A Generalized Approach For Practical Task Allocation Using A MAPE-K Control Loop

Reinout Eyckerman, Phillipe Reiter, Siegfried Mercelis, Steven Latré, Johann Marquez-Barja and Peter Hellinckx
IDLab - Faculty of Applied Engineering
University of Antwerp - imec
Sint-Pietersvliet 7, 2000 Antwerp, Belgium
Email: {firstname.lastname}@uantwerpen.be

***Abstract*—Fog and edge computing paradigms were adopted to enable Internet of Things (IoT) applications, improving response time and reducing network load. Task allocation algorithms are used on IoT-enabled networks to determine the optimal software placement. However, managing such a network is considerably more complex than allocating the tasks. To simplify management, we propose a general Monitor - Analyze - Plan - Execute over a Knowledge base (MAPE-K) framework in which all requirements for task allocation are fulfilled, and where components can easily be adapted to the use case at hand. This research identifies several pitfalls and proposes solutions. Additionally, we apply this approach to a distributed testbed, comparing it to traditional cloud approaches.**

result details

I. Introduction

The continuous increase of Internet of Things (IoT) devices allows us to better monitor, process and act on our environment. Due to the low computational capabilities of IoT devices, cloud servers are typically used for the actual data processing. However, as the number of IoT devices rises, the data load becomes a burden to the network. This can create considerable congestion, causing IoT applications to compete over the available bandwidth, while trying to maintain their stability when encountering potentially large network latencies. Wireless networks are a staple of the IoT concept, but can further complicate things when it comes to application reliability, with jitter and connection issues becoming commonplace. This can cause severe complexities for more demanding software that requires reliable and soft real-time connections.

Fog computing has been proposed to aid in reducing these issues [1]. This paradigm attempts to shift the software running on the cloud closer towards the end devices, decreasing the network load and improving application performance. The concept is based on making use of unused general compute resources close to the end user. A similar paradigm is edge computing, which moves the cloud computing to edge servers that are installed closer to the end users [2].

The perspective proposed in this study can be applied to both fog and edge computing, since in our case, as there is little difference in domain-specific requirements, objectives and constraints. However, allocating the tasks to devices is a considerable challenge, taking into account the available hardware and network resources and the required objectives. As the number of possible placements rises exponentially as tasks or devices are added, it becomes infeasible for a human user to find the best placement. Dynamic networks with devices joining and leaving the network further push the need for an automatic placement calculation. We propose a methodology for tackling the problem of placing tasks across the fog in a fully autonomous fashion. We do this by implementing a Monitor - Analyze - Plan - Execute over a Knowledge base (MAPE-K) control loop, and define issues that arise when looking at a practical implementation.

II. Problem Statement

The core difficulty in task allocation is finding the optimal placement. Large search spaces, multiple objectives and time constraints greatly influences this. However, the actual practical deployment can be considered an equally complex problem. The next section will look into the complexity of both, and defines how they fit together.

A. Task Allocation

The task allocation problem concerns the optimal placement calculation of each task across the device network according to one or multiple objectives while adhering to one or multiple constraints. These objectives can include latency minimisation, bandwidth usage minimisation, and also reducing the total Worst-Case Execution Time (WCET) across the devices, and minimising the energy footprint. The solution has to adhere to several constraints, such as available bandwidth, available device memory and maximally allowed WCET. There is extensive research on optimal placement calculation, each covering different metrics, objectives and constraints, and often working towards a specific use-case [3]–[5].

There is a large set of design choices when solving this problem, as defined in previous research [5], [6]. Examples include the choice of centralised versus distributed coordination, local versus global optimization, which objectives, which constraints, etc., all of which make defining a general methodology complex. This task allocation problem depends on several models, defined below.

1) Network: The network is considered a directed multigraph where every link is split into an up- and down-link. This allows independent modeling, as link directions can get throttled by network providers. The nodes represent the devices with their corresponding capabilities.

