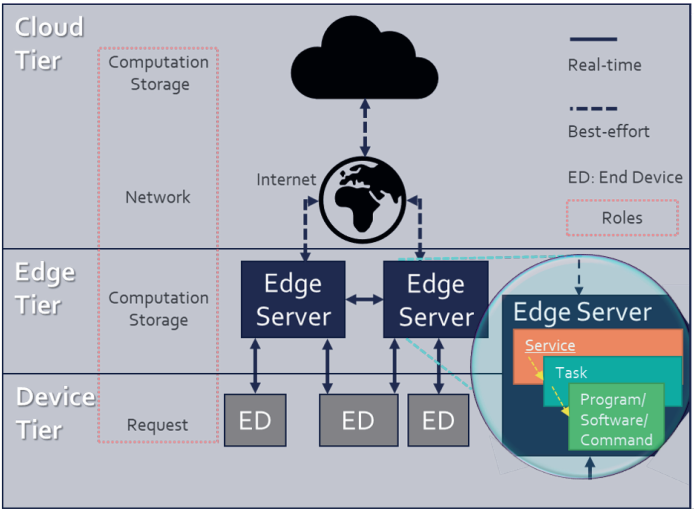

Figure 4.2: An example Edge Network and its participants (nodes). Dashed arrows show best-effort communication due to the behaviour of Cloud Computing, whereas solid arrows show real-time capable communications.

network interfaces to connect with. An End Device or Edge Server can establish as many connections to other Edge Servers as possible. Once a request is made, underlying decision mechanisms will evaluate several options choose an optimal Edge Server to complete this request and return its response to the initial requester. The options are throttling the Central Processing Unit (CPU) utilization of the tasks, scheduling them or offloading to their neighbouring servers. This process will be seamless, and the End Device will not need to implement anything special for this to happen. As it might be noted, the execution of the tasks is called ”requesting” tasks instead of ”starting” tasks. The reason for this is because the decision of execution moment and location being under the control of the decentralized Edge Servers. The optimal server location will be determined by the Edge Server that first receives the task request. Then, that optimal server will execute the task when it is appropriate.

Even though each server is independent, when there is an activity on a server, this activity is shared with all other connected servers. This information exchange continues until all servers are informed about this activity. As a result, whether Edge Servers have direct access to each other or via another Edge Server, each of them is aware of the currently available resources of all servers. Moreover, thanks to the synchronized information, the choices and results are always the same, regardless of the server. As it also might be noticed in Fig. 4.2, tasks are different from program/software/command (PSC). *Tasks* are instances of *Services* that define the execution behaviour of *PSCs* (See Sec. 4.2). Execution behaviours (See Sec. 4.2.5) are parameters such as execution time, deadline, CPU utilization, allowed CPUs, and so on. PSCs are developed

independently from the architecture using programming languages to connect the Edge Servers with the physical device, control it, or perform various computations on the Edge Server. A PSC can have multiple services defined for it to have different behaviours. Once a task is requested, the task itself will call the PSC that is linked with its service via command line parameter when it is time to start the task. All components and decision mechanisms are going to be explained in the upcoming sections.

The architecture is designed to be open, operating system (OS) neutral, and to have no proprietary standards which can hinder the usability or create vendor lock-in problems. It is also designed to be flexible, and scalable, allowing new servers to be connected with minimum effort. Moreover, the architecture supports multiple user interaction by separating each session but sharing the resources. Once it is deployed, it works with other participants in the Edge Network, interoperably. To fully exploit the features of the architecture and enable collaboration among the participants, all Edge Servers in the vicinity need to be incorporated into a shared network. This integration is abstracted from the lower-level operations by the introduced standard Application Programming Interface (API) methods.

The architecture suggests two roles for users: (1) framework users, or *operators*, who are allowed to set up the Edge Network, configure Edge Servers, and add/remove services, (2) *end-users* that are End Devices, or persons who use these End Devices. The following steps are endorsed for a successful setup for the operators:

1. Implement a framework based on the software reference architecture defined in this thesis and deploy it on the servers that are expected to participate in the network.
2. On these servers, install the user PSCs that should be available for End Devices.
3. Create services to specify the execution behaviour of each installed PSC.
4. Leave these services public or make private, deciding whether they should be accessible by other Edge Servers or not.
5. Connect Edge Servers to create the desired Edge Topology.

The steps above summarize the preparation phase of the Edge Network. A Cloud server can also run the same framework to participate in the execution. However, as mentioned before, the deployment in the Cloud will provide only best-effort service rather than a real-time response.

To completely establish connections between two Edge Servers or Edge Server to End Device, authentication is necessary to prevent unauthorized access. The authentication requires the creation of users in each server. These users must be created individually in the servers, together with their access levels. Once a user logs in, they are allowed to perform further actions.

During establishing set of connections between Edge Servers, each server shares its resources and public services with other participants automatically, to create a joint "knowledge base." An

End Device can request tasks from any of its connected Edge Servers; however, as mentioned earlier, the task will be instantiated on the most suitable Edge Server after a decision of location is made. This decision is made by performing different calculations on the initial Edge Server that receives the request (Sec. 4.3). If the chosen server is not authenticated, that server is not available for execution; hence, it will not be considered during the decision making. Requested tasks call and run the PSCs whose execution behaviours are defined by their services. These services can explicitly inform about execution duration, relative deadline, CPU affinity, and CPU execution capacity of each PSC. Once an Edge Server receives the request, first, the availability of this PSC within the network is queried, including the current server. Based on the behaviour of the service such as its deadline, the most suitable server in the network is chosen, again, also considering the current server. The initial server also evaluates whether the task is going to complete its execution until its defined deadline (See Sec. 4.3). The chosen server can have a direct or indirect connection to the original requester. Regardless of how they are connected, if the task sends a response upon execution, it will also be delivered back to the requester using the same connection path. All of these steps are going to be performed seamlessly. The requester will get serviced without noticing the underlying process.

Following sections explain the concepts to design such a software reference architecture to behave as described above. The architecture explained in this chapter is realized with a novel framework called Real-Time Edge Framework (RTEF) (Chapter 5). The RTEF is tested in a simulation environment, but it is implemented in such a way that it requires minimal modifications to work on hardware when deployed. During the tests, the hardware is assumed to be ideal, and the decisions mechanisms to have no overheads. The tests are going to be explained in detail in Chapter 5.

4.1 Edge Server and End Device Concepts

An Edge Network consists of End Devices, which the job requests are originated from, and Edge Servers, which execute these jobs. End Devices can be any source or destination device that requests a task in the architecture (See Definitions). The only requirement of being an End Device is to have a network interface to communicate with Edge Server(s). Edge Servers are physical hardware and communicate with the End Devices and other Edge Servers using a connection-oriented communication protocol. They can also be used as hubs to connect multiple devices together. These servers are regular computers, but converted into Edge Servers after they run a framework, developed based on the reference architecture explained in this chapter. If they are formally verified for real-timeliness, then, they can execute real-time jobs. Whether they are real-time capable or not, the servers will work collaboratively to handle the requests. The thesis aims to orchestrate the execution of real-time tasks in a decentralized environment; non-real-time task executions are out of the scope. To accomplish the objectives defined in Sec. 1.2,

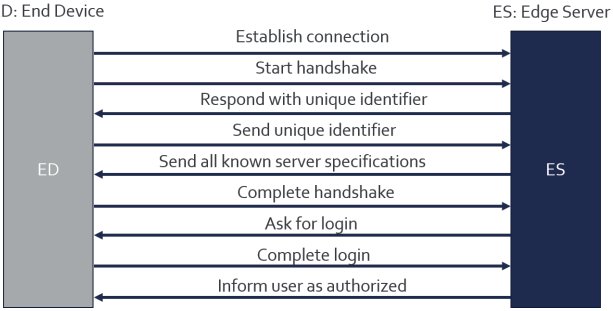

Figure 4.3: Handshaking, after establishing a connection between an End Device (ED) and Edge Server (ES).

the software reference architecture explained in this chapter needs to be instantiated.

Before or after an Edge Network is set up, the user PSCs on Edge Servers are installed. These software or programs can be installed on all Edge Servers, or some of them. *Services*, which can be thought of as wrappers for the PSCs, define how PSCs run when they are executed. They also enable interoperability between other Edge Servers. Whether the PSC is optimized for decentralization or not, the services allow execution of user PSCs from any of the available Edge Servers in the same network. Each service has several parameters, which need to be set. These parameters also include the PSC executable to link the service. These will be further discussed in the next section (Sec. 4.2). Once a service is created, it can be set as *public* and broadcast to the Edge Network so that other servers are also aware of this service. It is also possible to set the services as *private*, allowing only execution from the End Device that is directly connected to an Edge Server. *Public* is the default access type of services.

An Edge Network can be set up with at least one computer. The computer should also have network interface(s) to communicate with the End Devices. The architecture is designed to be realized on limited-resource computers. A deployable instance of an architecture is called a framework. When a framework based on the reference architecture is installed on a computer and run, this computer is considered as an Edge Server. To enable interworking, more than one Edge Server within the same network is necessary. Each Edge Server or End Device in an Edge Network is also called a node. Each node should have a unique identifier that is to be used for communication among all other servers. There is no limitation on the network topology, meaning the Edge Topology can be created using any of the available network topologies. To create an Edge Topology, the Edge Servers establish connections between each other using a connection-oriented communication method. The servers do not introduce discovery methods to find each other. Instead, one Edge Server initiates a connection using a set of messages. After each successful connection, the latest topology is updated and stored in each Edge Server. Once a connection is established, whether it is between an Edge Server and End Device or another

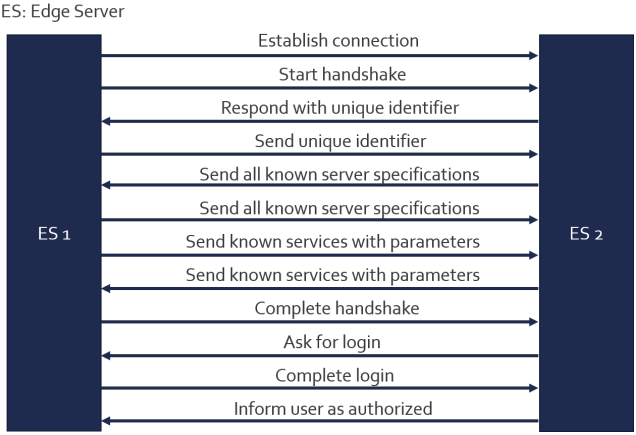

Figure 4.4: Handshaking, after establishing a connection between two Edge Servers (ES).

Edge Server, a handshaking process is initiated. The new connection is shared between all active participants of the network. The handshaking informs the nodes about their available resources (if any) before preparing them for collaborative work. If the connection is between an End Device and Edge Server, the handshaking is done as seen in Fig. 4.3. In this connection type, the connection initiator starts the handshake by sending a message to the Edge Server. Then, the Edge Server responds to this message by sending its unique identifier back. The initiator also sends its unique identifier to identify itself. At this stage, the topology is updated by adding the new End Device. Next, the server sends specifications such as CPU speed and core count of all known servers in the network. The End Device can ignore this message or use it for internal purposes. Then, it completes the handshaking process. Finally, the login process begins, and the user is authorized for further operations if the login is successful.

Establishing a connection between two Edge Servers is similar to the one with Edge Server and End Device connection. As seen from Fig. 4.4, handshaking follows the same procedure until the connected server sends its server specifications. In this case, server specifications are stored and used to determine the optimal location for requests. They are also used to estimate the execution time for a possible offloading to that server. Since the initiator is also an Edge Server, this time, instead of completing the handshake, it also sends its specifications to inform that it is a server with computing capability. Besides, the initiator further sends all known public services with their available locations. Naturally, if this is the first connection that the Edge Server is establishing, the only services sent will be its public services. Once the connected server receives the list, it updates its known service list and also informs other connected servers (if any) with the new services and their locations. This process is repeated until all servers contain up-to-date service information. Later, the connected server responds with the latest known service list. Finally, the handshake is completed, and the server proceeds with the login pro-

cess. The login process is the same as the one for End Devices. It could be either username and password-based or automated, using key-based authentication methods. The topology information, available services, and resources are also updated whenever a change on the servers is detected, not on a polling basis for the best accuracy.

As it might be noticed, all nodes, including Edge Servers, need to be authorized for further operations, right after a connection is established. Authorization can be password-based for manual input, or via shared keys. At the current status of the work, the users of a server are local. They are not stored in any database nor shared with other servers. At each server, users must be manually created and assigned an access level. However, in the future, it will be possible to use authentication servers or user directories to share users among the servers. End-users can perform only read-only operations and request tasks. Operators, on top of end-user rights, can change configurations, add, remove or modify services and users. No actions are allowed if the session is not yet authorized.

The architecture gives the control of the task execution to Edge Servers. From the End Device perspective, the architecture is seen as a whole. End Devices are not allowed to start tasks directly on the desired Edge Server. Instead, they *request* task execution from one of the directly connected Edge Servers. The execution is then performed seamlessly, and the task response is sent back to the caller if required. The reason for this is to give the decision rights to the Edge Servers for determining the most appropriate location to execute the task and to minimize the decision errors that could have been made by the End Devices. When a request arrives, the location of the execution is chosen based on several aspects such as the availability of the service, availability of the hardware resources of the servers, delay between the servers and requester, deadline of the service, etc. The static parameters are collected and stored during the handshake phase. However, as the resource availabilities, delays, and latencies depend on the load and network activity, this information is updated and broadcast to other neighbouring servers whenever a new task activity is detected. Since all information is up-to-date, a server can calculate and determine the optimal location to execute the task, without consulting other neighbouring servers. Requesting instead of executing also abstracts the complexity of execution procedure from the End Device. This abstraction requires the introduction of a *task request* method by the Edge Server, rather than to *start* one. An example of a task execution procedure is shown in Fig. 4.5.

Assume that *End Device 1* would like to execute *Program/Software/Command A* (*PSC A*) as seen in Fig. 4.5. To achieve that, first, it (1) requests *Service 1*, which defines the behaviour of *PSC A*, from *Edge Server 1*. Since this server does not contain this service, but only *Edge Server 2*, the request is (2) passed to *Edge Server 2*, which then (3) instantiates *Task 1*. Finally, *Task 1* (4) runs the *PSC A*. During the request passing, delays of each hop are also added to the total execution time, hence calculated to prevent deadline misses of the tasks.

Similarly, assume that *End Device 2* (5) requests *Service 1*, from *Edge Server 3* to execute *PSC A*,

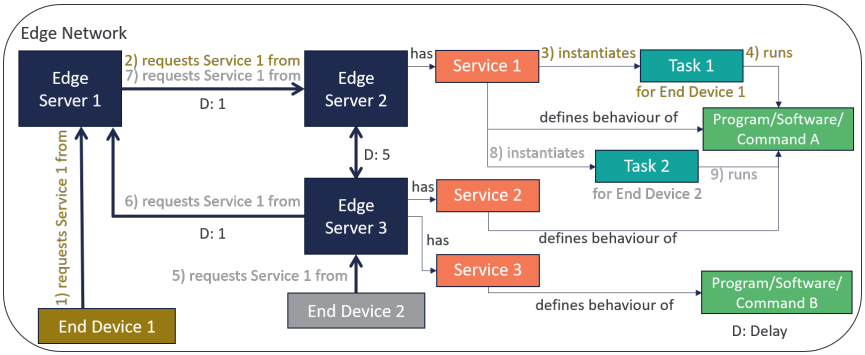

Figure 4.5: An arbitrary connection example to illustrate requesting a task in an Edge Network, containing three Edge Servers and two End Devices. Communication with the target server always follows the shortest path.

after *Task 1* completes its execution. This time, the request (6) follows *Edge Server 1* and then (7) *Edge Server 2* path, instead of going directly over *Edge Server 2*. This is due to a bigger delay between *Edge Server 3* and *Edge Server 2*. Later, another instance of *Service 1* is (8) instantiated with a new task (*Task 2*), and the *PSC A* is (9) run. This example is a very brief introduction to show how seamless the execution from the End Device perspective is. There are more underlying decision mechanisms that deal with resource availability issues in case multiple tasks are requested. These mechanisms are going to be introduced in Sec. 4.3.

4.2 Service and Task

A *service* is a piece of software that is reusable to perform a specific work (See Definitions). In this thesis, services specify the behaviours of a program, software, or command. Services must be created manually at each Edge Server where they are intended for use, by the operators. Once a service is created, it is permanent and available until deliberately removed. A *task* is an instance of a service, in the running or paused state. Services are linked to PSCs and tracked by the tasks. Instead of services, tasks execute the PSCs and control their lifecycle, after they have been requested. Each task receives a unique identification number after being requested. This number is used to find them during their lifecycle. Each new task instance increments this number, and the last number is stored until the Edge Server is rebooted. These numbers are unique only on the same server. The same number can be used as long as it is on another server. Tasks are automatically removed after they complete execution or they are terminated. Tasks can be used only once. Each request creates another task instance with a unique number. However, services can be reused. Multiple End Devices can request execution of a PSC multiple

times, using the same service.

Unlike End Devices, some other physical devices (e.g., a simple sensor or simple lights) may only accept inputs and/or return outputs. These devices may not have an ability to request tasks from Edge Servers. In such cases, a PSC will make the linkage between the Edge Server and physical device directly. Depending on the connection types that the Edge Server supports, such as OPC-UA or digital input/output, the PSC can be implemented as it would normally be developed for another environment. Its request must then be initiated locally on the server or by another End Device. A PSC can also be a package directly deployed as a Docker container. However, it should be noted that this may cause additional overheads and require more time to start it. If a container is used, its service should be linked with the container command that will directly start the PSC.

To execute a task (hence a PSC) through a service, the service must have been created in the Edge Server. Once the service is available, it can be broadcast inside the Edge Network to be known by other Edge Servers. Only services defined as public can be broadcast in the network. Private services can only be accessed if the End Device has a direct connection to that Edge Server. A created service is set as a public service by default. Services have unique names for their identifications later on. Although each task instance gets a new incremented unique identifier, services use the same name for reference. If a different name is used on another Edge Server, it cannot be used for offloading. Services are linked with one or bi-directional PSCs. This parameter is also used to calculate the delay, which is added up to the execution time of the PSC. If a PSC requires inputs only from one end and gives no response, it is defined as a one-directional service. However, if a response must be sent, then, it is defined as a bi-directional service. The responses of bi-directional services must be sent to the original caller, which are End Devices. The delivery of this response is done by the Edge Servers, not the service or PSC itself.

Tasks cannot change service parameters permanently, but only temporarily, as long as they are active. Tasks implement a *pause* method that pauses a PSC's thread group if they are running, a *resume* method that resumes the paused threads and a *stop* method that terminates its execution. Services work on tasks instead of directly dealing with the PSC that is tracked. Multiple services can be linked with a single PSC. For example, two services can be created for a PSC, one for running it as a single-threaded process and the other one with multi-threaded, by modifying its command line parameter.

Tasks in the proposed architecture have four states: (1) active or requested when they arrive, (2) started or running when the execution of the linked PSC begins, (3) paused or preempted, when a higher priority task is executed, and (4) terminated, when their execution is complete. Unique identifiers of tasks are assigned as soon as a task instance is created. Once a task is terminated, it is no more accessible.

A PSC is assumed to have the following properties for the architecture to function properly:

- All threads of the PSC start at the same time as the main process.
- All threads of the PSC stop execution at the same time as the main process.
- PSC is independent of other running PSCs, e.g., when preempted, it does not cause a critical section issue.

As mentioned in Sec. 2.2.2, according to arrival patterns, tasks in computing domain are classified in three categories: (1) periodic tasks that arrive at a constant rate and have an infinite sequence of identical activities, (2) aperiodic tasks that are usually event-driven and have no bound inter-arrival times, and (3) sporadic tasks that are aperiodic tasks with bounded inter-arrival times [Au91]. These categories define the relevant scheduling algorithms for the tasks. Based on these categories, this thesis defines three service types that are mapped with the tasks. These types are, namely, Legacy, Simple, and Simple Periodic. Each of them is explained below.

4.2.1 Legacy

Legacy services are for non-resumable PSCs. This kind of services can also be considered as wrappers for *aperiodic* tasks. Pausing this kind of tasks implies terminating the execution of PSC; thus, they cannot continue from the paused state. Regardless of the duration of the service ran until this point, resuming it starts from the beginning, ignoring the former execution time. Once the execution is completed, its task is removed.

4.2.2 Simple

Simple services are for the PSCs that get continuous or streaming data, or that can be paused. Similar to Legacy services, they also wrap *aperiodic* tasks. Different from Legacy services, they can be paused. Once they are started, pausing does not cause them to reset the execution time until that point. Resuming these tasks enables them to complete their remaining time. For example, a video application can be wrapped with a Simple service.

4.2.3 Simple Periodic

Simple Periodic services are repeating Simple services. As their name suggests, they wrap *periodic* tasks. They are usually used, e.g. to get status from a sensor or device, or for control loops. Since they are expected to arrive always at the specified intervals, the scheduling algorithm keeps enough resources allocated for these tasks at all times. Pausing their instances keeps their remaining times untouched, and they can be resumed.

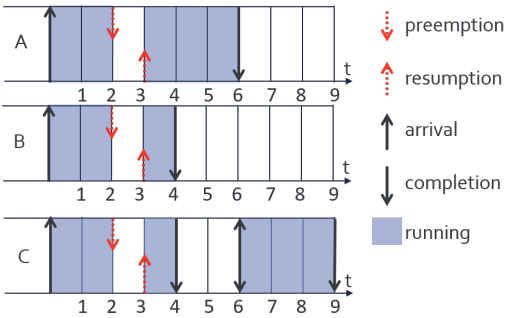

Figure 4.6: An example of the execution behaviours of independent PSCs in Legacy (A), Simple (B) and Simple Periodic (C) service types on three different computers when they are preempted at $t = 2$ and resumed at $t = 3$.

4.2.4 Service Behaviours

Previous sections explained the service types defined in the architecture and how tasks behave based on their types. This section will give a simple example of how services define PSC behaviour based on their types.

Table 4.1: A list of example PSCs with different service types.

PSC	Service Type	WCET	Period	Relative Deadline
A	Legacy	3	N/A	6
B	Simple	3	N/A	6
C	Simple Periodic	3	6	6

Table 4.1 lists three PSC examples, each defined with a different type of service. Assume that *A*, *B*, and *C* are independent PSCs having the same worst-case execution times (WCET) and running alone on different computers. As described, Legacy and Simple services do not have periods and their inter-arrival times are not known a priori. Then, if they are preempted at time $t = 2$ and resumed at $t = 3$, they are expected to run as illustrated in Fig. 4.6. As seen from the figure, when preempted, the previous runtime of A in Legacy service type is lost, causing it to run another three units until its completion at $t = 6$. B in Simple type remembered the runtime, causing it to run only one more unit until completion at $t = 4$. Similarly, C in Simple Periodic completed execution at $t = 4$ and its second period started at $t = 6$, idling the CPU for two units of time.

