Load Distribution for Mobile Edge Computing with Reliable Server Pooling*

Thomas Dreibholz[a], Somnath Mazumdar[b]

Abstract The energy-efficient computing model is a popular choice for both, high-performance and throughput-oriented computing ecosystems. Mobile (computing) devices are becoming increasingly ubiquitous to our computing domain, but with limited resources (true both for computation as well as for energy). Hence, workload offloading from resource-constrained mobile devices to the edge and maybe later to the cloud become necessary as well as useful. Thanks to the persistent technical breakthroughs in global wireless standards (or in mobile networks), together with the almost *limitless* amount of resources in public cloud platforms, workload offloading is possible and cheaper. In such scenarios, Mobile Edge Computing (MEC) resources could be provisioned in proximity to the users for supporting latency-sensitive applications. Here, two relevant problems could be: *i*) How to distribute workload to the resource pools of MEC as well as public (multi-)clouds? *ii*) How to manage such resource pools effectively? To answer these problems in this paper, we examine the performance of our proposed approach using the Reliable Server Pooling (RSerPool) framework in more detail. We also have outlined the resource pool management policies to effectively use RSerPool for workload offloading from mobile devices into the cloud/MEC ecosystem.

[a]Simula Metropolitan Centre for Digital Engineering
c/o OsloMet – storbyuniversitetet
Pilestredet 52, 0167 Oslo, Norway
dreibh@simula.no
https://orcid.org/0000-0002-8759-5603
[b]Department of Digitalization, Copenhagen Business School
Howitzvej 60, 2000 Frederiksberg, Denmark
sma.digi@cbs.dk
https://orcid.org/0000-0002-1751-2569

* This work has partly been supported by the 5G-VINNI project (grant no. 815279 within the H2020-ICT-17-2017 research and innovation program) and also partly by the Research Council of Norway (under project number 208798/F50).
The authors would like to thank Ann Edith Wulff Armitstead for her comments.

1 Introduction

Mobile devices are becoming an indispensable part of our daily life. Such devices are used for a large list of Internet-based activities (such as financial transactions, online conferencing, route navigation, Internet browsing, text/video messaging, and online gaming). These modern devices (including smart phones) are denoted as User Equipment (UE). Recent UEs have decent computational as well as storage capabilities. In some scenarios, they are limited by the steadily increasing amount of user data and computation demand. In such events, cloud computing become a viable solution to offload tasks that require large computation power and storage. Apart from that, the pay-as-you-go pricing model also offers cloud services at cheaper costs than overall on-premise clusters management. However, cloud platforms do not offer low latency between UEs, which may impact latency-sensitive applications performance. To counter such performance-related issues, Mobile Edge Computing (MEC) is becoming popular, thanks to the progress in the mobile network ecosystem. MEC can be placed between the cloud and the user, while resourcefully supported by the cloud.

MEC could be seen as a small subset of the cloud. Managing such geographically distributed resource pools without adding a large amount of management overhead is not easy. To date, MEC is not mature enough, thus offloading applications to MEC efficiently is not a trivial matter. In this work, we did *not* aim for a task offload solution where we add multiple abstraction layers and try to find the *almost* perfect mapping of tasks to resources. Instead, our proposed solution was inspired by the "Keep It Simple, Stupid" (KISS) approach, where we reuse the existing, lightweight Reliable Server Pooling (RSerPool) framework [1, 2]. In addition to that, we have adapted server selection policies by incorporating the structure of the MEC.

In our previous work [3], we already presented how RSerPool manages resource pools and also how it handles the application sessions effectively. Now, as an extension of the work, in this paper, we examine the use of different server selection policies in more detail (over a large parameter range). We conducted the performance evaluation of resource selection policies, using simulations of a MEC and (multi-)cloud setup with the RSerPool Simulation (RSPSIM) [2] model. We provided insights into how the proper choice and configuration of pool member selection policies could result in good performance. We also believe such a setup realistically resembles the serverless computing setups.

The rest of the paper is structured as follows: we first introduce RSerPool in Section 2. Then, we describe our approach in Section 3, followed by the description of our simulation setup and result in Section 4. Finally, we conclude our paper in Section 5.

