Internet of Things 21 (2023) 100651

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

Research article

Towards a lightweight task scheduling framework for cloud and edge platform

Thomas Dreibholz[a], Somnath Mazumdar[b,*]

[a] Centre for Resilient Networks and Applications, Simula Metropolitan, Pilestredet 52, Oslo 0167, Norway
[b] Department of Digitalization, Copenhagen Business School, Solbjerg Plads 3, Frederiksberg 2000, Denmark

ARTICLE INFO

Keywords:
Cloud
Edge
Framework
Placement
Task

ABSTRACT

Mobile devices are becoming ubiquitous in our daily lives, but they have limited computational capacity. Thanks to the advancement in the network infrastructure, task offloading from resource-constrained devices to the near edge and the cloud becomes possible and advantageous. Complete task offloading is now possible to almost *limitless* computing resources of public cloud platforms. Generally, the edge computing resources support latency-sensitive applications with limited computing resources, while the cloud supports latency-tolerant applications. This paper proposes one lightweight task-scheduling framework from cloud service provider perspective, for applications using both cloud and edge platforms. Here, the challenge is using edge and cloud resources efficiently when necessary. Such decisions have to be made quickly, with a small management overhead. Our framework aims at solving two research questions. They are: (i) How to distribute tasks to the edge resource pools and multi-clouds? (ii) How to manage these resource pools effectively with low overheads? To answer these two questions, we examine the performance of our proposed framework based on Reliable Server Pooling (RSerPool). We have shown via simulations that RSerPool, with the correct usage and configuration of pool member selection policies, can accomplish the cloud/edge setup resource selection task with a small overhead.

1. Introduction

Cloud computing has become the “go-to” solution for complex, latency-tolerant, and long-running applications. The pay-as-you-go pricing model also offers cloud services cheaper than on-premise clusters without management overheads. Generally, cloud computing is better suitable for latency-sensitive applications. For instance, it has been seen that cloud platforms are compatible towards applications that can tolerate delays up to 100 ms [1]. Edge Computing (EC) is becoming popular to counter such latency-related issues, thanks to the progress in the network infrastructure ecosystem [2,3]. In general, edge units are placed between the cloud and the user, while resourcefully supported by the cloud.

Mobile computing devices are becoming an indispensable part of our daily lives. Such devices are used for many online activities. In this paper, these mobile devices (including smartphones) are denoted as User Equipment (UE). Recent UEs have decent computational and storage capabilities. In some scenarios, they are limited by the steadily increasing amount of user data and computational demand. The cloud becomes a viable solution to offload tasks that require immense computation power and storage. Tasks[1] are offloaded to the nearby edge units to improve the applications’ performance [4]. Edge can be considered as a small subset

* Corresponding author.
E-mail address: sma.digi@cbs.dk (S. Mazumdar).
[1] We define a task as a single unit of work of an application. An application may consist of multiple tasks.

https://doi.org/10.1016/j.iot.2022.100651
Received 30 October 2022; Received in revised form 25 November 2022; Accepted 28 November 2022
Available online 11 December 2022

of the cloud platform. It is worth mentioning that cloud platforms do not provide low latency between the cloud and UEs, which may impact latency-sensitive applications' performance.

To date, EC is not mature and efficiently offloading tasks to EC is not trivial. Managing distributed compute resource pools without adding a large amount of management overhead is difficult. Existing literature can broadly be divided into two categories. One category primarily proposes strategies to optimise objective functions such as latency and energy using various algorithms or models. The other presents architectures and frameworks for improving the quality of service (QoS) via task offloading [2,5,6]. Lin et al. found four popular computation offloading techniques in their EC-focused survey. They are offloading mode-based, computing model, channel model, and energy harvesting model-based techniques [2].

In our previous work [7], we presented how Reliable Server Pooling (RSerPool) [8] manages resource pools and handles the application sessions effectively. RSerPool is an Internet Engineering Task Force (IETF) standard for a light server pooling approach that initially targeted server redundancy. As an extension of previous work, we examine RSerPool using different server selection policies with extensive parameter ranges by incorporating the edge structure. More specifically, this work aims to offload tasks from battery-powered, resource-constrained mobile devices to edge/cloud platforms. The tasks are small but latency-sensitive. Finding an optimal task-to-resource mapping process may consume more resources (such as time and computation). Therefore, our idea is to (re-)use a lightweight framework for task allocation. Current work covers mode-based and computing model-based offloading techniques. Specifically, it supports binary offloading mode, which means the whole task will be offloaded. The computing model focuses on latency, which we aim to optimise. We conducted the performance evaluation of resource selection policies, using simulations of an edge and multi-cloud setup with the RSerPool Simulation (RSPSIM) model [9, Chapter 6]. Our contributions are:

- We proposed a lightweight task scheduling framework from a cloud service provider perspective.
- We aimed to provide a simple, efficient, low-overhead, open-source solution which can perform well via the proper choice and configuration of policies.
- We compare the different policies using three popular models: round-robin, random and least used. We have reported pool usage, task processing speed, and queuing, startup and processing times of these policies for low, medium and high workload scenarios, also including overload scenarios.
- We have demonstrated the applicability of our proposed framework with a proof-of-concept using a cloud/edge research test bed.

2. Background: Reliable Server Pooling (RSerPool)

Offloading tasks from mobile devices to edge and even multi-cloud resources primarily depends on the availability of resources and workload type. Managing server pools and sessions (between clients and servers) is a traditional problem in computer networking. To avoid "reinventing the wheel" for each application, the IETF set up the RSerPool working group to develop a generic standard. RSerPool is an application-independent and open-source framework to be simple and lightweight. RSerPool is also suitable for devices with minimal resources, which is ideal for edge environments. RSPLIB[2] is the open-source implementation of RSerPool. Apart from that, the simulation model RSPSIM[3] has also been developed. Our approach combines existing tools to provide a simple, efficient, low-overhead, open-source solution. The RSerPool architecture is shown in Fig. 1, where a resource pool constitutes a set of servers providing a certain service. Servers in a pool are denoted as Pool Elements (PE). A pool is identified by its unique Pool Handle (PH, e.g. a string like "Offloading Pool") within its operation scope. The handlespace is the set of all pools of an operation scope. It is managed by Pool Registrars (PR, also denoted as Registrars). RSerPool setups should consist of at least two registrars, in order to avoid a single point of failure. The PRs synchronise the handlespace using the Endpoint haNdlespace Redundancy Protocol (ENRP) [10]. An operation scope is limited to an entity (such as a company). It does not scale to the whole Internet, in contrast to, e.g. the Domain Name System, which is a significant simplification that keeps RSerPool lightweight concerning the administrative overheads. Pools can be distributed over large geographic areas to achieve a high resilience of services.

PEs use the Aggregate Server Access Protocol (ASAP) [11] to dynamically register to, or de-register from, a pool at one of the PRs of their operation scope. The PR chosen for registration becomes the Home-PR (PR-H) of the PE. It also monitors the availability of the PE by a keep-alive mechanism. Clients, denoted as Pool Users (PU) in the context of RSerPool, use ASAP to access the resources of a pool. A PU can query a PR of the operation scope to select PE(s). This selection is performed by using a pool-specific pool member selection policy [12], shortly denoted as pool policy. We introduce pool policies later in Section 4.2. ASAP may also be used between PU and PE, then realising a *Session Layer* functionality between a PU and a pool. In this case, ASAP can also be used to assist the actual *Application Layer* protocol in handling failovers and helping with state synchronisation [12].

By default, ASAP and ENRP use the Stream Control Transmission Protocol (SCTP) [13] as their Transport Layer protocol. For 4G/5G networks, SCTP is already required for the signalling transport between the Enhanced Packet Core (EPC) components. The requirement of SCTP likely limited the deployment of RSerPool in the past, since e.g. Microsoft Windows only supports SCTP after installation of a third-party commercial implementation. However, Linux and FreeBSD-based UEs and servers usually provide stable SCTP support out of the box.

[2] RSPLIB: https://www.uni-due.de/~be0001/rserpool/#Download.
[3] RSPSIM: https://www.uni-due.de/~be0001/rserpool/#Simulation.

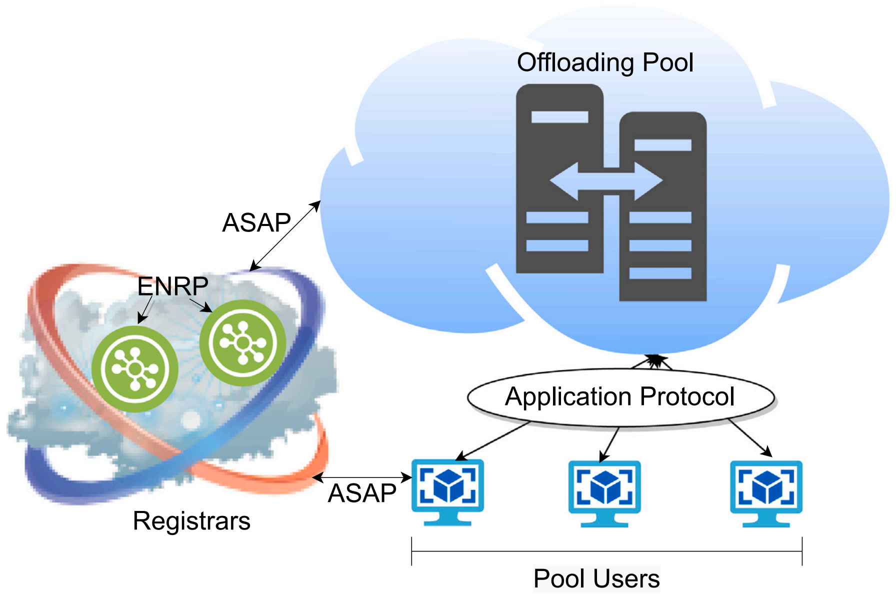


Fig. 1. Illustration of the RSerPool architecture.

