technische universität
dortmund

Bachelorthesis

Supporting Edge Computing Simulator for Real-Time Task Models

Roman Sazontow

May 20th, 2020

Supervisors:
Prof. Dr. Jian-Jia Chen
Dr.-Ing. Kuan-Hsun Chen

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Contents

1 Introduction

1.1 Motivation

Edge computing is a fast growing field of research and is becoming more relevant recently. Due to increasing data size, *cloud computing* is not as efficient anymore. It is faster and more efficient to process the data at the edge of the network rather than in the cloud. In order to test simulations in such environments, *EdgeCloudSim* was developed by C. Sonmez et al. [13] *EdgeCloudSim* is an extension of *CloudSim* and serves as a simulation tool for *edge computing* scenarios. It simulates mobile devices that move between various access points and creates tasks that have to be efficiently distributed and executed. In its current design, *EdgeCloudSim* does not support *Real-Time* task models. *Real-Time* is a relevant part of *edge computing*, since *edge computing* is a computing paradigm that offers fast response times, which are important for *Real-Time* scenarios. Similar to normal *edge computing* scenarios, it is desirable to have a simulation tool that can be used to test *Real-Time edge computing* systems. The purpose of this bachelor thesis is to extend *EdgeCloudSim* by adding *Real-Time* support. In order to do so, the necessary *Real-Time* characteristics have to be added to the source code.

1.2 Structure of the thesis

This thesis is organized as follows:

Chapter 2 focuses on the background of this thesis. It will go further into detail on the necessary parts to achieve its goal, such as *cloud computing*, *edge computing*, *Real-Time* systems and relevant scheduling methods.

Chapter 3 presents *EdgeCloudSim* by explaining its purpose and its original design. Additionally, this chapter will present the motivation of this thesis and the goal that has to be achieved.

Chapter 4 focuses on *CloudSim*, the base of *EdgeCloudSim*. In order to achieve the goal of this thesis, it is important to understand both *EdgeCloudSim* and *CloudSim*. For this reason, this chapter will go into detail on how the original design of *CloudSim* is implemented and which parts are relevant for this thesis.

Chapter 5 explains the implementation. This chapter goes into detail on which parts of the source code of *EdgeCloudSim* and *CloudSim* are relevant and have been changed. It also explains how the implementation works and presents the source code. The end of the chapter discusses the solution and its results.

Chapter 6 concludes the thesis. It summarizes the thesis and explains which parts have been focused on and what has been achieved. Furthermore, the presented solution is looked on in terms of how it can be improved. An outlook on future improvements on *Real-Time* support for *EdgeCloudSim* is given by looking into similar works.

2 Background

Henceforth, an introduction to relevant topics for this thesis is given. First, this chapter looks into *Real-Time* systems and its characteristics. The section after that explains what tasks are and, more specifically, what defines a *Real-Time* task. Following that, the topic of *fixed-priority* scheduling is looked into, while presenting the scheduling methods that are utilized in this thesis. The next section goes into detail on *cloud computing* to give the basics for *CloudSim*. Similarly, the section after that explains *edge computing*, which is the computing paradigm utilized in *EdgeCloudSim*. The end of this chapter presents related work to this thesis.

2.1 Real-Time Systems

A *Real-Time* system is one with explicit deterministic or probabilistic timing requirements, as described in [10]. *Real-Time* systems are computing systems, which main characteristic is its requirement to react within a strict time to events in the environment. While doing so, not only is time a relevant factor, but it is also important to produce correct results. In *Real-Time* environments it is very important to satisfy the time constraints, since reactions that take too long can be dangerous.
These days, *Real-Time* computing gains increasing relevance, since the number of systems that rely on computer control is growing. Examples for fields of applications that use *Real-Time* systems include automotive applications and flight control systems, telecommunication and multimedia systems, medical systems etc. Systems that are embedded with *Real-Time* computing can range from small devices (e.g. cellular phones and cameras) to larger systems (e.g. cars and aircraft) [5].

2.2 Real-Time Tasks

A task, sometimes also called a process or thread, is a potentially infinite sequence of jobs [10]. Tasks compete with each other over the control of a CPU. A task can have different states:

- an executing task is called *running*
- a waiting task is called *ready*

- a task that can be executed is *active*

Real-Time systems have strict time constraints, which are realized by tasks through a deadline. Each task has a deadline, which represents the time before it has to finish to not cause any damage. Tasks with a deadline are called *Real-Time* tasks. Depending on how consequential a missed deadline is, *Real-Time* tasks can be distinguished in three categories [5]:

Hard: A *hard Real-Time* task may cause damage to the system, if results are produced after its deadline.

Firm: A *firm Real-Time* task may be useless, if results are produced after its deadline.

Soft: A *soft Real-Time* task has still some utility, if results are produced after its deadline. The performance may degrade, though.

In order to realize *Real-Time* computing, an appropriate scheduling algorithm is required. In *Real-Time* scheduling each task is assigned a priority. Tasks have a release time, a computation time and a deadline. The deadline can either be an absolute time or relative to the release time of a task. In order for the tasks to finish before their deadline, tasks have to be scheduled according to their priorities. The assigned priority depends on the policy used.
In the following, a general characterization of a *Real-Time* task τ_i is given according to [5]:

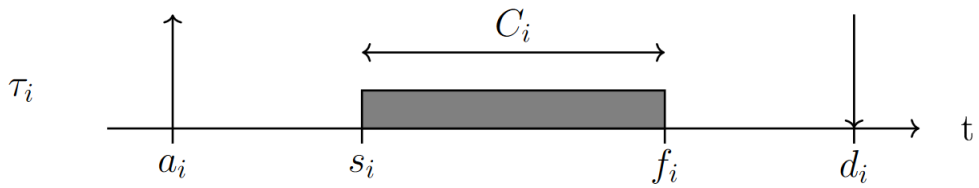

Figure 2.1: Parameters of Real-Time tasks [5]

Arrival time α_i refers to the time at which a task is ready for execution

Computation time C_i is the necessary time to complete the task.

Absolute Deadline d_i is the time before the task has to be finished.

Relative Deadline D_i is the difference between the absolute deadline d_i and the arrival time α_i.

Start time s_i refers to the time at which a task starts its execution.

Finishing time f_i refers to the time at which a task finishes its execution.

Response time R_i is the difference between the finishing time f_i and the arrival time α_i.

2.3 Fixed-Priority Scheduling

Fixed-priority scheduling is a *Real-Time* scheduling policy for handling tasks in a certain order. Each task is assigned a priority depending on the used policy. A task is periodic if it is time-triggered and has a regular release. The time between releases of successive jobs of a task is constant. This is called the period of a task. The deadline of a task is a certain amount of time after its release time [10]. In *fixed-priority* scheduling, the tasks are numbered so that task τ_i has priority i. Value one indicates the highest priority, while larger integers indicate tasks with lower priorities [10]. At all times it is ensured that the task with the highest priority is executed. *Fixed-priority* scheduling is one of the most commonly used scheduling algorithms [4] and its implementation is simple. For those reasons it was chosen as the scheduling algorithm used in this thesis.
Priorities and deadlines of tasks can be assigned in various ways and differ between task models. The task model used in this thesis is the *implicit-deadline* task model. In this model, the deadline of a task depends on its period. More specifically, the expected deadline is the sum of the task's release time and its period.