2 Reliable Server Pooling (RSerPool)

The management of server pools, as well as sessions (between clients and server pools) is a traditional problem in computer networking. To avoid "reinventing the wheel" for each application, the Internet Engineering Task Force (IETF) founded the Reliable Server Pooling (RSerPool) [1] working group to develop a generic standard. Therefore, RSerPool is an application-independent and open-source framework, with the goals of being simple and lightweight. RSerPool is also suitable for devices with very limited resources, which is ideal for MEC environments. RSPLIB[2] [2, Chapter 5] is the most widespread open-source implementation of RSerPool. Apart from that, a simulation model RSPSIM[3] [2, Chapter 6] has also been developed.

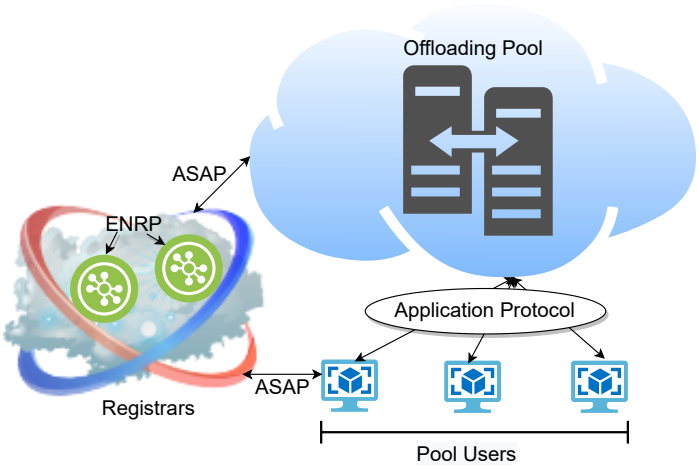

Fig. 1: Illustration of the Reliable Server Pooling (RSerPool) Architecture

Figure 1 illustrates the RSerPool architecture. 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). To avoid a single point of failure, RSerPool setups should consist of at least two registrars. The PRs synchronise the handlespace by using the Endpoint haNdlespace Redundancy Protocol (ENRP) [2, Section 3.10]. An operation scope is limited to an organisation or company. This means that it does not scale to the whole Internet, in contrast to the Domain Name System (DNS). This is a significant simplification, which keeps RSerPool very lightweight concerning the administrative overheads. Pools can be distributed over large geographic areas to achieve a high resilience of services.

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

PEs use the Aggregate Server Access Protocol (ASAP) [2, Section 3.9] to dynamically register to, or deregister 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 [4]. Clients, which are 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 [4], which is usually just denoted as pool policy (see more details in Subsection 3.2). 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 to handle failovers and help with state synchronisation [4].

3 Our Proposed Offloading Approach

In this section, we are explaining our offloading approach in two parts. The first part is primarily focused on setting up the RSerPool (refer to Subsection 3.1). Next, we will discuss the server pool member selection policies for our MEC setup (refer to Subsection 3.2).

3.1 Setting up Basic RSerPool

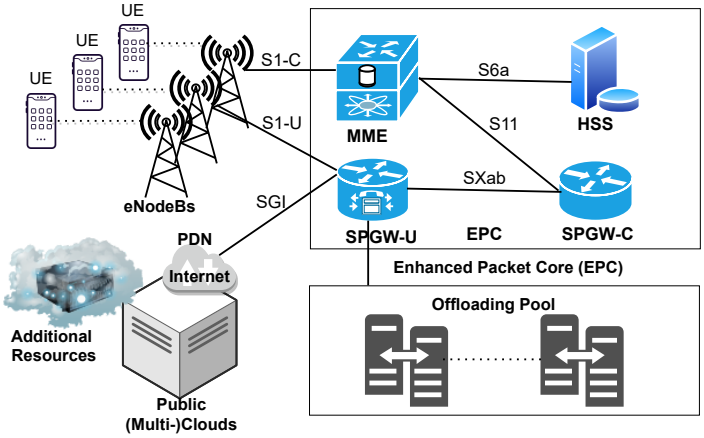

Fig. 2: Application Scenario with MEC and Public (Multi)-Clouds (PMCs)