To simplify the complexity of the problem, throughout the thesis, PSCs defined by the services are assumed to be independent of each other, and they do not allow preemption in the critical

section where other waiting tasks may attempt access. As it may also be noticed, *sporadic* tasks are not mentioned in the service types. The thesis considers sporadic tasks as aperiodic tasks. Each new task instance is assumed as a different task, and each task is taken into consideration during scheduling. There is no difference between creating a new task at arbitrary times or limiting a minimum task arrival time from the scheduler perspective.

While creating services, it is crucial to know the PSC characteristics that are to be linked with. For example, the type, duration, and deadline of the PSC must be validated before creating its service. These characteristics are set during service creation phase using parameters. The required parameters are explained in the next section.

4.2.5 Service Parameters

Services define how a PSC should behave. In addition to the PSC characteristics, services also define some parameters that are used for decisions as well as to improve the efficiency of these decisions. Some of these parameters can be changed during service design, or even during runtime. Others that are related to the PSC characteristics cannot be changed. In Sec. 3.2, the definition of a service (denoted as S) was depicted in Eq. 3.31. This section will recapitulate these definitions and include other parameters that are required in the proposed architecture, to solve decision and offloading problems.

The *command* is one of the parameters to define in services, to specify which program, software, or command to link/run, when a task instance is requested. This parameter can only be set during the creation of the service. The command can also specify command line parameters for the PSCs. This enables changing PSC behaviour by directly calling another service that is linked with the same PSC. For example, an argument in the command line can change whether the PSC is single-threaded or multi-threaded.

The service parameters that can be manipulated by the operators or the architecture itself during runtime are listed below:

- *Name*: This parameter is the identifier of a service. It is distributed all over the Edge Network with its location(s) to inform all Edge Servers. Each Edge Server can contain only one service with the same name, but other servers can use the same name for the same or different PSCs. Requesting tasks require this name as an argument. As long as the name is the same, different behaviours can be specified on different Edge Servers. If the name is different on other servers, it will not be found for offloading.
- *Publicity*: This value enables or disables the public use of the service. If a service is public, it is broadcast on the network and is available for use by all Edge Servers and End Devices. If private, it is available for use only by the directly connected End Devices. By default, all created services are set as public.

- *CPU utilization*: It defines how much CPU time in percentage should be given to this PSC when it is being executed. It must be greater than zero and less than or equal to 100. CPU utilization and execution time are inversely proportional. Reducing the utilization by a factor increases the execution time in the same ratio.
- *CPU mask*: This parameter defines which CPUs are allowed to be used by this PSC. It can be a bitmask or list of CPUs. If the PSC is multi-threaded, each allowed CPU executes the same number of threads.

PSC-related characteristics that cannot be changed are usually known in advance by performing in-depth analysis. Their service parameter equivalents are shown below:

- *Type*: It defines the type of PSC; hence, the service, as described in Sec. 4.2. Service type determines which scheduling algorithm is used when needed. Type can be one of Legacy, Simple, or Simple Periodic.
- *Direction*: It is used to determine whether the PSC sends a response after the execution is completed. The value is also used to calculate the delay to deliver the request and response. The service is either one or bi-directional.
- *WCET*: Worst-case execution time (WCET) is used to determine the longest execution duration of a PSC. The algorithms in this thesis use millions of instructions (MI) to calculate this value. For example, if an Edge Server's millions of instructions per second (MIPS) value is 1000, a PSC with the WCET of 10000 will be completed in 10 seconds. MIPS is a widely-known parameter to measure processor speed.
- *Relative deadline*: It defines the deadline of a task, relative to the arrival time. Arrival time is added to this value to determine the absolute deadline. Absolute deadline is the latest possible completion time that a PSC can run without missing its deadline. The thesis focuses only on execution of real-time tasks. However, keeping this value as high as possible enables execution of non-real-time tasks as well. In the architecture, this value is written in terms of seconds.
- *Memory*: The maximum amount of memory usage of this PSC when it is run. As mentioned in Sec. 3.2, this parameter is neglected due to the difficulty of its estimation during runtime and to reduce the complexity of the problem. However, it is considered as a parameter to be used in future work.
- *Thread per core*: This parameter defines how many threads for this PSC are going to run at each core of the Edge Server. Total thread count can be found by multiplying the core count of the server and this parameter value. Assuming that the CPU utilization is 100%, there are 2 allowed CPUs, and thread per core value is 2. CPU utilization for each core will be 100%, allocating 50% utilization for each thread in each core. However, the WCET will not be affected.

Thread per core parameter may seem to hinder wrapping a single-threaded legacy PSC. However, setting this parameter to 1 (one) and also limiting the CPU affinity to only one CPU defines a single-threaded PSC.

4.3 Decision Making

After a secure communication is established, End Devices are allowed to request tasks. An End Device requests a task by sending a request to one of its connected Edge Servers. However, that server may not have enough resources, power, or bandwidth to complete the request. The final server to execute this task is determined by the Edge Server that initially receives the request. If the initial server is not likely to execute this task on time, the task is forwarded to another server in the Edge Network. Forwarding a task request to another server is called *offloading*. Offloading arises due to limited resource availability, power limitation, mobility of the End Device, or network limitation [Sa96]. Decision mechanisms play a vital role during offloading for optimal performance.

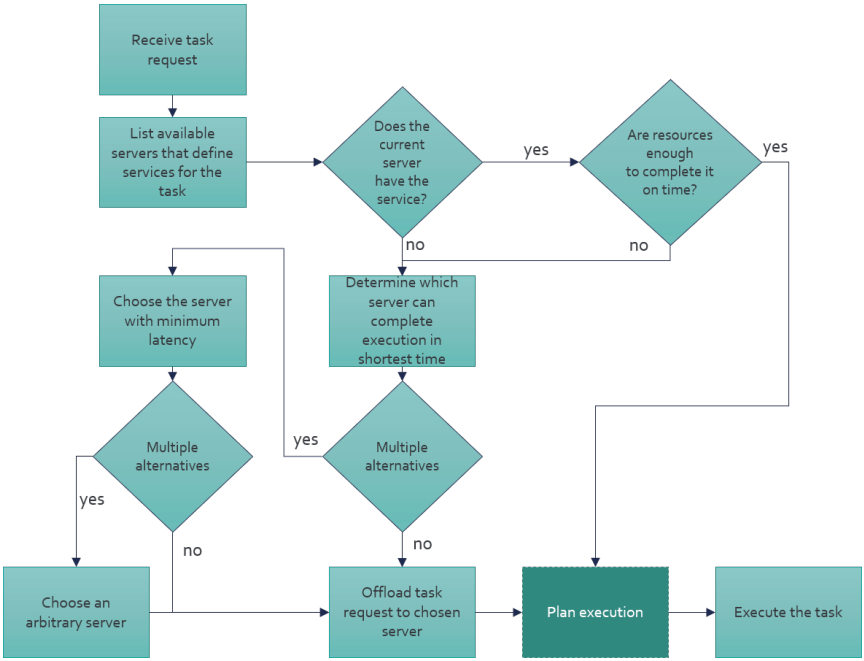

Figure 4.7: The flowchart of the decision mechanisms for selection of an optimal server to offload the task. *Plan execution* has its own flowchart depicted in Fig. 4.8.

The architecture suggests and follows the following steps in order to choose the most convenient server for task execution. If a step cannot determine a single server, the next step evaluates the situation. The steps are also shown as a flowchart in Fig. 4.7.

1. Since each Edge Server has a list of the available resources and services together with their locations, first, the current server queries the possible locations for the requested task, including itself.
2. If the current server does not contain the service for this task or does not have enough resources, then, an alternative server in the Edge Network is looked up.
3. In case multiple alternatives can execute the task on time, the server which can execute the task in the shortest time is chosen (the most idle server).
4. If the resources are the same, then, the server with the smallest delay is chosen.
5. If delays are also the same, then, one of the servers is chosen randomly.
6. If no server found that can execute the task, then the current server tries to plan the execution, following "no" in multiple alternatives branch. In this case, the chosen server will be itself.

Step (1) lists the possible servers for the requested task without any further actions. Step (2) checks whether this task can be executed on time using the available servers. This check uses the latest information about the resource availability such as CPU usage and the server specifications. If there are multiple alternatives whose resources are enough, steps (3), (4), and (5) decide which server to be used for offloading. Once the task request is offloaded, if the chosen server is busy with other tasks, an execution plan shown in Fig. 4.8 is devised. The possible plans are to scale down the CPU utilization of the newly requested task, already running tasks, or all tasks; or to schedule the running tasks in the server. WCET (denoted as x) is in MI unit and the maximum execution speed of the Edge Server j (denoted as p_{E_j}) is in MIPS unit. However, relative deadlines (denoted as d) are in seconds. For a basic comparison, it is necessary to equal both sides of the inequality. Since a relative deadline must be greater than or equal to the execution duration, $\frac{x}{p_{E_j}} \leq d$ can be used for comparison. x, d, p_{E_j} and execution utilization percentages denoted as u (where $0\% < u \leq 100\%$) of tasks are set during service definition and known a priori. If the new execution capacity (CPU utilization) of a task i is u'_i, where $0\% < u'_i \leq 100\%$, the new WCET x'_i can be calculated as in Eq. 4.1.

$$x'_i = \frac{x_i u_i}{u'_i} \tag{4.1}$$

With the new x'_i value, if $\frac{x'_i}{p_{E_j}} \leq d_i$ holds, then the new execution capacity does not cause task i to miss its deadline.

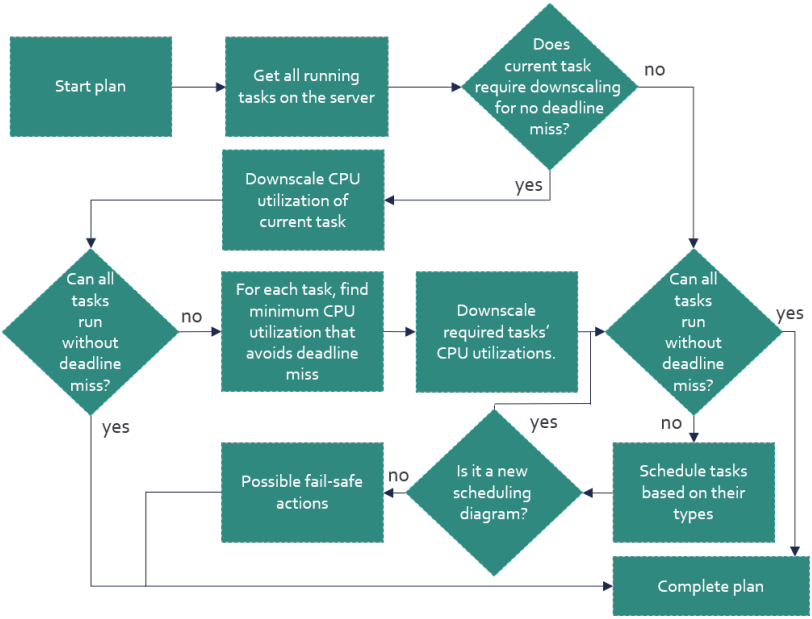

Figure 4.8: The flowchart of planning execution on a server, when a server is not entirely available for task execution.

The worst-case happens when $\frac{x_i'}{p_{E_j}} = d_i$. In this case, the worst-case execution utilization (WCEU) u_i'' can be calculated by replacing x_i' with $p_{E_j} d_i$ and using Eq. 4.1 for each CPU that the task has an affinity to (Sec. 4.2.5).

$$u_i'' = \frac{x_i u_i}{p_{E_j} d_i} \tag{4.2}$$

With the new WCEU(s) calculated using Eq. 4.2, if the sum of utilizations for the running tasks does not exceed 100%, the tasks can continue execution, and the new task can also be executed. If not, the tasks need to be scheduled.

Scheduling the tasks is a fallback situation, in the case that downscaling does not yield a feasible solution. To schedule tasks in Periodic type, the architecture uses the Earliest Deadline First (EDF) scheduling (Sec. 2.2.2). For tasks in Legacy and Simple type, Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling (Sec. 3.1) is used. Although the architecture currently introduces two schedulers, additional schedulers can also be implemented and integrated. They are required to implement two methods to: (1) return whether a task set is schedulable or not and (2) return the sorted task list. This procedure is followed after a server is chosen to complete the request. The execution plan above is also depicted in Fig. 4.8. If the scheduling is also not possible, the task fails to execute. At the moment, no precautions are taken if such a situation occurs. However, as future work, termination of lower priority tasks can be considered.

After the reception of the request, the server also follows the same steps to evaluate the situation, to check whether it is still the best server to continue execution. Prior to execution, each server uses decision mechanism to choose and validate the optimal server. Step (2) uses a satisfactory equation to determine the possible alternative servers that can execute the task. To find the most convenient server E which contains the task j in the network N, when an End Device A requests the task T_i, the satisfaction equation (S) seen in Eq. 4.3 and Eq. 4.4 can be used. The equation calculates the maximum duration to execute a task on a server, including the delay and currently used CPUs. Below, this satisfaction equation is explained. Assume

$$\forall E \in N \Longrightarrow S_j(E_j, T_i) \tag{4.3}$$

where

$$S_j(E_j, T_i) = max\left(\left\{\left(\frac{x_i u_i}{p_j F_{k,j}} + w_i D_{A,j}\right) f_{i,k} : k = 1 \dots c\right\}\right) \tag{4.4}$$

and where $F_{k,j}$ is the free available CPU percentage of core k in server j, $D_{A,j}$ is the minimum delay between the task requester A and Edge Server E_j. $f_{i,k}$ is 1 if T_i has an affinity to core k, 0 if not. w_i is the delay multiplier, and 1 if T_i is one-directional, or 2 if bi-directional. x_i is the WCET of the task in terms of MI, p_j is the total speed of the server j in terms of MIPS, and u_i is the CPU utilization allowed for task i, where $0\% < u_i \leq 100\%$.

Assume that S' is a subset of all S that satisfies $S_j \leq d_i + a_i$ (absolute deadline), where d_i is the relative deadline of the task i in terms of seconds and a_i is the arrival time instance of the request (in seconds) at the first receiving server. Then, the index of $min(S')$ defines which server should be chosen for execution.

If there are more than one indices that provide the same $min(S')$ value, as stated in step (3), Edge Server E_j giving $min(D_{A,j})$ receives the task. Finally, if step (3) also has multiple results, then the server with the smallest delay is chosen (step 4). If step (4) also has multiple alternatives, an arbitrary server is chosen. In any case, the chosen server will plan the execution for the new task; either downscaling or scheduling as defined above. Although not considered in this thesis, it is also possible to implement other decision algorithms. For example, current CPU utilization of the alternative servers can be ignored; only taking the server speed value from the specifications into consideration and choosing the most powerful server. Alternatively, minimum delay first or minimum hop first to reach the target server can be considered. Moreover, if the initial receiving server is planned to be used as a load balancer, others-first can be implemented to prioritize other servers and fallback to the current server in case other servers are not available.

As mentioned at the problem formulation (Sec. 3.2), the calculations are assumed to have no overhead, and the hardware is ideal. The decision using the equation gives a correct result at the time that the request is made. However, in real life, this calculation takes some time, which may use outdated information due to, e.g. a new execution of a task on another server. There are several research activities on offloading [Li15; Yo18], each focusing on possible policies such as power consumption, response time, availability of the server, or computing capability. These mostly focus on reducing the delay using probabilities and average values of the computation timing, which is not applicable in a real-time scenario.

4.4 Summary

This chapter explained how a technology and OS-independent architecture is to be designed to fulfil the requirements listed in Sec. 2.1.2. It described in detail how Edge Computing participants (also called nodes), namely Edge Servers and End Devices should communicate with each other to enable an interoperable, collaborative, and scalable Edge Network (Sec. 4.1). It further elaborated on what a Service or Task (Sec. 4.2) means for the architecture, and how they behave according to their characteristics. Moreover, the chapter explained how the servers are chosen for execution and the decision mechanisms used for offloading (Sec. 4.3). This chapter is a summary of the methods of the critical elements to realize a modular, collaborative, and extensible architecture for Edge Computing, which is the main contribution of the thesis.

5 Implementation and Validation

Previous chapters explained how technology and operating system (OS) independent software reference architecture could be designed for Edge Servers in the Edge Computing domain. The presented architecture enables execution of real-time tasks in a scalable, extensible, and collaborative Edge Network, with minimal effort. The final chapter of this thesis constitutes an implementation of the elaborated concepts from the previous chapters, by instantiating it as a framework. The created framework is later validated with two complex scenarios. This framework is called Real-Time Edge Framework (RTEF) and implemented using the Java programming language.

5.1 Edge Server Components

Edge Servers are physical computers that respond to the requests received from End Devices. They are the vital parts for the architecture to function. Chapter 4 explained how Edge Servers should behave to address these requests in a decentralized, but collaborative Edge Network. To realize the concepts defined in the previous chapter, Edge Servers need to implement several functions. No matter how complex the underlying decisions are, from the End Device perspective, this complexity must be hidden as much as possible. Moreover, Edge Servers in a network are expected to introduce themselves automatically and exchange information in case an offloading is necessary. Automation during communication requires them to implement a standard communication syntax, which needs to be understood by all nodes in the network. All of these concepts are realized in an exemplary framework called *Real-Time Edge Framework (RTEF)*. The RTEF must be deployed and run under a Real-Time Operating System (RTOS), installed on a computer.

The framework groups its functionalities in components. Each component is able to interact with others when needed. Some of them are configurable and can be disabled as per user request. These components are shown in Fig. 5.1 and namely, Configurator, Message Router, Security Protocols, Servers, Resource Monitor, Virtual Processors, Cache, Storage/Database, and Orchestrator along with its sub-components, Queue Manager, Scheduler, and Scaler. The RTEF is implemented using Java programming language. This section will introduce the components that are implemented or adapted/customized for RTEF, in detail.

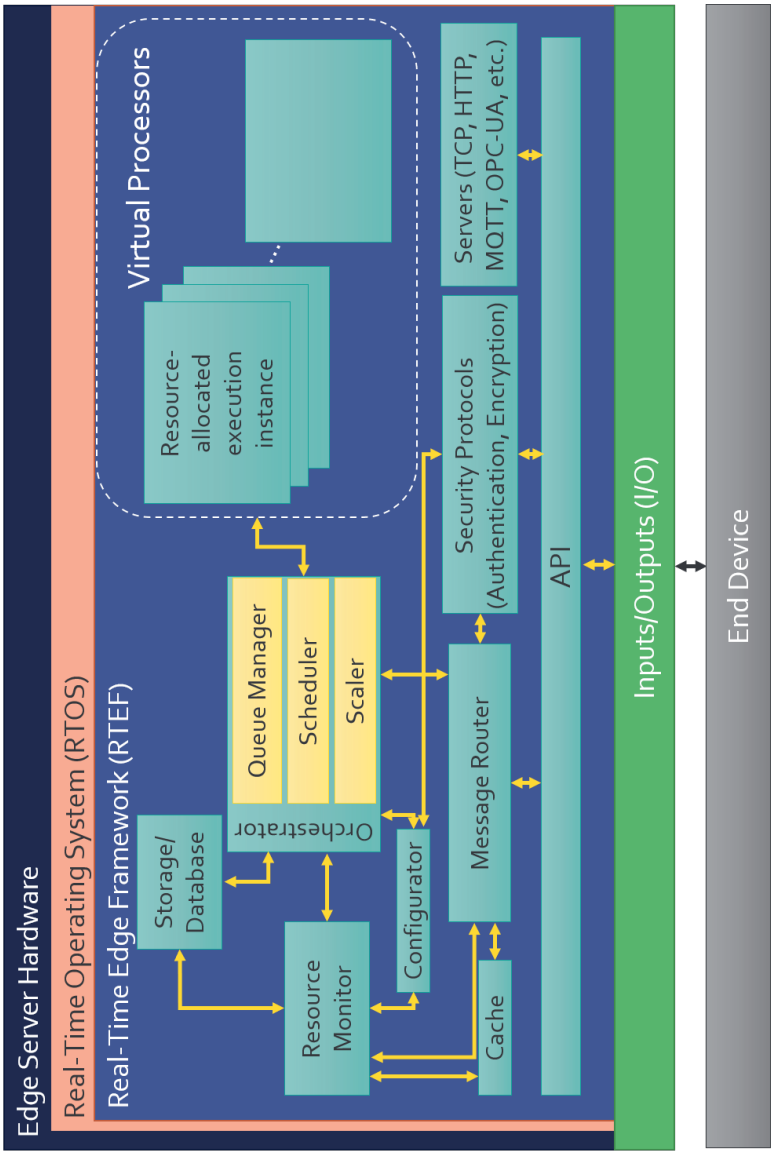

Figure 5.1: Software components of the proposed Edge Server architecture and their simplified communication.

5.1.1 Configurator

The Edge Servers configure themselves automatically as soon as they are started. However, manual configuration and tweaks may be necessary during runtime. This component handles automatic and manual configuration, as well as the detection of other Edge Servers in the Edge Network. Additionally, the Configurator logs server activities and stores them.

An Edge Server running RTEF requires a minimum of four specific configuration keys to function. These keys are reserved and case-sensitive:

- ID: A unique identifier number for the Edge Server. This number is going to be used in the Edge Network for referencing.
- PORT: A port number of the Edge Server, which will listen for connections and commands during TCP socket communication.
- NAME: A user-friendly name to be displayed on the console and logs.
- TOPOLOGYFILE: The full path to a file that is used to store/read the Edge Network topology.

Each Edge Server and End Device in Edge Computing is also called a *node*. TOPOLOGYFILE points to a simple text file that lists available nodes in the Edge Network and how they are connected. This file can be created manually using a plain text file or graphically via the Topology Designer which will be explained in Sec. 5.5. To fully automate the topology generation without human intervention, the file can also be left empty. If no topology file exists in the server or the file specified in TOPOLOGYFILE is empty, it is generated automatically as the connections are established. In the file, a hash "#" character at the beginning of each line is considered as a comment. Information about each node on each line should have the following format:

```
Node, <Unique Node ID>, <Node Name>[, <Free Text>]
```

The `Node` in the first field is a reserved keyword to let Configurator parse this line to define an Edge Server or End Device. Commas separate other inline fields. Field values are received from the reserved configuration keys for the current server. For other nodes in the Edge Network, this information is entered manually or retrieved during the handshake phase, automatically. As connections are established, during handshake phase, each node sends their unique identifier (See Sec. 4.1 and more in Sec. 5.2). If a name is also appended, then this information will be stored into the `Node Name` field of the topology file. During the handshaking phase, node type (either Edge Server or End Device) is also detected. If the name is omitted, it will be automatically generated based on the unique identifier of node and node type.

To specify a connection between nodes for creation of an Edge Network, the following format is used:

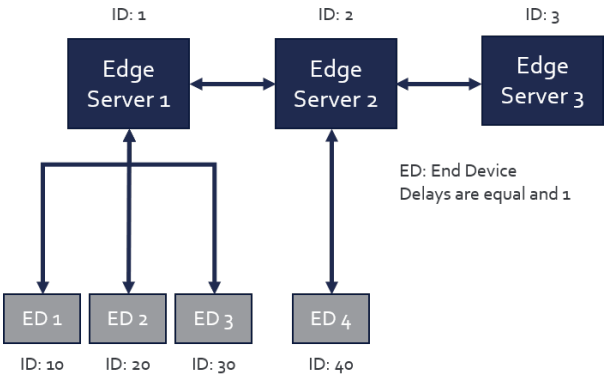

Figure 5.2: An example Edge Network consisting of four End Devices and three Edge Servers.

