Journal of Intelligent Manufacturing (2021) 32:2301–2317
https://doi.org/10.1007/s10845-021-01760-9

Real-time edge framework (RTEF): task scheduling and realisation

Volkan Gezer[1] · **Achim Wagner**[1]

Received: 21 November 2019 / Accepted: 12 March 2021 / Published online: 10 April 2021

Abstract
With the big success of the Cloud Computing, or the Cloud, new research areas appeared. Edge Computing (EC) is one of the recent paradigms that is expected to overcome the Quality of Service (QoS) and latency issues caused by the best-effort behaviour of the Cloud. EC aims to bring the computation power close to the end devices as much as possible and reduce the dependency to the Cloud. Bringing computing power close to the source also enables real-time applications. In this paper, we propose a novel software reference architecture for Edge Servers, which is operating system (OS) and hardware-agnostic. Edge Servers can collaborate and execute (near) real-time tasks on time, either by downscaling or scheduling them according to their deadlines or offloading them to other Edge Servers in the network. Decision making for offloading, resource planning, and task scheduling are challenging problems in decentralized systems. The paper explains how resource planning and task scheduling can be overcome with software approach. Finally, the article realises the architecture as a framework, called Real-Time Edge Framework (RTEF) and validates its correctness with a use case.

Keywords Real-time computing · Edge computing · Task offloading · Edge in manufacturing · Fog computing

Introduction

It is a well-known fact that Cloud Computing, the Cloud, was a big success, and it will be even more essential as the dependence on information grows. From its first initial concepts in the early 1960s, the idea was brought to life as Remote Job Entry (RJE) White (1971). Since then, different experimentations were made to exploit the usage of large-scale computing. The apparent success of the Cloud emerged in new application areas. The Cloud is well-used for daily tasks, such as emails, collaborative work, file or data storage, finance, or remote monitoring. Ubiquitousness, scalability, and accessibility are some of the significant reasons that make Cloud so popular. Cloud also brought different business models into life. The pay-per-use model reduces the infrastructure costs for the end-users. Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS), and Function-as-a-Service (FaaS) are some of how the Cloud provides service differently.

✉ Volkan Gezer
Volkan.Gezer@dfki.de

Achim Wagner
Achim.Wagner@dfki.de

[1] Innovative Factory Systems (IFS), German Research Center for Artificial Intelligence (DFKI), Kaiserslautern, Germany

Though not used as much as it is used in daily lives as stated above, the Cloud technologies are also used in manufacturing areas. One of the application areas in this context is Cloud Manufacturing (CM). Li and Zhang initially proposed the concept in 2010 Zhang et al. (2010). However, the manufacturing-as-a-service (MaaS) concept was first seen in literature in 1990 by Goldhar and Jelinek (1990). MaaS does not target controlling the factories or performing real-time computing. 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 users involved throughout the manufacturing lifecycle are serviced, on-demand Zhang et al. (2010).

Since the operation principle of Cloud Computing is mostly through the Internet, reactions to the requests use a best-effort approach. To perform (near) real-time operations, the computation needs to be close to the field or device tier. 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 Lhotka (2005). The paradigm, which adds an additional tier between the Cloud and the field tier, and moves the computational power near the user as close as possible is called *Edge Computing* (EC) Gezer et al. (2018).

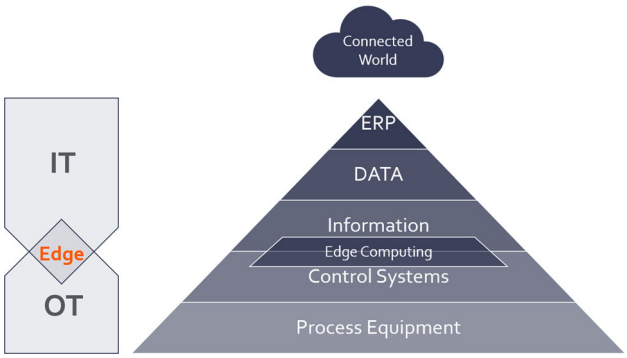

Fig. 1 Automation pyramid and where Edge Computing is located at

In the automation domain, this tier is located where the Information Technology (IT) meets Operational Technology (OT) as seen in Fig. 1. Different researches define EC as Edge Cloud, Fog Computing or Cloudlet. We believe that these terms are interchangeable, and throughout this paper, only EC will be used. EC targets reduced latencies and high QoS that are hindered by the Cloud solutions, mainly due to relying on the Internet connection. According to Shi et al. (2016), EC also reduces privacy and security risks of the confidential data being exposed over the Internet. In a networked system, most of the data is generated in the device tier. Another objective of the paradigm is to reduce the traffic of generated raw data from field tier to the Cloud. As EC has computing power at the field tier, it can preprocess the raw data.

In our previous paper on EC Gezer et al. (2018), we proposed a conceptual software reference architecture for Edge Servers, to realise a real-time capable server framework, together with its requirements and enablers. The objective of this architecture was to enable seamless execution of real-time tasks in a resource-aware Edge Network. The participants of the network were responsible for finding the best way and location to execute the requested tasks without causing them to miss their deadlines. The work in the previous paper was only theoretical and conceptual. This paper presents the progress since then; the decision algorithms, how the software components and Edge Servers interact with each other, their realisation, and implementation. All calculations and internal computations are considered to have no overheads. The realisation of this architecture is called Real-Time Edge Framework (RTEF). Following scalability, interoperability, extensibility, and collaboration features of the architecture, RTEF also has multi-user support. The paper also introduces a new novel preemptive and online scheduling algorithm called Non-preemptible And Preemptible Aperiodic TAsk (NAPATA) scheduling, which is integrated into the RTEF to decide on the execution order with negligible overhead and low complexity. The paper concludes after validating the concepts for their correctness after realisation/instantiation.

Related work

With the rise in popularity of the Internet and its derived technologies such as the renowned Internet of Things (IoT), there have been initiatives on finding new ways to increase usability. Most notably, Cloud Computing and Grid Computing can be given as two examples to the game-changers. Additionally, new terms such as Fog and Edge have emerged.

The origins of EC go back to 1999, where Akamai Technologies introduced Content Delivery Networks (CDN) to increase web performance (Dilley et al. 2002). They cached contents at different locations, aiming at reduced requests on the site's own infrastructure and faster response times for the users, by responding to the requests using nearby servers. Noble et al. (1997) first demonstrated the potential of EC by realising a speech recognition scenario on resource-limited devices. They offloaded the computation to a nearby server, and the results delivered an adequate performance. In 2014, Chang et al. (2014) proposed a new model for Cloud Computing, with the name *Edge Cloud*. Chang et al. then tested the performance of their architecture with indoor localisation application to evaluate latency, and with video monitoring application to measure the bandwidth. Their results showed a better performance compared to the existing Cloud solutions.

There exist several works done for computation and control in the Cloud as well (Kretschmer 2016; Givehchi et al. 2014; Goldschmidt et al. 2015). Realising 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, using virtual environments that can simulate several hours of real environment tasks in a couple of minutes, 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 centres, their application containers, costs as well as power consumption CLOUDS Laboratory (2019). One simulation tool to evaluate the reliability of the system is called iFogSim and implemented by Gupta, Dastjerdi, Ghosh, and Buyya Harshit et al. (2016). 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. Sonmez, Ozgovde, and Ersoy introduced another simulator called EdgeCloudSim Sonmez et al. (2017). It adds a mobility model and non-fixed delays into the network which is fixed in iFogSim. However, none of the simulators is targeted for real-time applications.