3. Related works

An effective task distribution technique is required to reduce latency and optimise bandwidth usage. We refer readers to surveys [2,6] related to existing offloading modelling techniques for edge and to [5] for machine learning-based offloading methods.

Alamouti et al. propose a hybrid cloud/edge model to minimise network bandwidth usage and reduces latency via resource delegation on other hardware devices of the cloud/edge platform [14]. Similarly, Sebrechts et al. analyse the impact by focusing on CPU usage, network latency and QoS while distributing microservice-based applications over a cloud/fog platform [15]. Nikolopoulos et al. propose a policy-based controller for fog nodes to achieve better scalability and context awareness, focusing on response time and task completion time [16]. Kishor et al. propose an ant colony optimisation-based task offloading algorithm to offload the IoT tasks in a fog environment [17]. It aims at minimising task offloading time by considering computation and communication latency.

Adhikari et al. propose an offloading strategy for IoT applications considering energy consumption and computation time [18]. It offers a method to select a computation device for each application. In another work, an edge process management is proposed for mobile devices [19]. It offers a delay-aware fog server selection scheme for cost optimisation based on user movement, load and location factors. Again, Adhikari et al. designed another delay-dependent priority-aware IoT-based task offloading strategy to reduce task waiting time. It assigns each task a priority based on its deadline and assigns it to a suitable multilevel-feedback queue [20].

Chen et al. propose a mixed integer non-linear programming-based task offload scheme for cloud/edge platforms [21]. It splits the problem into two sub-problems to minimise task completion and resource allocation. Chen et al. showed that reducing energy cost, computation cost, and the delays for a multi-user mobile cloud computing platform via optimal tasks offloading is an instance of a non-convex quadratically constrained quadratic program [22]. Zhang et al. propose one task offloading scheme to minimise task delay. It aims to follow fairness while offloading tasks among the fog nodes [23]. Jiang et al. present an energy-efficient task offloading mechanism by considering all devices' computation and communication resources in a fog environment [24]. Here, the tasks are composed of several sub-tasks with deadlines. A scheduler is proposed to meet response time requirements. Rashidi et al. propose an adaptive neuro-fuzzy inference system to assign tasks to the mobile cloud servers for optimising performance and QoS [25]. However, achieving an optimal task placement in a distributed computing platform is a constrained problem [26].

4. Proposed task offloading framework

Our proposed offloading framework has two parts. The first part is responsible for setting up the RSerPool framework, and the second part consists of server pool member selection policies.

4.1. RSerPool scenario with edge and (multi-)clouds

Fig. 2 shows the basic edge and (multi-)clouds platform setup. UEs are running applications that aim to offload tasks to the cloud. The cloud can instantiate a container that runs the assigned task and later releases it. Support of containerisation offers

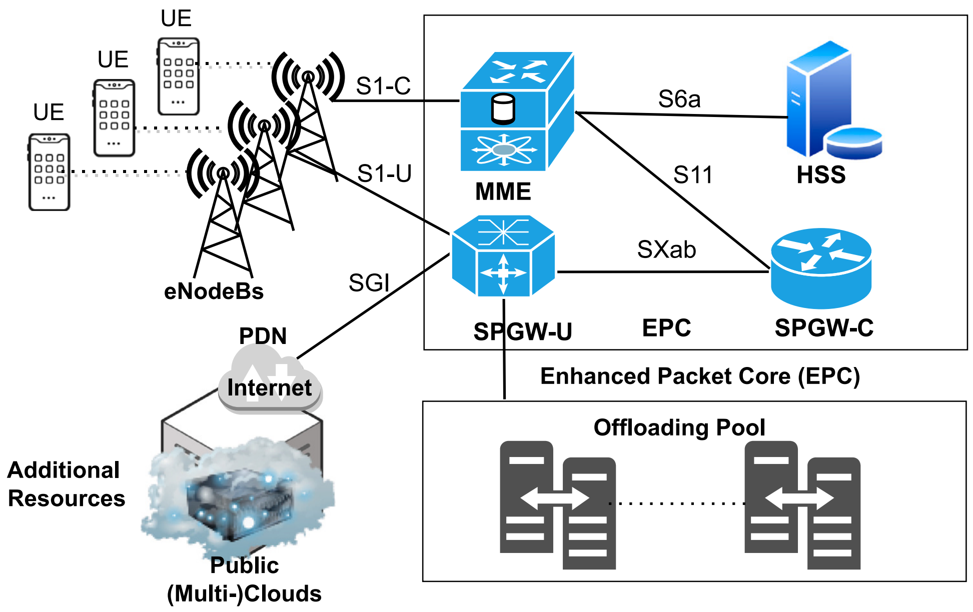


Fig. 2. Application scenario with edge and (multi-)clouds.

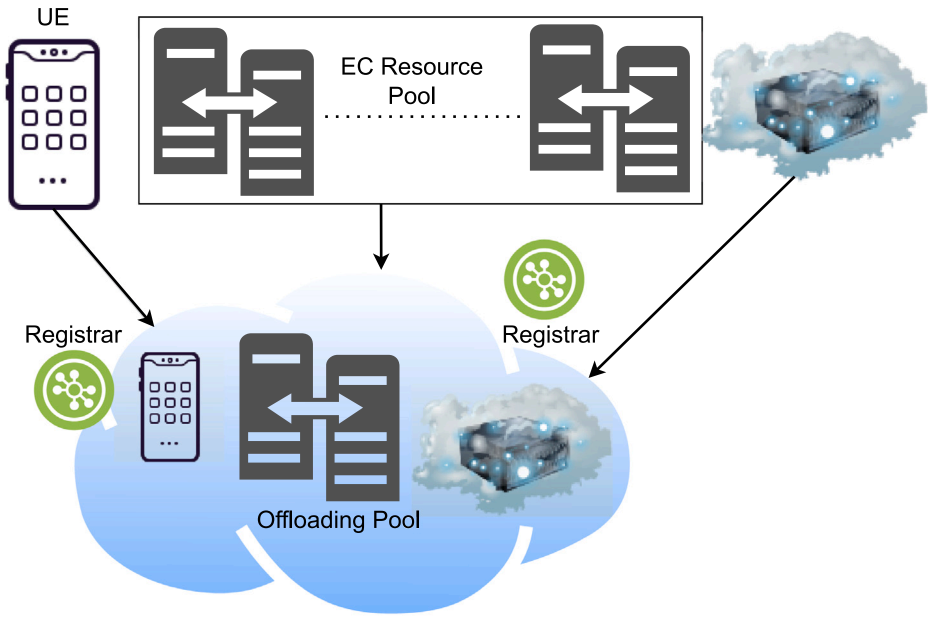


Fig. 3. RSerPool with edge computing and multi-cloud resources.

task offloading support for different application types. Cloud resources are provided by the local edge nodes and public (multi)-clouds (PMC). Supporting PMC not only offers a better cost perspective but also better latency. Distant cloud resources offer a high round trip time (RTT), as distance adds network latency to the service request handling. If available, the local edge resource is preferred. Now one question can arise: *how to manage such diverse resource pools, consisting of EC and PMC resources?*

In Fig. 3, we describe our approach to applying the RSerPool framework. Resources (i.e. servers) are added into a pool, identified by its PH (here: "Offloading Pool"). It means the pool consists of PEs in the EC and a PMC setup. Following the CAP theorem [27], a distributed system can only support two features of Consistency, Availability, and Partition tolerance. The primary purpose of RSerPool is high availability. Since RSerPool has a small management overhead, it is possible to instantiate a PE instance on the UE itself. This means it is possible to add UE resources to the pool. If everything fails, then the application on the UE can still "offload" a task to itself (i.e. the PE on the UE). It is similar to the situation when there is no network coverage. In such events, the application

Table 1
List of pool member selection policies.

Abbrev.	Full Name	Type
RAND	Random	Non-Adaptive
RR	Round-Robin	Non-Adaptive
LU	Least used	Adaptive
LUD	Least used with degradation	Adaptive
PLU	Priority least used	Adaptive
PLUD	Priority least used with degradation	Adaptive
PLU-DPF	Priority least used with distance penalty factor	Adaptive with DPF
PLUD-DPF	Priority least used with degradation and DPF	Adaptive with DPF

may still provide a valuable service for its user, with reduced performance and increased UE energy consumption. Edge and PMC resources are always preferable in these scenarios. It is worth noting that Edge and Fog both aim to add a computing resource layer that sits between the user and the cloud to offer lower network latency. The edge platform is more resource-constrained, while the Fog platform is better resourcefully equipped. RSerPool is simple and limited in its capabilities, while it offers a small overhead. Serverless computing supports function/method level execution with limited resources and may be ideal for one use case.