`Connection, <From Node ID>, <To Node ID>, <Delay>[, <Free Text>]`

Similarly, the `Connection` field is a reserved keyword to inform the Configurator to parse the line as connection information. Likewise, fields are separated by commas. `From Node ID` specifies who initiated the connection, which can be either End Device or Edge Server, `To Node ID` defines the target Edge Server to connect to, and `Delay` is the transmission delay in milliseconds (ms) required to transfer the data.

The topology file will be updated each time a new device added or removed. Similarly, changing connections will also update the file. This file is also used for decision mechanisms while calculating the delays among nodes. If a server is rebooted, all previous connections are lost. To re-establish pre-defined connections stated in TOPOLOGYFILE, Configurator accepts a reserved `AUTOCONNECT` key. This key takes a `true` or `false` value, which is `false` if not defined. If this key is `true` and the pre-defined connections are establishable, the TOPOLOGYFILE is read, and servers are reconnected to each other. To achieve that, the `Node` lines must be slightly modified. `Free Text` is an optional field to be used as an inline comment or note. If this field is replaced with `@IP:PORT`, when a server is booted up, it establishes connections provided that the target Edge Servers are up and the RTEF is running. An Edge Server cannot initiate a connection with an End Device; it should be other way around (See Sec. 4.1 and Sec. 5.2). An End Device should implement its own reconnection mechanisms to Edge Servers. Automatic connection is only possible between Edge Servers; therefore, it does not apply to End Devices.

Fig. 5.2 shows an example Edge Network consisting of four End Devices and three Edge Servers. This setup is also going to be used in Sec. 5.7 for validating the architecture.

Assuming delays between nodes are identical and 1 ms, the topology file to create the network seen in Fig. 5.2 can be defined in a file similar to Source code 5.1. Note that the definition of *Node 3* includes its address in the `Free Text` field (Line 7). If a connection **to** this server is

detected in the file, it will be automatically established, in case AUTOCONNECT key is set to true. This is the case when To Node ID matches with this server ID in any of the Connection strings, similar to Line 20. Swapping server IDs (namely 2 and 3) in that line would not re-establish the connection automatically because no address of the *Edge Server 2* is given.

Source code 5.1: Topology file to define the example Edge Network shown in Fig. 5.2.

```
1  # Defining available Edge Servers
2  ### Node, ID, Server Name, Optional Comment
3
4  Node, 1, Edge Server 1, Server 1
5  Node, 2, Edge Server 2, Server 2
6  #### 192.168.1.3 is an example IP to reach Edge Server 3, and 9093 is the
       ↪ Port that is listened by the RTEF
7  Node, 3, Edge Server 3, @192.168.1.3:9093
8
9  # Defining available End Devices (ED)
10 Node, 10, ED 1, End Device 1
11 Node, 20, ED 2
12 Node, 30, ED 3
13 Node, 40, ED 4
14
15 # Adding connections
16 ### Connection, From Node ID, To Node ID, Delay, Opt. Comment
17
18 ## Edge Server to Edge Server connections
19 Connection, 1, 2, 1.0, My connection comment
20 Connection, 2, 3, 1.0, This connection will be auto-established if
       ↪ AUTOCONNECT is true
21
22 ## End Device to Edge Server connections
23 Connection, 10, 1, 1.0
24 Connection, 20, 1, 1.0
25 Connection, 30, 1, 1.0
26 Connection, 40, 2, 1.0
```

The topology file is read as a whole and then parsed by the Configurator. Therefore, line ordering is not important. Moreover, each connection of pairs needs to be written only once. If AUTOCONNECT is false, values written in From Node ID and To Node ID are also interchangeable. For the example above, node 10 (End Device 1) is given for From Node ID and node 1 (Edge Server 1) for To Node ID (Line 23). However, connections are bi-directional. Therefore, another line containing node 1 as "from" and node 10 as "to" device makes no difference; hence, it can be omitted. However, if AUTOCONNECT is true, servers who are given in To Node

ID field must define their addresses and ports in the appropriate lines. Otherwise, automatic reconnection will not be possible.

5.1.2 TCP Server

Edge Servers need a communication protocol to communicate with End Devices, introduce themselves to other nodes in the Edge Network, exchange resource information, get informed about current tasks, and offload tasks to other servers in the network. Transmission Control Protocol (TCP) is one of the communication protocols that enable connection-oriented delivery, providing error detection and same-order transmission. In RTEF, TCP is chosen as the communication protocol because it empowers a reliable transmission, and it is a widely-used standard (See Sec. 2.1.3).

TCP Server is a software component in RTEF to enable communication with other Edge Servers or End Devices. Inter-communication between nodes is performed via raw TCP messages, without additional overheads. The End Devices must implement their TCP clients to communicate with the Edge Servers, either directly or using adapters/wrappers.

The TCP Server is the door to outside communication. The pre-login handshaking for resource introduction (defined in Sec. 4.1) is the responsibility of the TCP server. After login procedure, all commands received or sent are passed through this component. The TCP Server parses the received commands, translates them to internal API methods, and redirects them to the Message Router. Accepted commands by the server are going to be explained in Sec. 5.3.

This component also keeps the list of all connected End Devices or Edge Servers. Whenever a change in resource usage or services is detected, these changes are delivered to all concerned devices. If a connection drops unexpectedly, it is automatically re-established. If a node is permanently disconnected, it informs the Configurator to update the topology. If this server is rebooted, pre-defined connections are re-established in case AUTOCONNECT is enabled (See Sec. 5.1.1).

5.1.3 Message Router

The Message Router receives the parsed commands from the TCP Server and redirects them to their responsible components or Edge Servers. If the command is a task request from an End Device, it communicates with the *Resource Monitor* and *Orchestrator* to receive the Edge Server identifier that is decided to offload the request and execute the task. If the chosen Edge Server is not the current one, the task request is forwarded to the relevant Edge Server. If there are multiple alternative Edge Servers, the choice is made using the satisfaction equation shown in Sec. 4.3. If there is no direct connection with the target Edge Server, the shortest path to

the target is calculated using Dijkstra's algorithm, and the request is delivered to its destination via intermediate Edge Servers. This component also informs the End Device with the location `EdgeServerID` where the requested task `serviceName` is being executed on and its unique task identifier `UniqueTaskID`. This message uses the following format:

```
RUNNING:serviceName@EdgeServerID,UniqueTaskID
```

After execution is completed, `TASKCOMPLETED` message is also sent back to the End Device. If the task is defined with a bi-directional service, the task results are also appended, separated by commas. Again, if there is no direct link between the Edge Server and End Device, the message follows the same route back. If one or more servers on the route are no more available, it is delivered using an alternative route. Each intermediate server that receives a command or message calculates the shortest distance also using Dijkstra's algorithm considering only currently available Edge Servers.

5.1.4 Security Protocols

Giving external access to any server without any proper authorization is an extremely high risk. The role of the Security Protocols component is to perform authentication and maintain secure communication between the nodes. It introduces methods to add/remove users, set user roles, log a user in and out. After login, instead of carrying the password throughout the active session, it generates and assigns a unique token to the logged user. Whenever a method is to be called, this token is validated against active user token. Then, the user role is checked, and the command is forwarded. The sessions are invalidated if the client disconnects or the connection is broken. However, if `AUTOCONNECT` is enabled, the connection is re-established when available. As defined in Chapter 4, the architecture introduces two user roles. During user creation, either an *operator* or *end-user* role should be chosen along with a unique username. Created users on a server are local and not shared with other Edge Servers in the network. Consequently, each server should define its own users. In the future, user directories or other alternative methods can be used for authentication. Users can change their passwords. A successful change in the password keeps the connection intact but generates a new token.

5.1.5 Resource Monitor

Resource Monitor (RM) keeps track of available resources of the current Edge Server and other Edge Servers within the Edge Network, whether they are directly connected or not. Furthermore, it contains information about all public services in the network and how the Edge Topology is structured. Active tasks, whether they are in the running or paused state, are also tracked by this component and can be accessed via commands. Each change in the resource usage informs the TCP Server component, which later updates other connected Edge Servers. RM pre-

pares execution plans defined in Sec. 4.3 in conjunction with *Orchestrator*. RM performs the simple initial check whether the available resources of the current Edge Server are enough to execute the task until its deadline without any further actions, such as downscaling or scheduling. Two cases allow a direct execution: (1) The requested task is the only task to execute on that Edge Server, and there are no other running tasks on the same server. (2) There exist other running tasks on this server, but as defined in Sec. 4.2.5, the running tasks are defined to be executed on different Central Processing Unit (CPU) to avoid over 100% CPU utilization.

If further actions are required, the RM consults the *Orchestrator* for the last decision.

5.1.6 Orchestrator

RM evaluates the execution possibility based on the server specifications and resource availability. However, Orchestrator performs a more in-depth analysis to check whether this task can be executed on this server, on time. It is the last component that decides whether the task is executed here or not. Orchestrator itself is composed of three sub-components: (1) Scaler, (2) Scheduler, and (3) Queue Manager. Sub-components are called in order to evaluate the feasibility of running all tasks, including the requested one, without causing any deadline miss.

Scaler

Scaler is the first called sub-component of *Orchestrator*. It is called to check if the requested task or/and running tasks can be downscaled and still meet their absolute deadlines. The scaler uses downscaling formulas written as Eq. 4.1 and Eq. 4.2. The formula calculates the minimum execution utilization of a task in percentage, by using its relative deadline and runtime. The Scaler searches for a solution in three iterations. First, the possibility of downscaling of the requested task is evaluated. If an on-time execution is not possible, then, only other running tasks are considered for downscaling, using the same formula. If this iteration also fails to yield a successful execution plan, finally, all tasks, including the requested task, are evaluated for their minimum execution capacity. In case this repetition does not provide an adequate plan as well, the *Scheduler* component is called, to plan the execution order. Scaler upscales the downscaled tasks when possible, back to their original CPU execution capacities, as soon as more CPU is available. It does not, however, upscale them more than their CPU utilization values defined in their services parameters, even if it possible. By default, Scaler is enabled. It can be disabled during the configuration phase of the Edge Server via `setScalerEnabled=false` configuration. Then, the tasks will directly be passed to the Scheduler, without analysing the possibility of downscaling. If Scaler is desired to be disabled only for specific tasks, worst-case execution time (WCET) of such tasks should be set equal to their relative deadlines during service creation.

Scheduler

If scaling does not create a feasible plan to execute all tasks on time, another possibility is to

schedule them. The Scheduler is responsible for creating an internal scheduling plan for the execution of all tasks. This component is called when the sum of Worst-Case Execution Utilizations (WCEUs) of all tasks exceed 100%, and the Edge Server cannot guarantee an on-time execution only by downscaling (Eq. 4.2). The RTEF implements two types of schedulers: EDF scheduler (Sec. 2.2.2) for tasks defined with Periodic services and NAPATA scheduler (Sec. 3.1) for tasks defined with Legacy and Simple services. Running a combination of multiple service types on a single computer requires the implementation of a schedule server (See Sec. 2.2.2). The current framework does not implement one; thus, it is not yet fully supported. Nevertheless, this is proposed as future work. Currently, this can be achieved by limiting their allowed CPUs, setting a different CPU affinity for each type during service creation.

On multiprocessor systems, schedulers can change the CPU that the thread is running on. However, on hardware and OS level, changing CPUs of the processes/threads during their runtime can be costly due to context switching. Increasing the available CPU count for tasks also increases the parallel execution overheads, reducing efficiency [LHK03]. Moreover, finding an optimal scheduling diagram on multiprocessors is NP-hard [LW82]. Lee et al. [LHK03] introduced a scheduling algorithm for multiprocessor systems, which can schedule aperiodic tasks on-line. However, this requires the tasks to be non-preemptible and decomposable into subtasks. Since tasks in this thesis are both preemptible and non-resumable, slightly deviating from the plan in [GUR18], schedulers are prevented from modifying CPU affinities during execution, leaving this decision to service creator during creation time. The schedulers, then, calculate the scheduling possibilities for each core, retrieving the current activity information from the RM.

The architecture currently implements two schedulers, but additional schedulers can also be implemented and integrated. A scheduler is required to provide two methods: (1) A `schedule()` method which requires a list of tasks containing their WCETs and absolute deadlines as input, and returns `true` or `false` depending on the schedulability, (2) a `sort()` method which returns the sorted scheduling diagram of all tasks. The schedulers must be implemented in a way that they consider the CPU affinities of the tasks as well.

If tasks, including the newly requested task, can be scheduled, the sorted task list (scheduling diagram) will be passed to the *Queue Manager* and the task at the beginning of the list will be executed. Otherwise, the task *request* is immediately forwarded to an alternative Edge Server that contains the same service name, via *Message Router*. This alternative server, then, activates its Orchestrator, to repeat downscaling and scheduling procedures. The server selection mechanisms used by Message Router are elaborated in Sec. 4.3. If no alternative is found, the task cannot be executed. Nevertheless, this can be another topic to cover in future work. An algorithm can be implemented to determine what to do in such circumstances. Depending on the deadline, runtime, or remaining time of existing tasks, one or more of them can be terminated to allow execution of a more urgent task.

Queue Manager

Instead of storing the sorted list of tasks in the schedulers, a separate component, called Queue Manager, is created to enable access to the list by all schedulers as well as the Resource Monitor.

Schedulers pick the first task from the list and execute it. Tasks are executed only if they are on that list, and it is their turn to execute. Periodic tasks are automatically added to the list again when their next period starts. Tasks in Legacy and Simple types are removed from the list after they complete execution.

5.1.7 Virtual Processors

Services define how a program/software/command (PSC) behaves. These behaviours are set by the service parameters explained in Sec. 4.2.5. One of the parameters is the CPU mask, which defines the list of usable CPUs by the PSC. Similar to CPU set definition in UNIX-based systems [Li21b], in RTEF, this value is a bitmask between 1 and $2^n - 1$, where n is the CPU count of an Edge Server. The locations of ones after conversion of this value to the base-2 system specifies which CPUs are allowed to execute the task. For example, in a server with four (4) CPUs, the CPU mask can be between 1 and 15 ($2^4 - 1$). If a value of 14 is written, converting this number into the base-2 system gives $(1110)_2$. If CPU indexing starts with zero (0), the task and its threads can use CPU 1, 2, and 3 for the execution, but not CPU 0. The scheduling algorithms calculate the feasibilities considering these CPU masks.

Virtual Processors (VPs) work in a similar fashion to control groups (`cgroups`) in Linux as briefly explained in Sec. 2.2.2. `cgroups` organize a set of processes into hierarchical groups to limit their resource usage and monitor them. Likewise, VPs assign tasks to the CPUs and limit their execution capacities. A new VP can simply be added using *Configurator*. A *name* to call later, a *period* value to set a reference CPU unit of time, a *runtime* value to determine maximum allowed duration within each *period*, and a *CPU mask* in the format explained above are provided as input. After each *period*, the *runtime* resets. Result of $runtime/period$ determines the maximum allowed CPU load.

Multiple VPs can reuse the same CPU mask(s). This can be useful to allow tasks to use the same CPU(s) with different loads. For instance, assuming two tasks run using the same CPU, in five CPU units of time (period), one task can be given two, and the other task can be given three CPU time (runtime) for execution. The former task will then have 0.4 ($= \frac{2}{5}$) as load and the latter 0.6 ($= \frac{3}{5}$). In terms of CPU utilization, these will be equivalent to 40% and 60%, respectively.

Runtime and period values in RTEF are used to calculate the CPU utilization. Therefore, unlike stated in [Li21a], the runtime value in RTEF cannot exceed the period value, meaning $runtime/period$ cannot be greater than 1. A task can only be executed up at full load (1.0), or 100% CPU utilization. VPs can also be used to separate Periodic services from Legacy and Simple services from using the same CPU, as running a combination of all types is not entirely yet

supported. Tasks can be assigned to the VPs using a command after they start running or during service creation. This command requires the running task identifier stored in the RM, followed by the VP name that is given during creation. `DEFAULT` is a reserved VP name that removes the limitations, giving 100% execution capacity and access to all CPUs. To restore tasks to their original CPU(s) and CPU utilization, the VP assignment can also be removed. In this case, only the task identifier is given with the command.

As expected, a task can be assigned to only one VP at a time. Assigning another VP will invalidate the previous assignment, and the last given VP will be used instead. VPs assign the threads of a PSC by sorting them by their creation time. A specific choice for thread assignment is not possible at the moment.

5.1.8 Other Components

There are also other components such as *Cache* or *Storage/Database*. These components do not require anything specific to function and are indirectly used as they are. Moreover, the complexity of the problem is reduced by assuming the storage capacity in the Edge Servers is unlimited (See Sec. 3.2). Topology created by the Configurator is stored as a file. However, as an alternative, a database can also be used.

5.2 Communication

For inter-communication between nodes (End Devices or Edge Servers), first, they need to be connected. TCP Server component enables this communication via TCP messages. Communication between nodes is performed using raw TCP socket messages.

Each communication between nodes starts with handshaking. The handshaking between an End Device and Edge Server communication had been depicted in Fig. 4.3 and the one between two Edge Servers in Fig. 4.4. Following the concept in the reference architecture (See Sec. 4.1), Fig. 5.3 further explains how it is realized in RTEF.

During communication, the connection initiator (the ”from” device) is considered as a client. When a connection between two Edge Servers is established, the server that initiates the connection also acts as a client. Consequently, the initial communication sequence (or handshaking) is valid for both connection types: between End Device and Edge Server or two Edge Servers. As seen in Fig. 5.3, after a connection is established, the client first sends a plain `HELLO` message to an Edge Server to initiate the pre-login process. Then, the Edge Server responds with an `OLLEH:` appending its unique identifier (ID) and name, separating them using a colon ”:”. These values are used during the generation of the topology file and for further referencing. Next, the client responds to this message by sending its ID starting with ”`ID:`”, and

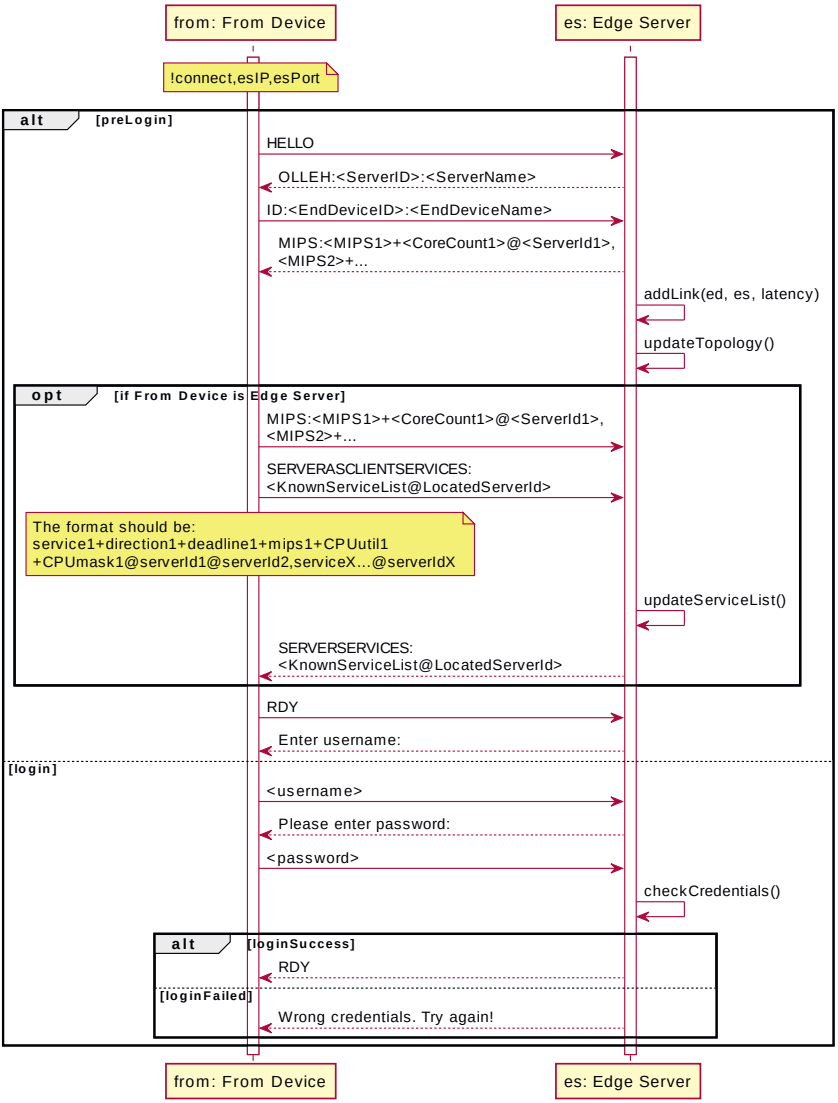

Figure 5.3: Initial communication sequence diagram between an End Device/Edge Server (as From Device) and Edge Server.

appending its name. The Edge Server validates whether other nodes in the Edge Network use this ID. Following, the connected Edge Server sends specifications of all known Edge Servers in the network. In RTEF, speed of servers is measured in terms of Millions of Instructions Per Second (MIPS). The value list starts with "MIPS:", followed by the MIPS value of one server, a plus "+" sign to separate the field and the core count of that server. To specify the server that these specifications belong to, an "@" character is appended, and the server ID is written. Finally, a comma "," is used to separate each server. If the communication is between two Edge Servers, the initiator also responds with all known specifications back to the connected server, using the same format. The whole line sent is similar to below:

```
MIPS1Value+CoreCount1@ServerID1,MIPS2Value+CoreCount2@ServerID2,...
```

It is essential that all known specifications are exchanged, instead of only the current server. With this information exchange, all Edge Servers become aware of available Edge Servers and their specifications, even if they are not directly connected. After exchanging the specifications and topology update, if the client is an Edge Server, it sends all known public services starting with "SERVERASCLIENTSERVICES:", followed by the service name, separating each mandatory service parameter by "+" and appending "@" characters to specify each located server ID. Mandatory service parameters to be used for decisions are whether the service is one or bi-directional, its deadline, WCET value and the maximum allowed CPU utilization. A comma "," also separates each public service. Parameter exchange is also necessary, as each Edge Server can define service behaviour differently. Concatenated information is sent in a single message and shown below:

```
SERVERASCLIENTSERVICES:serviceName1+direction1+deadline1+wcet1
  +CPUutil1+CPUmask1@serverId1@serverIdX,serviceNameX...
```

This line shows that `serviceName1` is available both on `serverId1` and `serverIdX`. The Edge Server receiving the services updates its service list along with their locations, broadcasts this information to all other connected servers, and responds to the client with its public services, using "SERVERSERVICES:" message, and following the same format. If the communication is between an End Device and an Edge Server, then, the service exchange is skipped, and the End Device completes the pre-login process by sending `RDY` to the Edge Server.

After the pre-login process is completed, the Edge Server asks for username and password to authorize the client to allow further commands. The client first sends its username, then sends the password associated with this username. If the authentication is successful, the server sends `RDY` message, enabling access to commands. Handshake messages with real information while setting up the validation environment is shown in Fig. 5.11.