Mohamed et al. (2017) proposed a Service-Oriented Middleware (SOM) for Cyber-Physical Systems (CPS). It provides a service-based infrastructure to develop and operate CPS applications. The approach also enables the integration of CPS with Cloud and EC. Pallasch et al. (2018) introduced

a concept to utilise Cloud and EC for industrial control. They refer to the devices connected to field tier devices as Edge Devices. Their setup uses Amazon Web Services (AWS) Cloud services for non-real-time, but computing-intensive tasks. One Edge Device (Edge Server in this work) can directly access to the AWS. This device has several sensors serially connected to it. Also, two robots and two IoT devices (that also act as Edge Devices) which connect the robot controllers with the Edge Device are connected. The research shows that industrial control using EC 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 to the original requester.

Vick et al. (2015) introduce a virtualised Robot Controller (RC) and a virtualised Programmable Logic Controller (vPLC), to enable outsourcing control functions of an industrial robot. They created Virtual Machines (VMs) to realise PLCs and a VM in the Cloud to perform control operations that require soft real-time capability. Horn and Krüger from the same group then test the feasibility of the novel architecture Horn and Krüger (2016). They performed this test using three different experiment setups. The results showed that the setup with the direct connection has the lowest latency. Nevertheless, using a direct connection with the solution is only possible with modern robots and machinery, which support communication interfaces such as OPC-UA.

Elbamby, Bennis, and Saad Elbamby et al. (2017) investigated the problems of the EC and a cache-enabled Edge Network. They proposed a clustering method to group end-users with the same interests on tasks. The idea was to track end-users and the popularity of the tasks that they request 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 end-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. Similarly, Sonmez et al. (2017) evaluated the performance of different three possible generic EC architectures: single-tier, two-tier, and two-tier with a load balancer. The evaluation analysed the performance of each architecture over wireless communication, and it was performed using a simulator based on CloudSim CLOUDS Laboratory (2019). The results showed that the two-tier setup with a load balancer had given the best results. As its name suggests, the architecture needed a 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 Edge Server. The scenario had a latency-intolerant application, but it did not carry out a real-time use case. In the real-world, failure in the load balancer would cause the whole network to stop servicing. Some companies and organisations proposed reference architectures in the domain to raise the interest and standardisation process of EC (IBM Cloud Architecture Center 2017; OpenFog Consortium 2017; EdgeX Foundry 2020; VMware 2017). Last, the work by Yin et al. (2020) uses an approach similar to this paper by considering the real-timeliness of tasks and available resources to determine the location to execute. However, only the resources of the current server are considered, and the offloading is performed only to the Cloud, not to the neighbouring Edge Servers.

Our proposed framework in this paper deals with problems that Cloud Computing has and combines the benefits of the existing work. On top of the state-of-the-art research, the proposed work enables seamless execution of real-time tasks within a created Edge Network, by abstracting low-level decisions and enabling offloading in case of resource unavailability. The framework can be used to execute real tasks or as a stand-alone simulator, with its dummy load generating features. The work brings several technologies together, such as IoT, grid computing, virtualization, and collaborative computing. To the best of our knowledge, there are no existing frameworks or scheduling algorithms that feature the specifications of our work.

Architecture overview

Rapid changes in industry and customer requirements need agile advancements due to market demands and shorter product cycles Feldhorst et al. (2009). With the idea of flexible and scalable servicing, EC targets simplifying the execution of complex tasks, without limiting their usability. To overcome the problems stated in "introduction" Section, an Edge Network containing Edge Servers and End Devices using an Edge Topology is going to be discussed. However, before advancing, it is worth clarifying the terms to avoid ambiguities. These terms are explained further in this section and summarised in Fig. 2.

Edge Servers are physical hardware and communicate with the End Devices and other Edge Servers using a connection-oriented communication protocol. They are regular computers, but converted into Edge Servers after they run an instance of the RTEF, developed based on the reference architecture explained in this section. 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. A source or destination device in a network is called an End Device The Computer Language Co Inc. (2021). An End Device in this paper is an end-user device that requests the execution of jobs through Edge Servers. It may have computing power or only

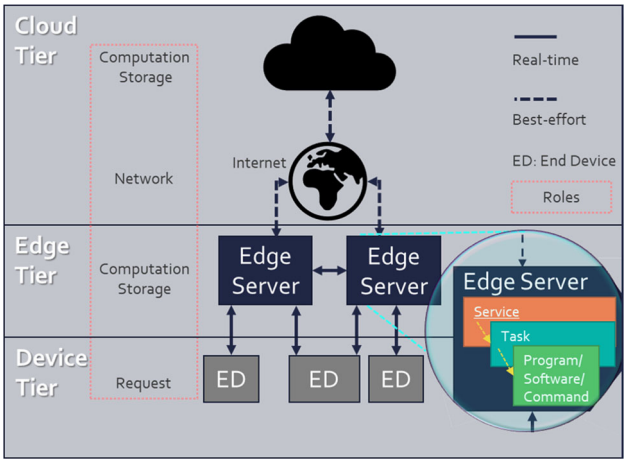

Fig. 2 An example Edge Network and its participants. Dashed arrows show best-effort communication due to the behaviour of Cloud Computing, whereas solid arrows show real-time capable communications

provides input to the Edge Server. The only requirement of being an End Device is to have a network interface to communicate with Edge Server(s). In the architecture, End Devices communicate with the servers using socket communication. An End Device can be a sensor, smart sensor, machine, computer, mobile phone, or worker assistance glasses. They are allowed to have connections to multiple Edge Servers, but they can request a job only from a single server at a time.

An Edge Network in this work is the combination of Edge Servers and End Devices which communicate with each other to request jobs and/or respond to jobs. Participant or node terms will also be used to refer to the Edge Servers and End Devices in the Edge Network. Edge Topology term in this research is the creation of a network structure by linking Edge Servers and End Devices in the Edge Network. The research does not limit the topology, and any available combinations can be used. The network communication between the Edge Servers are out of scope. Therefore, it is assumed that the connection between participants are using real-time communication standards.

A service is a piece of software that is reusable to perform a specific work. In this paper, a service is a wrapper of a program/software/command (PSC) that defines its behaviour and how it should be executed. One PSC may have several service definitions, each with a different execution behaviour. A program cannot be executed on 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 behaviour such as execution duration, relative deadline, Central Processing Unit (CPU) execution utilisation in percentage, and CPU affinities. These parameters will be discussed in further detail in "realization" Section. 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 work, tasks are the individual running instances of services. They perform requests that are carried out through services. Tasks may request the execution of a single command or program or multiple processes.

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 designed to be flexible, and scalable, allowing new servers to be connected with minimum effort. The realisation of the architecture for validation is done using the Java programming language and called Real-Time Edge Framework (RTEF). The participating End Devices in the Edge Network are abstracted from the lower-level operations by the introduced standard Application Programming Interface (API) methods by the architecture.

From the architecture perspective, there are two types of 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) Deploy and run the RTEF 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 define the execution behaviour of each installed PSC.
(4) Make these services private or leave public, deciding whether they should be accessible by other Edge Servers or not.
(5) Connect Edge Servers with each other to create the desired Edge Topology.

The steps above summarise the preparation phase of the Edge Network. After the network is set up, the End Devices establish connections to the Edge Servers. At any time later, new Edge Servers can be introduced to the network. During establishing connections between the Edge Servers, each server shares all known resources and public services (including its own) with other participants automatically, to collect resource information of the network. The exchange is performed using a set of messages during the handshake phase. These will be explained in the upcoming subsection. From the end-user perspective, the setup is completed when the desired connections between End Devices and Edge Servers are established. The network is then ready for accepting tasks.