The pool consists of three types of resources: EC, PMCs, and UE. In addition, PRs are needed for managing the handlespace. To avoid a single point of failure, it needs at least two: One in the EC and another one in the PMCs. In case of network coverage loss, a PR instance must run on the UE for standalone operations. This is possible, as PRs are lightweight, meaning they have only low memory and CPU requirements [9, Chapter 7]. A relevant question may be: *how can the PRs finally handle the different server types in the pool?*

4.2. Pool member selection policies

Handling the different server types in the pool mostly concerns how the pool member selection policy is applied. It performs the selection of a suitable PE, and it must achieve the following four goals in our scenario:

- Goal 1: Only use UEs if there is no other possibility (such as no network coverage or lack of EC/PMC servers).
- Goal 2: Use PMCs servers only when they are a suitable choice (such as when the edge servers are highly utilised).
- Goal 3: Otherwise, use the edge servers.
- Goal 4: Apply load balancing.

Table 1 lists the pool policies used in this article and their abbreviations [9, Section 3.11]. We briefly introduce them below:

- The policies Random (RAND) and Round Robin (RR) are non-adaptive. They are configured statically and do not need up-to-date information from the PEs. RAND randomly selects PEs of a pool, while RR selects them using a round-robin process. Since all PEs are handled equally, neither RAND nor RR can satisfy the first three goals.
- Least Used (LU) selects the PE p where its load L_p is lowest. Up-to-date load information is needed. It is an adaptive policy. In case of multiple PEs with the same lowest load (e.g. three PEs with load of 0%), round-robin or random selection is applied among these least-loaded PEs. It will not differentiate between edge units, PMC servers and UEs. LU also cannot satisfy the first three goals mentioned above.
- Priority Least Used (PLU) adds a PE-specific load increment constant I_p to LU. PEs are chosen based on the lowest sum $L_p + I_p$. By setting $I_{p_{\mathrm{EC}}} < I_{p_{\mathrm{PMC}}}$ for all EC PEs p_{EC} and PMCs PEs p_{PMC}, and $I_{p_{\mathrm{UE}}} = 100\%$ for the UE PE, PLU should achieve all four goals.
- Least Used with Distance Penalty Factor (LU-DPF) adds a PE-specific Distance Penalty Factor (DPF) constant D_p to LU. Next, the PEs are chosen based on the lowest sum

  $$L_p + \mathrm{RTT}_p * D_p$$

  where the RTT from the selecting PR to the PE p, RTT_p, is the approximated:

  $$\mathrm{RTT}_p = \begin{cases} \mathrm{RTT}_{\mathrm{PR-H}\leftrightarrow\mathrm{PE}} & \text{(on PR-H)} \\ \mathrm{RTT}_{\mathrm{PR}\leftrightarrow\mathrm{PR-H}} + \mathrm{RTT}_{\mathrm{PR-H}\leftrightarrow\mathrm{PE}} & \text{(on any other PR).} \end{cases}$$

  For simplicity, D_p can be the same for all PEs, making it a pool-specific constant. Assuming

  $$\mathrm{RTT}_{p_{\mathrm{EC}}} \ll \mathrm{RTT}_{p_{\mathrm{PMC}}}$$

  for all EC PEs p_{EC} and PMCs PEs p_{PMC}, LU-DPF should achieve all goals except the first one. However, the first goal is likely to be violated in this case since $\mathrm{RTT}_{p_{\mathrm{UE}}}$ may (mostly) be minimal. We extended LU-DPF to a new policy Priority Least Used with Distance Penalty Factor (PLU-DPF) by adding a PE-specific load increment constant I_p as for PLU.

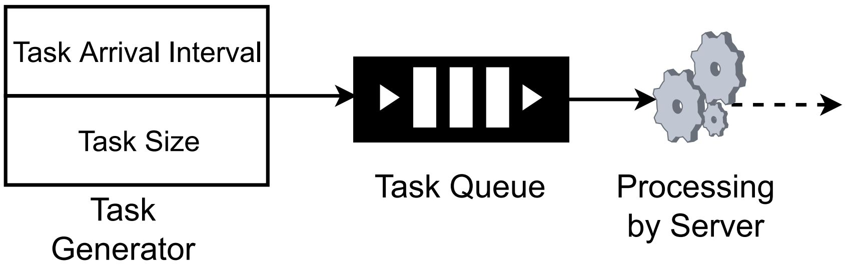


Fig. 4. The task generation and handling process.

- LU and its variants are adaptive policies. It means they require up-to-date load information from the PEs in the handlespace, which the PRs manage. A PR-H has to be updated with the load states of its PEs (by re-registration via ASAP). Then, it distributes the updates to the other PRs (via ENRP). So, propagating load updates takes time, leading to temporary inaccuracy. The Least Used with Degradation (LUD) [28] policy adds a load degradation variable X_p for each PE p. Each time a PR selects a PE, it increases the load degradation by the load increment constant I_p. On update from the PE, X_p is reset to zero. Then, PEs are chosen based on the lowest sum $L_p + X_p$. We combined the idea of PLU with Priority Least Used with Degradation (PLUD), choosing a PE by the lowest sum of $L_p + I_p + X_p$. Also, we combined it with DPF to a new policy Priority Least Used with Degradation and DPF (PLUD-DPF), selecting a PE by the lowest sum of

$$L_p + I_p + X_p + \mathrm{RTT}_p * D_p.$$

We have used the policies mentioned above to answer *Which policies are helpful for our use case? And which policy should be used for a UE/EC/PMC setup?*

5. Simulation and results

To examine the performance of the pool policies, we use the RSPSIM simulation model for RSerPool to create a setup as depicted in Fig. 3. RSPSIM is based on OMNeT++ 6.0.1 and has been extended to support policies such as PLUD, PLU-DPF and PLUD-DPF. The simulation is completed using Ubuntu Linux, running on 2x Intel(R) Xeon(R) CPU E7-8860 v4@2.20 GHz (2 CPUs, 18 cores per CPU, two threads/core means 72 threads). It is worth noting that the simulation results are entirely independent of the hardware.

5.1. Application description

The CalcAppProtocol application model [9, Section 8.3] is used as our benchmark application. It is a part of RSPSIM and also of RSPLIB. In the application model, a PE has a task handling *capacity* given in the abstract unit of calculations per second (calculations/s). An arbitrary application-specific metric for capacity may be mapped to this definition (such as CPU operations, processing steps, or disk space usage). Each task has a *task size* (in calculations), which is the number of calculations consumed by the processing of the task.

Tasks are generated and queued, for sequential processing as illustrated in Fig. 4. Following the multi-tasking principle, a PE can process multiple tasks simultaneously. The available capacity is shared among all currently processed tasks, as illustrated in Fig. 5: a single task gets 100% of the capacity. When another task is accepted, each gets 50%, doubling the required processing time per task. When a task is finished, the share is recomputed and the processing time per task reduces accordingly. The user-side performance metric is the handling speed. The total time for handling a task d_{Handling} (in seconds) is defined as the sum of:

1. Queuing time d_{Queuing},
2. Startup time d_{Startup} (de-queuing until reception of acceptance acknowledgement) and
3. Processing time $d_{\mathrm{Processing}}$ (acceptance until finish).

That is:

$$d_{\mathrm{Handling}} = d_{\mathrm{Queuing}} + d_{\mathrm{Startup}} + d_{\mathrm{Processing}}. \quad (1)$$

The *handling speed* (in calculations/s), for a given task size $\mathrm{TaskSize}$, is defined as:

$$\mathrm{HandlingSpeed} = \frac{\mathrm{TaskSize}}{d_{\mathrm{Handling}}}. \quad (2)$$

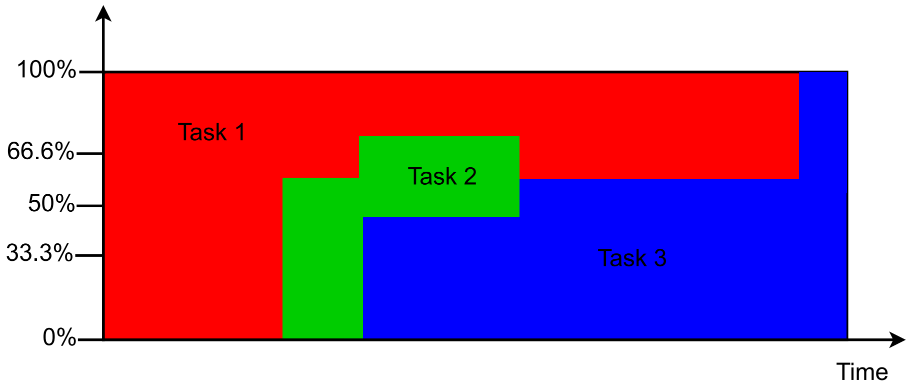


Fig. 5. The multi-tasking behaviour of the server side.

5.2. Simulation parameter setup

We use the below parameters for the performance analysis.