The client initiates disconnection by sending `QUIT` message to the server. The server responds with `QUIT` and closes the connection. All transmitted messages are case-sensitive.

5.3 Standard Commands

One of the objectives of the RTEF is to abstract low-level operations without limiting the functionality. As stated in Sec. 2.1.3, Application Programming Interfaces (APIs) increase backward compatibilities and provide standardized commands. These commands prevent incompatibilities with components when their behaviours or functionalities change.

The RTEF introduces several commands to be used both by the End Devices and the Edge Servers that participate in the Edge Network. The framework introduces two types of commands: (1) client-side commands and (2) remote commands. Client-side commands can be executed on the server locally, whether the connection is established or not. However, to execute remote commands, a connection with another Edge Server must be established. The client-side commands are entered using an interactive console provided by the framework when run. They start with an exclamation mark "`!`" and are followed by the command. Remote commands can be entered on the same interactive console on the servers after a connection is established or using a remote terminal. One Edge Server can be connected to multiple Edge Servers. Remote commands, therefore, start with the remote server ID, then the command, and followed by the arguments. In both command types, each argument of the command is separated by commas "`,`". The client-side commands and generic remote command syntax are summarized in Table 5.1.

Table 5.1: List of client-side commands that can be executed on the local server, with or without a connection to another server.

Command	Description	Example
`!help`	Prints available commands.	`!help`
`!bye`	Stops getting input from user.	`!bye`
`!connect,IP,PORT`	Connects to the Edge Server at `IP:PORT`.	`!connect,192.168.1.27,9090`
`!hosts`	Prints the ID(s) of connected server(s).	`!hosts`
`!id`	Prints the ID of the client.	`!id`
`!disconnect,ID`	Disconnects from the server `ID`.	`!disconnect,1`
`ID,COMMAND`	Executes a remote `COMMAND` on server `ID`.	`1,getServices`

Table 5.2 lists the remote commands that can be executed on an Edge Server through a client. The commands with an asterisk (*) can only be executed if the logged user is an *operator*. Similar to handshaking messages, all commands are also case-sensitive.

Table 5.2: Remote commands that can be executed on an Edge Server through an End Device or End Device. Commands with an asterisk (*) can only be executed if the user is an Operator.

Server Commands	**Description**	**Example**
*addLink,srcId,destId,delay	Manually connects two Edge Network participants.	addLink,1,2,1.2
checkTopology	Check if topology file is valid and the connections are established.	checkTopology
*connectTo,hostIp,hostPort	Connects an Edge Server with another Edge Server.	connectTo,192.168.1.22,9090
getLatency,srcId,destId	Prints the delay between servers srcId and destId.	getLatency,1,2
getServerId	Prints the current server ID.	getServerId
*kickClient,connectionId	Kicks a connected client with its connectionId.	kickClient,0
*printExecutionGraph	Prints the CPU execution activity graph until this point.	printExecutionGraph
*setLogLevel,SEVERE|WARNING| INFO|CONFIG|FINE|FINER|FINEST	Sets the logging detail level.	setLogLevel,WARNING
Service Related Commands	**Description**	**Example**
*addService,LEGACY|LPERIODIC|SIMPLE|SPERIODIC ,ONE|TWO,mips, deadline,memory, threadPerCore,command,cpuUtil,cpuMask,name	Adds a new service to the current Edge Server with the given parameters. Initially, newly added service is not publicly available in the network. See makeServicePublic.	addService, SIMPLE, ONE, 1, 10000, 30000, 102400, 1, ls, 100, 3, Test
*broadcastServices	Forces broadcasting all services in the Edge Network.	broadcastServices
getAllServices	See getServices.	
getAvailableServices	See getServices.	
getNetworkServices	Prints the available public services within the Edge Network together with their location.	getNetworkServices
getPublicServices	Prints the public services hosted in the current Edge Server.	getPublicServices
getServices	Prints the hosted services in the current Edge Server.	getServices
*makeServicePublic,serviceName,true|false	Makes this service publicly usable in the Edge Network if true or hides from the public usage if false.	makeServicePublic,Test,true
*removeService,serviceName	Removes a service from the current Edge Server. If it is public, also removes from the Edge Network.	removeService,Test
whereIsServiceLocated,serviceName	Prints a list of Edge Servers that hosts the service.	whereIsServiceLocated,Test
Task Commands	**Description**	**Example**
getRunningTasks	Prints the running tasks in the current Edge Server.	getRunningTasks
*killAllTasks	Kills all running tasks in the current Edge Server.	killAllTasks
*killTask,taskId[@remoteServerId]	Kill the task in the current Edge Server with the given taskId. Entering a remoteServerId kills the task on that server.	killTask,1@2
requestTask,serviceName	Requests task execution from the current Edge Server.	requestTask,Test
*setVPForTask,taskId,VPName	Assigns the task with ID taskId to Virtual Processor set VPName	setVPForTask,1,Set1

5.4 Requesting Tasks

One of the vital roles of an Edge Server is to execute a requested task on time. This execution should be performed on the current server, or the request should be offloaded to an alternative server. As stated in Sec. 4.1, execution moment and location is decided by the Edge Servers. Hence, the tasks and therefore user PSCs cannot be started by the End Devices directly, but only their executions can be *requested*. The End Device only sends `requestTask` command to one of the Edge Servers that it is connected to and appends the service name that this task is defined as. Then, the RTEF supervises the complete execution.

Starting a task follows a specific path within the Edge Server after the `requestTask` remote command is received. First, the Security Protocols component validates the authenticity of the incoming command. Next, the Message Router asks RM if the requested service is available on the current server. If not found, it locates the service within the Edge Network, through the RM and steps described in Sec. 4.3 are followed. If there are multiple alternatives, the request is forwarded to the Edge Server, which can execute the task on time. If there are multiple alternatives which contain this service, then the satisfaction equation (Eq. 4.4) is used to determine the optimal server. If the satisfaction equation yields the same results for more than one server, the one with minimum delay is chosen. If the delays are also the same, then the request is offloaded to an arbitrarily-chosen server. If there are no services available for the requested task, then the task fails to execute.

Offloading means asking for task execution from another server by forwarding the task request. An Edge Server offloads a request by using the same remote command as the End Device, `requestTask`. This means that an Edge Server can also be used as a client to execute remote commands on another server.

In the case that the service name is found in the current Edge Server, first, the availability of the resources is evaluated (See Sec. 4.3). If the task can be executed with the defined service parameters on time, it is directly executed. If the server is partially busy or the task can only be executed after planning, first, the Scaler component analyses whether the new, existing, or all tasks can be downscaled (See Sec. 5.1.6). Downscaling reduces the CPU usage by limiting a task's utilization factor and increases the runtime of the task, but no longer than its deadline. If all tasks can be downscaled without causing any deadlines misses, then, this step is taken. Otherwise, the Scheduler component goes active. Depending on the Service type of the task, Scheduler uses the Earliest Deadline First (EDF) or Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling algorithm to schedule the running tasks considering their deadlines. If the tasks are schedulable according to Eq. 3.39 (for EDF), or Eq. 3.12 (for NAPATA), then, the Queue Manager stores the execution order. Once the execution of a high priority task is completed, Queue Manager notifies the next task to be executed. If a task is to be preempted, the Scheduler monitors and acts as a supervisor.

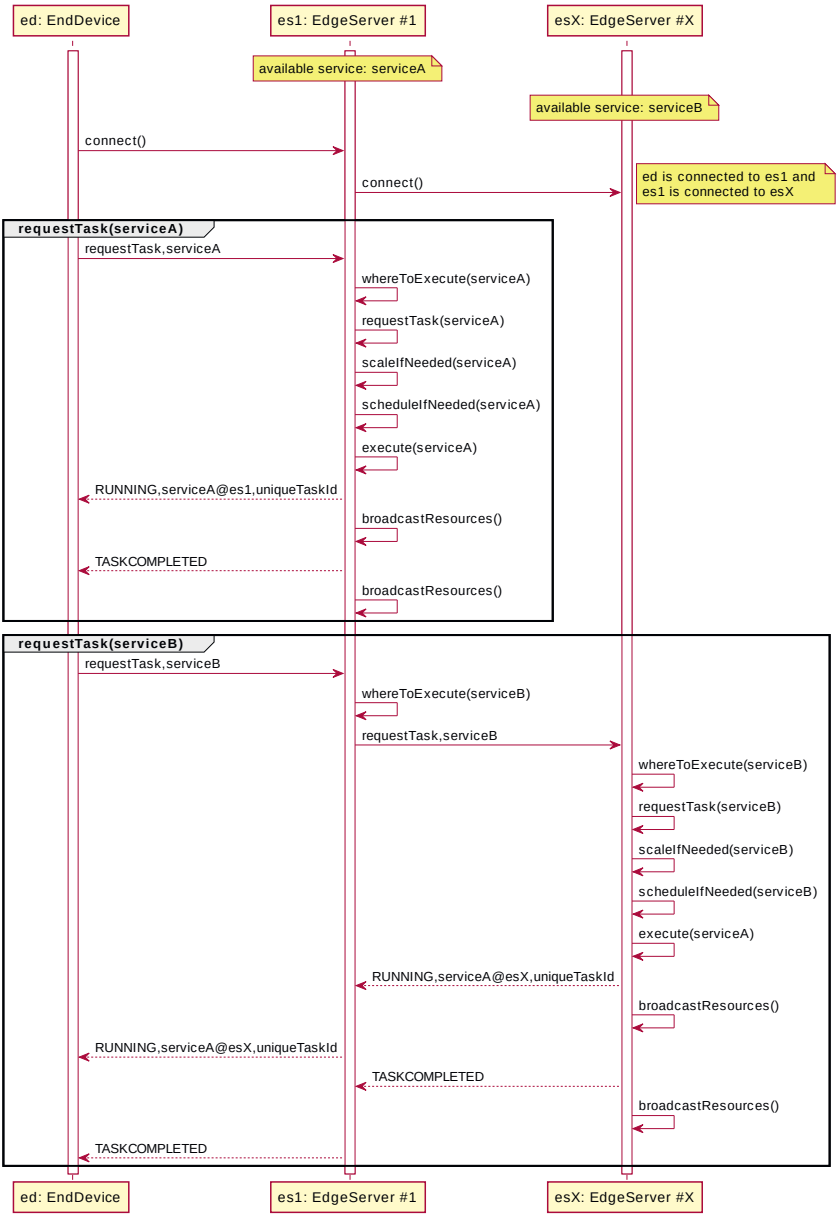

Figure 5.4: Simplified sequence diagram to show the process before and after starting a task, and after the task executes its completion.

Whenever a task is executed, the chosen server directly calls the PSC that the service is linked with. Then, the server announces its server ID and task ID to the task initiator (See Sec. 5.1.3). It also delivers its resource usage information to its neighbours. They are broadcast among all available Edge Servers in the network as soon as a change is detected. When the task execution is complete, the resource usage is broadcast again to keep all servers up-to-date. At the moment, only the CPU utilizations are broadcast as the memory is assumed to be unlimited to reduce problem complexity (See Sec. 3.2). At each task request, only the latest values are used for computations. CPU utilizations are broadcast via messages using the following format:

```
CPULOAD:CPU0UsePercentage,CPU1UsePercentage,...@EdgeServerID1,
 CPU0UsePercentage@EdgeServerIDX
```

At any time, the CPU activity diagram of an Edge Server can be written into a file using the remote command `printExecutionActivity` (See Table 5.2). This command is also useful to check if the CPUs are fully utilized or to change service parameters for better optimization. If the user is an operator, this command creates two different files in the working directory. The file names are generated following `YYYYMMDDhhmmss` format defined by W3C [WW97], where `YYYY` is four-digit year, `MM` is two-digit month (01 through 12), `DD` is two-digit day of month (01 through 31), `hh` is two digits of hour (00 through 23), `mm` is two digits of minute (00 through 59), and `ss` is two digits of second (00 through 59). If the command is executed on "29.12.2020 13:30:58", then the filenames will be "20201229133058". One file is with DAT extension, and it lists the task execution and CPU load information as a readable text. This file has four columns, and each one is separated with space. The columns are:

1. Relevant CPU ID: Zero-based CPU index on which the activity is detected.
2. The relative timestamp for this task activity: The time passed since the previous change in CPU load.
3. CPU load: The current load of the current CPU at the timestamp written in (2). A value between 0 and 1 (1 meaning 100% utilization).
4. The task ID: ID of the task that caused the CPU activity.

For each data, a separate line is used. An example data file generated by the command is shown in Source code 5.2.

Source code 5.2: An example data file generated automatically using the `printExecutionGraph` command.

```
# 20201229133058.dat
0 0 1 0
1 0 1 0
0 1 0.5 0
1 1 0.5 0
0 1 0.5 1
1 1 0.5 1
0 20000 0 0
1 20000 0 0
0 40000 0 1
1 40000 0 1
```

The second file created is a script file to convert this data into an interactive graph and has DEM extension. This file can be used with *gnuplot*[1], an open-source graphing utility, to display an interactive visual graph. The script file created automatically to parse and visualise the example data in Source code 5.2 file is shown in Source code 5.3.

Source code 5.3: An automatically created script file to parse Source code 5.2 and display an interactive visual graph.

```
# 20201229133058.dem
set datafile missing NaN
set offset graph 0.05, graph 0.05
set ylabel "CPU Load"
set xlabel "Time"
set key outside right
set grid y x
set xtics rotate
set style data linespoints
set autoscale noextend
set border back
set term qt
#set term png
#set output "executiongraph.png"
cpus = "0 1"
i = 0
set multiplot layout 2,1
do for [i in cpus] {
set title sprintf("CPU %s", i)
plot \
```

[1] Website: `http://gnuplot.info`

```
'20201229133058.dat' using (column(1) == i && column(4) == 0 ? $2 : NaN) :
    ↪ 3 with steps lw 2 title "Task 0", \
'20201229133058.dat' using (column(1) == i && column(4) == 2 ? $2 : NaN) :
    ↪ 3 with steps lw 2 title "Task 1", \
}
unset multiplot
pause -1 "Hit return to continue"
```

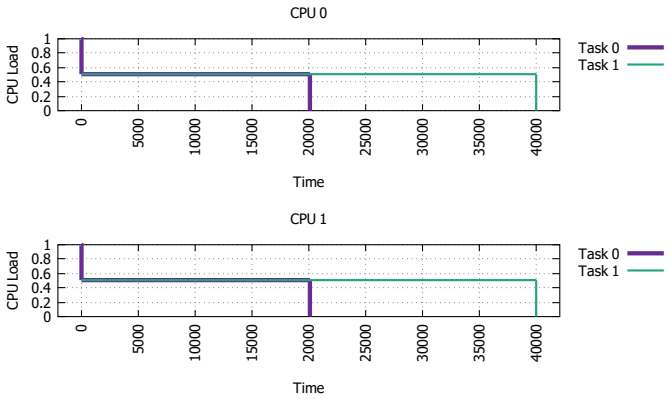

Figure 5.5: An example CPU activity diagram created using gnuplot. The required files can be created using `printExecutionGraph` remote command.

The activity diagram based on the data given in Source code 5.2 and parsed using Source code 5.3 is shown in Fig 5.5. It can be seen that *Task 1* is executed right after *Task 0* has started and *Task 0* CPU utilization is reduced to 50% due to downscaling. Then, both of the tasks run at 50% utilization until their executions are complete. It can also be seen that *Task 0* completes execution at timestamp 20000, and *Task 1* at timestamp 40000. From the figure, it is also clear that the tasks are multi-threaded, as both of them are seen in both CPUs. Finally, it can also be understood that the CPU utilization parameter for *Task 1* is set to 50% during service creation, as no upscaling after *Task 0* completes execution is seen. Otherwise, Scaler would upscale the utilization back to its original value.

It should be noted that the diagram of the CPU activity is reported only for the server that receives the command. If the tasks are executed on another server in the network, the `printExecutionGraph` command must be executed on that server instead.

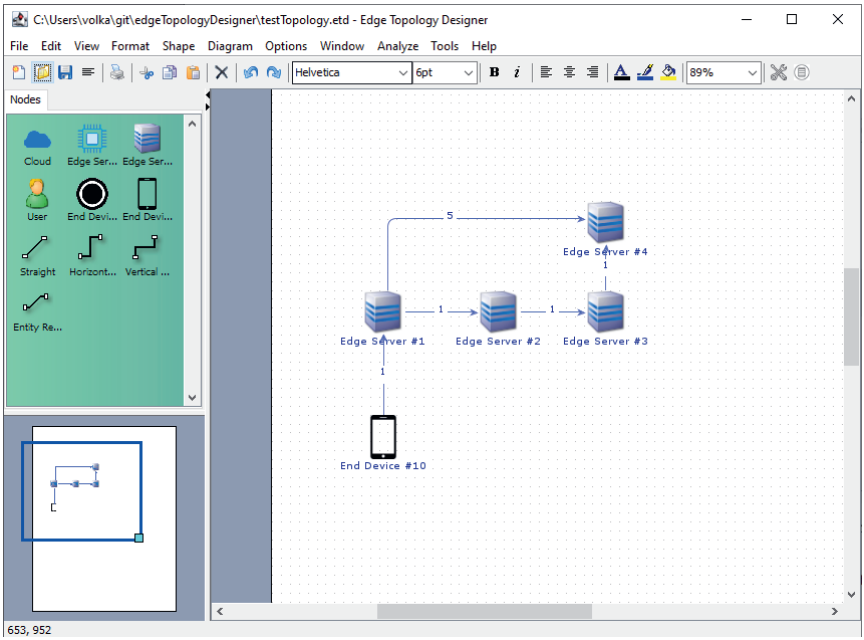

Figure 5.6: The user interface of Topology Designer used to create RTEF compatible topology file.

5.5 Topology Designer

As explained in Sec. 5.1.1, an Edge topology can be created manually by creating a plain text file and specifying the nodes, together with their connections. Alternatively, it is also possible to let topology be created automatically, as the connections between the nodes are established. A third way to create a topology compatible with the architecture is using the Edge Topology Designer, developed along with the thesis. It is based on jGraphX library[2] and was implemented in Java programming language. Its graphical interface can be seen in Fig. 5.6.

Using the designer is not different from using a drawing tool. Dragging the components from the list and dropping them into the canvas automatically assigns unique node IDs to them. These IDs can be changed by right-clicking on the desired node or using F3 from the keyboard. It is also possible to rename the nodes by double-clicking on their names, via the context menu or using F2 shortcut from the keyboard. Holding the left mouse button on a node and dragging it starts a link. Releasing the mouse button on another node creates a link between two nodes and a dialogue window to set the delay is shown. Similar to names, delays can also be modified

[2]Website: `https://github.com/jgraph/jgraphx`

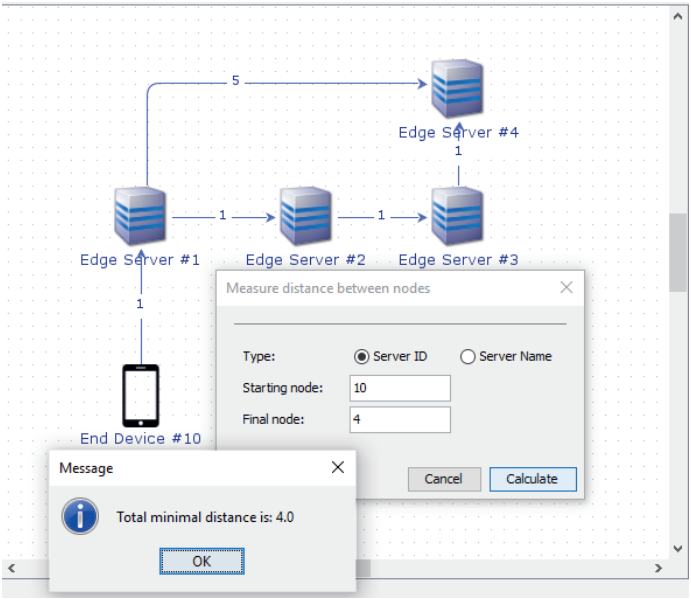

Figure 5.7: Result of minimum delay between End Device #10 and Edge Server #4 seen in the topology.

by double-clicking on them. If the mouse is released on a blank space, it creates another node of the same type with a unique name. Lastly, the `Free Text` field in the topology file can be set by right-clicking or choosing a node and pressing F4. This text will then be visible when hovered by the mouse and also when the topology file is saved. Multiple nodes and connections can be chosen by dragging a rectangle around them. Nodes or connections can be deleted by choosing one or more and pressing the Delete key on the keyboard or using the context menu.

Topology Designer provides several utilities to assist during topology creation. One of them is used to calculate the shortest route between two nodes. In complex topologies, it can be useful to figure out the path that the architecture would choose in the same scenario. This utility can be accessed via the Analyze menu. Fig. 5.7 shows the delay of the shortest route between *End Device #10* and *Edge Server #4* seen in the very same figure. In the same menu, another utility checks if all nodes have at least one connection. This is useful, especially in complex topologies. Tools menu provides utilities to make complementary connections of the existing ones, by inverting them. Moreover, other utility in this menu can make missing connections.

If participant count in a network raises, it becomes harder to manage the topology and the connections. The designer allows the creation of sub-topologies via groups. Chosen nodes can be grouped together and named. Their colour or images can also be specified to differentiate from others.

```
Java Class
  main()
    Core Resources = speed in MIPS, core count,
    max. memory in MB, max. memory usage in %,
    bandwidth upload capacity in %, bandwidth
    download capacity in %, max. allowed bandwidth
    utilization in %, available diskspace for RTEF-
    defined user program/software/command in MB
    HashMap<String, String> Username password pairs
    Configurator = ID, NAME, PORT, TOPOLOGYFILE
    Core Node (Configurator, Core Resources)
    Core Node.registerServer()
    Local Services
    Core Node.addService(Local Service)
    Core Node.run()
```

Figure 5.8: Pseudocode to create an Edge Server using RTEF in Java.

The topology can be printed or saved for editing later. If the topology is saved in Edge Topology Designer (ETD) format, it will be parsed by the Edge Servers and converted into a plain topology file. ETD file is in eXtensible Meta Language (XML) format and can be reopened with the designer to modify the topology later on, graphically. The content of the ETD file of the topology depicted in Fig. 5.6 can be seen in Source code A.1.

The Topology Designer also allows saving the topology as a plain topology file as defined in Sec. 5.1.1. During conversion, the repetition of the node IDs is checked and displayed as errors. It should be noted that the conversion from ETD file into the plain topology file is one-directional. Once it is saved as a plain file, it cannot be opened again by the Topology Designer as some of the cosmetic parameters are lost. These parameters are such as positioning of the nodes, sizes and their icons. Lastly, the topology can also be saved as an image file in PNG or JPG formats.

5.6 Edge Server Creation

A software reference architecture as a solution to the problems of Cloud Computing and decision making in decentralized environments is defined in Chapter 4. The functionalities of this architecture are grouped into several components, as detailed in Sec. 5.1. All components of the software reference architecture and their functionalities are combined as a framework. This framework is called Real-Time Edge Framework (RTEF) and developed using the Java programming language.