An End Device can request a task execution from any Edge Servers that it is connected to. However, the task will be executed on the most suitable Edge Server after the collective decision of other Edge Servers. This decision is made by using an algorithm in the Edge Server by evaluating whether the task completes its execution until its

defined deadline. This algorithm uses a satisfaction equation explained in Gezer and Wagner (2020). The chosen Edge Server to offload can have a direct or indirect connection to the original requester. No matter 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.

Requested tasks run the PSCs whose execution behaviours are defined by their services. These services can explicitly set the worst-case execution time (WCET) that is the longest execution duration, relative deadline, CPU affinity, and CPU execution utilisation for each PSC. Once an Edge Server receives the request, first the availability of this PSC within the network is queried. Based on the behaviour of the service such as its deadline, also considering the current server, the most suitable server in the closest distance is chosen. Once a server is chosen, it means that this task can be executed on this server without missing its deadline. However, if the server has other tasks that are already running, additional precautions must be taken. First, the server decides whether downscaling either in the new task, running tasks, or in both, due to availability of the CPU is necessary. When necessary, the possibility of downscaling the CPU execution utilisations is calculated. If downscaling is feasible, then the new task is executed with the newly calculated execution utilisation. Since WCET (denoted as x), relative deadlines (denoted as d, where $x \leq d$) and CPU utilisation percentage during execution denoted as u (where $0\% < u \leq 100\%$) of tasks are set during service definition and known a priori, if the new execution utilisation of a task i is u_i', where $0\% < u_i' \leq 100\%$, the new execution time x_i' becomes $\frac{x_i u_i}{u_i'}$. If $x_i' \leq d_i$ holds, then the new execution utilisation does not cause task i to miss its deadline. The worst-case execution utilisation (WCEU) c_i'' can be calculated by replacing x_i' with d_i and using the same equation, for each CPU that the task has an affinity to. If the sum of execution utilisations of all running tasks, including the requested task, does not exceed 100% at each CPU, then the tasks can run without further action. If not, this procedure is repeated for all tasks. If this iteration also not successful, the tasks need to be scheduled. If scheduling still does not solve the problem, the task offloaded to another Edge Server that has a public service defined with the same name.

According to arrival patterns, tasks, in general, are classified in three categories: (1) *periodic* tasks that arrive in a constant rate and have 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, but with bounded inter-arrival times Audsley et al. (1991). These types define whether a scheduling algorithm can be applied to them or not. Depending on the purpose of the tasks, their wrapper services can have different types. These types are explained below:

Table 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

Legacy

Legacy services wrap *aperiodic* non-preemptible PSCs. Non-preemptible means that preemption by other (especially with higher priority) tasks is not possible Berry (2007). Such tasks cannot be paused. Pausing such tasks means that their execution is terminated; thus, they cannot continue from the paused state. If resuming is requested, the tasks start their execution from their initial state.. They also reset their former execution time. Once the execution is completed, its task is removed.

Simple

Similar to Legacy services, Simple services wrap *aperiodic* tasks. Different than the Legacy services, the PSCs that are wrapped in this category can be paused. Some examples are the PSCs that read continuous or streaming data (e.g. video application). 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.

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. Tasks in this category are also preemptive. Pausing their instances keeps their remaining times untouched, and they can be resumed.

Table 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 WCET and running alone on different computers. As described, Legacy and Simple services do not have periods, as 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. 3. 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

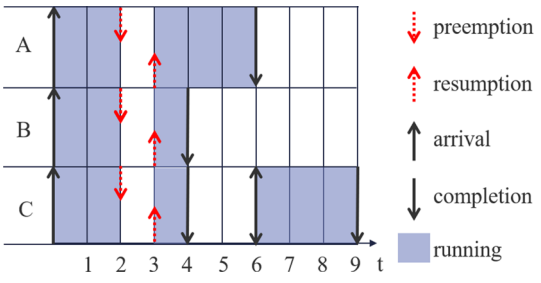

Fig. 3 An example of the execution behaviours of independent PSCs in Legacy, Simple and Simple Periodic service types on different computers when they are preempted at $t = 2$ and resumed at $t = 3$

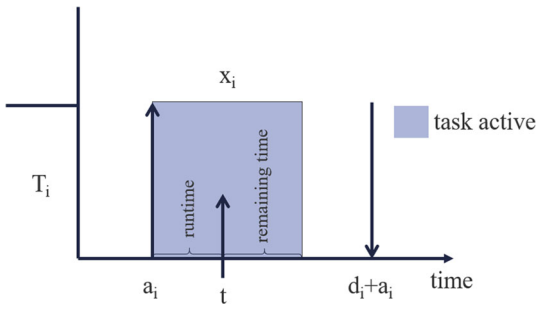

Fig. 4 Calculation of the remaining execution time (duration) for a task T_i at time t

Simple Periodic completed execution at $t = 4$ and its second period started at $t = 6$, idling the CPU for two units of time.

While defining services, it is essential to know the characteristics of the PSC that is to be wrapped. For example, the type, duration, and deadline of the PSC must be known before creating its service. These parameters are used to estimate the schedulability as well as for the decision of offloading location. The required parameters are explained in "services" Section.

If the chosen Edge Server cannot directly execute a task due to unavailable resources, it needs to schedule the incoming task, or the running tasks. Depending on the service type, RTEF uses two types of scheduling algorithms: (1) Earliest Deadline First (EDF) scheduling for Periodic services and (2) Non-preemptible And Preemptible Aperiodic TAsk (NAPATA) scheduling for Legacy and Simple services. Both of them are online, preemptive, and dynamic priority scheduling algorithms that can be used to schedule real-time tasks. NAPATA scheduling is a novel scheduling algorithm that has a negligible overhead and low complexity. Details on the algorithm will be explained in "stscheduler" Section. EDF and NAPATA scheduling algorithms are implemented for single CPU systems. For multiprocessor Edge Servers, the service creator is responsible for setting the most viable CPU mask during design. An Edge Server does not always receive requests for the same type of service. When running tasks with different types, an additional algorithm for scheduling is required. Such cases are solved by introduction of scheduling server algorithms (Sprunt et al. 1989; Lehoczky et al. 1987). At the moment, the RTEF does not implement any scheduling servers and requires that the combination of tasks with different types are isolated using different CPU affinities. Nevertheless, this leaves an open door for future work.

The following section will explain the novel scheduling algorithm used in the framework.

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*.

As mentioned earlier, three types of tasks are defined in the computing domain: periodic, aperiodic, and sporadic. For periodic tasks, one of the scheduling algorithms developed for periodic tasks can be used (e.g. EDF Scheduling). For aperiodic tasks, Least Slack Time (LST) Scheduling can be used. However, LST cannot optimally schedule non-preemptible tasks. A non-preemptible task is a task that cannot be preempted if a higher priority task is requested. 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-preemptible and preemptible tasks to run together. This requires the introduction of another scheduling algorithm. This algorithm is called Non-preemptible And Preemptible Aperiodic TAsk (NAPATA) scheduling.

NAPATA scheduling provides an online, dynamic priority, and preemptible scheduler with negligible overhead, and it uses counting sort for ordering the tasks. Unlike LST scheduling, NAPATA scheduling can work with non-preemptible and preemptible tasks together. Instead of slack time, NAPATA scheduling uses only the remaining times of the active tasks. If a non-preemptible 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. This section will elaborate how the algorithm works.

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 ($a_i + x_i \leq d_i + a_i$).

At any time t, the feasibility may be rechecked, regardless of the need. As seen in Fig. 4, if the feasibility at the time t is to be calculated, the remaining execution time (duration) of the task can be used, which should be between t and $d_i + a_i$.