- n PU instances may run on the UE. Each PE generates tasks with an average size of 1,000,000 calculations at an average interval of ten seconds (negative exponential distribution for both).
- There is one PE for *each* PU on the UE side, with a capacity of only 200,000 calculations/s (i.e. n PU instances mean n UE PEs).
- Four PEs, each with a capacity of 1,000,000 calculations/s, are deployed as edge resources, having a one-way delay between UE and PE within 5 ms and 15 ms (uniform distribution).
- In total, ten PEs are deployed as PMC resources, each with a capacity of 1,000,000 calculations/s, with a one-way delay between UE and PE between 30 ms and 300 ms (uniform distribution), with the above delays.
- Minimum processing speed per PE is 200,000 calculations/s, i.e. PEs in EC and PMC accept up to five tasks in parallel, while the UE PE can run at most a single one. When fully loaded, further tasks get rejected, and a new PE has to be selected.
- For policies with load increment (i.e. PLU, PLUD, PLU-DPF and PLUD-DPF):

$$I_{p_{\mathrm{EC}}} = 10\%, I_{p_{\mathrm{PMC}}} = 20\%, I_{p_{\mathrm{UE}}} = 100\%.$$

- For DPF policies (i.e. PLU-DPF and PLUD-DPF): DPF $D_p = 0.0001$ for all PEs.
- One PR in the UE network, one in the EC, and one in the PMC.
- The simulation duration for each run is 120 min of simulation time.
- For each simulation scenario, 64 runs are performed.

5.3. Performance results

5.3.1. Resource utilisation

First we examine the general properties of the different pool policies in our edge setup. Fig. 6 presents the utilisation of different PEs on UE, in edge and PMC (horizontal direction) for the different pool policies (vertical direction). On the x-axis, the number of PUs is varied. It is worth noting that n PUs also means n low-performance PEs at the UE side (“the more UEs, the more resources on UEs in total”), while the number of PEs in EC (4) and PMC (10) remain fixed. There is a line for the average utilisation of each PE, which mostly overlaps (explained later). Thick error bars mark the 10%- and 90%-quantiles, together with grey ribbons, for better visibility. Thin error bars show the absolute minima and maxima. The corresponding average handling speed is shown in Fig. 7, again with 10%- and 90%-quantiles (thick error bars and ribbons) and absolute minima and maxima (thin error bars).

From the utilisation of RAND and RR (refer to Fig. 6), it can be observed that all three types of resources, such as UE, EC and PMC, are used. Furthermore, the utilisation of UE PEs is significantly larger than for EC and PMC. It is because a UE PE only performs one task at a time (due to lower UE performance), while EC and PMC PEs can run up to five tasks in parallel. It violates the first goal which states that UE resources should not be used unless there is no other choice. It is confirmed by a low handling speed (refer to Fig. 7). Nevertheless, for up to around 20 PUs, there is still a better performance ($\geq$200,000 calculations/s) than just running on the UE itself, in addition to reduced battery consumption on the UE. Least Used (LU) is not much better, similar to policies such as RAND and RR.

Comparing the utilisation results of LU to the LUD, PLU and PLUD policies (refer to Fig. 6), there is a significant difference: at low loads, only edge resources are used as intended. With the edge resource utilisation increasing, there is also an increase in PMC utilisation. However, EC is the preferred choice, with higher utilisation than PMC. There is no usage of slow, expensive, battery-powered UE resources, as long as sufficient cloud and edge resources are available. The handling speed confirms the usefulness (Fig. 7), with superior values over the whole x-axis range. The two variants of policies with degradation (LUD and PLUD) perform

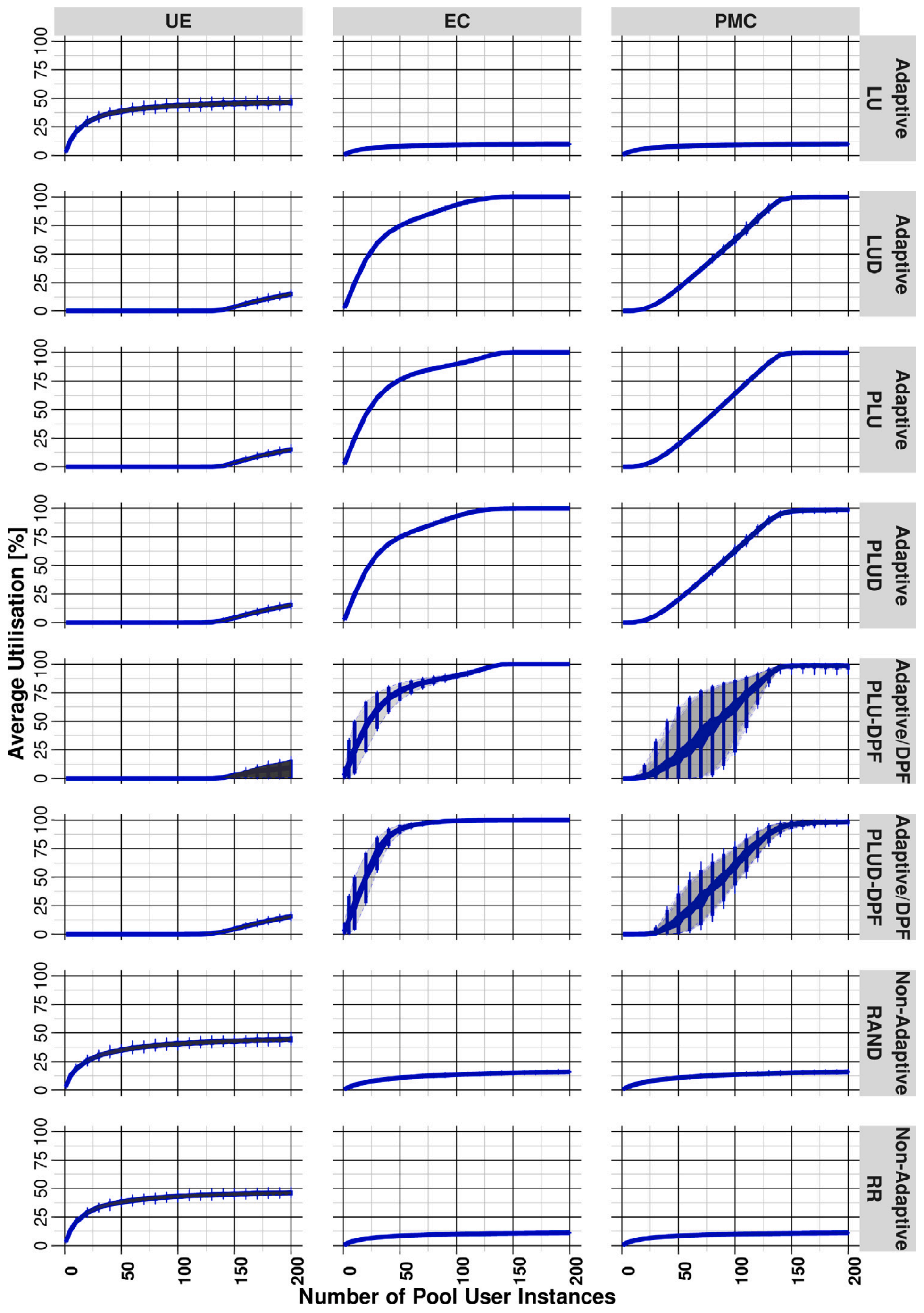


Fig. 6. Average utilisation of the three pool element types.

slightly better than standard PLU. The difference is subtle: multiple degradations for a PE p may occur before a load update resets the degradation variable X_p to zero, while plain PLU always adds *one* fixed load increment. As shown here, the compensation of handlespace load inaccuracy due to network delay performs slightly better. There is an advantage for PLUD compared to LUD at high loads (here: ≥130 PUs).

On the utilisation side, the degradation policies show an increased utilisation variation for higher loads: inaccuracy occurs due to network delay. There is a difference between the PEs having low and high PU↔PE RTT. It can easily be seen that LUD, PLU, and particularly PLUD fulfil all four goals.

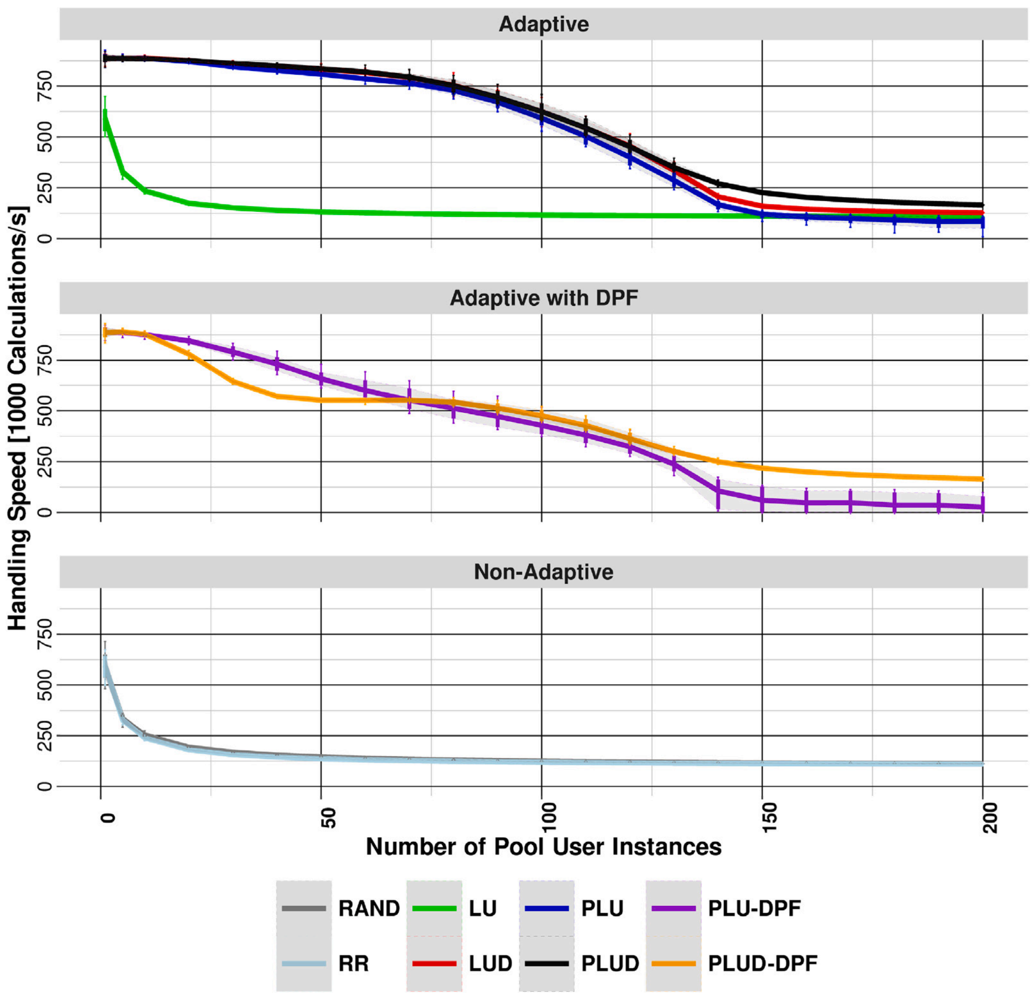


Fig. 7. Average task handling speed.