Figure 2 shows the basic MEC scenario [3,5]. Here, UEs are running applications, which aim at offloading tasks to the cloud instead of using the scarce and battery-powered UE resources. Particularly, this could be a serverless computing setup, where the application pushes a certain task for processing into the cloud. The cloud can instantiate an application-specific container that runs the assigned task and frees up the container after execution. That

is, the compute node is stateless, which allows for high flexibility of workload offloading for different application types. Cloud resources are provided by the local MEC, as well as by cheaper public (multi)-clouds (PMC). Particularly, it is from a cost-perspective an advantage to use PMCs, which are in vicinity to the user. Distant cloud resources offer a high round trip time (RTT), as distance adds network latency to the service request handling. Therefore, the local MEC resource is (if available) preferred. So, now the questions could arise: *how to manage such server pools, consisting of MEC as well as PMC resources?*

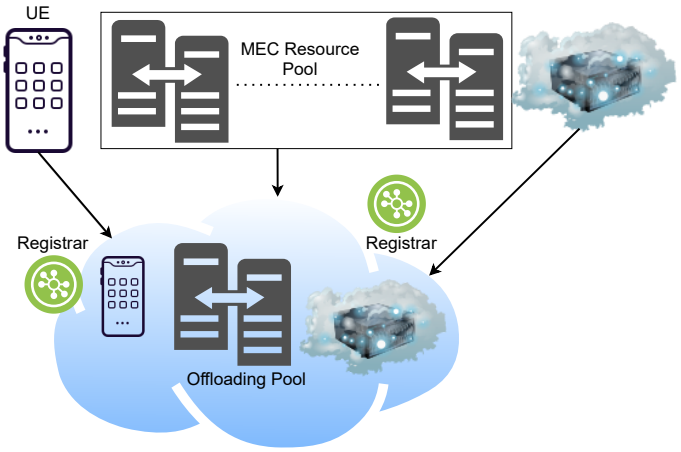

Fig. 3: RSerPool with Mobile Edge Computing and (Multi-)Cloud Resources

In Figure 3, we describe our approach [3, 5] to applying RSerPool. Resources are added into a pool, identified by its PH (here: "Offloading Pool"), which means the pool consists of PEs in the MEC as well as in a PMC setup. Since the original purpose of RSerPool is high availability, it may be straightforward to also run a PE instance on the UE itself. RSerPool is also lightweight, which means it has a very small management overhead. With RSerPool, it is also possible to add the UE resources to the pool. If everything fails, then the application on the UE is still able to "offload" a task to itself (i.e. the PE on the UE). This would be the case when there is no network coverage, which is not so unlikely for a mobile usage scenario. In such events, the application may still provide a useful service for its user with reduced performance and also with increased energy consumption. However, MEC or PMC resources are always preferable in these scenarios.

Overall, the pool consists of three different types of resources: MEC, PMCs, as well as UE resources. In addition, PRs are needed for managing the handlespace. To avoid a single point of failure, it needs to be at least two, e.g. one for MEC and another for PMCs. For standalone operations in case of loss of network coverage, also a PR instance needs to run on the UE itself. This is possible, since PRs are lightweight, meaning they have only low memory and CPU requirements. Now, in this case, the relevant question is: *how can the PRs finally handle the different resource types in the pool?*

3.2 Pool Member Selection Policies for MEC

The selection of a suitable PE is performed by the pool policy (as described in Subsection 3.3). In case of our UE/MEC/PMC setup, it has to achieve the following four goals [3]:

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

3.3 Policies

Two simple resource allocation algorithms, Random (RAND) and Round Robin (RR), neither have information about load state nor about resource type [2, Subsection 3.11.2]. They are used in this work for comparison purposes only. Apart from these two, we also have used different resource allocation policies with load states:

- Least Used (LU) [2, Subsection 3.11.3] selects the PE p where its load L_p is lowest. 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. Therefore, it will not differentiate between MEC, PMC and UE resources. In other words, it is not able to satisfy the first three goals mentioned above.
- Priority Least Used (PLU) [2, Subsection 8.12.2] adds a PE-specific load increment constant I_p to LU. This means, PEs are chosen based on the lowest sum $L_p + I_p$. Setting $I_{p_{\mathrm{MEC}}} < I_{p_{\mathrm{PMC}}}$ for all MEC PEs p_{MEC} and PMCs PEs p_{PMC}, as well as $I_{p_{\mathrm{UE}}}$=100% for the UE PE. Then, PLU should achieve our four goals.
- Least Used with Distance Penalty Factor (LU-DPF) [2, Subsection 8.10.2] adds a PE-specific Distance Penalty Factor (DPF) constant D_p to LU. Then, the PEs are chosen based on the lowest sum

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

  where RTT_p is the approximated[4] Round-Trip Time (RTT) to the PE. For simplicity, D_p can be the same for all PEs, then making it a pool-specific constant. Assuming

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

  for all MEC PEs p_{MEC} and PMCs PEs p_{PMC}, LU-DPF should achieve the goals 2 to 4. However, goal 1 is likely to be violated in this case, since

[4] On PR-H: $\mathrm{RTT}_{\mathrm{PR-H}\leftrightarrow\mathrm{PE}}$; on other PR: $\mathrm{RTT}_{\mathrm{PR}\leftrightarrow\mathrm{PR-H}} + \mathrm{RTT}_{\mathrm{PR-H}\leftrightarrow\mathrm{PE}}$.

$\mathrm{RTT}_{p_{\mathrm{UE}}}$ may (mostly) be minimal. LU-DPF can therefore be extended 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).

- LU and its variants are adaptive policies, i.e. they require up-to-date load information from the PEs in the handlespace, which is managed by the PRs. 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) [6] policy adds a load degradation variable X_p for each PE p. Each time a PE is selected by a PR, 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 $L_p + X_p$. This can be combined with the idea of PLU to a new policy Priority Least Used with Degradation (PLUD) choosing a PE by lowest sum of $L_p + I_p + X_p$, and with distance penalty factor to a new policy Priority Least Used with Degradation and DPF (PLUD-DPF) selecting a PE by lowest sum of

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

Now, one question arising from these different policies is about their performances: *Which policies are useful for our use case, and which policy should be used for a UE/MEC/PMC setup?*

4 Simulation and Results

For our simulation, we use the RSPSIM [2, Chapter 6] model for RSerPool to create a setup as depicted in Figure 3. RSPSIM is based on OMNeT++ 6.0pre15 and has been extended to support our new polices (PLUD, PLU-DPF and PLUD-DPF, see Subsection 3.3).

4.1 Application Description

As application, we use the CalcAppProtocol model from [2, Section 8.3]. This model is part of RSPSIM, as well as of the RSerPool implementation RSPLIB, which was used for our initial proof-of-concept real-world measurements in [3]. In CalcAppProtocol, a PE has a request 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, disk space usage). Each request has a request size, which is the number of calculations consumed by the processing of the request. Following the multi-tasking principle, a PE can process multiple requests simultaneously. The user-side performance metric is the handling speed. The total time for handling a request d_{Handling} is defined as the sum of queuing time, start-up time (de-queuing until reception of

acceptance acknowledgement) and processing time (acceptance until finish). The *handling speed* (in calculations/s) is defined as:

$$\text{HandlingSpeed} = \frac{\text{RequestSize}}{d_{\text{Handling}}}.$$

4.2 Simulation Setup

For our setup as depicted in Figure 3, we use below parameters, unless otherwise stated. For each simulation scenario, 64 runs are performed.