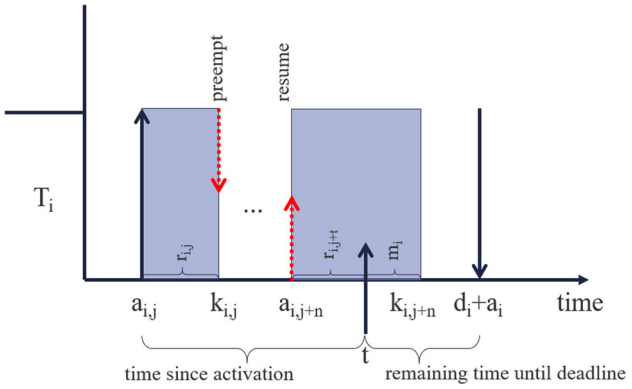

Fig. 5 Calculation of the remaining execution time of a preemptible task

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. 1 and 2 can be used to find them out.

$$r_i = t - a_i \tag{1}$$
$$m_i = x_i - r_i \tag{2}$$

$x_i \leq d_i$ can also be written in terms of t as seen in equations 3 to 7.

$$t = r_i + a_i \tag{3}$$
$$r_i = x_i - m_i \tag{4}$$

inserting r_i in Eq. 3

$$t = x_i - m_i + a_i \tag{5}$$

and solving for x_i

$$x_i = t + m_i - a_i \tag{6}$$

yields

$$t + m_i \leq d_i + a_i \tag{7}$$

However, if tasks are preemptible, equations 1 and 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. 5.

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. 8.

$$R_i = \sum_{j=0}^{t} r_{i,j} = \sum_{j=0}^{t} (k_{i,j} - a_{i,j}) \tag{8}$$

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

$$M_i = x_i - R_i \tag{9}$$

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 7, the feasibility algorithm F_i for task T_i is summarised in Eq. 10. 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.

$$F_i = t + \sum_{n=1}^{N} M_n \leq d_i + a_i \tag{10}$$

Let task $T_i = \{a_i, x_i, d_i\}$ have an arrival time of a_i, an execution time (duration) of x_i, and a relative deadline of d_i. Also, assume two tasks T_1 and T_2 as seen in Eq. 11 arrive at $t = 0$.

$$T_1 = \{0, 2, 5\}$$
$$T_2 = \{0, 3, 4\} \tag{11}$$

If they were periodic tasks whose deadlines are equal to their periods, using the feasibility equation for EDF given in Liu and Layland (1973) would yield:

$$\frac{x_1}{d_1} + \frac{x_2}{d_2} \leq 1 \tag{12}$$
$$\frac{2}{5} + \frac{3}{4} \leq 1$$
$$\frac{23}{20} \leq 1 \tag{13}$$

As inequality 13 does not hold, according to EDF Scheduling, the tasks are not schedulable. However, if these tasks are not periodic, Eq. 10 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. 14.

$$F_2 = t + (x_2 - R_2) \leq d_2 + a_2$$
$$F_2 = 0 + (3 - 0) \leq 4 \tag{14}$$

Table 2 An example set of tasks for the Non-preemptible And Preemptible Aperiodic TAsk (NAPATA) scheduling with mixed types

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

Since the inequality 14 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.

$$F_1 = (0 + (2 - 0)) + (0 + (3 - 0)) \leq 5 \tag{15}$$

Inequality 15 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, once non-preemptible services are preempted, they are terminated. Hence, their running time resets - also their remaining times. The following example will demonstrate another scenario with different service types.

Assume that the tasks arrive at an Edge Server as shown in Table 2.

Unlike the first example, this example cannot directly use the inequality 12, 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.

For $t = 0$

$$F_1 = 0 + (2 - 0) \leq 8 + 0 \tag{16}$$

At $t = 0$ only T_1 is active, and the inequality 16 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-preemptible 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:

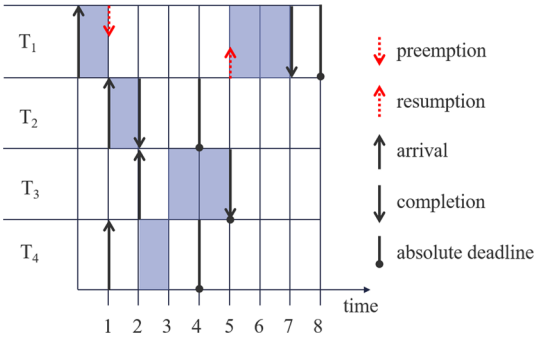

Fig. 6 Resulting scheduling diagram of the NAPATA scheduling example

For $t = 1$

$$F_2 = 1 + (1) + 1 \leq 3 + 1 \tag{17}$$
$$3 \leq 4$$
$$F_4 = 1 + (1) + 1 \leq 3 + 1 \tag{18}$$
$$3 \leq 4$$
$$F_1 = 1 + (2) + 1 + 1 \leq 8 + 0$$
$$5 \leq 8 \tag{19}$$

Inequalities from 17 to 19 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:

For $t = 2$

$$F_4 = 2 + (1) \leq 3 + 1 \tag{20}$$
$$3 \leq 4$$
$$F_3 = 2 + (2) + 1 \leq 3 + 2 \tag{21}$$
$$5 \leq 5$$
$$F_1 = 2 + (2) + 2 + 1 \leq 8 + 0$$
$$7 \leq 8 \tag{22}$$

Inequalities from 20 to 22 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. 6.

If T_1 were to be a preemptible type instead of non-preemptible, inequality 22 would be written as shown in inequality 23 and T_1 would be finished at $t = 6$ instead of $t = 7$.

$$F_1 = 2 + (1) + 2 + 1 \leq 8 \tag{23}$$
$$6 \leq 8 \tag{24}$$

Preemptible tasks always satisfy $M_i \leq x_i$ condition. As seen in Eq. 23, the time required to execute T_1 is less than the value in Eq. 22.

The following section will explain how the concept using this scheduler is implemented.

Realisation of the framework

"concept" Section explained how architecture should behave when implemented. This section will explain how the RTEF is realised to fulfil these requirements. To actualise a framework that can execute real-time tasks, the OS and hardware that the OS is running on, must be real-time capable. Successful testing of the hardware with random or long-running tests does not make it completely real-time. Formal verification is necessary to guarantee a real-time execution.

The implementation assumed that RTEF is working on an ideal and formally verified hardware that runs a real-time OS. An example implementation based on the architecture is realised using the Java programming language and run in simulation mode, without physical hardware.

Communication

An Edge Network following the architecture is set up only when it has at least one End Device and Edge Server connected to each other. If there are multiple Edge Servers, the creation of the network also requires connections established between them, to benefit from their computing resources. After the connection is established, they introduce themselves to each other by exchanging some set of messages using socket communication. To have an ordered and guaranteed end-to-end message delivery, the framework uses TCP to exchange messages among the participants. The introductory communication flow - handshaking - is both valid for End Device to Edge Server and Edge Server to Edge Server communication, and it is completed when the connection initiator logs in. The flow is illustrated in Fig. 7, and explained further below.