Our setup is based on our multi-country cloud/edge research test bed scenario. Thus, the RTT between PU and PMC PEs significantly varies, from 60 ms to 600 ms. With the DPF policy variants PLU-DPF and PLUD-DPF, the PE choice takes the RTT into account ($X_p = 0.0001$). In this case, the utilisation lines (Fig. 6) for the different PEs become distinct, leading to a significant variation of the 10%- and 90%-quantiles, as intended. Nearer PEs are preferred over far-away PEs concerning the RTT. However, the effect on the handling speed (Fig. 7) is not large, even leading to a lower performance than PLU, LUD and PLUD for up to 150 PUs (i.e. within the "normal" workload range). PLUD-DPF performs well comparable to PLUD at overload (≥150 PUs).

The performance of each task scheduling policy is presented in Fig. 8. It presents a bar plot for three load scenarios: 10 PUs (representing low load), 60 PUs (representing medium load), and 120 PUs (representing high load) for each policy. Each bar shows the three components of the average task handling time (see Eq. (1)): queuing, startup and processing time of the tasks completed during the simulation duration (120 min). For a detailed view, we also provide numerical values (mean, 10 % and 90 % quantiles) for queuing time and processing time in Tables 2 and 3.

The task handling time and speed is mainly influenced by queuing and processing. The startup time is short for all three scenarios. The performance is worst for the non-adaptive RAND and RR, including the adaptive LU policy. Mainly, the major contributor is queuing time. Tasks queue up and wait until the earlier task is finished. Thus, a significant amount of time is used for waiting. During this waiting time, a task does not make any computational progress. Finally, when a task gets processed, these policies may use an unsuitable PE (e.g. on the PU itself or a highly loaded one), leading to a longer processing time.

We found that LUD, PLU or PLUD leads to the lowest queuing times and processing times, with a slight advantage for PLUD, due to the optimised choice of PEs. Also, these policies perform well in the high load scenario (120 PUs). The DPF variants of the policies, i.e. PLU-DPF and PLUD-DPF, show a slightly longer processing time and result in a slightly increased queuing time. In some cases, the DPF policies do not make the best choice for a PE, leading to a longer duration for processing. It means choosing a nearby PE (in terms of RTT) is not beneficial in this setup. Nevertheless, the gap between the DPF and non-DPF policies (PLU/PLUD

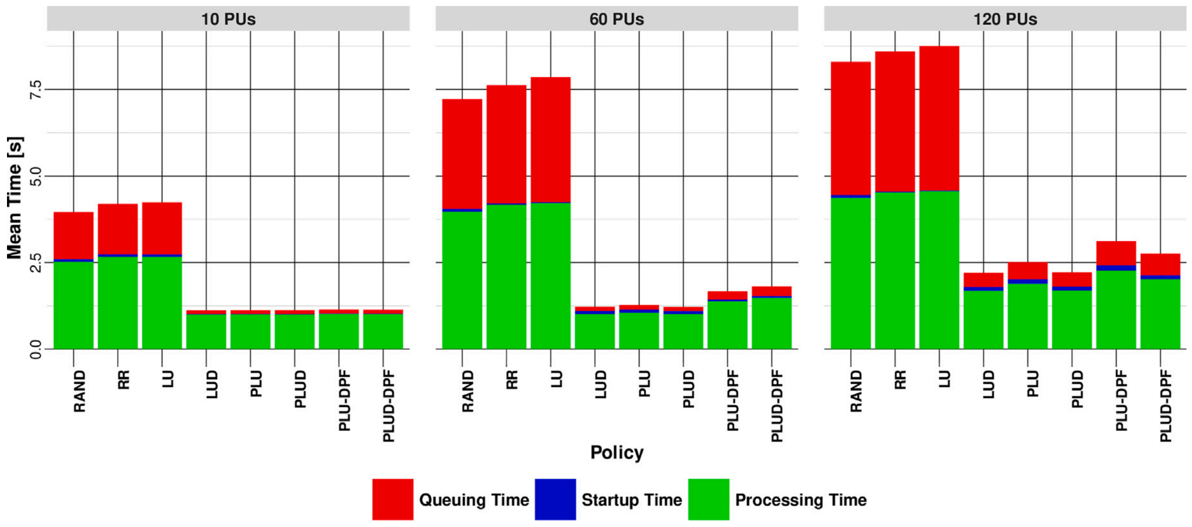


Fig. 8. Queuing, startup and processing times for low (10 PUs), medium (60 PUs) and high (120 PUs) workload.

Table 2
Queuing time for each policy.

PUs	PolicyType	Policy	Mean	$Q_{10\%}$	$Q_{90\%}$
10	Adaptive	LU	1.51	1.15	1.95
10	Adaptive	LUD	0.11	0.08	0.14
10	Adaptive	PLU	0.11	0.08	0.14
10	Adaptive	PLUD	0.11	0.09	0.14
10	Adaptive with DPF	PLUD-DPF	0.12	0.09	0.15
10	Adaptive with DPF	PLU-DPF	0.12	0.09	0.15
10	Non-Adaptive	RAND	1.36	0.99	1.78
10	Non-Adaptive	RR	1.46	1.11	1.82
60	Adaptive	LU	3.61	2.77	4.59
60	Adaptive	LUD	0.13	0.10	0.16
60	Adaptive	PLU	0.14	0.10	0.17
60	Adaptive	PLUD	0.13	0.10	0.16
60	Adaptive with DPF	PLUD-DPF	0.28	0.22	0.35
60	Adaptive with DPF	PLU-DPF	0.23	0.18	0.30
60	Non-Adaptive	RAND	3.18	2.46	3.98
60	Non-Adaptive	RR	3.42	2.66	4.22
120	Adaptive	LU	4.18	3.21	5.25
120	Adaptive	LUD	0.41	0.30	0.52
120	Adaptive	PLU	0.49	0.36	0.63
120	Adaptive	PLUD	0.41	0.31	0.53
120	Adaptive with DPF	PLUD-DPF	0.63	0.47	0.81
120	Adaptive with DPF	PLU-DPF	0.70	0.50	0.90
120	Non-Adaptive	RAND	3.85	2.97	4.81
120	Non-Adaptive	RR	4.05	3.12	5.06

vs. PLU-DPF/PLUD-DPF) is small. It leads us to the questions *at which load the policy performance is degrading?* and *are these policies performing reasonably even in case of overload scenarios?*

5.4. Performance in overload scenarios

Systems are designed to handle a certain workload. However, overload can occur, e.g. in case of unexpected events or denial of service attacks. To show the performance of the policies in such workload scenarios, and particularly to visualise how their performance degrades in such a scenario, we plot the mean queuing, startup and processing time of the tasks completed during the simulation duration (120 min) in Fig. 9.

Interestingly, the queuing and processing time for the non-adaptive policies RAND, RR, and adaptive LU handle each PE equally. They are non-optimal for our EC/PMC setup and converge to a result where each PE is fully loaded. Next, they mostly make a “trial

Table 3
Processing time for each policy.