- n PU instances may run on the UE. Each PE generates requests with an average size of 1,000,000 calculations, at an average interval of 10 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 MEC resources, having a one-way delay between UE and PE within 5 ms and 15 ms (uniform distribution). From testbed experiments [3, 5], these delays are realistic in local 4G setups.
- In total, ten PEs are deployed as PMC resources, each with a capacity of 1,000,000 calculations/s, with one-way delay between UE and PE between 30 ms and 300 ms (uniform distribution), with delays based on Internet measurements in [7].
- Minimum processing speed per PE is 200,000 calculations/s, i.e. PEs in MEC and PMC accept up to five requests in parallel, while the UE PE just can run at most a single one. When fully loaded, further requests get rejected, and a new PE has to be selected.
- For policies with load increment:

$$I_{p_{\text{MEC}}} = 10\%, I_{p_{\text{PMC}}} = 20\%, I_{p_{\text{UE}}} = 100\%.$$

- For DPF policies: DPF D_p=0.0001 for all PEs.
- One PR in the UE network, one in the MEC cloud, and one in the PMC.

4.3 Pool Member Selection Policy Performance

Our first simulation is examining the general properties of the different pool policies in our MEC setup (defined in Subsection 4.2). Figure 4 presents the utilisation of different PEs on UE in MEC as well as in PMC (vertical direction) for the different pool policies (horizontal direction). On the x-axis, the number of PUs is varied. Note, 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 MEC (4) and PMC (10) remain fixed.

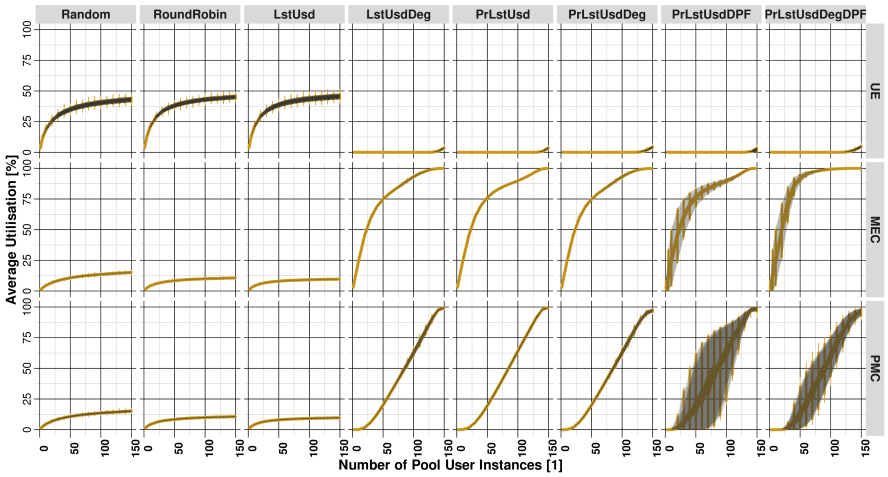

Fig. 4: Average Utilisation of the three Pool Element types

There is a line for the average utilisation of each PE, which mostly overlap (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 (as defined in Subsection 4.2) is shown in Figure 5, again with 10%- and 90%-quantiles (thick error bars and ribbons) as well as absolute minima and maxima (thin error bars).

From the utilisation of Random (RAND) and Round Robin (RR) (refer to Figure 4), it can be observed that all three types of resources, such as UE, MEC and PMC, are used. Furthermore, the utilisation of UE PEs is significantly larger than for MEC and PMC. This is due to the fact that a UE PE only performs one request at a time (due to lower UE performance), while MEC and PMC PEs can run up to five requests in parallel. This clearly violates goal 1 of Subsection 3.2, which states that UE resources should not be used unless there is no other choice. This is confirmed by a low handling speed (refer to Figure 5). Nevertheless, up to around 20 PUs, there is still a better performance ($\geq$200,000 calculations/s) than running on the UE itself, in addition to reduced battery consumption on the UE. Least Used (LU) is not much better, as expected (see Subsection 3.2), so these three policies (RAND, RR and LU) are not a good choice.