Once a connection is established, the connection initiator is considered as a client. Therefore, when a connection between two Edge Servers is established, the connecting Edge Server also acts as a client. As seen in Fig. 7, after the connection is established, the client starts the handshake by sending a message to an Edge Server. In this case it is a plain "HELLO". Then, the Edge Server responds with an "OLLEH:", immediately appending its ID and name. Next, the client acknowledges this message by sending its ID starting with "ID:", and adding its name to be used in informative messages. After that, the Edge Server sends all known Millions of Instructions Per Second (MIPS) value list of all known servers, separating each server's core count with "+"

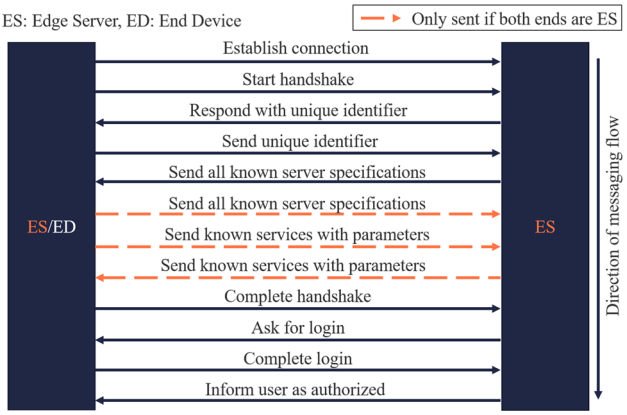

Fig. 7 Initial communication sequence diagram between two Edge Network participants (End Device to Edge Server or Edge Server to Edge Server). Messages shown using dashed arrows are only sent if both ends are Edge Servers

and appending "@" to specify the server, and comma "," to separate each server.

Following the ID exchange, if the communication is between two Edge Servers, then, the client responds with known MIPS values and also sends known available public services starting with "SERVERASCLIENTSERVICES:", followed by service parameters, separated by at sign "@" to specify the located server IDs, and separating these public services by comma ",". An example of this format can be given as:
"SERVERASCLIENTSERVICES:serviceName1+direction 1+ deadline1+mips1+CPUutil1+CPUmask1@serverId1@server-IdX,serviceNameX...". The Edge Server receiving this list updates its known service list along with their locations, broadcasts this information to all other connected servers, and responds to the client with updated 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. Services can define the same PSCs with different parameters; therefore, it is mandatory to exchange this information for decision making while computing the optimal location for the task execution.

After the pre-login process is completed, the Edge Server asks for username and password to authorise 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 an "RDY" message, enabling access to commands. The logged user can be either a regular *user* or an *operator*. A regular user can only execute client-side commands, read-only commands on the server, and request tasks. Operators, however, can execute all commands. Users in Edge Servers are local, meaning each

Table 3 Some commands that can be executed on Edge Servers that utilise RTEF

Command	Description
!connect,IP,PORT	Connects to the edge server at **IP:PORT**.
!disconnect,ID	Disconnects from the server **ID**.
ID,COMMAND	Executes a remote command at server **ID**.
ID,requesttask, servicename	Requests a task on a remote server **ID**.

server must define their own users along with their roles to enable execution.

Components that enable communication (Fig. 7) and play a role in task execution (Fig. 9) will be detailed in "components" Section. The following section will explain the minimal mandatory commands that participants should implement.

Commands

One of the objectives of RTEF is to abstract low-level operations without limiting the functionality. APIs allow backward compatibilities and provide standardised 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.

An Edge Server accepts two types of commands: (1) client-side commands and (2) remote commands. Client-side commands are executed on the server itself, whether a connection is established or not, using the interactive terminal. However, remote commands are executed on an Edge Server that is connected to. Client-side commands start with an exclamation mark "!" directly followed by a command. Remote commands start with the remote server ID, then the command, and the arguments. A comma "," separates each argument of the command. The minimal required client-side commands and remote command to request a task are listed in Table 3.

Services

As explained in "concept" Section, services wrap the PSCs that are installed on the Edge Servers and define how they behave when executed. The service creator must make sure that the behaviour perfectly suits to the PSC, and it is always the same. One PSC, however, can also be wrapped with more than one service, each with different behaviour. The RTEF introduces the following parameters for the software behaviour: *Direction* defines if the service sends a response (TWO) or only receives input (ONE). The *Type* defines the service type, that is either Legacy, Simple, or Simple Periodic. *Worst-Case Execution Time* (WCET) specifies the expected runtime of the software in worst-case conditions. Its given in millions of instructions (MI) as unit. *Relative deadline* limits the latest time that the software is allowed to run and it is converted to the absolute deadline when the software is executed as a task. This value is written in terms of seconds. *CPU utilisation* is the execution utilisation or CPU load in percentage that the PSC is allowed to use. *CPU mask* defines the list of granted CPUs for this software. It is also known as CPU affinity. *Thread per core* characterises how many threads are created on each allowed CPU. Assume that the CPU utilisation is 100%, there are 2 allowed CPUs, and thread per core value is 2. CPU utilisation for each core will be 100%, allocating 50% utilisation for each thread in each core. However, the WCET will not be affected. *Publicity* is a boolean value to determine if the service is visible from other Edge Servers and *public* by default. *Command* links the PSC that is defined by the service, and executed when task is started. *Name* is a user-friendly text to be used to request a task, logging in the server console. There is also a *memory* parameter which is not used at the moment to reduce the complexity of the problem. It is quite challenging to determine the memory used by a running PSC.

Services expect that the PSC and its threads adopt the following assumptions: (1) All threads of the PSCs start and stop at the same time as the main process, (2) threads do not alter low-level system configuration that may affect their execution behaviours, and (3) PSC is independent of other running PSCs, e.g., when preempted, it does not cause a critical section issue.

Tasks

Tasks are the instances of services that track the PSC of each request. 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. A PSC can be linked with multiple services, each defining a different behaviour. 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 or interrupted. As soon as a task instance is created, a unique identifier is assigned. Once a task is terminated, it is no more accessible.

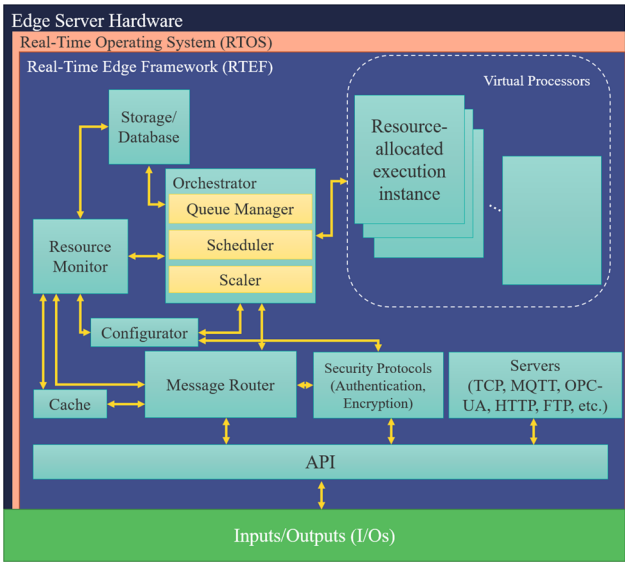

Fig. 8 Software components of the proposed Edge Server architecture in Gezer et al. (2018) and their simplified communication

Components

The proposed architecture in Gezer et al. (2018) embodies the ideas of EC and (near) real-time execution of tasks. The realisation of the entire framework to meet the requirements incorporates many components that work collaboratively. In the following section, these components shown in Fig. 8 will be briefly recalled and their implementations will be explained.

Configurator

RTEF configures itself automatically as soon as it is started. However, manual configuration and tweaks may be necessary during runtime. This component does automatic and manual configuration and detects other Edge Servers in the Edge Network. Additionally, the Configurator also prints log messages into the console.

An Edge Server running RTEF requires a minimum of four configuration keys to function. These values are:

- ID: A unique ID of the Edge Server, which is going to be used in the Edge Network.
- 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 path to a file that specifies the Edge Network Topology. If empty, it is created as the connections are established.