To exploit the benefits of the RTEF, a computer is converted into an Edge Server by running an instance of the framework. It can be achieved by creating a class with a `main` method and defining some mandatory properties. Pseudocode to set up and start an Edge Server is shown in Fig. 5.8. First, an Edge Server requires its resources, called Core Resources, to be defined. Those resources are the speed of the server in MIPS, total core count, maximum available memory in megabytes (MB), maximum allowed memory usage in percentage, maximum allowed bandwidth upload/download capacities in percentage, maximum allowed bandwidth utilization in percentage, and allowed maximum total disk space in MB. Even though only speed and total core count values are used at the moment, other definitions are still mandatory for forward compatibility. Next, the Edge Server needs at least a username and password pair for authentication. As explained before, the user creation is local; therefore, each server must create their own users. If a server does not have a valid user, it cannot be used for remote executions or offloading. Following, the server needs a *Configurator* with the four keys defined as seen in Sec. 5.1.1, namely a unique ID, name, port, and the path to a topology file to be used for creation or reading from it. To broadcast the Edge Server in the network for collaborative use, `registerServer()` method should be used. Then, local services can be defined for the PSCs, following the parameter description in Sec. 4.2.5. It is also possible to skip service definition and add them during runtime, using the `addService` remote command shown in Table 5.2. Finally, the server is run with the `run()` method. This method creates a TCP server to listen for remote commands coming from port defined in the configuration. If this method is omitted, `registerServer()` method is invalidated, as message exchange between Edge Servers are also performed using the TCP Server. Once this class file is run, the RTEF will be started, and the Edge Server will listen for the connections on port, defined in the configuration.

A minimal working example of creating an Edge Server is shown in Source code A.2. Inline comments describe each line of the code. In the working example, one local service and Virtual Processors (VPs) are also created for demonstration. However, creation of these is not mandatory, if not desired. Furthermore, logging settings are shown for possible debugging. The framework provides extensive API documentation, describing all methods, arguments, and parameters in detail. RTEF provides a `loadGenerator`, to be linked with the services, to create a dummy load. This internal load generator creates a task with the given parameters, such as runtime, thread count, and deadline. The next section will create several Edge Servers using the template seen in Source code A.2 with slight modifications. The servers will be using internal load generator to have an ideal execution environment. The concepts, architecture, and framework using different experimental setups will be validated.

5.7 Validation of Framework

This section is going to validate the framework, defining two different scenarios. As described in Sec. 4.2.5, services are linked with PSCs to call them when it is time to execute using defined parameters. For validation, instead of using a PSC, to have a setup close to an ideal environment, an internal load generator is implemented. The load generator is a dummy command that creates a CPU load with the given parameters, such as WCET, allowed CPU(s), and the number of threads, as defined in Sec. 4.2.5. This utility is integrated with the RTEF and used to test the architecture in different scenarios. The load generator can create a different type of tasks for all type of services; Legacy, Simple, or Periodic. The test environment uses this load generator and creates an Edge Network consisting of two Edge Servers and four End Devices.

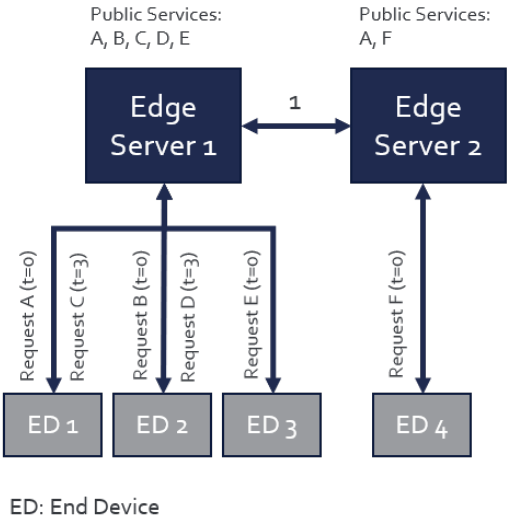

Figure 5.9: Experiment setup for an Edge Network using four End Devices and two Edge Servers.

Computers are converted into Edge Servers using classes similar to Source code A.2. As End Devices, simple TCP clients were used to send and receive TCP messages. Any TCP client can be used or implemented to behave as an End Device. In the experiments, PuTTY[3] is chosen.

The validation was performed on Linux-based systems with the real-time enabled kernel. The hardware that the RTEF working on is simulated using virtualization software and assumed to be ideal and real-time capable. The experiment setup and plan is summarized in Fig. 5.9. First server has two identical CPUs and the second server has only a single CPU. Each CPU at each server is running at a speed of 1 MIPS. The first server is connected to the second server, and

[3]Website: `https://putty.org`

```
[2020-06-06 17:21:02] [INFO   ] Changing log level to INFO
[2020-06-06 17:21:02] [INFO   ] User admin (Role: OPERATOR) is added.
[2020-06-06 17:21:02] [INFO   ] Resources are broadcast in the Edge Network.
[2020-06-06 17:21:02] [INFO   ] Node 1 -> Edge #1 added.
[2020-06-06 17:21:02] [INFO   ] Core registered in the Edge Topology.
[2020-06-06 17:21:02] [INFO   ] There are 1 registered Nodes in the Edge Network.
[2020-06-06 17:21:02] [INFO   ] Service A (Type: LEGACY) was registered on the server.
[2020-06-06 17:21:02] [INFO   ] Service A was broadcast in the Edge Network.
[2020-06-06 17:21:02] [INFO   ] Service B (Type: LEGACY) was registered on the server.
[2020-06-06 17:21:02] [INFO   ] Service B was broadcast in the Edge Network.
[2020-06-06 17:21:02] [INFO   ] Service C (Type: SPERIODIC) was registered on the server.
[2020-06-06 17:21:02] [INFO   ] Service C was broadcast in the Edge Network.
[2020-06-06 17:21:02] [INFO   ] Service D (Type: SPERIODIC) was registered on the server.
[2020-06-06 17:21:02] [INFO   ] Service D was broadcast in the Edge Network.
[2020-06-06 17:21:02] [INFO   ] Service E (Type: SIMPLE) was registered on the server.
[2020-06-06 17:21:02] [INFO   ] Service E was broadcast in the Edge Network.
[2020-06-06 17:21:02] [INFO   ] Starting Core Node. ID: 1, Name: Edge #1
[2020-06-06 17:21:02] [INFO   ] Server is ready (7.15.115.57) and waiting for connection at port: 9091
```

Figure 5.10: The console log of the Edge Server 1 created for the validation after it is started.

the delay between them is one time unit. The first server also has connections with the first three End Devices and the second server with the fourth one. The delays between the End Devices and their directly connected servers are neglected. The load generator is defined with services A, B, C, D, E, F each with different parameters listed in Table 5.3. All tasks, hence their services are one-directional. Services A and B are defined as Legacy services, C and D as Simple Periodic services, E and F as Simple services. Service A is created in both servers, using the same parameters. Services B, C, D, and E are created only on the first server, and Service F only on the second server.

Figure 5.11: Handshaking to establish a connection to the Edge Server 1 (with ID: 1) from an End Device 1 (with ID: 10) using PuTTY. The messages sent by the End Device (in this example by PuTTY) have a bounding box.

Fig. 5.10 shows the console log of the *Edge Server 1* after RTEF is started. Connection to the *Edge Server 1* from *End Device 1* is established using PuTTY as described in Sec. 5.2 and seen in Fig. 5.11. Other connections between nodes are established using the same handshaking messages seen in the figure, only using different IDs during the introduction. Green rectangles

surround the messages sent by the End Device. Remaining messages are received from the connected Edge Server.

As seen from the experiment setup (Fig. 5.9), End Device 1, 2, and 3 request a task from Edge Server 1 at time zero ($t = 0$) to execute A, B and E, respectively. At the same time, End Device 4 requests F from Edge Server 2. At $t = 3$, End Device 1 also requests an instance of C and End Device 2 requests D. As explained in Sec. 2.2.2, the architecture does not support running periodic tasks together with non-periodic tasks (Legacy and Simple) at the same time. Therefore, the experiment was carefully set up to avoid such cases. Parameters of the services and Edge Servers hosting these services are analysed in advance. With these considerations, EDF scheduling is used for tasks with Periodic services and NAPATA scheduling for non-periodic services.

Table 5.3: An example set of pre-defined services with defined execution behaviours of tasks. All tasks are one-directional.

Service	WCET	Rel. Deadline	Type	CPU Mask	Max. CPU Util. %	Thread/Core	Available on
A	6	8	Legacy	1	100	1	1,2
B	3	3	Legacy	1	100	1	1
C	4	8	S.Periodic	3	100	1	1
D	3	6	S.Periodic	3	100	1	1
E	2	5	Simple	2	100	1	1
F	2	9	Simple	1	100	1	2

At $t = 0$, multiple tasks arrive. The execution order of the tasks based on their deadlines on Edge Server 1 is B, E, and A. E uses CPU 2; however, A and B share the same CPU: CPU 1. B has higher priority than A. As there is no free resource at the only allowed CPU (CPU 1) to execute A before its deadline, the possibility of scaling is evaluated. Since the WCEU of B is 100% ($= \frac{3}{3}100$), the downscaling of B will not be possible. If Edge Server 1 were to be the only server in the network, NAPATA scheduling could have been used to evaluate whether it is possible to schedule these tasks. Although there is an alternative server (Edge Server 2) that can possibly execute this task, it is possible to evaluate the scheduling on Edge Server 1. B has higher priority than A, hence, the feasibility check starts with task B.

$$F_B = 0 + M_B \leq d_B + a_B \tag{5.1}$$
$$= 0 + 3 \leq 3 + 0 \tag{5.2}$$
$$= 3 \leq 3 \tag{5.3}$$

Inequality 5.3 holds. The algorithm then proceeds with task A.

$$F_A = 0 + M_A + M_B \leq d_A + a_A \tag{5.4}$$
$$= 0 + 6 + 3 \leq 8 + 0 \tag{5.5}$$
$$= 9 \leq 8 \tag{5.6}$$

The feasibility test fails since inequality 5.6 does not hold. As scheduling A after B would cause A to miss its deadline, the task would have failed. However, with the current setup, another alternative server within the network is searched. Meanwhile, E and B start execution on Edge Server 1 and F on Edge Server 2. Edge Server 2 also has service A and the request can be forwarded to that server. The transfer of the request takes one time unit due to the delay between the servers. Then, the first request of A at Edge Server 2 occurs at $t = 1$. However, as the initial request was on Edge Server 1, this delay must be subtracted from absolute deadline calculation. Hence, the arrival time of A (a_A) used on Edge Server 2 is zero (0). The scheduling diagram between $t = 0$ and $t = 1$ is seen in Fig. 5.12.

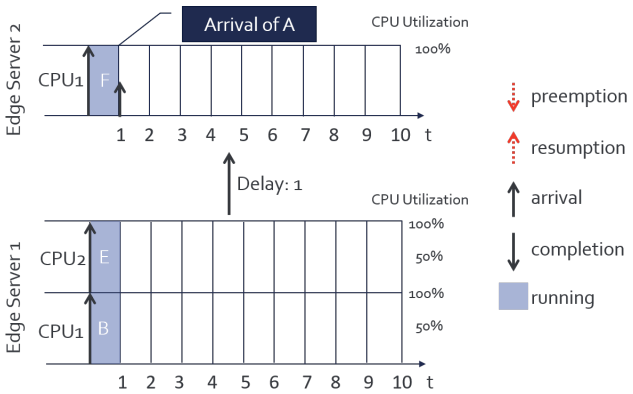

Figure 5.12: Scheduling diagram of the first scenario between $t = 0$ and $t = 1$.

At $t = 1$ on Edge Server 2, the possibility of on-time execution of tasks F and A are calculated using Eq. 3.12 from NAPATA scheduling. Sorting them by their priorities from high to low gives A and F. Starting from the highest priority, the feasibility calculation of task A (F_A):

$$F_A = 1 + M_A \leq d_A + a_A \tag{5.7}$$
$$= 1 + 6 \leq 8 + 0 \tag{5.8}$$
$$= 7 \leq 8 \tag{5.9}$$

passes. Similarly, feasibility of task F (F_F):

$$F_F = 1 + M_F + M_A \leq d_F + a_F \tag{5.10}$$
$$= 1 + 1 + 6 \leq 9 + 0 \tag{5.11}$$
$$= 8 \leq 9 \tag{5.12}$$

is satisfied. As scheduling is feasible, task F is preempted at $t = 1$ for having a lower deadline than task A. After A completes its execution, F will be resumed to complete its remaining

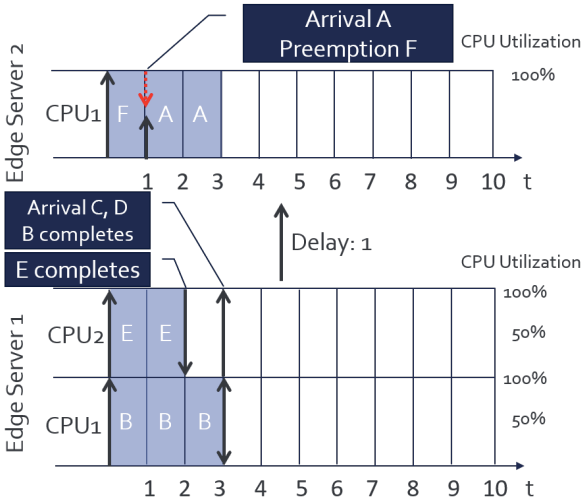

Figure 5.13: Scheduling diagram of the first scenario at $t = 3$, until C and D arrive.

execution time. C and D task requests arrive at $t = 3$. The scheduling diagram between $t = 0$ and until the time when two tasks arrive at $t = 3$ is seen in Fig. 5.13. Meanwhile, the task E completes its execution at $t = 2$ and task B at $t = 3$.

Both C and D have one thread at each CPU due to their CPU affinities and thread per core parameters. At $t = 3$, Edge Server 1 is available as E and B complete their executions. Both tasks, C and D can be scheduled according to EDF scheduling. However, the server downscales them first, as scheduling is considered as a fallback solution. Downscaling WCEUs of C ($\frac{4}{8}100$) and D ($\frac{3}{6}100$) without missing their deadlines yields 50% CPU utilization, which doubles execution times of both tasks. The scheduling diagram until $t = 7$ where A completes execution is shown in Fig 5.14. From this moment, F will be resumed until its completion.

F completes its execution at $t = 8$; C at $t = 11$, and D at $t = 9$. Since C and D are defined as Periodic services, they start execution again as soon as they complete execution. The complete scheduling diagram is shown in Fig. 5.15.

As soon as the tasks start execution, each Edge Server returns the ID of the server running the task and the unique task ID, back to the original End Device (Sec. 5.1.3). Assuming that that F has an ID of 1 and A of 2, for example, at $t = 1$, End Device 1 would then receive `RUNNING:A@2,2`. This is interpreted as *Service A is running on Edge Server 2, with task ID 2*. Similarly, End Device 4 would receive "`RUNNING:F@2,1`", meaning *Service F is running on Edge Server 2, with task ID 1*.

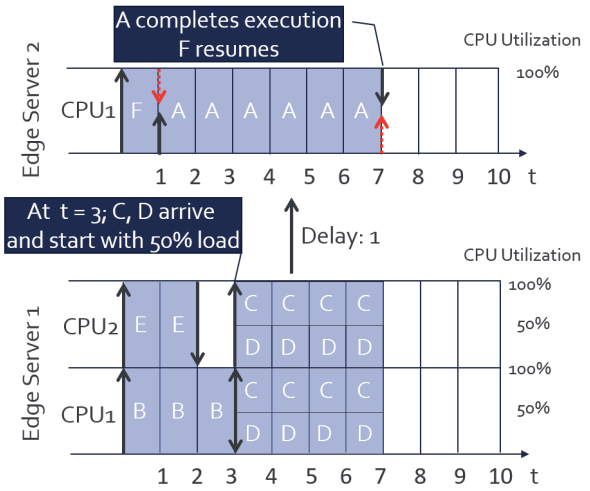

Figure 5.14: Scheduling diagram of the first scenario from the beginning until $t = 7$.

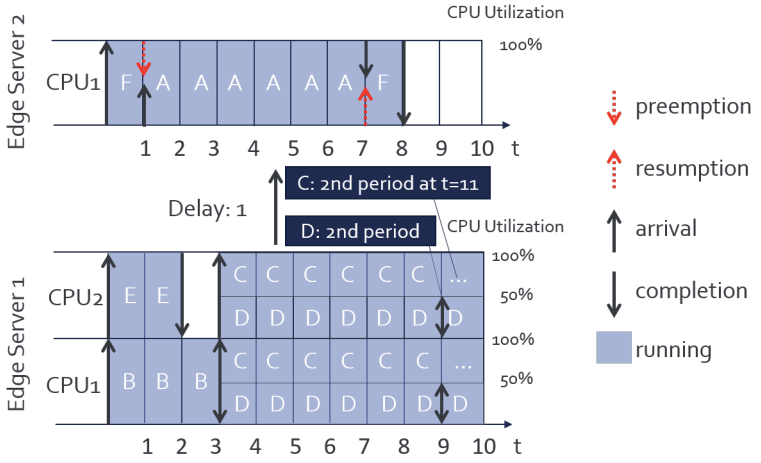

Figure 5.15: Resulting scheduling diagram based on the example defined in Fig. 5.9 and Table 5.3 on Edge Server 1 and Edge Server 2.

The operators are expected to create a feasible distribution of tasks in Edge Servers during service creation. If the example above had a shorter deadline for F (e.g., at $t = 7$), then scheduling F would not be feasible. Since there is not another alternative server to execute F, it would miss its deadline. An alternative fallback would be a Cloud server connected to the Edge Servers with RTEF installed; however, in that case, the execution would have been made using the best-effort approach. Another possibility would be terminating lower priority tasks, if any. However, task termination is not considered in this thesis. Instead, it is listed as an open point for future work.

In the example above, the downscaling of tasks C and D was possible. However, if the *Scaler* was disabled (See Sec. 5.1.6), the only way to plan execution would be by scheduling these tasks. Since the tasks arrive at the same time and they are periodic, the Eq. 3.36 could be used. The feasibility equation for these tasks results in 1 ($= \frac{4}{8} + \frac{3}{6}$) for each core, meaning that the tasks could be scheduled, without changing the affinity.

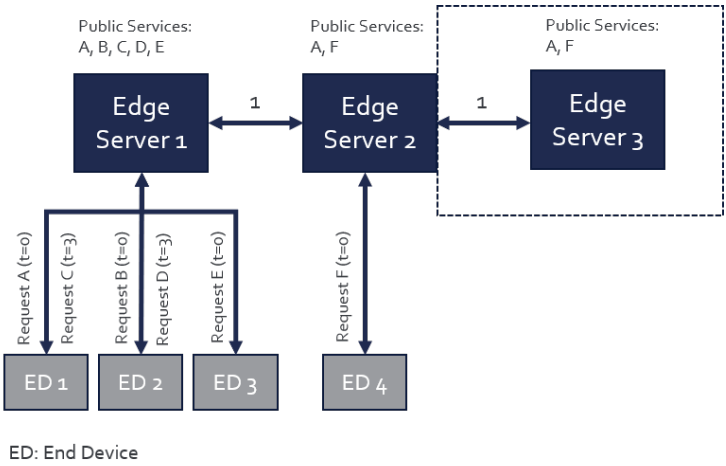

Figure 5.16: A second example scenario that adds another Edge Server to the Edge Network and connects it to the Edge Server 2 with a delay of 1 unit.

As the second experiment, another identical Edge Server with ID 3 is added, and the test is repeated. As seen in Fig. 5.16, the newly added server is only connected to the Edge Server 2 with the delay of one time unit. Similar to Edge Server 2, this server also has a single CPU with 1 MIPS speed. Likewise, it has A and F as public services.

Theoretically, there are two possible solutions to this scenario. One solution is not using Edge Server 3 at all, as the previous scheduling was feasible, facilitating only two Edge Servers, as seen in Fig. 5.15. However, with the existence of Edge Server 3, the decision mechanism comes

into the play (See Sec. 4.3) as the second solution. The decision mechanism will evaluate the resource availabilities of the servers and choose the most available server first, using the satisfaction equation (Eq. 4.4).

Evaluating the availabilities of each three server using Eq. 4.4 for the service A gives:

$$S_1(E_1, T_1) = max\left(\left\{\left(\frac{x_1 100}{(1)F_{k,1}} + (1)(0)\right) f_{1,k} : k = 1, 2\right\}\right) \tag{5.13}$$

$$= max\left(\left\{\left(\frac{(6)(100)}{(1)(0)} + (1)(0)\right) 1, \left(\frac{(6)(100)}{(1)(0)} + (1)(0)\right) 0\right\}\right) \tag{5.14}$$

$$= \text{Not Feasible} \tag{5.15}$$

$$S_2(E_2, T_1) = max\left(\left\{\left(\frac{x_1 100}{(1)F_{k,1}} + (1)(1)\right) f_{1,k} : k = 1\right\}\right) \tag{5.16}$$

$$= max\left(\left\{\left(\frac{(6)(100)}{(1)(0)} + (1)(1)\right) 1\right\}\right) \tag{5.17}$$

$$= \text{Not Feasible} \tag{5.18}$$

$$S_3(E_3, T_1) = max\left(\left\{\left(\frac{x_1 100}{(1)F_{k,3}} + (1)(2)\right) f_{1,k} : k = 1\right\}\right) \tag{5.19}$$

$$= max\left(\left\{\left(\frac{(6)(100)}{(1)(100)} + (1)(2)\right) 1\right\}\right) \tag{5.20}$$

$$= 8 \leq 8 + 0 \tag{5.21}$$

As seen in the results of Eq. 5.13 and Eq. 5.16, Edge Server 1 and Edge Server 2 have no available CPU for another task, but only Edge Server 3. Therefore, when A is requested from Edge Server 1 at $t = 0$, Edge Server 3 is chosen. The delay between End Device 1 and Edge Server 2 is one time unit whereas between End Device 1 and Edge Server 3 is two time units. Consequently, A arrives at Edge Server 3 at $t = 2$ when this server is chosen. However, initially, the task arrived at Edge Server 1 at $t = 0$. Hence, while calculating the absolute deadline, this value is taken into consideration. Task A can be executed on Edge Server 3 since its absolute deadline $(8 + 0)$ allows it. Meanwhile, F completes its execution at $t = 2$, instead of $t = 8$. The complete diagrams of Edge Server 2 and Edge Server 3 can be seen in Fig. 5.17. The scheduling diagram of Edge Server 1 is not changed, hence, omitted.

These two scenarios validated the solutions for the problems defined in the earlier chapters. These problems were offloading decisions in decentralized environments and scheduling of non-resumable and preemptible aperiodic tasks. The scenarios were completed successfully and the validation gave the expected results. Therefore, the conceptual architecture was proven for

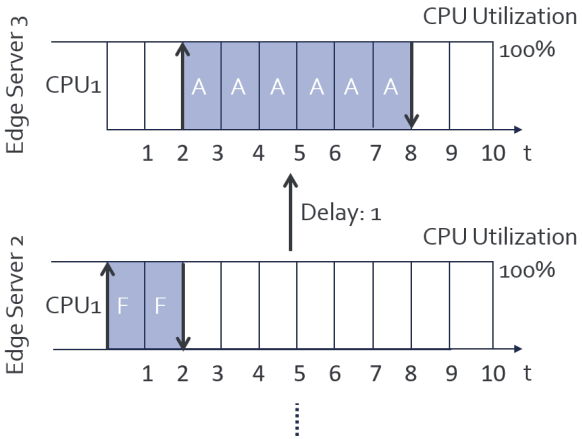

Figure 5.17: Resulting scheduling diagram based on the example defined in Fig. 5.16 and Table 5.3 on Edge Server 2 and Edge Server 3. Edge Server 1 results are omitted since they stayed the same.

correctness. The next chapter will conclude the thesis and give an outlook for future research.