2.3.1 Rate Monotonic Scheduling

Furthermore, the implementation in this thesis uses the *Rate Monotonic (RM)* scheduling algorithm. This algorithm implies that the priority of a task is higher the lower its period is. *RM* is one of the most used algorithms in industrial systems. It is simple and efficient to implement and intuitive [4].

2.3.2 Preemptive and Non-preemptive scheduling

Scheduling algorithms can be divided in two classes: *preemptive* and *non-preemptive*.

Preemptive: With *preemptive* scheduling, an executed task can at all times be overtaken by another task with a higher priority. That means that the CPU of an assigned task can be taken away and assigned to another task. How and when a preemption occurs depends on the scheduling policy.

Non-preemptive: With *non-preemptive* scheduling, an executed task can not be taken away from its CPU. The next task can only start its execution once the current task has finished, regardless of its priority. Compared to *preemptive* scheduling, *non-preemptive* scheduling has longer response-times [12]. Due to its simplicity though, the implementation in this thesis uses *non-preemptive* scheduling.

2.4 Cloud Computing

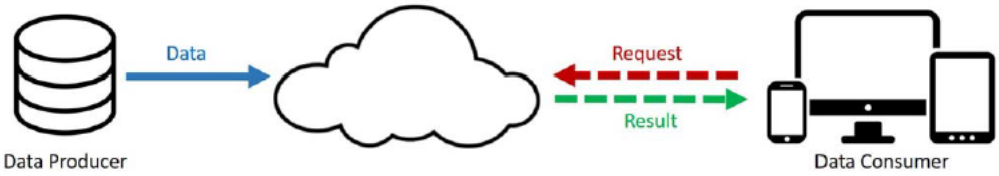

Figure 2.2: Cloud Computing Paradigm [11]

Cloud computing is a computing paradigm that gained increasing interest over the last few years. Since its inception around 2005 it had a big influence on the way we live, work and study [1]. *Cloud computing* allows offering IT-infrastructure, platforms and applications as services to end-users. These services are referred to as *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS) in industries [6] and are offered by providers such as Amazon, Google and Microsoft [2]. The report from the University of Berkeley highlighted the importance of these services as: " *Cloud computing*, the long-held dream of computing as a utility has the potential to transform a large part of the IT industry, making software even more attractive as a service" [1].
With *cloud computing*, the execution of tasks can be relocated to a more powerful server in a cloud. This can improve the performance and is useful to process tasks that are too complex and extensive for devices that are not as powerful. Those tasks include mathematical and imaging tools, face and object recognition, image search, optical character readers, games, etc. [3, 9].
Figure 2.2 shows the conventional structure for the *cloud computing* paradigm. Data producers generate data and send it to a cloud server. Data consumers request results from the cloud (indicated by the red line) and the results produced by the cloud are sent to the data consumers (represented by the green line) [11].
Thanks to *cloud computing*, acquiring powerful hardware is not as necessary anymore. Scalable infrastructures as well as processing engines developed to support cloud service are significantly influencing the way of running business: Google File System [8], MapReduce [7] etc.
In order to simulate *cloud computing* scenarios, *CloudSim* was developed by Calheiros et al. [6] *CloudSim* will be further looked upon in Chapter 4.

2.5 Edge Computing

Edge computing is a new computing paradigm in which data is processed at the edge of the network as opposed to a server in a cloud. This means that the computing services are provided by a server that is near the geographical location of the client. *Edge computing*

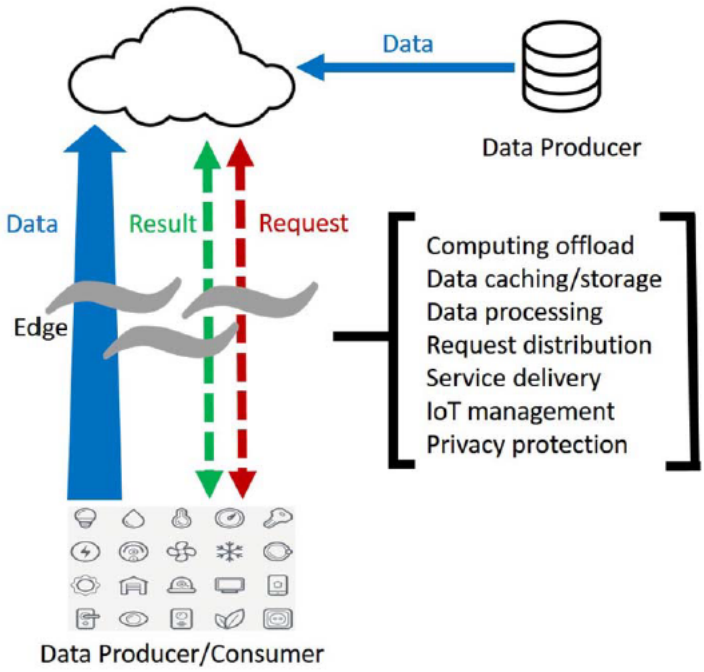

Figure 2.3: Edge Computing Paradigm [11]

proves to be advantageous for time sensitive applications or situations where no cloud is accessible. This is especially beneficial for mobile devices, since it allows them to handle complex operations while consuming less battery [13]. *Cloud computing* is not always efficient enough to support applications that require very short response time or produce a heavy load on networks. With growing quantity of data generated at the edge, speed of data transportation is becoming the bottleneck for cloud-based computing paradigms [11]. This makes it increasingly more efficient to process the data at the edge of the network.

To put it in perspective, about 5 Gigabyte of data can be generated by a Boeing 787 every second [11]. The bandwidth between the airplane and satellite or base station is not sufficient for this amount of data transmission. As another example, autonomous vehicles generate one Gigabyte of data every second. For those kind of scenarios, *Real-Time* processing is needed in order to make fast and correct decisions [11].

In these and many other cases, the response time may be too long when data is sent to the cloud for processing. Processing data at the edge however can shorten the response time, while being more efficient and putting less pressure on the network. Using *edge computing* addresses response time requirement, battery life constraints, bandwidth cost saving and data safety [11]. As shorter response times and better performance are benefits of *edge computing*, it is a useful computing paradigm for *Real-Time* systems.

Figure 2.3 shows the two-way computing streams in *edge computing* paradigms. Compared to *cloud computing*, the data consumers also serve as data producers. This means that

they not only request services from the cloud, but can also perform the computing of tasks from the cloud. The edge of the network can perform computing offloading, data storage, caching and processing, as well as distributing requests and delivering services from the cloud to the users. For those jobs, the edge needs to meet requirements such as reliability, security and privacy protection [11].

2.6 Related Work

The work "Adaptive Quality Optimization of Computer Vision Tasks in Resource Constrained Devices using Edge Computing" by Toma et al. [14] presents an approach to improve the performance of computer vision tasks in resource-constrained devices, such as mobile devices. The general idea is to create different execution versions of one task and compare them with the purpose to find the version that maximizes the total quality based on available resources. The execution versions are generated by reducing the quality of input information to a certain degree. In many cases, the quality can be reduced with having small to no impact on the results.
Toma et al. [14] propose an adaptive algorithm based on dynamic programming. The algorithm utilizes *edge computing* and selects execution configuration and execution location of each task based on the available resources. The presented approach is not only limited to computer vision tasks and improves total computing performance under resource and time constraints. These characteristics make it perfect for *Real-Time* systems.