TOPOLOGYFILE is a simple text file that contains information about the participants in the Edge Network. A hash "#" character at the beginning of each line is a comment. Information about an Edge Server or an End Device on each line should have the following format:

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

To add a connection between participants for creation of Edge Topology, the following format is used:

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

Additionally, Configurator also accepts a reserved *AUTOCONNECT* key, which takes *true* or *false* value. If *true*, the TOPOLOGYFILE is read, and pre-defined connections are established, if possible. To achieve that, the *Node* lines must be slightly modified. If the *Free Text* field is replaced with *@IP:PORT*, then the servers establish their previous connections provided that the target Edge Servers are up and running.

Server

Each Edge Server implements a TCP server to communicate. This component is responsible for performing pre-login communication. It also calls the relevant components by parsing the received commands, which can also be called API methods. End Devices or Edge Servers connected to another Edge Server also send their commands through this server. It keeps the list of all connected clients and keeps them updated when a change in resource usage or services are detected. If a connection is disconnected, it is automatically re-established. Some of the accepted commands by the server are explained in "commands" Section.

Message router

Receives the commands and redirects them to their responsible components or Edge Servers. If the command is a task request from an End Device, it communicates with *Resource Monitor* and *Orchestrator* to receive the Edge Server ID that is decided to execute this task. When the chosen Edge Server is not the same as the current one, the task request is forwarded to the relevant Edge Server (See "concept" Section). 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 Edge Device with information on where the requested task is being executed and using which unique task ID. The message uses the following format: "RUNNING:serviceName@EdgeServerID,UniqueTaskID". After

execution is completed, "TASKCOMPLETED" message is also sent. Similarly, at each resource usage, each Edge Server broadcasts the current resources to all connected servers.

Security protocols

The role of this component is to perform authentication and maintain secure communication between the participants. 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 method is called. 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.

Resource monitor

Resource Monitor (RM) keeps track of available resources in the current Edge Server and the connected Edge Servers within the Edge Network. Additionally, it contains the information on the existing services in the network and how the Edge Topology is structured. Each change in the resource usage informs the *Server* component, which updates other Edge Servers. RM makes execution decisions in conjunction with *Orchestrator*. RM performs the simple initial check whether the available resources of the current Edge Server is enough to execute the task until its deadline without any further actions, such as scaling or scheduling. This is usually the case when the requested task is the only task to execute on that Edge Server, or the task has a CPU affinity value that its allowed CPUs do not exceed 100% CPU utilisation when executed. If further actions are required, the RM consults *Orchestrator* for the last decision.

Orchestrator

If RM finds a possibility for the execution of the task in this Edge Server, but with some actions, this component becomes active. This 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. It computes the WCEU following the formula in "concept" Section. If downscaling succeeds, then the tasks can run without further action. Otherwise, the *Scheduler* component is activated. Scaler upscales the downscaled tasks when possible, back to their original CPU execution utilisations, as soon as CPU utilisation is available.

On the hardware and OS level, changing CPUs of threads during runtime can be costly due to context switching. Increasing the available CPU count for tasks also increase the parallel execution overheads, reducing efficiency Lee et al. (2003). Moreover, finding an optimal scheduling diagram on multiprocessors is NP-hard Leung and Whitehead (1982). Lee, Hong, and Kim Lee et al. (2003) introduce a scheduling algorithm that can schedule aperiodic tasks online on multiprocessor systems. However, this requires that the tasks are non-preemptive and they can be decomposed into subtasks. Therefore, slightly deviating from our previous plan in Gezer et al. (2018), we decided to keep the CPU affinities unchanged during execution, leaving the decision to service creator at the beginning.

Scheduler

This component is called when WCEUs of all tasks exceed 100%, and the Edge Server cannot guarantee an on-time execution only by downscaling. The RTEF implements two types of schedulers: EDF scheduler for Simple Periodic services and NAPATA scheduling for Legacy and Simple services. A combination of multiple service types, running their tasks in the Edge Server is not yet fully supported. Currently, this can be achieved by isolating their CPUs, setting a different CPU affinity for each type during service definition. Additional schedulers can also be implemented and integrated. A scheduler is required to provide two methods: (1) A *schedule()* method which returns true or false depending on the schedulability and requires a list of tasks containing their WCETs and absolute deadlines as input, (2) a *sort()* method which returns the sorted scheduling diagram. The schedulers must be implemented in a way that they consider the CPU affinities of the tasks as well. If tasks, including the new task, can be scheduled, this sorted 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 directly forwarded to the most available Edge Server containing the same service, via *Message Router*.

Queue Manager

If tasks are schedulable, the sorted list of scheduling diagram is stored in this component. Then, *Scheduler* picks the first task from the list and executes 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.

Virtual processors

"services" Section explained the parameters that a service defines for software. One of the parameters is the CPU mask,

which lists the allowed CPUs for a task. The scheduling algorithms calculate the feasibilities considering these CPU masks.

Virtual Processors (VPs) work in a similar fashion to control groups (cgroups) in Linux Linux manuals (2020). Cgroups organise the set of processes into hierarchical groups to limit their resource usage and monitor. Likewise, VPs assign tasks to the CPUs and limit their execution utilisations. A new VP can be added during Edge Server creation. A *name* to call later, a *runtime* x and *period* value p to determine the maximum given execution utilisation ($\frac{x}{p}$) Linux Documentation (2020), and a *CPU mask* Linux Documentation (2021) is provided as input. Multiple VPs can reuse the same CPU(s). The tasks in this work can get up to 100% CPU utilization and unlike stated in Linux Documentation (2021), runtime value in this work cannot exceed the period value, meaning $\frac{x}{p}$ cannot be greater than 1. VPs can be used to isolate Periodic services from Legacy and Simple services, as running a combination of all types is not entirely yet supported. Tasks can be assigned to the VPs while they are running or during service creation.

Other components

There are also other components such as *Cache* or *Storage/Database*. These components are not explained again since they do not require anything specific to function, and are indirectly used as they are.

This section completed explaining the components. The next section will validate the architecture as well as the RTEF using a complex scenario with multiple Edge Servers.

Requesting tasks

The architecture prevents execution of tasks directly as they arrive. This is due to the decision algorithms that come into the play. Once a request is received, set of calculations are made in the background. As it may be noticed, this operation is not called "start", but a "request". The flow that is followed by the RTEF after getting a task request is shown in Fig. 9 and explained below.

After each connection between Edge Servers a list of the available resources and services together with their locations are exchanged ("communication" Section). Once a request is received, first, the current server queries the possible locations for the requested task, including itself. 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. In case multiple alternatives can execute the task on time, the server which can execute the task in the shortest time is chosen. If the resources are the same, then, the server with the smallest delay is chosen. If delays are also the same, then, one of the servers is chosen randomly.

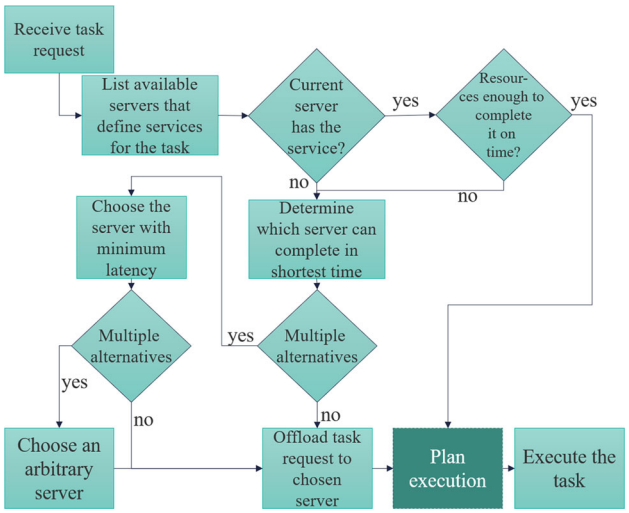

Fig. 9 Simplified flow that shows the process of executing a task after its request

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.