2) Application: The application is a directed multigraph as well, with tasks as nodes and their communication links as edges. This application is placed over the network. If a single monolithic application were to be deployed, little gain would be obtained due to the lack of degrees of freedom. In an optimal scenario, the application is a graph that contains tasks small enough that they can be run almost everywhere.

3) Constraints: There is a multitude of constraints when placing tasks, which are problem specific. We chose for the subset considering Central Processing Unit (CPU) usage, memory usage, Network Interface Card (NIC) usage, bandwidth usage and application deadlines, both communication and execution time, as defined in [6]. All these constraints must be met, otherwise the placement is considered invalid.

4) Objectives: Once the constraints are adhered, the objectives are minimised, which are also chosen depending on the problem at hand. These are often conflicting objectives, as minimizing WCET pushes tasks towards the cloud whereas minimizing latency pushes towards the end user. The trade-off made when selecting the most important objective creates a set of optimal solutions, called the Pareto Front, where no objective can be further optimized without degrading another objective. Out of this Pareto front, one solution must be selected to use as an actual placement. We will dive into solving approaches in Section III.

B. Task Management

The task placement problem has been shown to be quite complex. There is, however, a considerable gap between finding the optimal placement and the actual deployment. One often overlooked complexity is obtaining the network and application information for the algorithm. Fog devices are generally a heterogeneous set of devices. Thus, the monitoring of these devices should generate comparable metrics across different devices with potentially different architectures. This is, however, still open research, as the current methodologies either examine software for very specific architectures without resource competition, or try to estimate a bound by monitoring the resource usage of the application [7]. Due to resource constraints, the monitoring should use as little resources as possible, to make room for other processes and to keep energy usage at a minimum. Similarly, the application metrics need to be known per device. The accuracy of all metrics must be considered as well, as measuring detailed, yet costly, device info might have little added value if the application metrics lack precision. To support all approaches, we propose a general framework to obtain these metrics.

III. Methodology

To support dynamic environments, where devices can continuously join and leave the network, the placement and monitoring are required to work without any user interaction. To achieve this, we defined a Monitor - Analyze - Plan - Execute over a Knowledge base (MAPE-K) control loop framework [8]. This control loop divides the problem into several steps, each with their own challenges, as shown in Fig. 1. A central coordination mechanism provides the computational services required to determine optimal placements (Analysis), and configures the tasks to adhere to this placement (Plan). Monitoring and software execution are done by on-device agents, which register themselves at the coordination mechanism. In the case of distributed coordination mechanisms, the Analysis and Plan phases become part of the device-specific agents. We will now discuss the several phases of the control loop.

A. Monitoring

Monitoring the fog capabilities is in itself a considerable challenge. The monitor should be able to determine the device and network capabilities, which is not an easy task when considering devices running without an Operating System (OS). Moreover, the monitor should be able to determine the running process load on the device and network. All this has to happen with a minimal resource footprint, to prevent overloading the device and to ensure other processes can still run on-device. Recent state of the art is attempting to develop efficient monitoring tools, with for example Fogmon, provided by Brogi et al. [9]. We chose to develop a monitoring tool which runs on Linux devices capable of running the Java Virtual Machine (JVM). We used the Operating System and Hardware Information (OSHI) library to monitor the device capabilities and the device load [10]. This software library enables access to the device resource metrics, such as CPU load, memory availability and NIC usage. As we are working with a set of heterogeneous CPU, we have to keep in mind that tasks might use more CPU power and take longer on different devices. We tackle this issue a priori by creating a look-up table, which is provided to the coordinator. The table contains a dynamically estimated CPU load, memory load and WCET per task per device. This WCET can be measured using research such as that of Huybrechts et al., who provide a hybrid WCET measurement methodology [11].