5.8 Summary

This chapter explained how the software reference architecture described in Chapter 4 is realized as a framework, called Real-Time Edge Framework (RTEF). It also explained its components (Sec. 5.1), how the communication between the nodes can be performed (See Sec. 5.2), commands implemented by the framework (Sec. 5.3), and how the End Devices can request tasks (Sec. 5.4). The chapter continued explaining another contribution of the thesis, the Topology Designer (Sec. 5.5) together with its features. Moreover, this chapter exemplified the creation of an Edge Server using the framework (Sec. 5.6). Finally, the chapter was concluded by validating the concept using two different scenarios (Sec. 5.7).

6 Conclusion and Outlook

This thesis introduced a concept to create a modular, and extensible architecture for the Edge Servers in the Edge Computing domain, to handle real-time task requests, and execute these tasks on-time in a decentralized environment.

The thesis began with an introduction to Edge Computing and continued with a problem definition, objectives of the thesis, and followed by the approach in Chapter 1. Chapter 2 started from Cloud Computing history and listed some issues that started arising due to the increased device count connected to the Cloud. Then, the same chapter moved to Edge Computing history, together with its advantages, and initial ideas of the domain. Further, it elaborated on the related work done both in Cloud Computing and Edge Computing, followed by enablers and requirements of Edge Computing. Later, it moved to the real-time computing domain where the thesis is mostly focused on. It defined the real-time term, listed its challenges, problems, and how they can be overcome. Chapter 3 brought the first three chapters together, introduced the novel scheduling algorithm, called Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling, and formulated one of the problems: server selection on decentralized environments. Chapter 4 explained the conceptual idea of the architecture, explaining the concepts of Edge Servers, End Devices, real-time task execution, and decision mechanisms. In Chapter 5, the concepts were proven creating a framework based on the reference architecture and using two complex scenarios.

This final chapter of the thesis concludes the findings of the thesis, highlights the features of the reference architecture, and lists assumptions and limitations to be considered for the realization of the theory. Furthermore, the chapter concludes with an outlook for future research.

6.1 Conclusion

Edge Computing is a recent paradigm to overcome the limitations and issues of Cloud Computing, due to the increasing number of connected devices, and the amount of data transferred to the Cloud. It adds a new tier between Operational Technology (OT) and Information Technology (IT). Thanks to the closeness to the source of the data, the data owners have more control over the sensitive information and reduced latency. Moreover, with the computing power added closer to the devices, dependency to the Internet is reduced. However, there exists no

accepted standard that can perform real-time calculations in the Edge, nor a widely-accepted definition of Edge Computing in literature. This makes it hard to create interoperable and flexible intercommunication methods for the devices, which could otherwise benefit from Edge Computing.

Edge Servers perform computation at the edge, and they can be close to the source as much as possible. This also enables more critical tasks to be executed on these Edge Servers. There exist several reference architectures from different initiatives in the domain; however, none of them deals with real-time characteristics of the actors. The ability to execute real-time tasks on these servers increase the reliability of the systems in case of resource starvation.

The main contribution of this thesis is a software reference architecture for Edge Servers to create an Edge Network with a framework featuring decentralization, modularity, and scalability. The architecture proposes several concepts to enable real-time execution of tasks on Edge Servers. These servers can execute the tasks on them, or offload the task requests to their neighbouring servers using decentralized decisions. Regardless of the execution location, the results of the tasks are also sent back to their original requesters. Offloading is performed when an Edge Server is expected to miss the deadline of a real-time task, or if the server is overloaded. Tasks run a program/software/command (PSC) when executed. Their behaviours are defined using services. The PSCs can also be containers which include the complete stack for an application. However, to have a full control on the PSC and to avoid additional overheads, it must be used with caution.

The thesis followed the assumptions below to fulfil the requirements of a real-time execution in a decentralized Edge Network:

- The hardware to realize the architecture is ideal, meaning formally verified for real-time computing.
- The hardware has identical processors if consisting of multiple cores.
- The used hardware has enough storage and memory to handle all requests.
- The scheduling algorithms have no overhead, which may delay the calculation.
- Decision mechanisms have no overhead, which may delay the calculation.
- The worst-case execution time (WCET) of a PSC is known a priori.
- The PSC does not suspend itself before its execution is completed.
- Running PSCs are independent of each other, and they have protection for critical section usage.
- The network message exchange is real-time, e.g. uses real-time communication technologies such as Time-Sensitive Networking (TSN).

The contributed architecture is realized as an exemplary framework. This framework implemented all features of the architecture and tested the correctness of the theoretical approach. The framework can be used as a simulator with its integrated dummy load generator features or can be run on a computer, which complies with the hardware assumptions listed above.

To create an Edge Network, first Edge Servers establish a connection with each other. After each connection, all known information by an Edge Server is shared among all participants. When an End Device requests a task execution from any Edge Server, the server first calculates whether the execution of the received task within its deadline is possible, by evaluating its resources. If not, the task request is forwarded to another server, which can execute this task in the shortest time. If the timings are the same, then the request is passed to the server with minimum delay. To choose a server, the server initially received the request checks whether the task can be executed on time, without any behaviour change, including other running tasks (if any) on that server. If not, the possibility of scaling down the processor utilization is evaluated. When the scaling down causes a deadline miss, then the running tasks are scheduled according to the service type that the task belongs to, using an appropriate scheduling algorithm. Depending on the periodicity of the tasks, two scheduling algorithms are supported in the architecture and the framework. For periodic tasks, it uses the Earliest Deadline First (EDF) scheduling algorithm. Preemptive scheduling of aperiodic tasks is performed using a novel scheduling algorithm, called Non-resumable And Preemptible Aperiodic TAsk (NAPATA) scheduling, contributed by this thesis. This scheduling algorithm is an online scheduler that prioritizes aperiodic tasks according to their deadlines and types.

The architecture and framework that the architecture is based of, are tested with two complex scenarios, to prove whether they solve the problem defined in Sec. 1.1 and formulated in Sec. 3.2. The scenario results were the same as expected in the theoretical calculations; thus, the concept was proven. As a result, if Edge Servers are defined in accordance with the architecture concepts, they can collaboratively work to execute real-time tasks on time.

6.2 Outlook

The limitations and assumptions of this thesis can be used as a guide for further research in the Edge Computing domain. This section will include the possible extension topics for the thesis.

Task migration is a way of moving a running task into another computer for completion. Migration can be performed for a higher priority task, or to speed up the completion of the task, if another server has more resources than the current one. In this thesis, task migration is not possible, meaning once a server starts a task, only that server can complete it. Additionally, optimal scheduling of the combination of Legacy, Simple, and Simple Periodic tasks cannot be performed. Currently, the processor affinities must be set by the operator during service cre-

ation time manually, to prevent such circumstances. An optimal algorithm that can schedule multiple types of tasks or a schedule server to allow such conditions is desirable.

At the moment, if no alternative server is found for the newly arriving task, it fails to execute. An algorithm can be implemented to determine whether an existing task should be terminated to enable execution of the new task, based on the properties, deadline, or priority. Furthermore, the delays between Edge Servers and End Devices are considered to be static. However, the load on the network traffic affects the transmission time. The decision mechanisms can also model the network and make decisions analysing the traffic as well. The tasks running in the framework are considered to be independent. Nevertheless, tasks may request, execute a task, or depend on another task. Currently, this dependency has to be handled at the End Device side. This work can also be extended by introducing artificial intelligence (AI) or machine learning (ML) for decision mechanisms and the prediction of the server resources. Moreover, the resulting framework can be deployed on the hardware. Although the architecture is hardware and operating system (OS) agnostic, the framework was implemented in the Java programming language. The performance of the architecture and the framework on different hardware can be evaluated.

Finally, although the required time to offload the task is calculated, the overheads of the scheduling and decision making algorithms are neglected. When the framework is deployed on hardware, the durations that these algorithms require can be evaluated for a better time estimation.

A Appendix

A.1 Edge Topology Designer File Example

Source code A.1: Edge Topology Designer file that can be reopened with Topology Designer. The contents are from the topology seen in Fig. 5.6.

```
<mxGraphModel>
  <root>
    <mxCell id="0" />
    <mxCell id="1" parent="0" />
    <mxCell id="2" parent="1" serverid="1" style="image;image=/com/mxgraph
    ↪ /examples/swing/images/server.png" value="Edge Server #1" vertex="1"
    ↪  warningmessage="Using auto-generated Server ID: 1">
      <mxGeometry as="geometry" height="50.0" width="50.0" x="100.0" y="
    ↪ 320.0" />
    </mxCell>
    <mxCell id="3" parent="1" serverid="2" style="image;image=/com/mxgraph
    ↪ /examples/swing/images/server.png" value="Edge Server #2" vertex="1"
    ↪  warningmessage="Using auto-generated Server ID: 2">
      <mxGeometry as="geometry" height="50.0" width="50.0" x="230.0" y="
    ↪ 320.0" />
    </mxCell>
    <mxCell id="5" parent="1" serverid="3" style="image;image=/com/mxgraph
    ↪ /examples/swing/images/server.png" value="Edge Server #3" vertex="1"
    ↪  warningmessage="Using auto-generated Server ID: 3">
      <mxGeometry as="geometry" height="50.0" width="50.0" x="350.0" y="
    ↪ 320.0" />
    </mxCell>
    <mxCell edge="1" id="9" parent="1" source="2" style="" target="3"
    ↪ value="1">
      <mxGeometry as="geometry" relative="1">
        <mxPoint as="sourcePoint" x="130.0" y="350.0" />
        <mxPoint as="targetPoint" x="260.0" y="350.0" />
      </mxGeometry>
    </mxCell>
    <mxCell edge="1" id="10" parent="1" source="3" style="" target="5"
    ↪ value="1">
```

```
21        <mxGeometry as="geometry" relative="1">
22          <mxPoint as="sourcePoint" x="260.0" y="350.0" />
23          <mxPoint as="targetPoint" x="360.0" y="350.0" />
24        </mxGeometry>
25      </mxCell>
26      <mxCell id="11" parent="1" serverid="10" style="image;image=/com/
        ↪ mxgraph/examples/swing/images/telephone.png" value="End Device #10"
        ↪ vertex="1" warningmessage="Using auto-generated ID: 10">
27        <mxGeometry as="geometry" height="50.0" width="50.0" x="100.0" y="
        ↪ 460.0" />
28      </mxCell>
29      <mxCell edge="1" id="12" parent="1" source="11" style="" target="2"
        ↪ value="1">
30        <mxGeometry as="geometry" relative="1">
31          <mxPoint as="sourcePoint" x="130.0" y="490.0" />
32          <mxPoint as="targetPoint" x="110.0" y="370.0" />
33        </mxGeometry>
34      </mxCell>
35      <mxCell id="13" parent="1" serverid="4" style="image;image=/com/
        ↪ mxgraph/examples/swing/images/server.png" value="Edge Server #4"
        ↪ vertex="1" warningmessage="Using auto-generated Server ID: 4">
36        <mxGeometry as="geometry" height="50.0" width="50.0" x="350.0" y="
        ↪ 220.0" />
37      </mxCell>
38      <mxCell edge="1" id="14" parent="1" source="5" style="" target="13"
        ↪ value="1">
39        <mxGeometry as="geometry" relative="1">
40          <mxPoint as="sourcePoint" x="370.0" y="350.0" />
41          <mxPoint as="targetPoint" x="380.0" y="270.0" />
42        </mxGeometry>
43      </mxCell>
44      <mxCell edge="1" id="15" parent="1" source="2" style="" target="13"
        ↪ value="5">
45        <mxGeometry as="geometry" relative="1">
46          <mxPoint as="sourcePoint" x="130.0" y="350.0" />
47          <mxPoint as="targetPoint" x="360.0" y="240.0" />
48          <Array as="points">
49            <mxPoint x="130.0" y="240.0" />
50          </Array>
51        </mxGeometry>
52      </mxCell>
53    </root>
54  </mxGraphModel>
```

A.2 Creation of an Edge Server Using RTEF: An Example

Source code A.2: A minimal working example to create an Edge Server using RTEF.

```java
import java.io.IOException;
import java.util.HashMap;

import de.dfki.edgesim.edgeserver.Configurator;
import de.dfki.edgesim.edgeserver.CoreNode;
import de.dfki.edgesim.edgeserver.CoreResource;
import de.dfki.edgesim.edgeserver.SecurityProtocol.Role;
import de.dfki.edgesim.service.Service;
import de.dfki.edgesim.utils.SimLogger;

/**
* This class is a minimal example to create an Edge Server using RTEF.
    ↪ Each line is explained in-line. Comments starting with [OPTIONAL]
    ↪ defines that they are not mandatory for the Edge Server to function.
*/
public class TestEdgeServer {

/**
* Creates an Edge Server and starts its listening server.
*
* @param args for terminal arguments.
* @throws IOException in case an important config parameter is not set.
*/
public static void main(final String[] args) throws IOException {
/**
* Logging level. INFO is for production, FINE and above are for debugging.
    ↪  As a parameter, a filename can also be given to log into a file as
    ↪ well. E.g. SimLogger.setFormatting("INFO", "log.log");
*/
SimLogger.setFormatting("INFO");
/**
* Define the core resources
*/
final CoreResource coreResources = new CoreResource(
10000, // speed of the server in MIPS, to calculate task execution time.
2,     // total CPU count of this server
10240, // maximum available memory in MB, Not Implemented Yet
100,   // maximum allowed memory usage in percentage, Not Implemented Yet
100,   // bandwidth allowed upload capacity in percentage, Not Implemented
    ↪  Yet
```

```
100,   // bandwidth allowed download capacity in percentage, Not Implemented Yet
100,   // maximum allowed bandwidth utilization in percentage, Not Implemented Yet
102400 // available disk space in MB, Not Implemented Yet
);

/**
* Each Core Node needs at least one user name password pair for login.
*
*/
final HashMap<String, String> userCredentials = new HashMap<>();
userCredentials.put("username", "password"); // Action only with those credentials possible.

/**
* Additional users can be added later on.
*/
// userCredentials.put("seconduser", "password2"); // Username should be unique.

/**
* Set some pair of configurations with at least one user credentials pair. The
* Core must have at least ID, NAME, PORT, and TOPOLOGYFILE keys. See class document for more details.
*/
final Configurator configurator = new Configurator(userCredentials);
configurator.setUserRole("username", Role.OPERATOR); // Change user role to OPERATOR for full access
configurator.setConfig("ID", "1"); // Mandatory unique ID
configurator.setConfig("NAME", "Edge #1"); // A name for referencing
configurator.setConfig("PORT", 9091); // Listening port for remote commands
configurator.setConfig("TOPOLOGYFILE", "src/latMatrixConv"); // The file to store available nodes and their connections
configurator.setConfig("AUTOCONNECT", "false"); // [OPTIONAL] if set to true, known connections are reestablished if connection drops.

/**
* Combine core resources and configuration values together and create the server.
*/
final CoreNode coreNode = new CoreNode(configurator, coreResources);
```

```
69 /**
70
71 * [OPTIONAL] Create two VPs each assigning to one CPU and giving 50% and
       ↪ 100% execution capacity, respectively.
72 */
73 coreNode.addVirtualProcessor("VP0", 50, 100, 3); // All CPUs are allowed
       ↪ with 50% execution cap.
74 coreNode.addVirtualProcessor("VP1", 100, 100, 1); // Only CPU0 is allowed
       ↪ with 100% execution cap.
75
76 /**
77 * [OPTIONAL] If there exists a topology file created using Topology
       ↪ Designer it can be parsed using the following command.
78 */
79 // coreNode.parseTopologyDesignerFile("src/latencyExample.etd");
80
81 /**
82 * Register the Server in the network. Whenever a change in resources is
       ↪ detected, the network is informed with the changes.
83 */
84 coreNode.registerServer();
85
86 /**
87 * [OPTIONAL] Create local services. Services can be created while
       ↪ instantiating the Edge Server or added later using remote commands.
88 * If no CPU mask is used, then all CPUs are allowed for execution
89 * If publicity is not set, it is a public and can be accessed by other
       ↪ servers.
90 *
91 * MIPS: 10000, Relative Deadline: 30000, Type: SIMPLE, Direction: ONE
92 */
93 final Service service1 = new Service(Service.Type.SIMPLE, // Type
94 Service.Direction.ONE, // Direction: ONE no response needed.
95 30000,          // Relative deadline in terms of MIPS.
96 10000,          // WCET of the task in MI. In this server it will take:
       ↪ 10000/10000 = 1 second to execute if free.
97 1024,           // Required Memory. Not Implemented Yet
98 1,              // Thread per core. Creates a thread on each core
99 "loadGenerator",// Command to execute. loadGenerator uses internal load
       ↪ generator. This parameter is used to link the service to an actual
       ↪ software/program/command.
100 100,           // How much percentage of the CPU is allowed for the
       ↪ execution.
101 "TestService"   // Name to recall later. Used also to request tasks.
```

```
102 );
103
104 /**
105 * If a service created in the code, creating them do not integrate them
        ↪ directly into the current server. Services must be added using
        ↪ addService(service). However, creating services via remote commands
        ↪ perform this automatically. If a service is to be created using a
        ↪ remote command, this line can also be omitted.
106 */
107 coreNode.addService(service1);
108
109 /**
110 * Start core node with a server thread. If used, a TCP server is created.
        ↪ If not used, the server only executes commands/methods defined in
        ↪ the main() method. In this case, if collaborative execution is
        ↪ desired, it is also necessary to establish connections via coreNode.
        ↪ connectTo(...) method and create links using coreNode.addLink(...)
        ↪ method. See method documentation for details.
111 */
112 coreNode.run();
113
114 /**
115 * [OPTIONAL] Scaler provides editable settings. See related component
        ↪ description for details. They can be omitted. Default values are
        ↪ given below.
116 */
117 // coreNode.setScalerEnabled(true);
118
119 }
120 }
```

A.3 Definitions

In IT and engineering, the meaning of some terms may differ from other domains or their daily usage. Even in the same domain, some terms may be used in different contexts. To avoid confusion and prevent ambiguity, in this section, some terms are explained. Whenever these terms are used, the meanings defined in this section should be considered, unless stated otherwise.

Edge Computing

According to the Oxford dictionary [Ox20], an *edge* is the outside limit of an object; the outside border. When two objects intersect, their edges contact each other.

Edge Computing then means the computation at the borders. In this thesis, it means the computation performed at the point where the IT meets Operation Technology (OT) (Fig. 2.1). Edge Computing is also named as Edge Cloud, Fog Computing, or Cloudlets [GUR18]. However, in this thesis, only the term *Edge Computing* will be used.

Edge Server

A server is a software or hardware that provides functionalities to one or multiple programs or devices. Servers can be hosting databases, files, or mails.

Edge Server in this thesis is hardware that utilizes a Real-Time Edge Framework (RTEF) and provides its functionalities to End Devices.

End Device

A source or destination device in a network is called an End Device [Th21]. An End Device in the RTEF is a resource-limited end-user device that requests execution of jobs through Edge Servers. It may have computing power, or only provide input to the Edge Server. In RTEF, they communicate with the servers via a socket communication. An End Device can be a sensor, a smart sensor, a machine, a computer, a mobile phone, or smart glasses. They can also be called clients. They are allowed to have connections to multiple Edge Servers, but they ask for a job only from one server. They also must obey the decisions made by the Edge Server.

Edge Network

A network, in general, is the interconnection of units to share information [Pe21]. A computer network is a digital communication network that allows computing units to share resources with each other via data links. The data links are established over cable media such as wires or optic cables, or wireless media such as Wi-Fi.

An Edge Network in this thesis is the collection of Edge Servers and End Devices which communicate with each other using cable media to request jobs and/or respond to jobs.

Edge Topology

Network topology is the arrangement of the set of participants in a network by creating links between them and creating a network structure. Edge Topology in this thesis is the creation of the network structure by linking Edge Servers and End Devices in the Edge Network. Network topologies can be classified as point-to-point, bus, star, ring, mesh, daisy chain, and hybrid. This thesis does not limit the Edge Topology, and the network can be organized using any of the available topologies.

Process and Thread

A process is an instance of a runnable program, which consists of an executable object code, usually read from some hard media and loaded into memory [SBG09].

A thread, which is also known as a ”light-weight process”, is an execution context. Each process contains at least one thread. Once a process contains several execution threads, it is said to be a multi-threaded process. Multiple threads allow concurrent programming. On multi-core systems, this is called true parallelism [BC05].

Each process gets a unique ID as soon as it is executed. If the process is single-threaded, the process ID is equal to the thread ID. In the multi-threaded process, each thread gets a unique ID [Li20c], but they share their address spaces between each other. This allows them to communicate with each other without using any kind of Inter-Process Communication (IPC) methods; thus, context switching is inexpensive and fast.

To illustrate the different kind of threads, a POSIX-compliant system can be considered. Under these systems, there are two types of threads: User threads and kernel threads. User threads that work on user-space are created with user-level library Application Programming Interface (API) method, `pthread` [IE18b; IS09]. Kernel threads, however, work in the kernel space and are created by the kernel itself. They reside solely in the kernel space. Nevertheless, similar to user threads, they can be scheduled and preempted. The only difference of kernel threads is that they do not have a limited address space [Lo10].

Service

A service is a piece of software that is reusable to perform a specific work. According to Russell [Ru14], a *service* is a combination of reusable software functionalities and the policies that define its usage. In this work, a service is the wrapper of a program that defines the program’s behaviours and how it should be executed. One program may have several service definitions,

each with a different execution behaviour. A program cannot be executed in an Edge Server if it does not have a service definition for it. A service has several parameters to be set, based on the software characteristics such as execution duration, relative deadline, execution capacity, Central Processing Unit (CPU) affinities, etc. A service also announces the Edge Server that this program exists and is available for use. These parameters will be discussed in further detail in Sec. 4.2.5.

The architecture defines three types of services: Legacy, Simple, and Simple Periodic. Legacy and Simple services assume that the program runs once, but they slightly differ from each other. Once the Legacy services are preempted, their execution starts from the beginning. However, Simple services can resume execution after being preempted, and their runtime since the beginning is remembered. As the name suggests, Periodic services assume the program runs repeatedly, and each repetition starts after its period. They are also preemptible. More details on services and their types are explained in Sec. 4.2.

Task

The definition of the task is ambiguous. It may mean a process, a thread, the process of a thread, or a set of threads.

In this thesis, *tasks* are the individual running instances of services; hence, the programs. They carry out requests that are defined by services. Tasks may request execution of a single command or process, or multiple processes.

Another term that falls under the task category is ”multitasking.” Multitasking is an ability to run multiple tasks by switching between the tasks during their life cycles. This is different from parallelism; hence, it does not necessarily require multiple cores. During multitasking, only a single task is active, and the other tasks are paused or suspended. Task switching is performed seamlessly without the user noticing.