3 EdgeCloudSim

This chapter focuses on *EdgeCloudSim*. First, an introduction is given, followed by the purpose of the presented simulation tool. The next section presents the original design of *EdgeCloudSim*. At the end of the chapter, the problem is explained, which will be solved in this thesis.

3.1 Introduction

EdgeCloudSim [13] is a simulation tool that is based on *CloudSim* [6]. As *CloudSim* is a tool to simulate *cloud computing*, *EdgeCloudSim* enables the simulation of *edge computing* scenarios. It extends *CloudSim's* design with a modular structure that adds mobilization, network modeling etc.

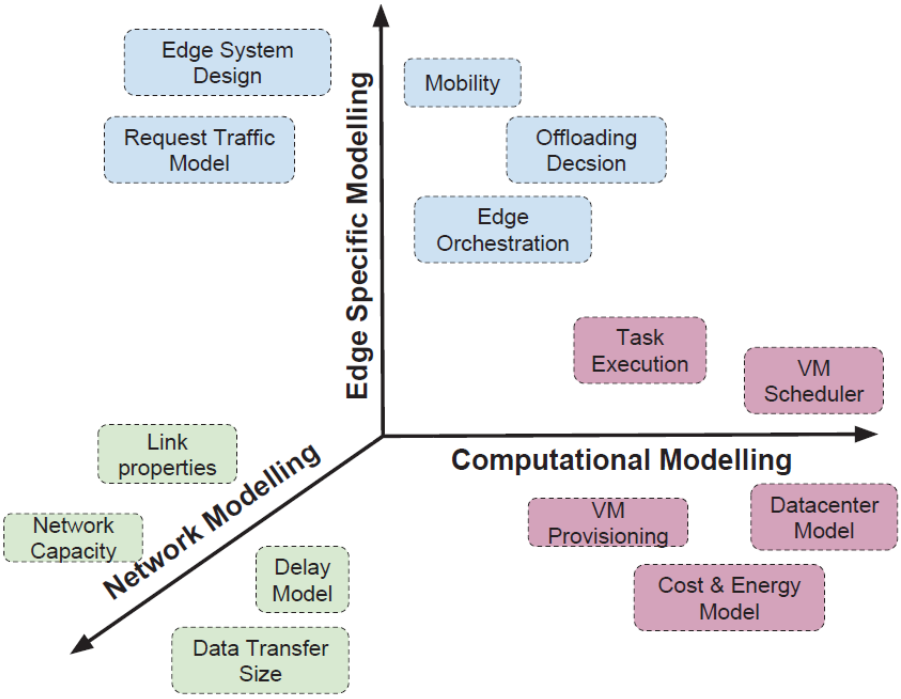

Figure 3.1: Aspects of Edge Computing simulation modeling [13]

3.2 Purpose

Edge computing is a fast growing field of research and involves technically more sophisticated setup than *cloud computing* [13]. Since testing *edge computing* in real life environments is too costly and unreliable, a simulation tool for those scenarios is necessary. *EdgeCloudSim*, developed by C. Sonmez et al. [13], fills that purpose by adding the necessary functionality in terms of computation and networking abilities to *CloudSim* [6]. In its original design, *CloudSim* does not include a WLAN and WAN communication model. It also does not support mobility and has no realistic VM utilization model. *EdgeCloudSim* addresses those lacks and provides the necessary functions to simulate *edge computing* scenarios. In Figure 3.1 the main aspects of *edge computing* simulation modeling are shown. The axes describe different aspects that are addressed by *EdgeCloudSim*.
EdgeCloudSim simulates multi-tier scenarios with multiple edge servers that run in coordination with upper layer cloud servers. Additionally, the Edge Orchestrator module allows the modelling of orchestration actions in the *edge computing* simulations.
The computational tasks, such as VM creation, still happen within the *CloudSim's* source code. *EdgeCloudSim* is open-source and its modular design allows to easily modify the code for the user's own needs.
During simulations, different architectures are examined and the effects of networking and computational parameters on the *edge computing* performance are displayed.

3.3 Original Design

EdgeCloudSim has a modular structure. Each module serves its own specific purpose in *edge computing* scenarios. In this section the various modules of *EdgeCloudSim* are presented according to [13]. The modules that are relevant for this work are explained in further detail. In Figure 3.2 the relationship between the presented modules is displayed.

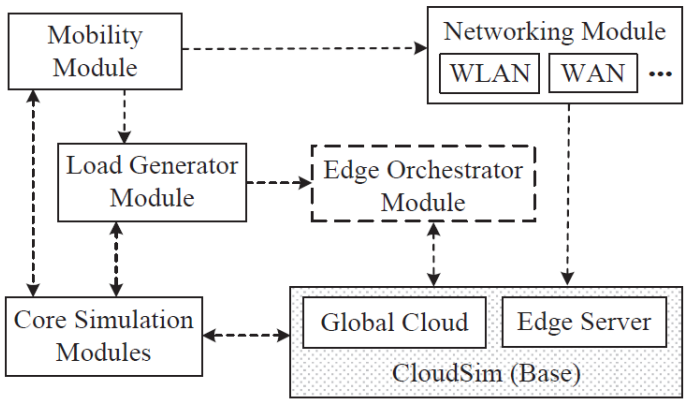

Figure 3.2: Relationship between EdgeCloudSim modules [13]

Core Simulation Module: This module serves as the core of the simulation and is responsible for loading and running the *edge computing* scenarios from the configuration files. It is also responsible for logging the results of the simulations, which are saved in comma-separated value (CSV) data. The logging mechanism is relevant for this thesis, since the implementation adds additional results based on tasks that have failed due to a missed deadline. This will be further explained in Chapter 5.

Networking Module: The Networking Module is responsible for the networking part of a simulation, such as handling transmission delay in the WLAN and WAN. It considers upload and download data. The Networking Module has no relevance for this work and as such has not been changed.

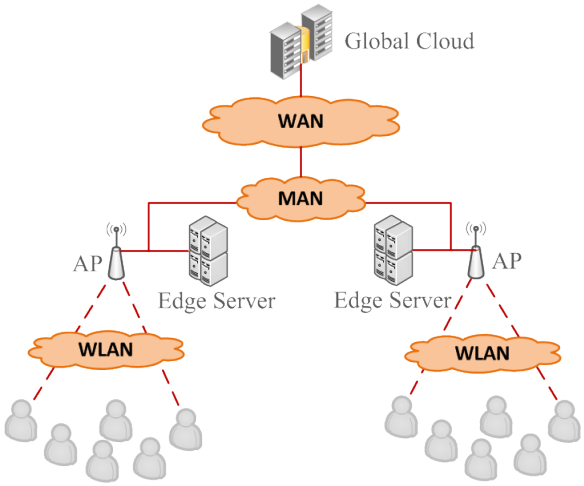

Figure 3.3: Networking Module [13]

Edge Orchestrator Module: The Edge Orchestrator module can be seen as the decision maker of the system. It is connected to the other modules and uses their information to decide where to handle incoming tasks. Since the solution in this work only focuses on how to handle tasks on each VM, the Edge Orchestrator is not relevant and has not been changed.