Different techniques are used to measure the network capabilities. First we read the Linux forwarding table and routing table. Based on the forwarding table, each agent knows its directly connected neighbours. Using this information, the agent measures those links. Both the routing and forwarding tables are shared with the coordinator. Link latency is measured using the *ping* Linux command [12]. NIC packet measuring is used to determine the available bandwidth. An alternative, more intrusive methodology is iperf [13]. Other approaches exist, such as the In-band Network Telemetry (INT)-enabled architecture proposed by Haxhibeqiri et al [14]. They measure the overhead of WiFi communication, and provide this information to the applications interested. The agent monitors the running tasks as well. We will go into this in more detail in the Execution stage. All information determined by the monitoring tools is consequently sent to the analyzer for processing and logging.

B. Analysis

The analysis stage receives all monitoring information and stores it in its knowledge store for future reference. This stored

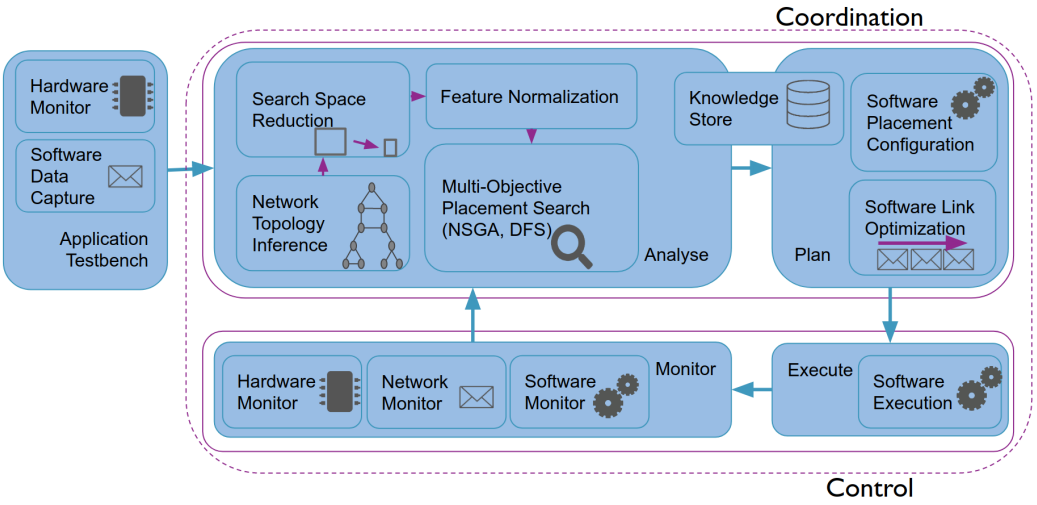

Fig. 1. MAPE-K Control Loop.

information allows future research in regards with predictive maintenance and control. From the routing and forwarding table, the analyzer infers the network graph, and then uses this information to determine an optimal placement. The placement is then transferred to the planning stage. Multi-Tenancy is inherently supported, as one or more analyzers can be used, which can support multiple applications.

1) Network Reconnection: Using the resource information shared by the monitor stage, the analyzer first tries to create a network graph. It determines the connections between the vertices using their routing tables. For this we work on the network layer implementing the Internet Layer and the Internet Protocol. This choice reduces complexity of handling multiple addressing protocols and multiple routing types. Using the forwarding table, the analyzer creates links between directly connected devices. The routing table allows the analyzer to determine which path the data takes when crossing the network, allowing for more accurate transfer estimates. This is done by determining a path matrix, which contains the shortest paths between a source device and a target IP. If a device is not running an agent, and thus not sharing its network information, the analyzer might no longer see how all devices are connected. This results in an incomplete network view, creating small islands of agents which are connected. This is seen on the left of Fig. 2. To ensure isolated agents can communicate, we provide Algorithm 1, where N represents the set of devices and E the set of network links. This algorithm uses the routing table to determine which devices have network interfaces towards other networks. All those network interfaces then create a fully connected graph, linking the islands together, as shown on the right in Fig. 2. Notable here is that multiple links can be created between two devices, depending on the number of NICs connected to the edge of the network. Targeting a different NIC on the same device can cause the packets to follow a different route with different resources, which can be exploited by the analyzer.