Jobs

In real-life, a job is a set of tasks [Ca20b]. However, in IT, the definition of the job is ambiguous. Based on the operating system (OS) concepts, a job may also mean a set of tasks or each step of a task. In this thesis, the *job* has the same meaning as the task.

Bibliography

Print Resources

[AA06] Adams, K.; Agesen, O.: A Comparison of Software and Hardware Techniques for x86 Virtualization. SIGARCH Comput. Archit. News 34/5, pp. 2–13, Oct. 2006, ISSN: 0163-5964, URL: http://doi.acm.org/10.1145/1168919.1168860.

[APZ18] Ai, Y.; Peng, M.; Zhang, K.: Edge computing technologies for Internet of Things: a primer. Digital Communications and Networks 4/2, pp. 77–86, 2018, ISSN: 2352-8648.

[Ar10] Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; Zaharia, M.: A View of Cloud Computing. Commun. ACM 53/4, pp. 50–58, Apr. 2010, ISSN: 0001-0782, URL: http://doi.acm.org/10.1145/1721654.1721672.

[Au91] Audsley, N. C.; Burns, A.; Richardson, M. F.; Wellings, A. J.: Hard Real-Time Scheduling: The Deadline-Monotonic Approach. IFAC Proceedings Volumes 24/2, pp. 127–132, 1991, ISSN: 1474-6670.

[AWG03] Albert, A.; Wolter, B.; Gerth, W.: Distinctness of Reaction - Ein Messverfahren zur Beurteilung von Echtzeitsystemen (Teil 2) (Distinctness of Reaction - A Method for the Evaluation of Real-Time Systems (Part 2)). At-automatisierungstechnik - AT-AUTOM 51/, pp. 445–452, Oct. 2003.

[Ba14] Bai, H.: Zen of Cloud. Routledge, Boca Raton, Florida, 2014, ISBN: 978-1-4822-1580-9.

[BC05] Bovet, D. P.; Cesati, M.: Understanding the Linux Kernel, Third Edition. O'Reilly Media, 2005, ISBN: 978-0131876712.

[BK01] Bampis, E.; Kono11, A.: On the Approximability of Scheduling Multiprocessor Tasks with Time-dependent Processor and Time Requirements. In: Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium. IPDPS '01, IEEE Computer Society, Washington, DC, USA, pp. 200–, 2001, ISBN: 0-7695-0990-8, URL: http://dl.acm.org/citation.cfm?id=645609.662324.

[Bo12] Bonomi, F.; Milito, R.; Zhu, J.; Addepalli, S.: Fog Computing and Its Role in the Internet of Things. en,/, p. 3, 2012.

[Br16] Bruneo, D.; Distefano, S.; Longo, F.; Merlino, G.; Puliafito, A.; D'Amico, V.; Sapienza, M.; Torrisi, G.: Stack4Things as a fog computing platform for Smart City applications. In: 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). Pp. 848–853, Apr. 2016.

[Br89] Braden, R.: RFC-1122 Requirements for Internet Hosts – Communication Layers, Internet Standard, [retrieved: Jan 2021], Internet Engineering Task Force, Oct. 1989, URL: https://tools.ietf.org/html/rfc1122.

[Bu95] Burchard, A.; Liebeherr, J.; Yingfeng Oh; Son, S. H.: New strategies for assigning real-time tasks to multiprocessor systems. IEEE Transactions on Computers 44/12, pp. 1429–1442, Dec. 1995.

[Ca84] Carlow, G. D.: Architecture of the Space Shuttle Primary Avionics Software System. Commun. ACM 27/9, pp. 926–936, Sept. 1984, ISSN: 0001-0782, URL: http://doi.acm.org/10.1145/358234.358258.

[CDO17] Cozzolino, V.; Ding, A. Y.; Ott, J.: FADES: Fine-Grained Edge Offloading with Unikernels. In: Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems. HotConNet '17, ACM, Los Angeles, CA, USA, pp. 36–41, 2017, ISBN: 978-1-4503-5058-7.

[Ch14] Chang, H.; Hari, A.; Mukherjee, S.; Lakshman, T. V.: Bringing the cloud to the edge. In: 2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). Pp. 346–351, Apr. 2014.

[Ch17] Choi, N.; Kim, D.; Lee, S.; Yi, Y.: A Fog Operating System for User-Oriented IoT Services: Challenges and Research Directions. IEEE Communications Magazine 55/8, pp. 44–51, Aug. 2017, ISSN: 1558-1896.

[Co12] Corbett, J. C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J. J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; Hsieh, W.; Kanthak, S.; Kogan, E.; Li, H.; Lloyd, A.; Melnik, S.; Mwaura, D.; Nagle, D.; Quinlan, S.; Rao, R.; Rolig, L.; Saito, Y.; Szymaniak, M.; Taylor, C.; Wang, R.; Woodford, D.: Spanner: Google's Globally-distributed Database. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. OSDI'12, USENIX Association, Hollywood, CA, USA, pp. 251–264, 2012, ISBN: 978-1-931971-96-6, URL: http://dl.acm.org/citation.cfm?id=2387880.2387905.

[Co18] Coutinho, A.; Greve, F.; Prazeres, C.; Cardoso, J.: Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing. In: 2018 IEEE International Conference on Communications (ICC). Pp. 1–7, May 2018.

[CW97] Cohen, A.; Woodring, M.: Win32 Multithreaded Programming. O'Reilly Media, New York, NY, USA, 1997, ISBN: 1565922964.

[DD17] Dolui, K.; Datta, S. K.: Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In: 2017 Global Internet of Things Summit (GIoTS). IEEE, Geneva, Switzerland, pp. 1–6, June 2017, ISBN: 978-1-5090-5873-0, URL: http://ieeexplore.ieee.org/document/8016213/.

[Di02] Dilley, J.; Maggs, B.; Parikh, J.; Prokop, H.; Sitaraman, R.; Weihl, B.: Globally distributed content delivery. IEEE Internet Computing 6/5, pp. 50–58, 2002.

[DL78] Dhall, S. K.; Liu, C. L.: On a Real-Time Scheduling Problem. Operations Research 26/1, pp. 127–140, 1978.

[DS12] Dazhong Wu J. Lane Thames, D. W. R.; Schaefer, D.: Towards a Cloud-Based Design and Manufacturing Paradigm: Looking Backward, Looking Forward. In: ASME 2012 Computers and Information in Engineering Conference. Vol. 2, pp. 315–328, Aug. 2012, ISBN: 978-0791845011.

[DS13] Dazhong Wu Matthew J. Greer, D. W. R.; Schaefer, D.: Cloud Manufacturing: Drivers, Current Status, and Future Trends. In: ASME 2013 International Manufacturing Science and Engineering Conference. June 2013, ISBN: 978-0791855461.

[EBS17] Elbamby, M. S.; Bennis, M.; Saad, W.: Proactive edge computing in latency-constrained fog networks. In: 2017 European Conference on Networks and Communications (EuCNC). Pp. 1–6, June 2017.

[ELA94] El-Rewini, H.; Lewis, T. G.; Ali, H. H.: Task Scheduling in Parallel and Distributed Systems. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994, ISBN: 0-13-099235-6.

[Fe09] Feldhorst, S.; Libert, S.; ten Hompel, M.; Krumm, H.: Integration of a legacy automation system into a SOA for devices. Proceedings of the IEEE Conference on Emerging Technologies Factory Automation/, pp. 1–8, 2009.

[Fl89] Flaviu, C.: Probabilistic clock synchronization. Springer - Distributed Computing 3/3, Aug. 1989, ISSN: 0178-2770.

[Fr14] Frotzscher, A.; Wetzker, U.; Bauer, M.; Rentschler, M.; Beyer, M.; Elspass, S.; Klessig, H.: Requirements and current solutions of wireless communication in industrial automation. In: 2014 IEEE International Conference on Communications Workshops (ICC). Pp. 67–72, June 2014.

[Ga99] Garfinkel, S. L.: Architects of the Information Society: Thirty-five Years of the Laboratory for Computer Science at MIT. MIT Press, 1999, ISBN: 0262071967.

[Ge09] Gemebr, A.: Real-Time TCP for Embedded Devices. In: Marquette University, Dept. of Mathematics, Statistics, and Computer Science. 2009.

[Ge19] Gezer, V.; Zietsch, J.; Weinert, N.; Ruskowski, M.: A Field Study: The Perception of Edge Computing for Production Industry. In: Thirteenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies. International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM-2019), Special Session on Edge Computing. International Academy, Research, and Industry Association (IARIA), IARIA, Sept. 2019, ISBN: 978-1-61208-736-8.

[Gi14] Givehchi, O.; Imtiaz, J.; Trsek, H.; Jasperneite, J.: Control-as-a-service from the cloud: A case study for using virtualized PLCs. In: 2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014). Pp. 1–4, May 2014.

[GJ79] Garey, M. R.; Johnson, D. S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA, 1979, ISBN: 0716710447.

[GJ90] Goldhar, J. D.; Jelinek, M.: Manufacturing as a service business: CIM in the 21st century. Computers in Industry 14/1, Special Issue Josef Hartvany Memorial, pp. 225–245, 1990, ISSN: 0166-3615.

[GMS15] Goldschmidt, T.; Murugaiah, M. K.; Sonntag, C.: Cloud-Based Control: A Multi-Tenant, Horizontally Scalable Soft-PLC. In: IEEE 8th International Conference on Cloud Computing. 2015.

[Go17] Goldin, E.; Feldman, D.; Georgoulas, G.; Castano, M.; Nikolakopoulos, G.: Cloud computing for big data analytics in the Process Control Industry. In: 2017 25th Mediterranean Conference on Control and Automation (MED). Pp. 1373–1378, July 2017.

[Gr69] Graham, R. L.: Bounds on Multiprocessing Timing Anomalies. SIAM Journal on Applied Mathematics 17/2, pp. 416–429, 1969.

[GUR18] Gezer, V.; Um, J.; Ruskowski, M.: An Introduction to Edge Computing and A Real-Time Capable Server Architecture. The International Journal on Advances in Intelligent Systems 11(1&2)/, pp. 105–114, July 2018.

[Ha14] Ha, K.; Chen, Z.; Hu, W.; Richter, W.; Pillai, P.; Satyanarayanan, M.: Towards Wearable Cognitive Assistance. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services. MobiSys '14, ACM, Bretton Woods, New Hampshire, USA, pp. 68–81, 2014, ISBN: 978-1-4503-2793-0, URL: http://doi.acm.org/10.1145/2594368.2594383.

[Ha16] Harshit, G.; Dastjerdi, A. V.; Ghost, S. K.; Buyya, R.: iFogSim: A Toolkit for Modeling and Simulation of Resource Management Techniques in Internet of Things, Edge and Fog Computing Environments. In: Software Practive and Experience. June 2016.

[HCK11] Hwang, M.; Choi, D.; Kim, P.: Least Slack Time Rate First: an Efficient Scheduling Algorithm for Pervasive Computing Environment. J. UCS 17/, pp. 912–925, Jan. 2011.

[He61] Heller, J.: Sequencing Aspects of Multiprogramming. J. ACM 8/3, pp. 426–439, July 1961, ISSN: 0004-5411, URL: http://doi.acm.org/10.1145/321075.321086.

[HHG16] Holm, H. H.; Hjelmervik, J. M.; Gezer, V.: CloudFlow - An Infrastructure for Engineering Workflows in the Cloud. In: UBICOMM 2016: The Tenth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies. IARIA, pp. 158–165, Oct. 2016.

[HK16] Horn, C.; Krüger, J.: Feasibility of connecting machinery and robots to industrial control services in the cloud. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). Pp. 1–4, Sept. 2016.

[Ho08] Hoopes, J.: Virtualization for Security. Syngress, 2008, ISBN: 978-1-59749-305-5.

[HPO16] Hermann, M.; Pentek, T.; Otto, B.: Design Principles for Industrie 4.0 Scenarios. In: 2016 49th Hawaii International Conference on System Sciences (HICSS). ISSN: 1530-1605, pp. 3928–3937, Jan. 2016.

[IE08] IEEE: IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, tech. rep., July 2008.

[IE18a] IEEE: Standard for Adoption of OpenFog Reference Architecture for Fog Computing, tech. rep., June 2018.

[IE18b] IEEE: The Open Group Base Specifications Issue 7, 2018 edition/IEEE Std 1003.1-2017, Standard, [retrieved: Jan 2021], The Open Group, 2018.

[IE19] IEC: IEC 61131: Programmable controllers - ALL PARTS, tech. rep., Apr. 2019.

[IS09] ISO: ISO9945 Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7, Standard, [retrieved: Jan 2021], International Organization for Standardization, Sept. 2009.

[IS94] ISO: ISO7498 Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, Standard, [retrieved: Jan 2021], International Organization for Standardization, Nov. 1994.

[ISB86] Iqbal, M. A.; Saltz, J. H.; Bokhari, S. H.: Performance Tradeoffs in Static and Dynamic Load Balancing Strategies, Contractor Report, NASA Aeronautics and Space Administration, Mar. 1986, URL: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19860014876.pdf.

[Ja55] Jackson, J. R.: Scheduling a Production Line to Minimize Maximum Tardiness./, July 1955.

[JSM91] Jeffay, K.; Stanat, D.; Martel, C.: On non-preemptive scheduling of period and sporadic tasks. In: [1991] Proceedings Twelfth Real-Time Systems Symposium. IEEE Comput. Soc. Press, San Antonio, TX, USA, pp. 129–139, 1991, ISBN: 978-0-8186-2450-6, URL: http://ieeexplore.ieee.org/document/160366/.

[KDN17] Kendrick, B. A.; Dhokia, V.; Newman, S. T.: Strategies to realize decentralized manufacture through hybrid manufacturing platforms. Robotics and Computer-Integrated Manufacturing 43/, pp. 68–78, 2017, ISSN: 0736-5845, URL: http://www.sciencedirect.com/science/article/pii/S073658451530137X.

[KK12] Khera, I.; Kakkar, A.: Comparative Study of Scheduling Algorithms for Real Time Environment. In. 2012.

[KLB06] Kaldewey, T.; Lin, C.; Brandt, S.: Firm real-time processing in an integrated real-time system. REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS 398/, p. 5, 2006.

[KM15] Khaitan, S. K.; McCalley, J. D.: Design Techniques and Applications of Cyberphysical Systems: A Survey. en, IEEE Systems Journal 9/2, pp. 1–16, 2015, ISSN: 1932-8184, URL: https://www.academia.edu/23178627/Design_Techniques_and_Applications_of_Cyber_Physical_Systems_A_Survey.

[Ko09] Kordon, F.: Reliable Software Technologies - Ada-Europe 2009: 14th Ada-Europe International Conference. Springer, 2009, ISBN: 978-3642019234.

[Ko16] Konieczek, B.; Rethfeldt, M.; Golatowski, F.; Timmermann, D.: A Distributed Time Server for the Real-Time Extension of CoAP. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC). Pp. 84–91, May 2016.

[KO87] Kopetz, H.; Ochsenreiter, W.: Clock Synchronization in Distributed Real-Time Systems. IEEE Transactions on Computers C-36/8, pp. 933–940, Aug. 1987, ISSN: 0018-9340.

[KY08] Kato, S.; Yamasaki, N.: Global EDF-Based Scheduling with Efficient Priority Promotion. In: 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. Pp. 197–206, Aug. 2008.

[LDG04] Lopez, J.; Díaz, J.; Garcia, F. D.: Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. Parallel and Distributed Systems, IEEE Transactions on 15/, pp. 642–653, Aug. 2004.

[LGJ16] Lera, I.; Guerrero, C.; Juiz, C.: YAFS: A simulator for IoT scenarios in fog computing. en, 4/, p. 14, 2016.

[LHK03] Lee, W. Y.; Hong, S. J.; Kim, J.: On-line scheduling of scalable real-time tasks on multiprocessor systems. Journal of Parallel and Distributed Computing 63/12, pp. 1315–1324, 2003, ISSN: 0743-7315.

[Li15] Li, B.; Pei, Y.; Wu, H.; Shen, B.: Heuristics to allocate high-performance cloudlets for computation offloading in mobile ad hoc clouds. The Journal of Supercomputing 71/8, pp. 3009–3036, Aug. 2015, ISSN: 1573-0484.

[LL73] Liu, C. L.; Layland, J. W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of the ACM Vol. 20/, 1973.

[Lo10] Love, R.: Linux Kernel Development (Developer's Library) (3rd Edition). Addison-Wesley Professional, 2010, ISBN: 978-0672329463.

[LR80] Lampson, B. W.; Redell, D. D.: Experience with Processes and Monitors in Mesa. Commun. ACM 23/2, pp. 105–117, Feb. 1980, ISSN: 0001-0782, URL: http://doi.acm.org/10.1145/358818.358824.

[LS10] Li, B.; Shilong, W.: Cloud manufacturing: A new service to oriented networked manufacturing model. English, Computer Integrated Manufacturing Systems 16/1, pp. 1–7, Jan. 2010.

[LSS87] Lehoczky, J. P.; Sha, L. R.; Strosnider, J. K.: Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In: Unknown Host Publication Title. IEEE, pp. 261–270, 1987.

[LW82] Leung, J. Y.-T.; Whitehead, J.: On the complexity of fixed-priority scheduling of periodic, real-time tasks. Performance Evaluation 2/4, pp. 237–250, 1982, ISSN: 0166-5316.

[Ma04] Maroti, M.; Kusy, B.; Simon, G.; Ledeczi, A.: The Flooding Time Synchronization Protocol. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems. SenSys '04, ACM, Baltimore, MD, USA, pp. 39–49, 2004, ISBN: 1-58113-879-2, URL: http://doi.acm.org/10.1145/1031495.1031501.

[Ma08] Mauerer, W.: Professional Linux Kernel Architecture. Wiley, 2008, ISBN: 978-0470343432.

[Ma17] Mayer, R.; Graser, L.; Gupta, H.; Saurez, E.; Ramachandran, U.: EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures. In: 2017 IEEE Fog World Congress (FWC). Pp. 1–6, Oct. 2017.

[MC70] Muntz, R. R.; Coffman Jr., E. G.: Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems. J. ACM 17/2, pp. 324–338, Apr. 1970, ISSN: 0004-5411, URL: http://doi.acm.org/10.1145/321574.321586.

[Me05] Menon, A.; Santos, J. R.; Turner, Y.; Janakiraman, G. (; Zwaenepoel, W.: Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. VEE '05, ACM, Chicago, IL, USA, pp. 13–23, 2005, ISBN: 1-59593-047-7, URL: http://doi.acm.org/10.1145/1064979.1064984.

[MG11] Mell, P.; Grance, T.: The NIST Definition of Cloud Computing. In: National Institute of Standards and Technology Technical report. Sept. 2011.

[MM98] Manimaran, G.; Murthy, C. S. R.: An efficient dynamic scheduling algorithm for multiprocessor real-time systems. IEEE Transactions on Parallel and Distributed Systems 9/3, pp. 312–319, Mar. 1998.

[MMR98] Manimaran, G.; Murthy, C. S. R.; Ramamritham, K.: A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. Real-Time Systems 15/1, pp. 39–60, 1998, ISSN: 09226443.

[Mo17] Mohamed, N.; Lazarova-Molnar, S.; Jawhar, I.; Al-Jaroodi, J.: Towards Service-Oriented Middleware for Fog and Cloud Integrated Cyber Physical Systems. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). Pp. 67–74, June 2017.

[MSV14] Mahmud, N.; Sandström, K.; Vulgarakis, A.: Evaluating industrial applicability of virtualization on a distributed multicore platform. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA). Pp. 1–8, Sept. 2014.

[MUK00] Mehrabi, M. G.; Ulsoy, A. G.; Koren, Y.: Reconfigurable manufacturing systems: Key to future manufacturing. en, Journal of Intelligent Manufacturing 11/4, pp. 403–419, Aug. 2000, ISSN: 1572-8145, URL: https://doi.org/10.1023/A:1008930403506.

[MWT19] McChesney, J.; Wang, N.; Tanwer, A.: DeFog: Fog Computing Benchmarks. en,/, p. 13, 2019.

[Ne17] Networks, F.: Load Balancing 101: Nuts and Bolts, White paper, May 2017, URL: https://f5.com/resources/white-papers/load-balancing-101-nuts-and-bolts-31392.

[No97] Noble, B. D.; Satyanarayanan, M.; Narayanan, D.; Tilton, J. E.; Flinn, J.; Walker, K. R.: Agile Application-aware Adaptation for Mobility. SIGOPS Oper. Syst. Rev. 31/5, pp. 276–287, Oct. 1997, ISSN: 0163-5980.

[O-97] O-Hoon Kwon; Jong Kim; SungJe Hong; Sunggu Lee: Real-time job scheduling in hypercube systems. In: Proceedings of the 1997 International Conference on Parallel Processing (Cat. No.97TB100162). Pp. 166–169, Aug. 1997.

[OD08] Olderog, E.; Dierks, H.: Real-Time Systems: Formal Specification and Automatic Verification. Cambridge University Press, 2008, ISBN: 978-1139474603.

[Pa09] Patnaik, D.; Krishnaku03, A. S.; Krishnan, P.; Singh, N.; Yajnik, S.: Performance Implications of Hosting Enterprise Telephony Applications on Virtualized Multi-core Platforms. In: Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications. IPTComm '09, ACM, Atlanta, Georgia,

8:1–8:11, 2009, ISBN: 978-1-60558-767-7, URL: http://doi.acm.org/10.1145/1595637.1595648.

[Pa18] Pallasch, C.; Wein, S.; Hoffmann, N.; Obdenbusch, M.; Buchner, T.; Waltl, J.; Brecher, C.: Edge Powered Industrial Control: Concept for Combining Cloud and Automation Technologies. In: 2018 IEEE International Conference on Edge Computing (EDGE). Pp. 130–134, July 2018.

[PG74] Popek, G. J.; Goldberg, R. P.: Formal Requirements for Virtualizable Third Generation Architectures. Commun. ACM 17/7, pp. 412–421, July 1974, ISSN: 0001-0782, URL: http://doi.acm.org/10.1145/361011.361073.

[PH13] Patterson, D. A.; Hennessy, J. L.: Computer Organization and Design MIPS Edition. Morgan Kaufmann, 2013, ISBN: 978-0124077263.

[Po81] Postel, J.: Transmission Control Protocol, Internet Standard, [retrieved: Jan 2021], Internet Engineering Task Force, Sept. 1981, URL: %7Bhttps://tools.ietf.org/html/rfc793%7D.

[PY98] Park, J.; Yoon, Y.: An extended TCP/IP protocol for real-time local area networks. Control Engineering Practice 6/1, pp. 111–118, 1998, ISSN: 0967-0661.

[Ra98] Rajagopalan, S.; Pinilla, J.; Losleben, P.; Tian, Q.; Gupta, S.: Integrated Design and Rapid Manufacturing over the Internet. Proceedings of DETC98 17/2, Sept. 1998.

[Re15] Ren, L.; Zhang, L.; Tao, F.; Zhao, C.; Chai, X.; Zhao, X.: Cloud manufacturing: from concept to practice. Enterprise Information Systems 9/2, pp. 186–209, 2015.