Comparing the utilisation results of LU to the Least Used with Degradation (LUD), Priority Least Used (PLU) and Priority Least Used with Degradation (PLUD) policies (refer to Figure 4), there is a significant difference: at low loads, only MEC resources are used (as intended). With the MEC resources utilisation increasing, there is an increase in PMC utilisation as well. However, MEC is the preferred choice, with higher utilisation than for PMC. Apart from that, there is no usage of slow, expensive, battery-powered UE resources, as long as sufficient MEC/PMC resources are available. The useful-

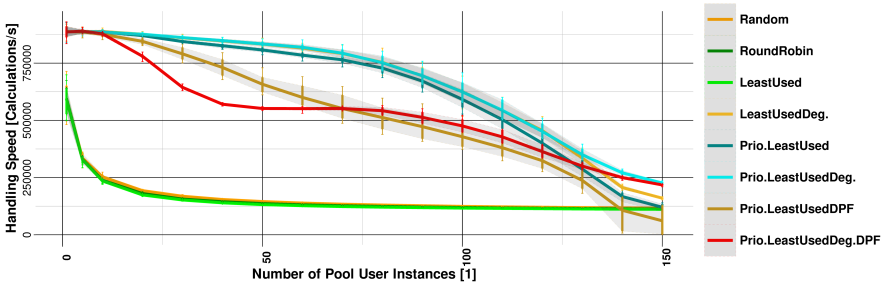

Fig. 5: Average Request Handling Speed

ness is confirmed by the handling speed (Figure 5), with superior values over the whole x-axis range. The two variants of policies with degradation (LUD and PLUD) perform slightly better than plain PLU. Note, that 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. And, there is an advantage for PLUD compared to LUD at high loads (here: $\geq$130 PUs). On the utilisation side, the degradation policies show an increased utilisation variation for higher loads: inaccuracy occurs due to network delay, so there is a difference between the PEs having low and high PU$\leftrightarrow$PE RTT. Nevertheless, it can easily be seen that LUD, PLU, and particularly PLUD fulfil all four goals (for goals refer to Subsection 3.2).

Our setup defined in Subsection 4.2 is based on our multi-country Internet testbed scenario already mentioned in our previous work [3]. Therefore, particularly the RTT between PU and PMC PEs significantly varies, from ca. 60 ms to ca. 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 (Figure 4) 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 with respect to RTT. However, in this scenario, the effect on the handling speed (Figure 5) is not large, even leading to a lower performance than PLU, LUD and PLUD for a larger number of PUs. Such findings could lead to the question *whether the DPF policies are useful at all for a MEC/PMC scenario?*

4.4 Performance with Distance Penalty Factor

We performed simulations for LUD, PLU, PLUD, PLU-DPF and PLUD-DPF to further examine the usefulness of the DPF policies. Figure 6 presents the utilisation results for MEC and PMC PEs (vertical direction; UE PEs omitted, as they are unused, i.e. utilisation at 0%), while Figure 7 presents the corresponding handling speed results. We varied the number of PUs (hor-

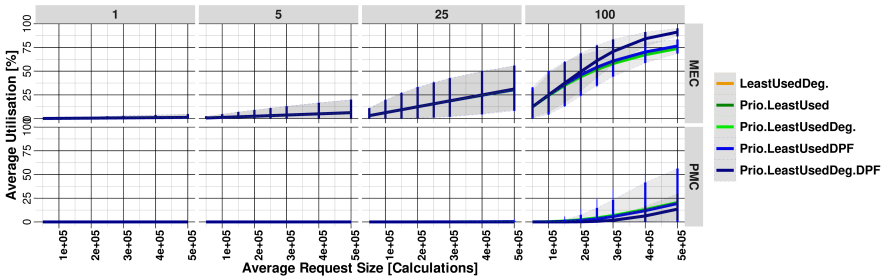

Fig. 6: Average Utilisation for varying Request Sizes

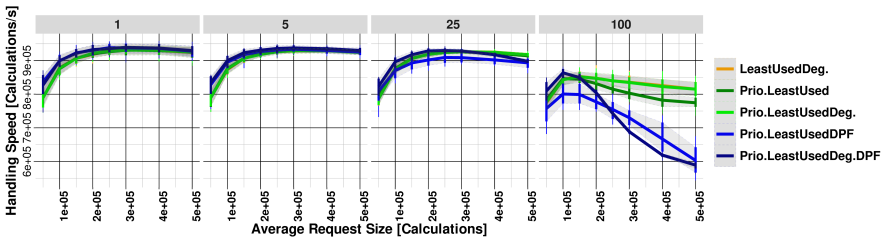

Fig. 7: Average Request Handling Speed for varying Request Sizes

izontal direction). On the x-axis, the average request size in calculations is varied. Note, that we use particularly small requests here, from 50,000 (5e4) to 500,000 (5e5) calculations, which is small compared to a PE capacity of 1,000,000 (1e6) calculations/s.