Mobility Module: The Mobility Module handles the location of the mobile devices in a simulation. Each mobile device and the Wi-Fi access points have x and y coordinates. The coordinates of the mobile devices are updated regularly during the simulation according to a dynamically managed hash table, while the coordinates of the Wi-Fi access points are fixed. This work does not focus on the mobility within a simulation, so the Mobility Module is not relevant.

Load Generator Module: The Load Generator module's purpose is the generation of tasks for the given configuration. The tasks are generated according to a Poisson distribution. Since this work focuses on adding *Real-Time* support, the tasks are a relevant

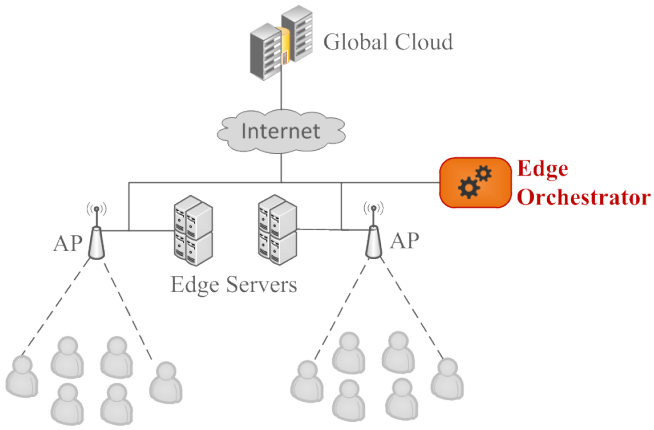

Figure 3.4: Edge Orchestrator Module [13]

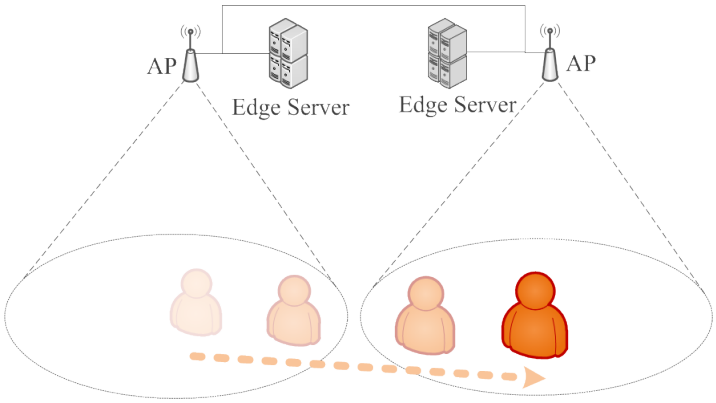

Figure 3.5: Mobility Module [13]

aspect and have to be changed to be suitable for *Real-Time* purposes. The Load Generator module is tied to the mobile device manager, which also has to be changed.

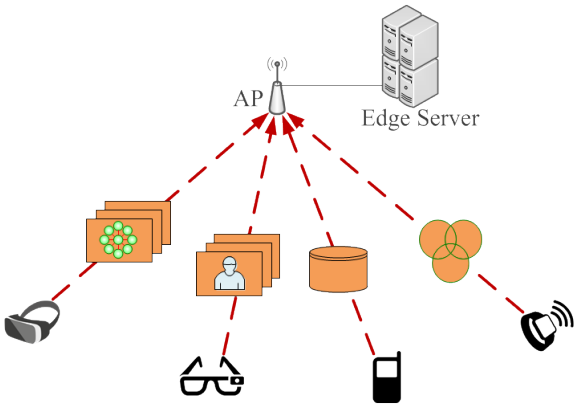

Figure 3.6: Load Generator Module [13]

3.4 Simulation Environment

EdgeCloudSim allows testing the performance of different *edge computing* architectures. In order to do this, a virtual environment is created where mobile devices move around and request services from nearby edge servers.
Three different architectures are compared, which are:

- single-tier
- two-tier
- two-tier with edge orchestrator (EO)

With the *single-tier* architecture, mobile devices utilize the nearby edge server. The *two-tier* architecture allows the mobile devices to send tasks to the global cloud using the WAN connection. The *two-tier with EO* additionally enables the mobile devices to utilize edge servers that are connected to the same network as opposed to only utilizing the nearby edge server. A generic *edge computing* architecture is shown in Figure 3.7.

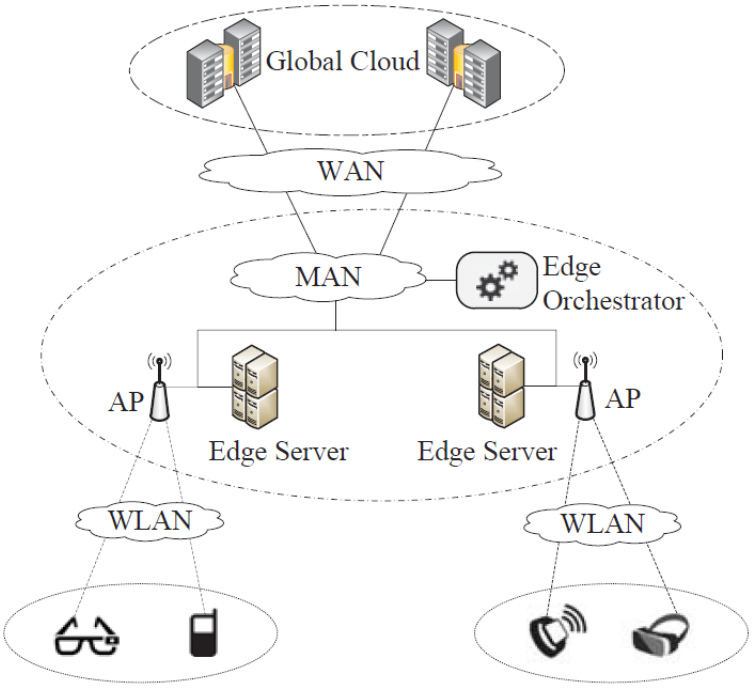

Figure 3.7: Generic Edge Computing architecture [13]

The mobile devices move between Wi-Fi access points that are connected to an edge server. After connecting to the WLAN, the mobile devices start sending to the edge server. In case a task is offloaded to the global cloud, the mobile device uses the WAN connected to the Wi-Fi access point. The tasks are randomly created. In Figure 5.1 some examples for results of *EdgeCloudSim* are given.