Algorithm 1: Network Reconnection.

links = E;
potential_links = {};
for $src \in N$ **do**
  for $dest \in N \backslash \{src\}$ **do**
    $out\_nics = queryRoutingTable(src, dest)$
    if $out\_nics = \{\}$ **then**
      $out\_nics = \{src_{defaultgateway}\}$
    end
    for $nic \in out\_nics$ **do**
      if $\{src, dest, nic\} \notin links$ **then**
        $potential\_links \cup \{src, dest, nic\}$
      end
    end
  end
end
for $link \in potential\_links$ **do**
  for $back\_link \in potential\_links$ **do**
    if $link_{dest} = back\_link_{src}$ ***and*** $link_{src} = back\_link_{dest}$ **then**
      $E \cup \{link_{nic}, back\_link_{nic}\}$
      $E \cup \{back\_link_{nic}, link_{nic}\}$
    end
  end
end

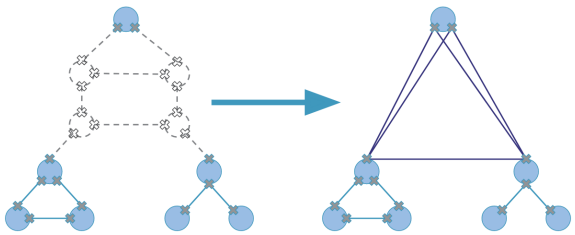

Fig. 2. Reconnection graphs when devices are not sharing information.

2) Search Space Reduction: Due to the potentially large search spaces, reduction methodologies can be applied to facilitate search algorithms. One possibility is using the constraints to remove invalid solutions in the search space, reducing the constraint-checking load in the placement algorithm. Another possibility is using graph matching to reduce the application graph, as proposed in previous research [6]. This reduces the number of possibilities while maintaining a decent solution.

3) Feature Normalization: As previously mentioned, there are often multiple objectives. These objectives tend to have different magnitudes, making it difficult to compare them and weigh them together using a weighted sum approach. This can be solved using weight-induced normalisation, where the weights are chosen so that they scale up / scale down the objective. This approach, however, requires in-depth knowledge about the specific problem at hand, something that is often unavailable. Another approach exploits the Pareto front's edge values to determine the minima and maxima of each objective on this front, as defined in [6]. These can then be used to normalize the objective space. These minima and maxima are found by running a modified NSGA-II in advance, which estimates the border solutions of the Pareto front. These solutions estimate the best and worst values for all objectives on the Pareto front, giving a good range for minimization. This approach is especially useful in scenarios where the normalization has to happen inside of the algorithm (e.g. Single Objective Algorithms).

4) Placement Calculation: In previous research, we focused on generalizing this problem, allowing for placements which are problem-independent for greater flexibility, with the disadvantage of generally having a larger search space [6]. This larger search space can be attributed to the inability of adding problem-dependent knowledge to the search algorithm which could reduce the search space. Multiple methodologies exist to solve multi-objective problems. One common approach is using (Multi-Objective) Evolutionary Algorithms, where Single Objective Genetic Algorithm and the Multi Objective NSGA-II are key techniques [15]. Decentralized approaches exist as well, such as Distributed Reconnaissance Ant Colony Optimization (DRACO) [5], with a growing focus on Multi-Objective Reinforcement Learning (RL) as well [16]. Using such algorithms, which can be fine-tuned to the problem, we select an optimal placement based on a single-objective function. Several approaches exist for this, as defined in [17], including weighted sum and lexicographic ordering.

As the network changes continuously, so do the objective and constraint values. The analyzer has to consider this separately to keep placements optimal, by determining if the current task allocation still adheres to the constraints. If it does not, a redistribution is determined and executed. In the case that the constraints are still adhered, the analyzer still determines a new placement, which is then compared with the original placement and checked to see if it better adheres to the objectives, while considering migration cost. Note that migration is not taken into account during placement search. This is due to the difference between task allocation and task migration problems, as defined in [18]. Task migration problems try to minimise the effect of migration for existing placements, whereas task allocation problems try to optimize the location of the tasks over the network. The short term perspective of task migration maps poorly onto the longer term perspective of task allocation. Although this research uses task allocation, this can easily be swapped out by a related task migration analyzer. We assume a minimal impact of application migration, as solely the application state will be migrated. The new task can be loaded early, requiring reconfiguration of the connected tasks once the state is reloaded. All this, however, depends on the specific application at hand. An additional complexity is software failure detection, prevention and recovery, and is left out of scope.