Once the task request is offloaded, if the chosen server is busy with other tasks, the *Orchestrator* determines whether scaling, scheduling, queueing is necessary. Finally, the task is executed. If task returns a result, it is sent back to its original requested.

Test case

The framework is tested on a virtual environment in simulation mode, with an Edge Network of two Edge Servers and four End Devices. In the simulation mode, the hardware is assumed to be ideal and real-time capable. First server has two identical CPUs and the second server has only a single CPU. The first server is connected to the second server, and 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. Services A, B, C, D, E, F are defined with the parameters listed in Table 4. These services are linked with the load generator of the framework to be run when called. Load generator allows creation of ideal tasks based on the defined parameters. Service A is created in both servers, using the same behaviour. Services B, C, D, and E are created only on the first server, and Service F only on the second server. End Device 1, 2, and 3 request tasks from Edge Server 1 at $t = 0$ 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

Table 4 An example set of pre-defined services with defined execution behaviours of tasks.

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

All tasks are one-directional

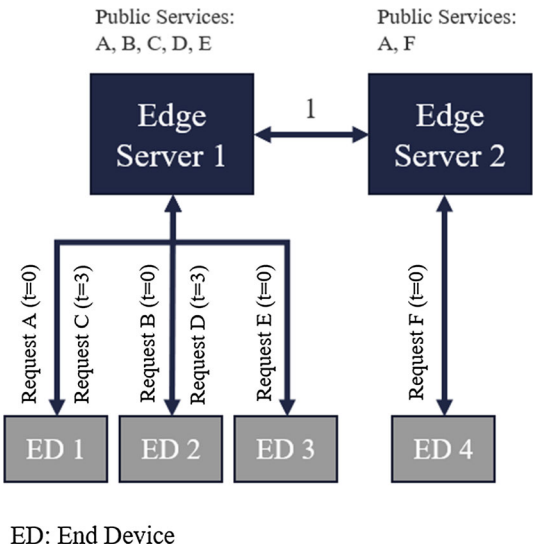

Fig. 10 Experiment setup and plan for an Edge Network using four End Devices and two Edge Servers

End Device 2 requests D. The experiment setup and plan are summarised in Fig. 10.

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. As there is no available resource at the only allowed CPU (CPU 1) to execute A before its deadline, the possibility of scaling is evaluated. Since the WCEU B is 100% ($\frac{3}{3}100$), the downscaling of B will not be possible. Since A and B are not periodic tasks, NAPATA scheduling can evaluate whether it is possible to schedule these tasks. B has higher priority than A, hence, the feasibility check starts with task B.

$$F_B = 0 + M_B \leq d_B + a_B \tag{25}$$
$$= 0 + 3 \leq 3 + 0 \tag{26}$$
$$= 3 \leq 3 \tag{27}$$

As the inequality 27 holds, the algorithm then proceeds with task A.

$$F_A = 0 + M_A + M_B \leq d_A + a_A \tag{28}$$
$$= 0 + 6 + 3 \leq 8 + 0 \tag{29}$$
$$= 9 \leq 8 \tag{30}$$

The feasibility test for A fails since inequality 30 does not hold. Scheduling A after B causes A to miss its deadline. Therefore, another alternative server within the network is searched. 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).

At $t = 1$ on Edge Server 2, the possibility of on-time execution of tasks F and A are calculated using Eq. 10 from NAPATA scheduling. Starting from the highest priority, the feasibility calculation of task A (F_A):

$$F_A = 1 + M_A \leq d_A + a_A \tag{31}$$
$$= 1 + 6 \leq 8 + 0 \tag{32}$$
$$= 7 \leq 8 \tag{33}$$

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

$$F_F = 1 + M_F + M_A \leq d_F + a_F \tag{34}$$
$$= 1 + 1 + 6 \leq 9 + 0 \tag{35}$$
$$= 8 \leq 9 \tag{36}$$

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 is resumed to complete its remaining execution time. C and D task requests arrive at $t = 2$. Both have one thread at each CPU due to their CPU mask and thread per core parameters. At this time, Edge Server 1 is available. Downscaling WCEUs of both tasks to 50% doubles execution times of both tasks, but they can still complete their executions before missing their deadlines. The resulting scheduling diagram is shown in Fig. 11. The framework uses scheduling as a fallback in case downscaling does not

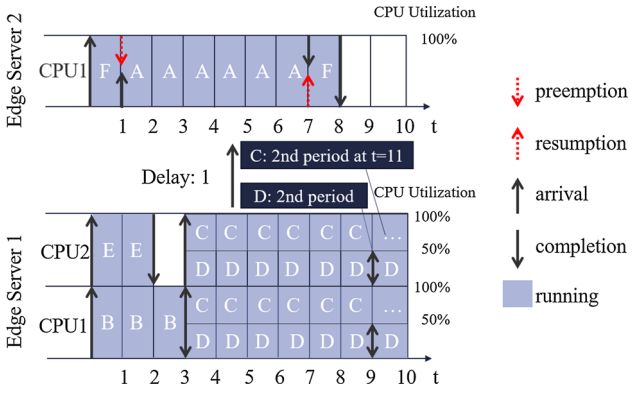

Fig. 11 Resulting scheduling diagram based on the experiment defined in Fig. 10 and Table IV

work. Nonetheless, it was also possible to schedule *C* and *D* if scaler had been disabled.

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 (See "requesting-tasks" Section). Assuming that that *F* has an ID of one (1) and *A* two (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*.

The service creator is expected to create a feasible distribution of tasks in Edge Servers. 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 a 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.

Conclusion and future work

Edge Computing (EC) is a paradigm that introduces a new tier between the device tier and the Cloud tier. It enables lower-delay servicing by providing computation power close to this tier. This work implemented the novel architecture in the EC domain, which was formerly conceptually proposed in Gezer et al. (2018). In this paper, the realised framework of this architecture is called the Real-Time Edge Framework (RTEF). The framework allows the execution of (near) real-time tasks within their defined deadlines. The hardware that utilises this framework is called an Edge Server. If a network contains multiple Edge Servers connected using any available topology arrangement, the framework is able to offload task requests to another server if resources are insufficient to execute a task on time. The network can also include a Cloud server. In this case, as a fallback, the task will be executed using a best-effort approach in the Cloud. This work also introduced a novel scheduling algorithm, called Non-preemptible And Preemptible Aperiodic TAsk (NAPATA) scheduling. As its name suggests, it is used to schedule non-preemptible and preemptible aperiodic tasks. The algorithm is provided with a feasibility equation and explained using two examples.

This work follows Liu and Layland's assumptions Liu and Layland (1973) for task scheduling. One of the assumptions is that the tasks are independent of each other. However, tasks can depend on each other, or require that a previous task is completed before its execution. In the future, the algorithms will support a graph to schedule the tasks in a specific order, defined by their services. Another plan is to offload a *running* task to another Edge Server if a higher priority task needs to run on a specific Edge Server. Currently, running tasks can be preempted, but cannot be offloaded to continue from where they left off. Moreover, the research will be conducted to fully support scheduling of aperiodic and periodic tasks together. Lastly, it is assumed that the algorithmic calculations and internal computations have no impact on time and cause no overheads. Nevertheless, in real environments, this is not the case. Performance evaluations of the architecture and framework will be performed on different hardware to get a view on time performance.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

Audsley, N. C., Burns, A., Richardson, M. F., & Wellings, A. J. (1991). Hard real-time scheduling: The deadline-monotonic approach. *IFAC Proceedings Volumes*, *24*(2), 127–132.