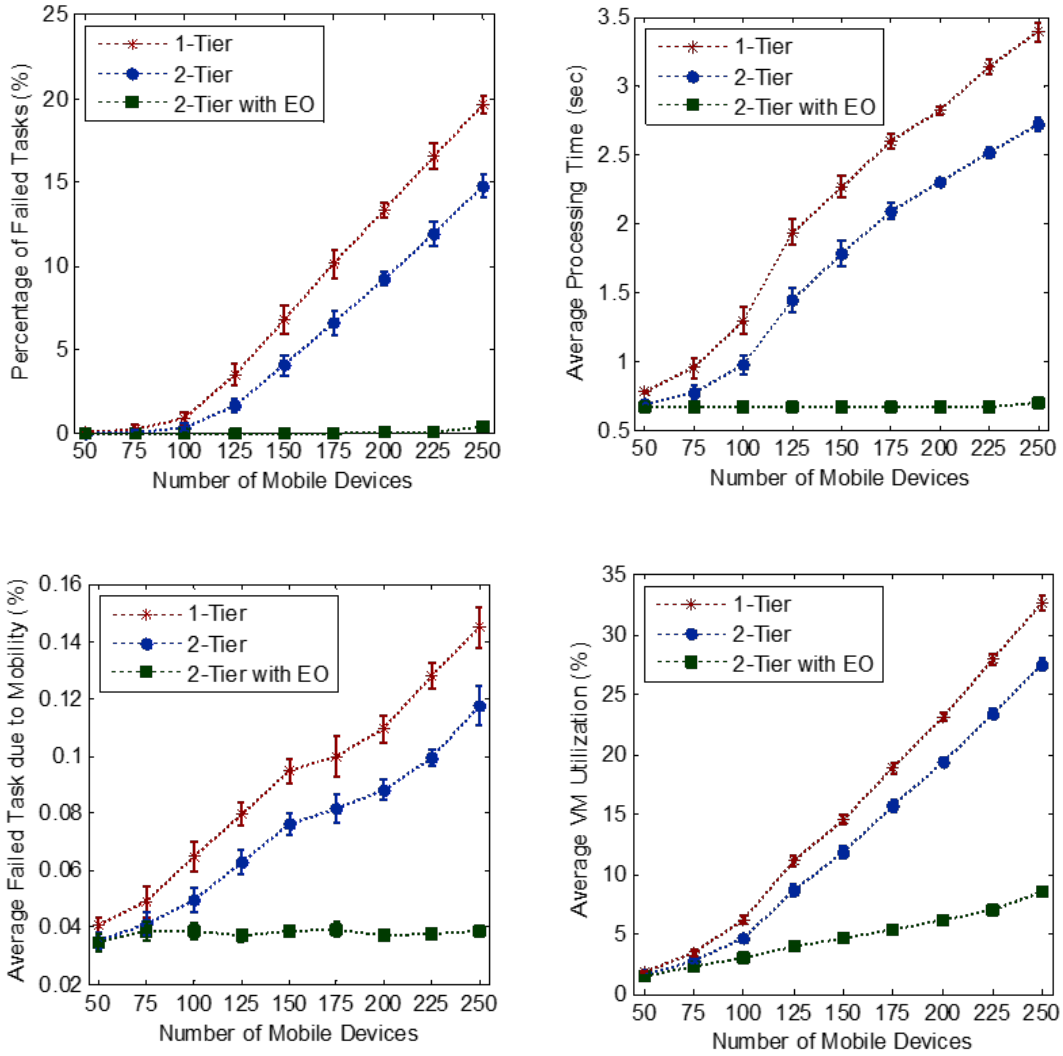

Figure 3.8: EdgeCloudSim result examples [13]

3.5 Problem

In its current design, *EdgeCloudSim* does not support *Real-Time* task models. As *Real-Time* is an important factor for *edge computing* scenarios in real environments (such as autonomous vehicles), it is desirable to have a simulation tool that supports it. In order to make this possible, the goal of this thesis is to add *Real-Time* support to *EdgeCloudSim*. This will be achieved by adding necessary parameters to the tasks generated in *EdgeCloudSim* and then scheduling them accordingly within *CloudSim*. This will be further explained in Chapter 5.

4 CloudSim

In this chapter a look into *CloudSim* is presented. In the first section a brief introduction is given. After that, the original design of *CloudSim* is explained. The last section describes the internal processing to give an idea on how the tasks within *CloudSim* are handled and updated.

4.1 Introduction

CloudSim is an extensible simulation toolkit that enables modeling and simulation of *cloud computing* systems and application provisioning environments [6]. It was developed by Calheiros et al. [6] to help evaluating the performance of *cloud computing* models under various configurations and requirements. Its generic application provisioning techniques allow for easy extensions. Because of that, *CloudSim* served as a base of multiple simulation tools, such as *EdgeCloudSim*, which this thesis focuses on.

Performing tests in a real *cloud computing* environment (e.g. Amazon EC2, Microsoft Azure, Google App Engine [6]) under various configurations is very challenging and time-consuming. There are several limitations for test runs in a real environment, such as:

- Clouds can vary a lot in terms of resources (hardware, software, network), system sizes and demands
- Users have different requirements
- The requirements for applications can differ in terms of performance and workload

For those reasons it is useful to have a simulation tool to perform benchmarking experiments in a controlled environment in which it is possible to easily reproduce results. *CloudSim* is such a simulation tool and offers an extensible simulation framework.

In order to achieve the goal of this thesis, the scheduling of tasks during the simulation of *EdgeCloudSim* has to be adjusted. Since the actual scheduling of the tasks happens within *CloudSim* rather than *EdgeCloudSim*, the source code of *CloudSim* has to be edited as well.

4.2 Original Design

In this section the original design of *CloudSim* is explained according to [6], while going into detail on the parts that are relevant for this work. In Figure 4.1 the overall Class design diagram is shown.

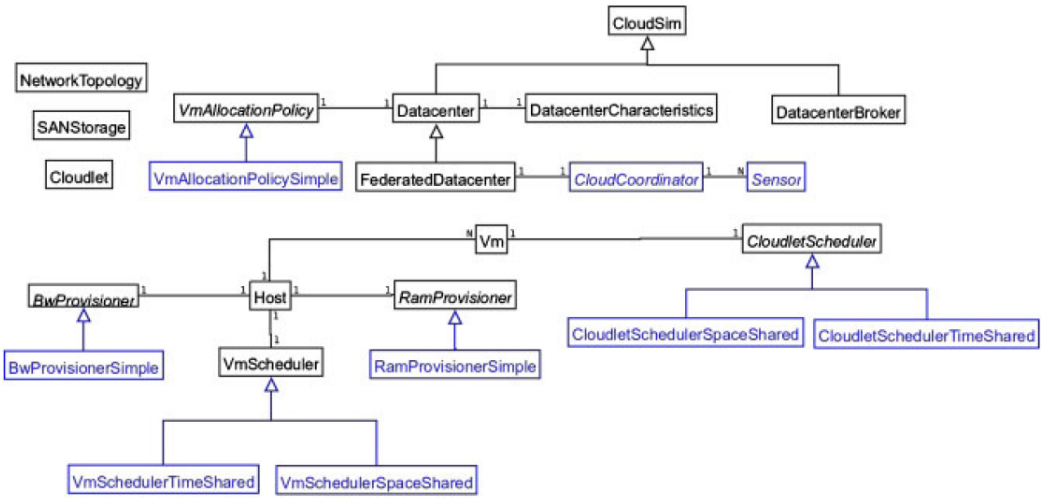

Figure 4.1: CloudSim class design diagram [6]

Cloudlet: This class is the equivalent to the tasks of the *EdgeCloudSim*. It models the cloud-based application services, such as content delivery and social networking. Each Cloudlet has a pre-assigned length and data transfer overhead that it needs to undertake during its life cycle.