C. Plan

After analyzing the situation, the software execution is planned. Increasing the number of configurable parameters available for the tasks at hand improves the gain which can be used by an advanced planner. For example, Distributed Uniform Streaming (DUST) is designed with this high configurability in mind [19]. DUST is a middleware intended to simplify the creation and maintenance of distributed software. It allows the dynamic manipulation of the communication stack through configuration files, using a modular approach of message transformers and transport protocols.

Using the placement defined by the analysis phase, the planner maps the tasks to the agents and configures the communication channels between the tasks. For channels between tasks on the same device, the planner can configure the DUST-oriented tasks to use IPC sockets, reducing the overhead. For channels between tasks hosted on different devices, the planner writes configurations for the tasks, configuring the target IP addresses using the ZeroMQ protocol. Different protocols are supported by DUST as well, but we leave research into scenario-specific protocol selection depending on scenario for future research. Given models of the communication links, this can be further enhanced by adding compression onto communication links which benefit from compression and where the network is close to being congested. This concept is further elaborated in [20]. Finally, the planner communicates this configuration to the executing agents.

D. Execute

Depending on the information received by the planner, the executing agent loads the tasks it has to run from storage and launches these with the configuration imposed by the planner. Storage can be both local or remote. To reduce complexity, we use Docker containers for task deployment. We assume every agent has enough storage. Upon launch, the agent starts monitoring the deployed tasks by checking the return codes. We assume to work with tasks which keep listening for input. If such tasks finish, we assume a crash and re-launch the process. Repeated crashes are alerted to the analyzer, which then looks for a new executing agent to run the task.

E. Application Testbench

Although not a part of the MAPE-K control loop, an application testbench can provide a considerable benefit in controlled environments. Used before deployment, such a testbench measures the task resource consumption, enabling accurate placements. This does, however, require that access to the devices in the network be obtained in advance, which is unrealistic in highly dynamic environments, such as fog environments with e.g. smartphones. In controlled environments, however, we can determine the resource consumption and WCET in advance. Moreover, the application testbench can provide application bandwidth requirements. Using each application's communication configuration provided by the DUST framework [19], the testbench can discover and connect to these communication channels and measure the amount of data generated to determine the worst case bandwidth consumption. All this information is provided to the coordinator, to provide more accurate and optimal placements.

IV. Use Cases

We will describe the use-case and subsequently the testbed.

A. Distributed AI

With rapid advancements in Artificial Intelligence (AI), interest is gathering in applying it to everyday life. One example is smart traffic lights, where multi-agent Reinforcement Learning (RL) is used to optimize traffic throughput. Another is predictive maintenance, where hardware monitoring is combined with pre-trained neural networks. It allows predicting when a certain component will fail, which enables the preventive replacement of the defective component, avoiding any potential downtime. We find our distributed AI use case within Industrial IoT. Let us consider a container terminal where cranes are continuously moving containers around and people are simultaneously walking around to monitor the placements and items. Multiple cameras monitor the terminal and employees. Using facial recognition, activity logging and behaviour prediction, unregistered people and unusual behaviour is reported. Moreover, using object tracking and movement prediction, alerts are sent to alarm employees when they walk in the path of a moving crane. We optimized for the subset of minimising bandwidth usage, application latency, WCET and energy consumption. We will assign objective preference by weighted sum using the ranking method.