Berry, G. (2007). "SCADE: Synchronous Design and Validation of Embedded Control Software," In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and P. Sampath, Eds. Dordrecht: Springer Netherlands, (pp. 19–33).

Chang, H., Hari, A., Mukherjee, S., & Lakshman, T.V. (2014). "Bringing the cloud to the edge," In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 04, (pp. 346–351).

CLOUDS Laboratory. (2019). CloudSim: A Framework for modeling and simulation of cloud computing infrastructures and services. [retrieved: Jan 2021]. [Online]. Available: http://www.cloudbus.org/cloudsim/.

Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., & Weihl, B. (2002). Globally distributed content delivery. *IEEE Internet Computing*, *6*(5), 50–58.

EdgeX Foundry. (2020). EdgeX foundry architectural tenets. [retrieved: Jan 2021]. [Online]. Available: https://docs.edgexfoundry.org/2.0/.

Elbamby, M. S., Bennis, M., & Saad, W. (2017). "Proactive edge computing in latency-constrained fog networks," In *2017 European Conference on Networks and Communications (EuCNC)*, 06, (pp. 1–6).

Feldhorst, S., Libert, S., ten Hompel, M., & Krumm, H. (2009). "Integration of a legacy automation system into a SOA for devices," *Proceedings of the IEEE Conference on Emerging Technologies Factory Automation*, (pp. 1–8).

Gezer, V., & Wagner, A. (2020). "Real-time edge framework (rtef): Decision making for offloading and task scheduling," *Manuscript submitted for publication*.

Gezer, V., Um, J., & Ruskowski, M. (2018). An introduction to edge computing and a real-time capable server architecture. *The International Journal on Advances in Intelligent Systems*, *11*(1&2), 105–114,07.

Givehchi, O., Imtiaz, J., Trsek, H., & Jasperneite, J. (2014). "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)*, 05, (pp. 1–4).

Goldhar, J. D., & Jelinek, M. (1990). Manufacturing as a service business: Cim in the 21st century. *Computers in Industry*, *14*(1), 225–245. special Issue Josef Hartvany Memorial.

Goldschmidt, T., Murugaiah, M.K., & Sonntag, C. (2015). "Cloud-based control: a multi-tenant, horizontally scalable soft-PLC," In *IEEE 8th International Conference on Cloud Computing*.

Harshit, G., Dastjerdi, A.V., Ghost, S.K., & Buyya, R. (2016). "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*, 06.

Horn, C.,& Krüger, J. (2016). "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)*, 09, (pp. 1–4).

IBM Cloud Architecture Center. (2017). IBM: Internet of Things. [retrieved: Jan 2021]. [Online]. Available: https://www.ibm.com/cloud/garage/architectures/iotArchitecture/reference-architecture.

Kretschmer, F. (2016, 09). Projekt – pICASSO. [retrieved: Jan 2021]. [Online]. Available: https://industrie40.vdma.org/documents/4214230/21848134/piCASSO_1510147125829.pdf/9a64d3eb-c397-401f-893a-9994c70bcc12.

Lee, W. Y., Hong, S. J., & Kim, J. (2003). On-line scheduling of scalable real-time tasks on multiprocessor systems. *Journal of Parallel and Distributed Computing*, *63*(12), 1315–1324.

Lehoczky, J.P., Sha, L.R., & Strosnider, J.K. (1987). "Enhanced aperiodic responsiveness in hard real-time environments." In *Unknown Host Publication Title*. IEEE, (pp. 261–270).

Leung, J. Y.-T., & Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, *2*(4), 237–250.

Lhotka, R. (2005). *Should all apps be n-tier?* Blog [retrieved: Jan 2021]. [Online]. Available: http://www.lhotka.net/weblog/ShouldAllAppsBeNtier.aspx.

Linux Documentation. (2020). Real-time group scheduling. [retrieved: Jan 2021]. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt.

Linux Documentation. (2021). CFS bandwidth control. [retrieved: Jan 2021]. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt.

Linux Documentation. (2021). CPUSETS. [retrieved: Jan 2021]. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt.

Linux manuals. (2020, 03). Linux Programmer's Manual - cgroups(7). [retrieved: Jan 2021]. [Online]. Available: http://man7.org/linux/man-pages/man7/cgroups.7.html.

Liu, C.L., & Layland, J.W. (1973). "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM Vol. 20*.

Mohamed, N., Lazarova-Molnar, S., Jawhar, I., & Al-Jaroodi, J. (2017). "Towards service-oriented middleware for fog and cloud integrated cyber physical systems," In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 06, (pp. 67–74).

Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., & Walker, K. R. (1997). Agile application-aware adaptation for mobility. *SIGOPS Operaton System Review*, *31*(5), 276–287, 10.

OpenFog Consortium. (2017). OpenFog consortium reference architecture for fog computing. [retrieved: Jan 2021]. [Online]. Available: https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf.

Pallasch, C., Wein, S., Hoffmann, N., Obdenbusch, M., Buchner, T., Waltl, J., & Brecher C. (2018). "Edge powered industrial control: Concept for combining cloud and automation technologies," In *2018 IEEE International Conference on Edge Computing (EDGE)*, 07, (pp. 130–134).

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, *3*(5), 637–646, 10.

Sonmez, C., Ozgovde, A., & Ersoy, C. (2017). "Edgecloudsim: An environment for performance evaluation of edge computing systems," In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, 05, (pp. 39–44).

Sonmez, C., Ozgovde, A., & Ersoy, C. (2017). "Performance evaluation of single-tier and two-tier cloudlet assisted applications," In *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, 05, (pp. 302–307).

Sprunt, B., Sha, L., & Lehoczky, J. (1989). "Scheduling sporadic and aperiodic events in a hard real-time system," Software Engineering Institute, Carnegie Mellon University. *Tech. Rep.* (**CMU/SEI-89-TR-011**).

The Computer Language Co Inc. (2021). Definition of end device. PC Magazine - Website. [retrieved: Jan 2021]. [Online]. Available: https://www.pcmag.com/encyclopedia/term/64886/end-device.

Vick, A., Horn, C., Rudorfer, M., & Krüger, J. (2015). "Control of robots and machine tools with an extended factory cloud," In *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, 05, (pp. 1–4).

VMware. (2017). VMware Introduces Liota. [retrieved: Jan 2021]. [Online]. Available: https://www.vmware.com/radius/vmware-introduces-liota-iot-developers-dream/.

White, J. (1971). "Network Specifications for Remote Job Entry and Remote Job Output Retrieval at UCSB," Internet Engineering Task Force, Internet Standard, 03 1971, [retrieved: Jan 2021]. [Online]. Available: https://tools.ietf.org/html/rfc105.

Yin, S., Bao, J., Zhang, J., Li, J., Wang, J., & Huang, X. (2020). Real-time task processing for spinning cyber-physical production

systems based on edge computing. *Journal of Intelligent Manufacturing*, *31*(8), 2069–2087, 12. https://doi.org/10.1007/s10845-020-01553-6. [Online]. Available:

Zhang, L., Guo, H., Tao, F., Luo, Y. L., & Si, N. (2010). "Flexible management of resource service composition in cloud manufacturing," In *2010 IEEE International Conference on Industrial Engineering and Engineering Management*, 12, (pp. 2278–2282).

Zhang, L., Luo, Y. L., Tao, F., & Ren, H. G. L. (2010). Key technologies for the construction of manufacturing cloud. *Computer Integrated Manufacturing Systems*, *16*(11), 2510–2520.