CloudletScheduler: This is an abstract class that can be extended to realize different scheduling policies. Each VM has its own instance of a CloudletScheduler to run its Cloudlets. The scheduling policies determine how the processing power of a VM is shared among the Cloudlets. In its original design, CloudSim offers two scheduling policies: CloudletSchedulerSpaceShared and CloudletSchedulerTimeShared. The implementation in this thesis introduces a third scheduling policy: CloudletSchedulerPriority.

CloudletSchedulerSpaceShared: In this policy each VM only executes one Cloudlet at a time. It is non-preemptive, so the VM will only start the execution of the next Cloudlet once the previous one has finished.

CloudletSchedulerTimeShared: In this policy each Cloudlet has only a certain amount of time for its execution. Once the time is over, the next Cloudlet is assigned to the VM. If a Cloudlet has not finished during its assigned time, it is postponed until it is assigned its next time frame.

CloudletSchedulerPriority: In order to introduce scheduling for Real-Time scenarios, a new scheduling policy will be implemented in *CloudSim*. This one will be called CloudletSchedulerPriority (PriorityScheduler) and will be further looked at in Chapter 5.

Datacenter: The Datacenter models the core infrastructure-level services offered by cloud providers (e.g. Amazon, Azure). It represents the hardware and its characteristics (memory, cores, bandwidth, etc.). The Datacenter consists of multiple hosts which get assigned various Vms.

DatacenterBroker: This class acts as the decider of the system. It communicates between SaaS and the cloud providers. It decides which Cloudlets are assigned to which VM in order to maximize the overall performance of the data center.

VM: This class represents a VM, which is managed and hosted by a cloud host component. Each VM has following characteristics: accessible memory, processor, storage size and the VM's internal provisioning policy, which is its own instance of a *CloudletScheduler*.

VmScheduler: This is an abstract class implemented by a Host component. It represents the policy that is used to share processing power among VMs running in a Host. Each Host has its own instance of a VmScheduler. Since this thesis only focuses on scheduling the tasks on a VM, the VmScheduler is irrelevant.

SimEntity: This is an abstract class which represents a simulation entity. An entity is able to send events to other entities and process received messages as well as fire and handle events.

SimEvent: This entity represents a simulation event that is passed between two or more entities. It contains the information about an event, such as: type, init time, time at which the event should occur, finish time, time at which the event should be delivered to its destination entity, Ids of the sources and destination entities, tag of the event, and data that have to be passed to the destination entity.

A SimEvent can be anything that is needed during a simulation: starting necessary entities, creating tasks, etc.

CloudSim: CloudSim is the main class. It is responsible for managing event queues and controlling step-by-step execution of simulation events. Every event that is generated by the CloudSim entity at run-time is sorted by their time parameter and then stored in a queue called future events. Each event that is scheduled is removed from the future events queue and transferred to the deferred event queue. Following this, each event is processed according to its event type.

The tasks that are generated within *EdgeCloudSim* are sent to the CloudSim class, which then creates a SimEvent for each TaskProperty object. Once such a SimEvent is executed, a Task object is created, which is an extension of the Cloudlet class. This can then be assigned to a VM and is later scheduled by the *PriorityScheduler*.

4.3 Datacenter internal processing

Processing Cloudlets is handled by their respective VMs. At each simulation step, their progress must be updated. The method *updateVmsProcessing()* is called by each Datacenter entity at each simulation step to inform the Datacenter of the status of each Cloudlet. At host level, triggering the *updateVmsProcessing()* method directs every VM to update its task unit status (finish, suspended, executing) with the Datecenter entity. The VMs then return the next expected completion time of the task units currently managed by them. Figure 4.2 shows this process in form of a sequence diagram.
The *updateVmsProcessing()* method is relevant for the implementation of this work, since checking a task's status is important to determine if it finishes before its deadline.

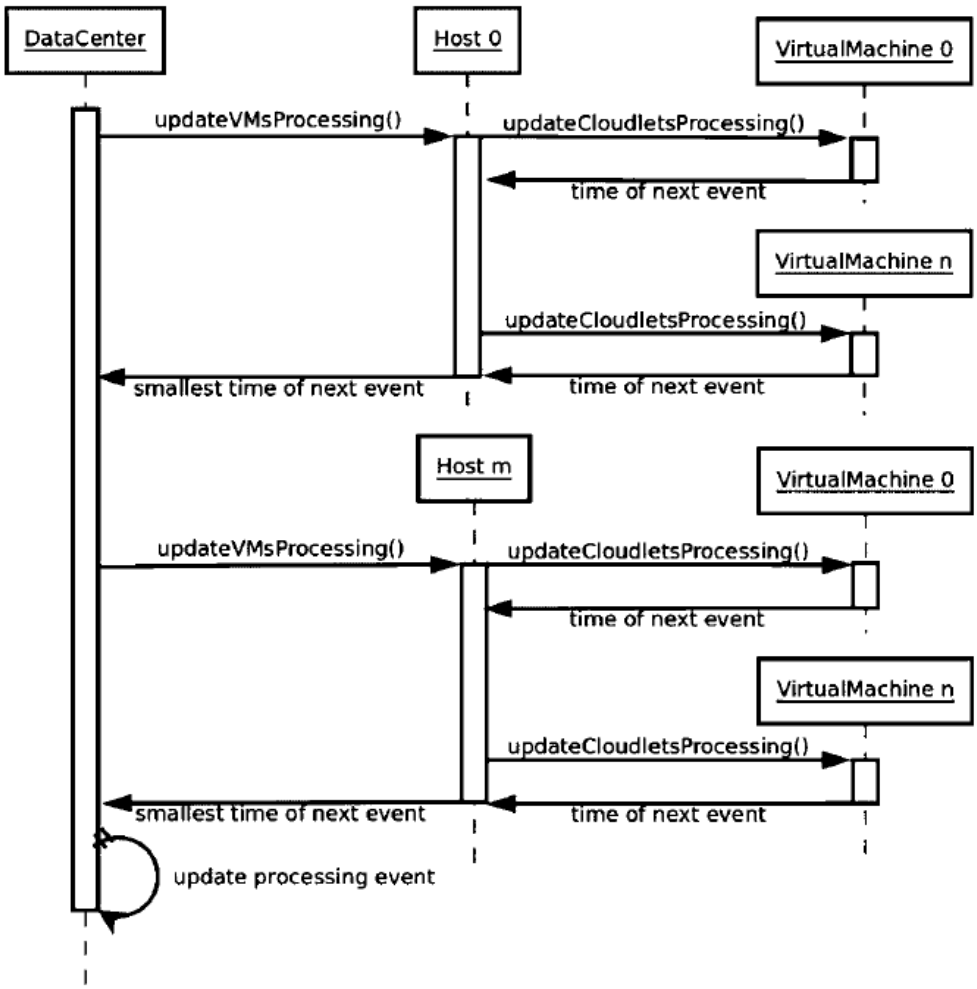

Figure 4.2: Cloudlet processing update process [6]

5 Implementation

In this chapter the implementation process is explained. First, the overall objective is given. The section after that describes the changes that had to be made to the tasks generated in *EdgeCloudSim* and their parameters in both *EdgeCloudSim* and *CloudSim*. Following that, the new *Real-Time* scheduling policy is presented. In the next section the changes to the output given by *EdgeCloudSim* are shown. The last section concludes this chapter by presenting the results of the implementation.