A benefit of using PLUD-DPF in comparison to LUD or PLU is visible with small requests, where the network latency significantly affects the handling speed performance (Figure 7). This could be the case for real-time cloud processing of interactive applications (e.g. handling audio/video data) on the UE. For request sizes of up to 150,000 (1e5) 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 requests, load information in the handlespace becomes inaccurate, e.g. a request 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 MEC PEs (Figure 6) is highest for PLUD-DPF with a large number of PUs. That is, PLUD-DPF combines the information from DPF and degradation variable X_p to prefer MEC PEs. In summary, PLUD-DPF can provide better performance than for instance PLUD in scenarios with short requests. However, the setup has to be configured carefully. For longer-running requests, LUD, PLU or particularly PLUD will be better and less complicated choices.

5 Conclusion and Future Work

Mobile devices are resource-constrained. Serverless computing offers the possibility to offload tasks into a public cloud platform. But cloud computing does not offer low latency. To counter the latency issue, another resource level abstraction has been created known as Mobile Edge Computing (MEC) that sits in proximity to its users. However, load distribution and the management of computing resource pools with different resource types is a challenging task. Our proposed approach is to reuse the lightweight, simple, yet powerful Reliable Server Pooling (RSerPool) framework to distribute the tasks onto the resource pool by satisfying four basic goals. It also supports multiple resource allocation policies, and we added three new ones: PLUD, PLU-DPF and PLUD-DPF. We have presented our simulated results and showed that three policies – LUD, PLU and particularly PLUD – are better for longer-running requests, while PLUD-DPF achieves a better handling speed for short requests. As future work, we plan to extend the analysis with real applications/benchmarks set up in a *true* geographically distributed scenario including OpenAirInterface-based EPC. We also intend to contribute our results into the IETF standardisation process of RSerPool, as well as the development of orchestration frameworks, particularly Open Source MANO.

References

1. Lei, P., Ong, L., Tüxen, M., Dreibholz, T.: An Overview of Reliable Server Pooling Protocols. Informational RFC 5351, IETF (September 2008)
2. Dreibholz, T.: Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture. PhD thesis, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems (March 2007)
3. Dreibholz, T., Mazumdar, S.: Reliable Server Pooling Based Workload Offloading with Mobile Edge Computing: A Proof-of-Concept. In: Proceedings of the 3rd International Workshop on Recent Advances for Multi-Clouds and Mobile Edge Computing (M2EC) in conjunction with the 35th International Conference on Advanced Information Networking and Applications (AINA). Volume 3., Toronto, Ontario/Canada (May 2021)
4. Dreibholz, T., Rathgeb, E.P.: Overview and Evaluation of the Server Redundancy and Session Failover Mechanisms in the Reliable Server Pooling Framework. International Journal on Advances in Internet Technology (IJAIT) **2**(1) (June 2009)
5. Dreibholz, T., Mazumdar, S.: A Demo of Workload Offloading in Mobile Edge Computing Using the Reliable Server Pooling Framework. In: Proceedings of the 46th IEEE Conference on Local Computer Networks (LCN), Edmonton, Alberta/Canada (October 2021)
6. Zhou, X., Dreibholz, T., Rathgeb, E.P.: 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 (February 2008)
7. Dreibholz, T.: HiPerConTracer - A Versatile Tool for IP Connectivity Tracing in Multi-Path Setups. In: Proceedings of the 28th IEEE International Conference on Software, Telecommunications and Computer Networks (SoftCOM), Hvar, Dalmacija/Croatia (September 2020)