PUs	PolicyType	Policy	Mean	$Q_{10\%}$	$Q_{90\%}$
10	Adaptive	LU	2.66	2.49	2.83
10	Adaptive	LUD	1.00	0.95	1.04
10	Adaptive	PLU	1.00	0.96	1.05
10	Adaptive	PLUD	1.00	0.95	1.05
10	Adaptive with DPF	PLUD-DPF	1.01	0.96	1.06
10	Adaptive with DPF	PLU-DPF	1.01	0.97	1.06
10	Non-Adaptive	RAND	2.51	2.33	2.71
10	Non-Adaptive	RR	2.66	2.49	2.84
60	Adaptive	LU	4.22	3.98	4.45
60	Adaptive	LUD	1.01	0.96	1.06
60	Adaptive	PLU	1.05	1.00	1.10
60	Adaptive	PLUD	1.01	0.96	1.06
60	Adaptive with DPF	PLUD-DPF	1.48	1.40	1.55
60	Adaptive with DPF	PLU-DPF	1.38	1.27	1.47
60	Non-Adaptive	RAND	3.97	3.75	4.19
60	Non-Adaptive	RR	4.17	3.95	4.39
120	Adaptive	LU	4.56	4.32	4.80
120	Adaptive	LUD	1.68	1.55	1.81
120	Adaptive	PLU	1.88	1.72	2.05
120	Adaptive	PLUD	1.69	1.56	1.82
120	Adaptive with DPF	PLUD-DPF	2.02	1.86	2.18
120	Adaptive with DPF	PLU-DPF	2.26	2.02	2.47
120	Non-Adaptive	RAND	4.37	4.14	4.59
120	Non-Adaptive	RR	4.52	4.29	4.75

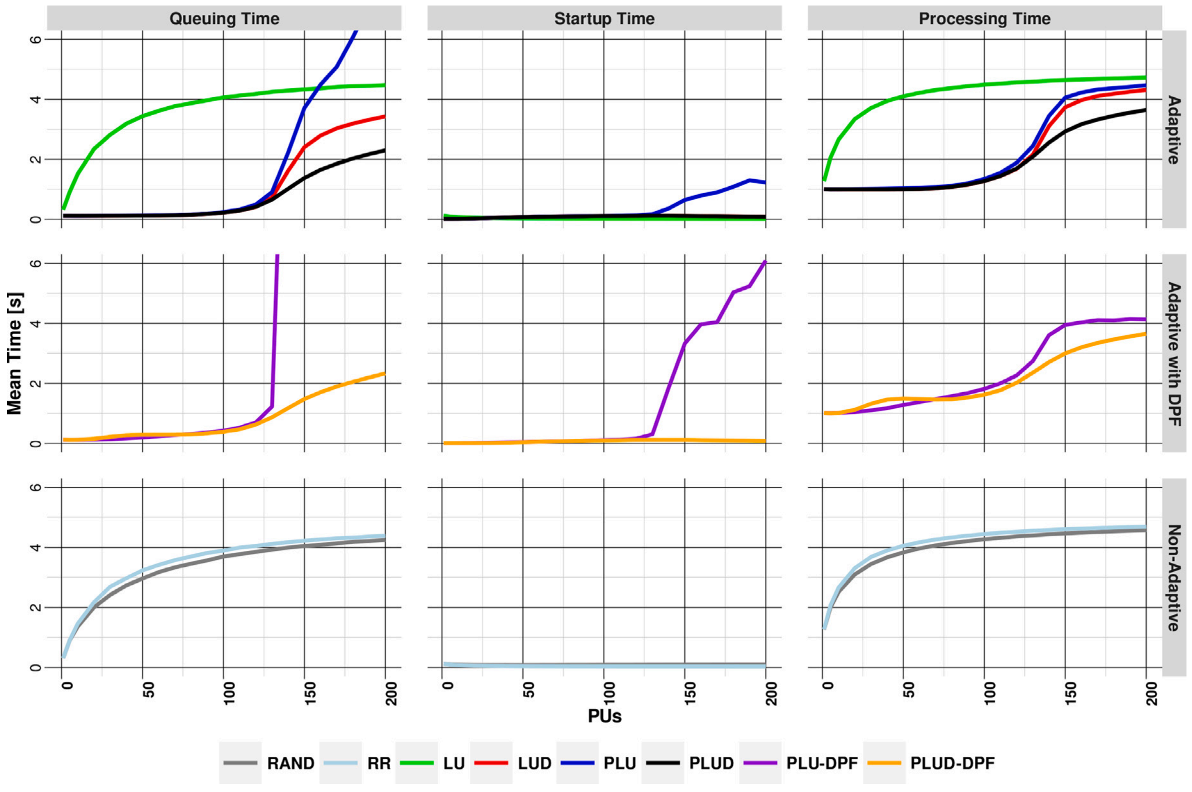


Fig. 9. Queuing, startup and processing times for varying workloads.

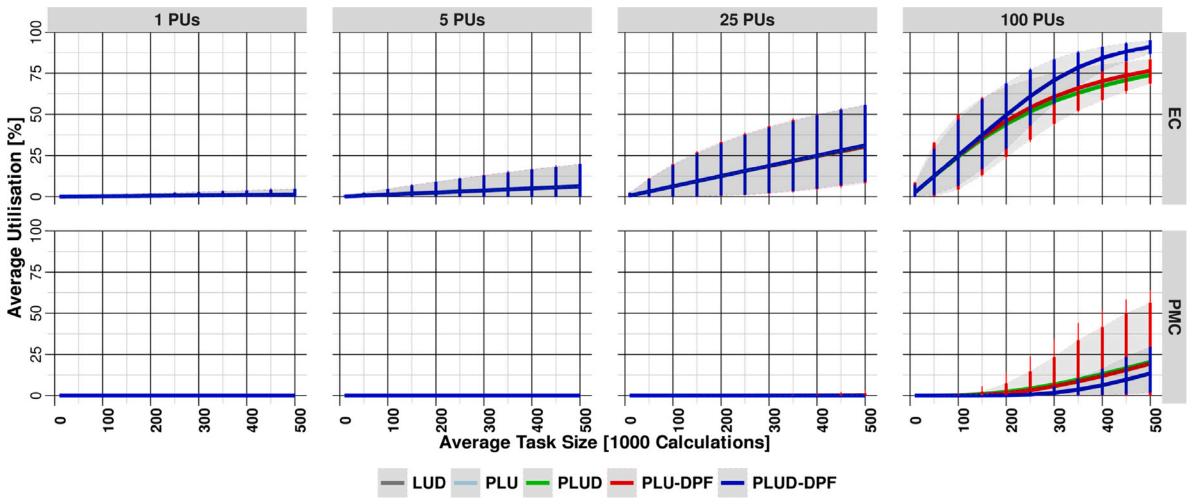


Fig. 10. Average utilisation for varying task sizes.

and error" for getting a task distributed to a PE. Note that for LU, when most PEs are at 100 % load, it performs round-robin selection[4] between these PEs. Only in rare cases, LU can make a slightly better selection when a task has just been finished. These policies mostly do not perform very well in the EC/PMC setup. On the other hand, their performance does not degrade unexpectedly in high overload.

It strongly contrasts PLU and PLU-DPF, where the performance degrades remarkably: for around 140 PUs, the queuing time goes "through the roof". The reason is that the PE load information in the handlespace becomes inaccurate. When there is a PE with a slightly lower load (due to a just-finished task), there will likely be multiple PUs to select this lower-loaded PE within a very short time. Since each EC/PMC PE can only handle up to five tasks simultaneously, further tasks get rejected. It leads to additional queuing, and further selection rounds, for all rejected tasks. Degradation used by LUD, PLUD and PLUD-DPF prevents this effect. Unless there is a load update from the PE, a PR performing a selection increases the degradation variable X_p (see Section 4.2). Then, consecutive selections of an already-selected PE become less likely, reducing the number of rejections. So, in high-load scenarios, having a policy with the degradation feature (like LUD, PLUD, PLUD-DPF) becomes crucial. Such findings can lead to the next question *whether the DPF policies are useful, particularly for "normal" workload?*.

5.5. *Performance with distance penalty factor*

To further examine the usefulness of the DPF policies, we performed simulations for LUD, PLU, PLUD, PLU-DPF and PLUD-DPF. Fig. 10 presents the utilisation results for EC and PMC PEs (horizontal direction; UE PEs omitted, as they are unused, i.e. utilisation at 0%), while Fig. 11 presents the corresponding handling speed results. We varied the number of PUs (vertical direction). On the x-axis, the average task size in calculations (in 1000 calculations) is varied. Note that we use small tasks here, from 10,000 to 500,000 calculations, which is small compared to a PE capacity of 1,000,000 calculations/s.

A benefit of using PLUD-DPF compared to LUD or PLU is visible with small tasks, where the network latency significantly affects the handling speed performance (Fig. 11). It can be the case for real-time cloud processing of interactive applications (e.g. handling audio/video data) on the UE. For task sizes of up to 500,000 calculations, PLUD-DPF achieves a better handling speed, even with 100 PUs. On the other hand, PLU-DPF (without degradation) performs worse than PLU, LUD and PLUD, even with only 25 PUs. Because of the short tasks, load information in the handlespace becomes inaccurate, e.g. a task is already finished when its increased load state gets propagated to a PR selecting a new PE. So, policies with degradation are essential for handling inaccurate load information here.

As expected from the handling speed results above, the utilisation of the EC PEs (Fig. 10) is highest for PLUD-DPF with many PUs. PLUD-DPF combines the information from DPF and degradation variable X_p to prefer EC PEs. In summary, PLUD-DPF can provide better performance than for instance PLUD in scenarios with short tasks. For longer-running tasks, LUD, PLU or particularly PLUD will be better and less complicated choices. However, the setup has to be configured carefully.

An IoT application may be a use case for a short task. IoT devices can collect and deliver data to a collector service (i.e. a pool of servers) hosted at edge or cloud. For instance, it can accept the data in JSON format and store it in a NoSQL/key–value database for real-time processing or further analysis. Minimising the RTT using DPF policies ensures the quick processing of such tasks and minimises the active network communication duration (i.e. energy consumption).

[4] RSPLIB and RSPSIM use round-robin between equally least-loaded PEs since this performs slightly better. The RSerPool standard alternatively allows random selection.