5.1 Objective

The goal of this thesis is adding *Real-Time* support to *EdgeCloudSim*. *Real-Time* includes giving the tasks that are generated in the simulation a deadline and scheduling them according to *Real-Time* methods. In this work a *fixed-priority non-preemptive* scheduling algorithm is used as a *Real-Time* scheduling method. Finally, an additional output is needed for *EdgeCloudSim*, so the users can analyze the results of the *Real-Time* scheduling. In order to achieve the goal of this thesis, the source code of both *EdgeCloudSim* and *CloudSim* had to be changed.

5.2 Task Generation

This section shows the changes to the parameters of the generated tasks. In order to use the generated tasks in *Real-Time* scheduling, priority and deadline parameters had to be added.

TaskProperty: The TaskProperty class represents the characteristics of the tasks that are generated for a simulation. It serves as a template for tasks and contains the necessary parameters. For *Real-Time* support, each task needs a deadline and a priority. For this, a deadline and a priority parameter were added to the TaskProperty class.

Now whenever a task is generated by the load generator module, the created TaskProperty object will include a priority and a deadline. This implementation uses the *Rate Monotonic* method, so the deadline and priority of a generated task will be equal to its length.

IdleActiveLoadGenerator: This class handles the generation of random tasks. It creates tasks according to a Poisson distribution. Once all tasks are created, the task list is sent to the CloudSim class, which creates and schedules a SimEvent for each TaskProperty object. Since nothing about the generation of tasks had to be changed other than the priority and deadline parameter, this class did not have to be modified.

Task: The Task class is an extension of the Cloudlet class from *CloudSim*. Once a scheduled TaskProperty SimEvent is executed, a Task object will be created using the information contained by the TaskProperty object. Since the Task class is an extension of the Cloudlet class, the scheduler within *CloudSim* can then handle the created Task object. For this, the Task class was changed by adding priority and deadline parameters, while also adding the necessary constructor.

Cloudlet: The Cloudlet class is the *CloudSim* equivalent to the Task class of *EdgeCloudSim*. Similar to the Task class, priority and deadline parameters were added to the Cloudlets. The necessary constructor was added as well.

5.3 Scheduling

This section presents the new scheduling policy created to enable *Real-Time* support for *EdgeCloudSim*. It describes the scheduling process by explaining how the created class is structured compared to the original scheduling policies in *CloudSim*.

CloudletSchedulerPriority: The CloudletSchedulerPriority class (PriorityScheduler) is a newly created scheduling policy. It is an extension of the abstract CloudletScheduler class and is similarly implemented to the CloudletSchedulerSpaceShared policy.

Each VM has its own instance of a PriorityScheduler to schedule the Cloudlets (Tasks) that are assigned to the VM. When a Cloudlet is assigned to a VM, it is added to its waiting list. The waiting list consists of all the Cloudlets that have yet to be executed by the VM.

The *updateVMsProcessing()* method is called at each simulation step to check the status of all Cloudlets and then inform the Datacenter. When a Cloudlet is finished, it is removed from the execution queue and the next Cloudlet from the waiting list is added to the execution queue.

Sorting the waiting list: In order to realize the proper scheduling, the *updateVMsProcessing()* method sorts the Cloudlets in the waiting list according to their priority, so that the Cloudlet with the highest priority is always executed first. Listing 5.1 shows the source code of the sorting algorithm.

```
ArrayList<ResCloudlet> tempList = new ArrayList<ResCloudlet>();
List<ResCloudlet> priorityList = new ArrayList<ResCloudlet>();

for(ResCloudlet rcl : getCloudletWaitingList()) {
        temporaryList.add(rcl);
}

for(int j=0; j< tempList.size(); j++) {
        ResCloudlet maxPriority = tempList.get(0);

        for (ResCloudlet rcl: tempList) {

                if (rcl.getCloudlet().getCloudletPriority() <
                maxPriority.getCloudlet().getCloudletPriority())
                {
                        maxPriority = rcl;
                        }
}

priorityList.add(maxPriority);
tempList.remove(maxPriority);
}
```

Listing 5.1: Sorting algorithm for waiting list

Check deadline of tasks: Additionally, every time the *updateVMsProcessing()* method is called, it checks if the current Cloudlet misses its deadline. A deadline is missed, if following function returns true:

$$currentTime > expectedDeadline \tag{5.1}$$

currentTime is the current system time. *expectedDeadline* is the sum of the *arrivalTime* and the *deadline* of a task, which is calculated as shown in Listing 5.2.

```
public double getExpectedDeadline(ResCloudlet rcl) {
        double capacity = 0.0;
        int cpus = 0;
        for (Double mips : getCurrentMipsShare()) {
                capacity += mips;
                if (mips > 0) {
                        cpus++;
                }
        }
        currentCpus = cpus;
        capacity /= cpus;

        double expDeadline = rcl.getCloudletArrivalTime()
                + (rcl.getCloudlet().getCloudletDeadline()
                / (capacity * rcl.getNumberOfPes()));

        return expDeadline;
}
```

Listing 5.2: Method to calculate the expected deadline of a task

If the current Cloudlet misses its deadline, it is removed from the execution queue and it is tagged as *CANCELED*. The *CANCELED* status is important for the SimLogger, which creates the output of *EdgeCloudSim*. Table 5.1 shows a few examples of task values that are important to determine if a task missed its deadline.

arrivalTime	deadline	expectedDeadline	currentTime	status
446.143	1002	447.145	446.312	SUCCESS
454.873	375	455.248	456.178	CANCELED
519.053	1487	520.540	521.113	CANCELED
549.718	9520	559.238	556.439	SUCCESS
648.374	1815	650.189	649.385	SUCCESS
1026.515	3941	1030.456	1027.279	SUCCESS
1362.612	12190	1374.802	1366.845	SUCCESS
1511.534	2712	1514.246	1514.260	CANCELED
1799.312	1643	1800.955	1799.794	SUCCESS

Table 5.1: Examples for task values

5.4 Simulation Output

This section explains how the new output for *EdgeCloudSim* with *Real-Time* support is created. More specifically, it lists the classes that have been modified to recognize canceled tasks and adjust the results from the simulations.

MobileDeviceManager: This class handles the mobile devices. Among other functions, it creates Task objects according to their TaskProperty characteristics and sends them to a VM. When a task returns, it checks the task's status.

An additional condition was added to the method *processOtherEvent()*. It now checks if a task was canceled, which is done by checking for the *CANCELED* tag set by the *updateVMsProcessing()* method.

If a task has been detected as canceled, the MobileDeviceManager informs the SimLogger, which is responsible for creating the results.

SimLogger: This class handles the output of the simulation. It takes the information gathered by the MobileDeviceManager and creates the result by differentiating if a task succeeded or failed, the reason it failed and its transmission delay. The results are saved in CSV (comma-separated values) format.