[Ri60] Richards, P. B.: Timing properties of multiprocessor systems. In. Aug. 1960.

[Ru14] Russell, T.: The IP Multimedia Subsystem: Session Control and Other Network Operations. CTI Reviews, 2014, ISBN: 978-1467279840.

[Sa17] Satyanarayanan, M.: The Emergence of Edge Computing. Computer 50/1, pp. 30–39, Jan. 2017.

[Sa96] Satyanarayanan, M.: Fundamental challenges in mobile computing. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. PODC '96, Association for Computing Machinery, Philadelphia, Pennsylvania, USA, pp. 1–7, May 1996, ISBN: 978-0-89791-800-8, URL: https://doi.org/10.1145/248052.248053.

[SB17] Shinde, V.; Biday, S. C.: Comparison of Real Time Task Scheduling Algorithms. In. 2017.

[SB96] Spuri, M.; Buttazzo, G.: Scheduling aperiodic tasks in dynamic priority systems. Real-Time Systems 10/2, pp. 179–210, Mar. 1996, ISSN: 1573-1383.

[SBG09] Silberschatz, A.; B., G. P.; Gayne, G.: Operating System Concepts with Java, 8th Edition. Wiley, 2009, ISBN: 978-0470509494.

[Sh16] Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L.: Edge Computing: Vision and Challenges. en, IEEE Internet of Things Journal 3/5, pp. 637–646, Oct. 2016, ISSN: 2327-4662.

[SJ18] Siderska, J.; Jadaan, K.: Cloud manufacturing: a service-oriented manufacturing paradigm. English, Engineering Management in Production and Services 10/1, pp. 22–31, 2018.

[SLR87] Sha, L. R.; Lehoczky, J. P.; Rajku03, R.: Task Scheduling in Distributed Real-Time Systems. In: Unknown Host Publication Title. IEEE, pp. 909–916, 1987.

[SOE17a] Sonmez, C.; Ozgovde, A.; Ersoy, C.: EdgeCloudSim: An environment for performance evaluation of Edge Computing systems. In: 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC). Pp. 39–44, May 2017.

[SOE17b] Sonmez, C.; Ozgovde, A.; Ersoy, C.: Performance evaluation of single-tier and two-tier cloudlet assisted applications. In: 2017 IEEE International Conference on Communications Workshops (ICC Workshops). Pp. 302–307, May 2017.

[SSL89] Sprunt, B.; Sha, L.; Lehoczky, J.: Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System, tech. rep. CMU/SEI-89-TR-011, Software Engineering Institute, Carnegie Mellon University, 1989.

[Ta15] Tavakoli, A.; Cabello, A.; Zukowski, M.; Bourennane, M.: Quantum Clock Synchronization with a Single Qudit. In: Scientific reports. Vol. 5, 2015.

[Vi15] Vick, A.; Horn, C.; Rudorfer, M.; Krüger, J.: Control of robots and machine tools with an extended factory cloud. In: 2015 IEEE World Conference on Factory Communication Systems (WFCS). Pp. 1–4, May 2015.

[Wa10] Wagner, T.; Haußner, C.; Elger, J.; Löwen, U.; Luder, A.: Engineering Processes for Decentralized Factory Automation Systems. In. 2010.

[Wa17] Wang, N.; Varghese, B.; Matthaiou, M.; Nikolopoulos, D. S.: ENORM: A Framework For Edge NOde Resource Management. IEEE Transactions on Services Computing/, pp. 1–1, 2017, ISSN: 2372-0204.

[WAG03] Wolter, B.; Albert, A.; Gerth, W.: Distinctness of Reaction – Ein Messverfahren zur Beurteilung von Echtzeitsystemen (Teil 1) (Distinctness of Reaction – A Method for the Evaluation of Real-Time Systems). At-automatisierungstechnik - AT-AUTOM 51/, pp. 396–403, Sept. 2003.

[We04] Wellings, A.: Concurrent and Real-Time Programming in Java. Wiley, 2004, ISBN: 978-0470844373.

[WL88] Welch, J. L.; Lynch, N.: A new fault-tolerant algorithm for clock synchronization. Information and Computation 77/1, pp. 1–36, 1988, ISSN: 0890-5401.

[Wu13] Wu, D.; Greer, M.; Rosen, D.; Schaefer, D.: Cloud Manufacturing: Strategic Vision and State-of-the-Art. English, Journal of Manufacturing Systems 32/4, pp. 564–579, 2013, ISSN: 0278-6125.

[Xu12] Xu, X.: From cloud computing to cloud manufacturing. Robotics and Computer-Integrated Manufacturing 28/1, pp. 75–86, 2012, ISSN: 0736-5845.

[YDC17] Yuan, M.; Deng, K.; Chaovalitwongse, W. A.: Manufacturing Resource Modeling for Cloud Manufacturing. International Journal of Intelligent Systems 32/4, pp. 414–436, 2017.

[Yi15] Yi, S.; Hao, Z.; Qin, Z.; Li, Q.: Fog Computing: Platform and Applications. In: 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb). Pp. 73–78, Nov. 2015.

[Yi17] Yi, S.; Hao, Z.; Zhang, Q.; Zhang, Q.; Shi, W.; Li, Q.: LAVEA: Latency-Aware Video Analytics on Edge Computing Platform. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). Pp. 2573–2574, June 2017.

[Yo18] Yousefpour, A.; Ishigaki, G.; Gour, R.; Jue, J. P.: On Reducing IoT Service Delay via Fog Offloading. IEEE Internet of Things Journal 5/2, pp. 998–1010, Apr. 2018, ISSN: 2372-2541.

[Ze13] Zeng, L.; Wang, Y.; Shi, W.; Feng, D.: An Improved Xen Credit Scheduler for I/O Latency-Sensitive Applications on Multicores. In: 2013 International Conference on Cloud Computing and Big Data. Pp. 267–274, Dec. 2013.

[Zh10a] Zhang, L.; Guo, H.; Tao, F.; Luo, Y. L.; Si, N.: Flexible management of resource service composition in cloud manufacturing. In: 2010 IEEE International Conference on Industrial Engineering and Engineering Management. Pp. 2278–2282, Dec. 2010.

[Zh10b] Zhang, L.; Luo, Y. L.; Tao, F.; L. Ren, H. G.: Key Technologies for the Construction of Manufacturing Cloud. In: Computer Integrated Manufacturing Systems. Vol. 16. 11, pp. 2510–2520, 2010.

[Zh17] Zhang, W.; Chen, J.; Zhang, Y.; Raychaudhuri, D.: Towards Efficient Edge Cloud Augmentation for Virtual Reality MMOGs. In: Proceedings of the Second ACM/IEEE Symposium on Edge Computing. SEC '17, ACM, San Jose, California, 8:1–8:14, 2017, ISBN: 978-1-4503-5087-7.

[ZT01] Zhang, C.; Tsaoussidis, V.: TCP-real: Improving Real-time Capabilities of TCP over Heterogeneous Networks. In: Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video. NOSSDAV '01,

ACM, Port Jefferson, New York, USA, pp. 189–198, 2001, ISBN: 1-58113-370-7, URL: http://doi.acm.org/10.1145/378344.378371.

[Zu10] Zuehlke, D.: SmartFactory-Towards a factory-of-things. Annual Reviews in Control 34/1, pp. 129–138, 2010, ISSN: 1367-5788.

Online Resources

[Ap07] Apple Inc.: OS X - Affinity API Release Notes for OS X v10.5, [retrieved: Jan 2021], Oct. 2007, URL: https://developer.apple.com/library/archive/releasenotes/Performance/RN-AffinityAPI/index.html.

[Aq10] Aquosa: AQuoSA: Adaptive Quality of Service Architecture, Aug. 2010, URL: http://aquosa.sourceforge.net/documentation-architecture.php.

[Be21] Beckhoff: BECKHOFF New Automation Technology, 2021, URL: https://www.beckhoff.com/en-us/products/ipc/pcs/c60xx-ultra-compact-industrial-pcs/c6015.html.

[Bj20] Björn Brandenburg: Litmus RT About, 2020, URL: https://www.litmus-rt.org/.

[Bu18] Burris, M.: Selecting Between I2C and SPI for Your Project, [retrieved: Jan 2021], Aug. 2018, URL: https://www.lifewire.com/selecting-between-i2c-and-spi-819003.

[Ca20a] Canonical Ltd.: LXC - Introduction, [retrieved: Jan 2021], 2020, URL: https://linuxcontainers.org/lxc/introduction/.

[Ca20b] Cappelli, P.: Tasks vs. Jobs, (Timestamp: 1:26) [retrieved: Jan 2021], 2020, URL: https://www.coursera.org/lecture/wharton-social-human-capital/tasks-vs-jobs-2LrEW.

[CL19] CLOUDS Laboratory: CloudSim: A Framework For Modeling And Simulation Of Cloud Computing Infrastructures And Services, [retrieved: Jan 2021], 2019, URL: http://www.cloudbus.org/cloudsim/.

[Co15] Collins, K.: Know your real-time protocols for IoT apps, [retrieved: Jan 2021], Aug. 2015, URL: https://www.infoworld.com/article/2972143/real-time-protocols-for-iot-apps.html.

[Co19] Comedi: comedi - linux control and measurement device interface, 2019, URL: http://www.comedi.org/hardware.html, [retrieved: Jan 2021].

[CO21] CODESYS: Home - CODESYS Automation Server, en-US, [retrieved: Jan 2021], 2021, URL: https://www.automation-server.com/en/.

[Co82] Comeau, L. W.: CP/40 - The Original of VM/370, Sept. 1982, URL: http://www.garlic.com/~lynn/cp40seas1982.txt.

[De20] Dell: Dell Edge Gateways for IoT | Dell USA, en-US, [retrieved: Jan 2021], 2020, URL: https://www.dell.com/en-us/work/shop/gateways-embedded-computing/sf/edge-gateway.

[Do21] Docker Website: What is a container?, [retrieved: Jan 2021], 2021, URL: https://www.docker.com/resources/what-container.

[DSS21] Dague, S.; Stekloff, D.; Sailer, R.: xm - Xen management user interface, 2021, URL: https://man.cx/xm.

[Ea17] Eagar, M.: What is the difference between decentralized and distributed systems?, en, Nov. 2017, URL: https://medium.com/distributed-economy/what-is-the-difference-between-decentralized-and-distributed-systems-f4190a5c6462.

[Ed20] EdgeX Foundry: EdgeX Foundry Architectural Tenets, [retrieved: Jan 2021], 2020, URL: https://docs.edgexfoundry.org/2.0/.

[Eu21] Eurotech Website: Everyware Software Framework (ESF), [retrieved: Jan 2021], 2021, URL: https://esf.eurotech.com/.

[Ev11] Evans, D.: The Internet of Things - Cisco, [retrieved: Jan 2021], Apr. 2011, URL: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

[Fo19] Folding@Home Website: About Us, [retrieved: Jan 2021], 2019, URL: https://foldingathome.org/about/.

[Gl17] Gleixner, A.-M.: HOWTO setup Linux with PREEMPT_RT properly, [retrieved: Jan 2021], June 2017, URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt%5C_setup.

[HB04] Haas, H.; Brown, A.: Web Services Glossary, [retrieved: Jan 2021], Feb. 2004, URL: https://www.w3.org/TR/ws-gloss/.

[IB17] IBM Cloud Architecture Center: IBM: Internet of Things, [retrieved: Jan 2021], 2017, URL: https://www.ibm.com/cloud/garage/architectures/iotArchitecture/reference-architecture.

[IB68] IBM: IBM System/360 Operating System Remote Job Entry Program Number 360S-RC-536, Nov. 1968, URL: http://www.bitsavers.org/pdf/ibm/360/rje/C30-2006-2_RmJobEntry_Nov68.pdf.

[IB71] IBM: Control Program-67/Cambridge Monitor System (CP-67 /CMS) Version 3.1 CMS Program Logic Manual, Oct. 1971, URL: http://www.bighole.nl/pub/

mirror/www.bitsavers.org/pdf/ibm/360/cms/GY20-0591-1_CMS_PLM_Oct71.pdf.

[In18] Indiana University: What is the history of Microsoft Windows?, Jan. 2018, URL: https://kb.iu.edu/d/abwa#386.

[Ja21] Jansen, P. G.; Mullender, S. J.; Havinga, P. J.; Scholten, H.: Plan 9 from Bell Labs Real Time Capabilities, 2021, URL: http://doc.cat-v.org/plan%5C_9/real%5C_time/.

[JP98] JPL Pathfinder team: What really happened on Mars?, [retrieved: Jan 2021], Feb. 1998, URL: https://trs.jpl.nasa.gov/bitstream/handle/2014/19020/98-0192.pdf?sequence=1&isAllowed=y.

[Ki05] Kiszka, J.: The Real-Time Driver Model and First Applications, 2005, URL: http://www.cs.ru.nl/lab/xenomai/RTDM-and-Applications.pdf.

[Kl21] Klingler-Deiseroth, C.: Industrial IoT for brownfields | B&R Industrial Automation, en, [retrieved: Jan 2021], 2021, URL: https://www.br-automation.com/en/about-us/press-room/technology-highlights/from-your-brownfield-to-the-cloud/.

[Kr16] Kretschmer, F.: Projekt – pICASSO, de-DE, [retrieved: Jan 2021], Sept. 2016, URL: https://industrie40.vdma.org/documents/4214230/21848134/piCASSO_1510147125829.pdf/9a64d3eb-c397-401f-893a-9994c70bcc12.

[Lh05] Lhotka, R.: Should all apps be n-tier?, [retrieved: Jan 2021], 2005, URL: http://www.lhotka.net/weblog/ShouldAllAppsBeNtier.aspx.

[Li18a] Lim, G.: Worst Case Latency Test Scenarios, [retrieved: Jan 2021], Aug. 2018, URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/worstcaselatency.

[Li18b] Linux Foundation: RTLinux Technical Basics, [retrieved: Jan 2021], 2018, URL: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/start.

[Li19a] Linux Foundation: About Linux Kernel, [retrieved: Jan 2021], 2019, URL: https://www.kernel.org/linux.html.

[Li19b] Linux manuals: Linux Programmer's Manual - sched(7), [retrieved: Jan 2021], Mar. 2019, URL: http://man7.org/linux/man-pages/man7/sched.7.html.

[Li20a] Linux Documentation: Real-Time group scheduling, [retrieved: Jan 2021], 2020, URL: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt.

[Li20b] Linux Foundation: What is Kubernetes?, [retrieved: Jan 2021], 2020, URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[Li20c] Linux manuals: gettid(2) - Linux man page, [retrieved: Jan 2021], 2020, URL: https://linux.die.net/man/2/gettid.

[Li20d] Linux manuals: Linux Programmer's Manual - cgroups(7), [retrieved: Jan 2021], Mar. 2020, URL: http://man7.org/linux/man-pages/man7/cgroups.7.html.

[Li21a] Linux Documentation: CFS Bandwidth Control, [retrieved: Jan 2021], 2021, URL: https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt.

[Li21b] Linux Documentation: CPUSETS, [retrieved: Jan 2021], 2021, URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt.

[Ma16] Marr, B.: What Everyone Must Know About Industry 4.0, en, Section: Innovation, June 2016, URL: https://www.forbes.com/sites/bernardmarr/2016/06/20/what-everyone-must-know-about-industry-4-0/.

[Mi14] Microsoft Corp.: Windows Version History, Dec. 2014, URL: https://web.archive.org/web/20141219013444/http://support.microsoft.com/kb/32905.

[Mi18] Microsoft Corp.: SetThreadAffinityMask function (winbase.h) - Win32 apps, en-us, [retrieved: Jan 2021], May 2018, URL: https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setthreadaffinitymask.

[Mo20] Moore, K.: Counting Sort | Brilliant Math & Science Wiki, en-us, [retrieved: Jan 2021], 2020, URL: https://brilliant.org/wiki/counting-sort/.

[Na21] National Science Foundation: Cyber-Physical Systems (CPS) (nsf21551), [retrieved: Jan 2021], Jan. 2021, URL: https://www.nsf.gov/pubs/2021/nsf21551/nsf21551.pdf.

[NC14] NCTA: The Growth of The Internet of Things, [retrieved: Jan 2021], May 2014, URL: https://www.ncta.com/platform/industry-news/infographic-the-growth-of-the-internet-of-things/.

[Op17] OpenFog Consortium: OpenFog Consortium Reference Architecture for Fog Computing, [retrieved: Jan 2021], 2017, URL: https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf.

[Or11] Oracle: Solaris - man pages section 1M: System Administration Commands, [retrieved: Jan 2021], 2011, URL: https://docs.oracle.com/cd/E23824_01/html/821-1462/pbind-1m.html.

[Ox20] Oxford Dictionary: edge: Definition of edge in English by Oxford Dictionaries, [retrieved: Jan 2021], 2020, URL: https://www.lexico.com/definition/edge.

[Pe21] Pearson Education: network: Longman Dictionary of Contemporary English, [retrieved: Jan 2021], 2021, URL: https://www.ldoceonline.com/dictionary/network.

[Re14] Renwick, K.: How to use priority inheritance?, [retrieved: Jan 2021], May 2014, URL: https://www.embedded.com/how-to-use-priority-inheritance/.

[RT21] RTEMS Project: 5. Scheduling Concepts, 2021, URL: https://docs.rtems.org/releases/rtems-docs-4.11.2/c-user/scheduling_concepts.html.

[Ru13] Russell, A. L.: OSI: The Internet That Wasn't, [retrieved: Jan 2021], July 2013, URL: https://spectrum.ieee.org/tech-history/cyberspace/osi-the-internet-that-wasnt.

[SE19] SETI Website: SETI Institute: Mission, [retrieved: Jul 2019], 2019, URL: https://www.seti.org/about-us/mission.

[SH21] SHaRK: S.Ha.R.K. Modules, 2021, URL: http://shark.sssup.it/modules.shtml.

[Sm09] Smith, J.: Harnessing the Power of Cloud Computing for M2M, May 2009, URL: https://connectedworld.com/wp-content/uploads/2014/07/m2mCloudComputing.pdf.

[Sy13] Systems, R.: What really happened to the software on the Mars Pathfinder spacecraft?, [retrieved: Jan 2021], 2013, URL: https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft.

[Th21] The Computer Language Co Inc.: Definition of end device, [retrieved: Jan 2021], 2021, URL: https://www.pcmag.com/encyclopedia/term/64886/end-device.

[To91] Torvalds, L.: Notes for Linux release 0.01, Sept. 1991, URL: https://mirrors.edge.kernel.org/pub/linux/kernel/Historic/old-versions/RELNOTES-0.01.

[TT16] TTTech: TTTech Announces Nerve – a New Product Platform for Edge Computing and Control, en-US, [retrieved: Jan 2021], Nov. 2016, URL: https://www.tttech.com/press/tttech-announces-nerve-a-new-product-platform-for-edge-computing-and-control/.

[VM17] VMware: VMware Introduces Liota, [retrieved: Jan 2021], 2017, URL: https://www.vmware.com/radius/vmware-introduces-liota-iot-developers-dream/.

[Wa14] Watson, B.: The Design of the Everyman Hard Real-time Kernel, 2014, URL: http://www.barrywatson.se/download/Everyman.pdf.

[WW97] Wolf, M.; Wicksteed, C.: Date and Time Formats, en, [retrieved: Jan 2021], 1997, URL: https://www.w3.org/TR/NOTE-datetime.

Lebenslauf

Persönliche Daten

Name:	Volkan Gezer
E-Mail:	volkangezer@gmail.com
Staatangehörigkeit	türkisch
Geburtsdatum	26.02.1989
Geburtsort	Istanbul

Ausbildung

2012-09 – 2014-11	Studium Elektro- und Informationstechnik (M.Sc.) an der TU Kaiserslautern
2009-09 – 2011-06	Studium (seitlich) System Engineering an der Eskisehir Osmangazi University
2009-09 – 2010-02	Studium Electrical and Electronics Engineering (ERASMUS) an der Vilnius Gediminas Technical University
2007-09 – 2012-06	Studium Electrical and Electronics Engineering (B.Sc.) an der Eskisehir Osmangazi University
2003-09 – 2007-07	Gymnasium "A. Rifat Canayakin Lisesi" in der Türkei (Istanbul)

Berufliche Erfahrungen und Praktika

2015-03 –	Researcher im Forschungsbereich Innovative Fabriksysteme beim Deutschen Forschungszentrum für Künstliche Intelligenz GmbH in Kaiserslautern
2012-02 – 2015-02	Wissenschaftliche Hilfskraft bei Fraunhofer ITWM in Kaiserslautern
2011-06 – 2011-08	Praktikant bei CERN (European Organization for Nuclear Research) in Genf, Schweiz
2011-01 – 2011-02	Praktikant bei Turkish Telecommunication AS

VDI verlag

Alle 23 Reihen der „Fortschritt-Berichte VDI“
in der Übersicht – bequem recherchieren unter:
elibrary.vdi-verlag.de

Und direkt bestellen unter:
www.vdi-nachrichten.com/shop

Reihe 01 Konstruktionstechnik / Maschinenelemente
Reihe 02 Fertigungstechnik
Reihe 03 Verfahrenstechnik
Reihe 04 Bauingenieurwesen
Reihe 05 Grund- und Werkstoffe/Kunststoffe
Reihe 06 Energietechnik
Reihe 07 Strömungstechnik
Reihe 08 Mess-, Steuerungs- und Regelungstechnik
Reihe 09 Elektronik/Mikro- und Nanotechnik
Reihe 10 Informatik/Kommunikation
Reihe 11 Schwingungstechnik
Reihe 12 Verkehrstechnik/Fahrzeugtechnik
Reihe 13 Fördertechnik/Logistik
Reihe 14 Landtechnik/Lebensmitteltechnik
Reihe 15 Umwelttechnik
Reihe 16 Technik und Wirtschaft
Reihe 17 Biotechnik/Medizintechnik
Reihe 18 Mechanik/Bruchmechanik
Reihe 19 Wärmetechnik/Kältetechnik
Reihe 20 Rechnergestützte Verfahren
Reihe 21 Elektrotechnik
Reihe 22 Mensch-Maschine-Systeme
Reihe 23 Technische Gebäudeausrüstung

OHNE PROTOTYP GEHT NICHTS IN SERIE.

Unser Podcast ist das Werkzeug, mit dem Sie Ihre Karriere in allen Phasen entwickeln – vom Studium bis zum Chefsessel. Egal, ob Sie Ingenieur*in, Mechatroniker*in oder Wissenschaftler*in sind: Prototyp begleitet Sie. Alle 14 Tage hören Sie die Redaktion von INGENIEUR.de und VDI nachrichten im Gespräch mit prominenten Gästen.

INGENIEUR.de
TECHNIK - KARRIERE - NEWS

PROTOTYP

Karriere-Podcast

JETZT REINHÖREN UND KOSTENFREI ABONNIEREN:
WWW.INGENIEUR.DE/PODCAST

IN KOOPERATION MIT VDI NACHRICHTEN

VDI

REIHE 10
INFORMATIK/
KOMMUNIKATION

NR. 874 **ISBN 987-3-18-387410-1**

BAND
1|1

VOLUME
1|1