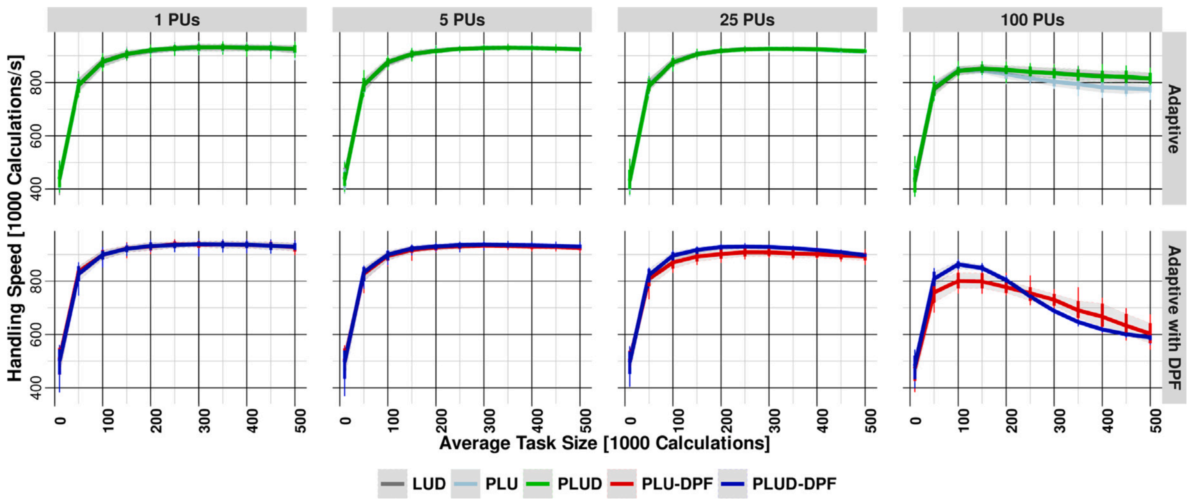


Fig. 11. Average task handling speed for varying task sizes.

6. Prototyping

We have built a prototype to demonstrate the applicability of RSerPool in a real cloud/edge setup.

6.1. Scenario setup

Based on RSPLIB, we have set up our EC/PMC prototype as depicted in Fig. 2: an OpenAirInterface-based Enhanced Packet Core (EPC) is the core of our 4G testbed network. OpenAirInterface[5] is an open-source implementation for 4G/5G EPC and eNodeB. It is the only specialised hardware necessary for the software-defined radio. UEs, like a laptop with a 4G modem, can connect via radio to the eNodeB. The components of the EPC run in virtual machines in an OpenStack[6] cluster, orchestrated by Open Source MANO.[7] The OpenStack cluster hosted in Oslo, Norway, also hosts virtual machines for two edge servers. External cloud servers on the Internet, running in the NorNet Core infrastructure [29], provide virtual machines as PMC servers, distributed at two locations in Norway (Gjøvik and Tromsø), one location in Germany (Essen), and another in China (Haikou). All of our machines are running Linux.

6.2. Application

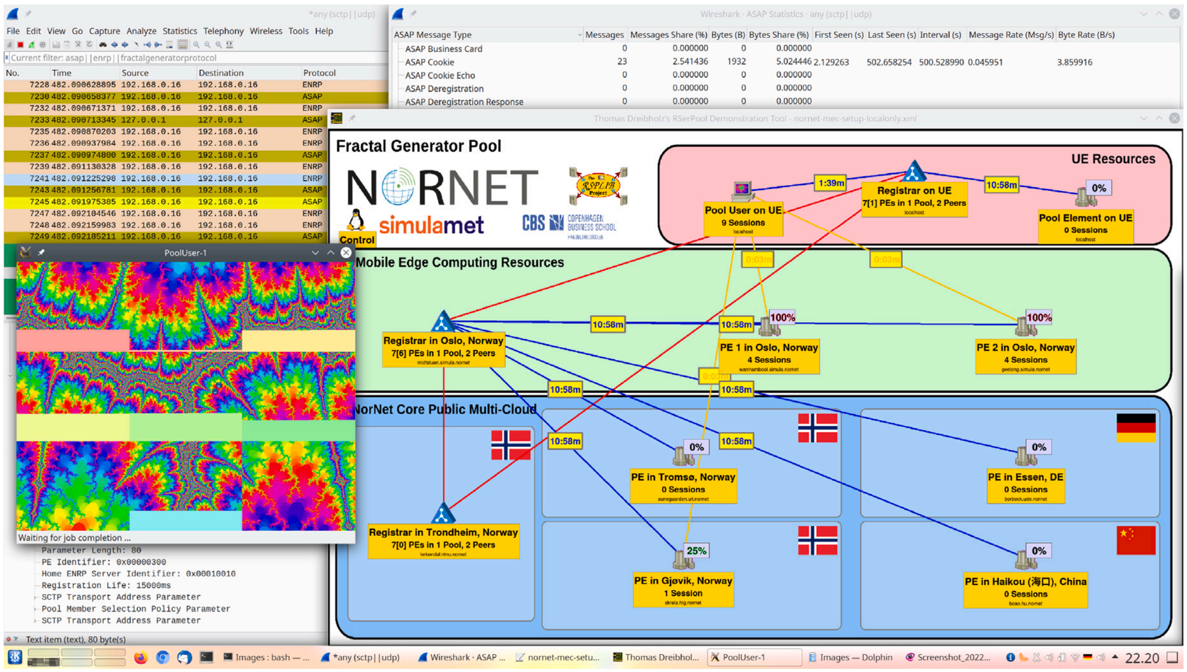
The application running is the FractalGeneratorProtocol (FGP) application of RSPLIB. FGP provides the computation of fractal graphics [30] based on Mandelbrot sets. It computes the set of numbers $c \in \mathbb{C}$ for which the iteration $z_{n+1} = z_n^k + c$, $z_0 = 0, k \in \mathbb{R}$ remains bounded within a configured number of iteration steps. In the FGP application, the coordinates of a pixel represent the real and imaginary parts of such a number c. It leads to a simple per-pixel computation, allowing to easily split the task of computing one image into the computation of multiple independent blocks. The PE service provides the computation of such blocks. The client side can run one or more PU instances for this service, with each session requesting another block of the image computation.

The image computation is requested by the client application running on the UE. The PEs in edge and PMC provide a processing capacity of four simultaneous sessions, while the PE on the UE can only handle one session. Recent cloud-based applications are mostly microservices, which can be instantiated dynamically [31], so the application is representative for such type of application.

6.3. Application interaction

Fig. 12 presents a screenshot of the UE screen running our prototype. The client (left-hand side) runs the PU instance(s). It can create an adjustable number of PU sessions (here: 9). Each session requests the computation of a part of a fractal graphics image (represents 3 × 3 blocks for 9 sessions) and continuously displays the progress of these computations. The RSerPool prototype tool GUI (right-hand side) presents the status of PU, PRs and PEs in EC, PMC and UE. Two PEs in the edge are handling four sessions each, leading to a load of 100 % in the EC. A PMC PE processes one session (here: on the PE in Gjøvik, Norway), while PLUD is the

[5] OpenAirInterface: https://www.openairinterface.org.
[6] OpenStack: https://www.openstack.org.
[7] Open Source MANO: https://osm.etsi.org.



```
Current filter: asap||enrp||fractalgeneratorprotocol
No.   Time            Source          Destination     Protocol
7228 482.090628895 192.168.0.16    192.168.0.16    ENRP
7230 482.090658377 192.168.0.16    192.168.0.16    ASAP
7232 482.090671371 192.168.0.16    192.168.0.16    ENRP
7233 482.090713345 127.0.0.1       127.0.0.1       ASAP
7235 482.090870203 192.168.0.16    192.168.0.16    ENRP
7236 482.090937984 192.168.0.16    192.168.0.16    ENRP
7237 482.090974800 192.168.0.16    192.168.0.16    ASAP
7239 482.091130328 192.168.0.16    192.168.0.16    ENRP
7241 482.091225298 192.168.0.16    192.168.0.16    ENRP
7243 482.091256781 192.168.0.16    192.168.0.16    ASAP
7245 482.091975385 192.168.0.16    192.168.0.16    ASAP
7247 482.092104546 192.168.0.16    192.168.0.16    ENRP
7248 482.092159983 192.168.0.16    192.168.0.16    ENRP
7249 482.092185211 192.168.0.16    192.168.0.16    ASAP
```

```
    Parameter Length: 80
    PE Identifier: 0x00000300
    Home ENRP Server Identifier: 0x00010010
    Registration Life: 15000ms
  › SCTP Transport Address Parameter
  › Pool Member Selection Policy Parameter
  › SCTP Transport Address Parameter
Text item (text), 80 byte(s)
```

ASAP Message Type	Messages	Messages Share (%)	Bytes (B)	Bytes Share (%)	First Seen (s)	Last Seen (s)	Interval (s)	Message Rate (Msg/s)	Byte Rate (B/s)
ASAP Business Card	0	0.000000	0	0.000000					
ASAP Cookie	23	2.541436	1932	5.024446	2.129263	502.658254	500.528990	0.045951	3.859916
ASAP Cookie Echo	0	0.000000	0	0.000000					
ASAP Deregistration	0	0.000000	0	0.000000					
ASAP Deregistration Response	0	0.000000	0	0.000000					

Fig. 12. Representation of the FGP application with the control GUI.

applied pool policy. Our prototype is interactive, and the RSerPool GUI also allows us to start/stop the local/remote components. The client parameters (such as the number of simultaneous PU sessions) can be adjusted interactively. In addition to the prototype, the network protocol analyser tool Wireshark[8] can be used to observe (main window on the left-hand side, in the background) and analyse (RSerPool analysis tool window on the right-hand side, in the background) the underlying RSerPool- and FGP-related network traffic. All software introduced in this article is open-source. As part of this work, we added the new policies PLUD, PLU-DPF and PLUD-DPF for both, RSPLIB and RSPSIM. In addition, we implemented the Wireshark dissector code to dissect and display policy information for PLUD, PLU-DPF and PLUD-DPF in the RSerPool traffic.