For this implementation, an additional output was created. The output for missed deadlines was created according to the other outputs.

LogItem: An additional constructor for the cloudletStatus was created. This is necessary to recognize canceled tasks.

5.5 Evaluation Results

In this chapter the implemented changes to *EdgeCloudSim* were explained. The tasks generated in *EdgeCloudSim* now have the necessary characteristics of *Real-Time* tasks and are scheduled according to *non-preemptive fixed-priority* scheduling, while detecting missed deadlines. Furthermore, the output now considers canceled tasks.
In order to run simulations, the developers of *EdgeCloudSim* provided sample applications, which can be found in the *applications* folder. The provided sample applications have been adjusted to use the implemented *Real-Time* scheduling. Running the *Main* class starts the simulation and the output files are saved in the *sim_results* folder. It may be necessary to create a folder with the name *ite1* inside the *sim_results* folder first. *ite_n* represents the results for iteration *n*.
The output files are in CSV format and can be used to plot graphs. In Figure 5.1 an example for the results of the *EdgeCloudSim* with *Real-Time* support is shown. The graph was plotted with MATLAB R2020a using the scripts provided by the developers

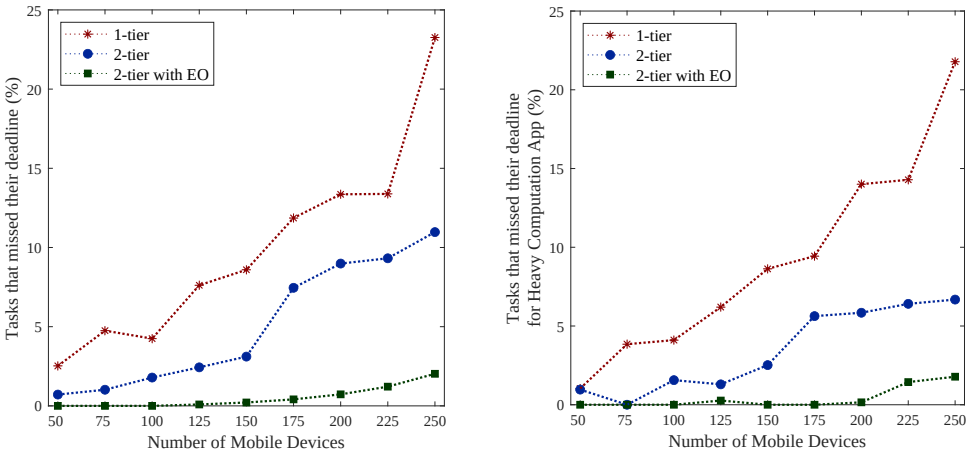

Figure 5.1: Result examples for EdgeCloudSim with Real-Time support

of *EdgeCloudSim* [13]. As seen in Figure 5.1, the single-tier architecture shows way more missed deadlines as compared to the two-tier and two-tier with EO architectures. The two-tier with EO architecture resulted in the least missed deadlines by far.

With the addition of *Real-Time* support to *EdgeCloudSim*, users are now able to use this simulation tool for *Real-Time edge computing* scenarios. The adjusted result output allows them to analyze and compare different architectures and policies in regards to missed deadlines.

6 Conclusion

6.1 Summary

In this work we looked into *Real-Time* systems and its necessary characteristics to realize *Real-Time* support for *EdgeCloudSim*. We introduced *fixed-priority* scheduling as a *Real-Time* scheduling paradigm that can be implemented in *EdgeCloudSim*. The thesis then presented the original design of *EdgeCloudSim* and *CloudSim* in order to explain which parts of their source code are relevant and have to be modified to reach the goal of this thesis. In Chapter 5 a solution is suggested and implemented. The task model of *EdgeCloudSim* and *CloudSim* were changed so that tasks now include a priority and a deadline. Furthermore, a new task scheduling policy was implemented in *CloudSim*, so that the tasks created during the simulation are scheduled according to their priority. Additionally, the implemented PriorityScheduler checks the tasks' status to detect missed deadlines. Finally, the SimLogger class in *EdgeCloudSim* is now informed about tasks that missed their deadline and adjusts its results to consider those failed tasks.

6.2 Conclusion and Future Work

In conclusion, the proposed solution works as expected and enables *Real-Time* support to *EdgeCloudSim*. The tasks are now created with a correct priority and deadline and the scheduling algorithm works according to *non-preemptive fixed-priority* scheduling. The proposed solution is a very simple but effective approach, since *fixed-priority* scheduling is a popular *Real-Time* scheduling algorithm. This leaves a lot of room for improvements, though. The scheduling can be more efficient and other scheduling paradigms can be realized.

For example, the algorithm proposed by Toma et al. [14] and mentioned in Section 2.6 can be implemented to possibly achieve a more efficient solution. The algorithm is based on dynamic scheduling and uses different execution versions for tasks, which can improve overall performance. Since the algorithm utilizes *edge computing* and serves useful for *Real-Time* systems, it can be an appropriate approach to enable *Real-Time* support for *EdgeCloudSim*. Since the algorithm is very complex and the time was too limited to implement it to *EdgeCloudSim*, it was not possible to realize it in this thesis. It can be a desirable approach for future works, though.

All in all, the implementation proposed in this thesis was successful and the *EdgeCloudSim* can now be used for *edge computing* scenarios with *Real-Time* task models.

Appendix

List of Figures

List of Tables

List of Source Codes

Bibliography

[1] M. Armbrust. A view of cloud computing. *Communications of the ACM*, 53:50–58, 2010.

[2] C. Baun, M. Kunze, et al. *Cloud Computing: Web-basierte dynamische IT-Services*. Springer, 2011.

[3] A. Bhattacharya and P. De. A survey of adaptation techniques in computation offloading. *J. Netw. Comput. Appl.*, 78(C):97115, Jan. 2017.

[4] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.

[5] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[6] R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software Practice and Experience*, 41:23–50, 01 2011.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107113, Jan. 2008.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):2943, Oct. 2003.

[9] A. U. R. Khan, M. Othman, S. Madani, and S. Khan. A survey of mobile cloud computing application models. *IEEE Communications Surveys Tutorials*, 16:393–413, 02 2014.

[10] L. Sha, T. Abdelzaher, et al. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004.

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

[12] M. Short. The case for non-preemptive, deadline-driven scheduling in real-time embedded systems. *IAENG International Journal of Computer Science*, 1, 06 2010.

[13] C. Sonmez, A. Ozgovde, and C. Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 39–44, 2017.

[14] A. Toma, J. Wenner, J. E. Lenssen, and J. Chen. Adaptive quality optimization of computer vision tasks in resource-constrained devices using edge computing. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 469–477, 2019.

Eidesstattliche Versicherung
(Affidavit)

Sazontow, Roman
Name, Vorname
(Last name, first name)

171392
Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/~~Masterarbeit~~* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/~~Master's~~* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/~~Masterarbeit~~*:
(Title of the Bachelor's/ Master's* thesis):

Supporting Edge Computing Simulator for Real-Time Task Models

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Bochum, 18.05.2020
Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Bochum, 18.05.2020
Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.