7. Discussion

Optimal task placement is an instance of the multi-objective optimisation problem. Resource orchestration techniques for cloud/edge platforms are essential for optimisations. Similar to developing the task offloading techniques mentioned in [2,5], evaluating its performance under realistic conditions is also important [32]. For cloud/edge platforms, energy cost, network bandwidth and latency are the primary functional requirements [26], while performance, reliability, availability, scalability, maintainability, security, and privacy are non-functional requirements [33]. Ashouri et al. in their survey, found that response time (or throughput) and resource utilisation are the most frequently used quality attributes, but such requirements lack established metrics. At the same time, simulation is the primary tool used for performance assessments [34]. To evaluate our proposed lightweight task scheduling framework, we primarily used RTT as a performance metric, which offers speed and reliability checks of network connections, and our simulation platform is hardware-independent.

From the simulations, we learned that the existing, lightweight RSerPool framework could be adapted with the right choice and configuration of pool policies. RSerPool framework can work reasonably well in a cloud/edge environment with a small management overhead. Implementing it in a real cloud/edge platform need lots of tuning, which cannot be anticipated now, but the framework is based on open-source software and developed following industry standards. It is worth noting that the workload scenarios are realistic but not accurate.

8. Conclusion and future work

A unified resource orchestration framework offers the possibility to offload tasks seamlessly into a cloud/edge platform. Public cloud platforms do not provide low latency, and another resource-level abstraction has been created to offer low latency, known as Edge Computing (EC). EC sits in proximity to its users. Managing workload and heterogeneous resource pools is a challenging

[8] Wireshark: https://www.wireshark.org.

task. Our proposed approach is to reuse the lightweight, simple RSerPool framework to schedule the tasks into the resource pool by satisfying four basic goals. It should be used by cloud service providers and can support multiple resource allocation policies. We added three new ones: PLUD, PLU-DPF and PLUD-DPF. We have presented our simulated results and showed that the three policies LUD, PLU and PLUD are better for longer-running tasks, while PLUD-DPF achieves a better handling speed for short tasks. As future work, we plan to extend the benchmarking process by setting up a larger, geographically distributed scenario including OpenAirInterface-based EPC. It includes an application-specific comparison of different scheduling approaches to show the performance penalty of applying simple compared to more complicated ones. We also intend to contribute our results to the IETF standardisation process of RSerPool, and the development of orchestration frameworks, particularly Open Source MANO.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] I. Pelle, J. Czentye, J. Dóka, B. Sonkoly, Towards latency sensitive cloud native applications: A performance study on AWS, in: IEEE 12th International Conference on Cloud Computing, CLOUD, IEEE, 2019, pp. 272–280.
[2] H. Lin, S. Zeadally, Z. Chen, H. Labiod, L. Wang, A survey on computation offloading modeling for edge computing, J. Netw. Comput. Appl. 169 (2020) 102781.
[3] P. Mach, Z. Becvar, Mobile edge computing: A survey on architecture and computation offloading, IEEE Commun. Surv. Tutor. 19 (3) (2017) 1628–1656.
[4] M.S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, F. Hussain, Machine learning at the network edge: A survey, ACM Comput. Surv. (CSUR) 54 (8) (2021) 1–37.
[5] A. Shakarami, M. Ghobaei-Arani, A. Shahidinejad, A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective, Comput. Netw. 182 (2020) 107496.
[6] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, P. Mohapatra, Edge cloud offloading algorithms: Issues, methods, and perspectives, ACM Comput. Surv. (CSUR) 52 (1) (2019) 1–23.
[7] T. Dreibholz, S. Mazumdar, Load distribution for mobile edge computing with reliable server pooling, in: Proceedings of the 4th International Workshop on Recent Advances for Multi-Clouds and Mobile Edge Computing (M2EC) in Conjunction with the 36th International Conference on Advanced Information Networking and Applications, AINA, Sydney, New South Wales/Australia, 2022.
[8] P. Lei, L. Ong, M. Tüxen, T. Dreibholz, An Overview of Reliable Server Pooling Protocols, Informational RFC, 5351, IETF, 2008.
[9] T. Dreibholz, Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture (Ph.D. thesis), University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, 2007.
[10] Q. Xie, R.R. Stewart, M. Stillman, M. Tüxen, A.J. Silverton, Endpoint Handlespace Redundancy Protocol, RFC, 5353, ENRP, IETF, 2008.
[11] R.R. Stewart, Q. Xie, M. Stillman, M. Tüxen, Aggregate Server Access Protcol, RFC, 5352, ASAP, IETF, 2008.
[12] T. Dreibholz, E.P. Rathgeb, Overview and evaluation of the server redundancy and session failover mechanisms in the reliable server pooling framework, Int. J. Adv. Internet Technol. (IJAIT) 2 (1) (2009).
[13] R.R. Stewart, Stream Control Transmission Protocol, RFC 4960, IETF, 2007.
[14] S.M. Alamouti, F. Arjomandi, M. Burger, Hybrid edge cloud: A pragmatic approach for decentralized cloud computing, IEEE Commun. Mag. 60 (9) (2022) 16–29.
[15] M. Sebrechts, B. Volckaert, F.D. Turck, K. Yang, M. Al-Naday, Fog native architecture: Intent-based workflows to take cloud native toward the edge, IEEE Commun. Mag. 60 (8) (2022) 44–50.
[16] V. Nikolopoulos, M. Nikolaidou, M. Voreakou, D. Anagnostopoulos, Fog node self-control middleware: Enhancing context awareness towards autonomous decision making in Fog colonies, Internet Things 19 (2022) 100549.
[17] A. Kishor, C. Chakarbarty, Task offloading in Fog computing for using smart ant colony optimization, Wirel. Pers. Commun. (2021) 1–22.
[18] M. Adhikari, H. Gianey, Energy efficient offloading strategy in Fog-cloud environment for IoT applications, Internet Things 6 (2019) 100053.
[19] J. Mass, C. Chang, S.N. Srirama, Edge process management: A case study on adaptive task scheduling in mobile IoT, Internet Things 6 (2019) 100051.
[20] M. Adhikari, M. Mukherjee, S.N. Srirama, DPTO: A deadline and priority-aware task offloading in Fog computing framework leveraging multilevel feedback queueing, IEEE Internet Things J. 7 (7) (2019) 5773–5782.
[21] M. Chen, Y. Hao, Task offloading for mobile edge computing in software-defined ultra-dense network, IEEE J. Sel. Areas Commun. 36 (3) (2018) 587–597.
[22] M.-H. Chen, B. Liang, M. Dong, Multi-user multi-task offloading and resource allocation in mobile cloud systems, IEEE Trans. Wirel. Commun. 17 (10) (2018) 6790–6805.
[23] G. Zhang, F. Shen, Y. Yang, H. Qian, W. Yao, Fair task offloading among Fog nodes in Fog computing networks, in: IEEE International Conference on Communications, ICC, IEEE, 2018, pp. 1–6.
[24] Y.-L. Jiang, Y.-S. Chen, S.-W. Yang, C.-H. Wu, Energy-efficient task offloading for time-sensitive applications in fog computing, IEEE Syst. J. 13 (3) (2018) 2930–2941.
[25] S. Rashidi, S. Sharifian, Cloudlet dynamic server selection policy for mobile task off-loading in mobile cloud computing using soft computing techniques, J. Supercomput. 73 (9) (2017) 3796–3820.
[26] R. Eyckerman, S. Mercelis, J. Marquez-Barja, P. Hellinckx, Requirements for distributed task placement in the Fog, Internet Things 12 (2020) 100237.
[27] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News 33 (2) (2002) 51–59.
[28] X. Zhou, T. Dreibholz, E.P. Rathgeb, A new server selection strategy for reliable server pooling in widely distributed environments, in: Proceedings of the 2nd IEEE International Conference on Digital Society, ICDS, Sainte Luce/Martinique, 2008.
[29] E.G. Gran, T. Dreibholz, A. Kvalbein, NorNet core – a multi-homed research testbed, Comput. Netw. Special Issue Future Internet Testbeds 61 (2014).
[30] H.-O. Peitgen, P. Richter, the Beauty of Fractals, XII ed., Springer-Verlag, Heidelberg, Baden-Württemberg/Germany, 1986.

[31] L. Chen, Microservices: Architecting for continuous delivery and DevOps, in: IEEE International Conference on Software Architecture, ICSA, IEEE, 2018, pp. 39–397.
[32] M.S. Aslanpour, S.S. Gill, A.N. Toosi, Performance evaluation metrics for cloud, Fog and edge computing: A review, taxonomy, benchmarks and standards for future research, Internet Things 12 (2020) 100273.
[33] S. Gupta, Non-functional requirements elicitation for edge computing, Internet Things 18 (2022) 100503.
[34] M. Ashouri, P. Davidsson, R. Spalazzese, Quality attributes in edge computing for the Internet of Things: A systematic mapping study, Internet Things 13 (2021) 100346.