B. Testbed

The use case will be validated using custom load generators. *Stress-ng* allows configurable CPU and memory loads to generate the devices [21]. Network load was generated by sending the determined amount of random data towards the target task. WCET is simulated data. These load generators were built upon the DUST Framework, and wrapped with Docker containers [22]. This DUST framework is currently available under an open-source license, and enabled our Execution stage to reconfigure the software depending on the position of the devices. To test this software, we designed a distributed testbed where multiple devices provide Virtual Machines (VMs). Using multiple devices has the benefit of upscaling the network while ensuring all devices have their own resources. The testbed makes use of IP-over-IP tunnels to ensure communication between VMs on different devices but on the same virtual network.

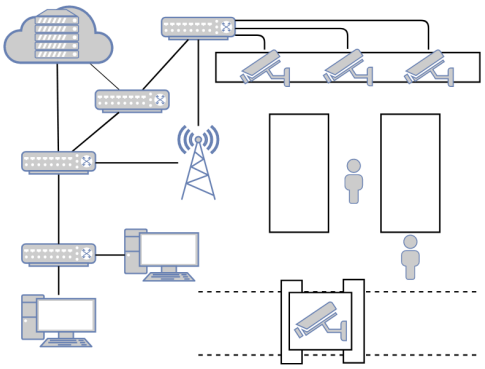

Fig. 3. Industrial centre example.

V. Experiment

Uitbreide

We will compare the continuously running placement methodology to a centralised cloud methodology. In the Cloud Placement scenario, an application chain consisting of 14 tasks is placed on the cloud server, with sensors and actuators sending and receiving data, respectively. In a MAPE-K scenario, this same application chain is distributed across the devices. In our use-case, Fig. 3, the application is constrained in that it cannot run in the cloud, due to latency demands. The NSGA-II algorithm is used with a ranking-based weighting where total application latency has the highest priority followed by bandwidth, WCET and energy in descending order.

We can see some preliminary results in Fig. 4. Here, the vertical axis represents the objectives of the algorithm, normalized and summed together, using the ranking defined above. This allows directly comparing placements. We can see that the NSGA-II algorithm does slightly better than the cloud placement. Although this result might seem not impressive, one does have to take into account that it tends to achieve at least the same performance as the cloud placement, while being able to adhere to the constraints.

Do note that with domain knowledge, one might use more efficient algorithms which work better on the scenario at hand. We chose NSGA-II as it is a stable baseline. Also note the instability of the cloud placement solution. Although one would suspect this to be a straight curve, with the solution staying optimal, this is not the case, even in a relatively static network. This is due to the dynamic behaviour of the devices themselves and measurement precision errors. This line fluctuates depending on the measurement quality and the amount of other services running in the network. In multi-tenant scenarios, this line might fluctuate intensely due to the impact of other applications on the measurements. Using more accurate methods to measure bandwidth, such as the

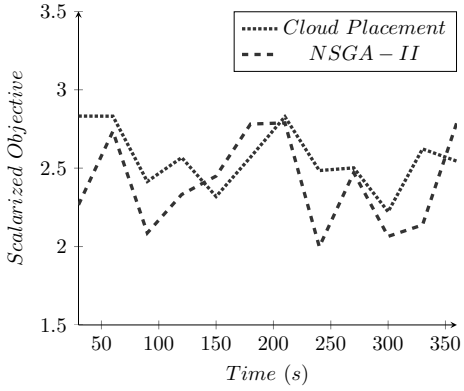

Fig. 4. Cloud Placement vs MAPE-K Approach.

previously mentioned INT, will further stabilise this line [14]. Over an average of 20 runs, we noticed that our MAPE-K approach has improvement of 11% over the cloud placement, with an average of 150ms total application latency, compared to an average of 190ms with a cloud placement.

VI. Discussion

In this paper, we defined a general architecture for MAPE-K control loops for task allocation problems. We defined the requirements and complexities of the task allocation problem in a practical context, and proposed several methodologies for providing these requirements in an autonomous fashion. There is still considerable room for improvements on every part of the MAPE-K loop, but these are left for future research as they are problem-specific. Although task allocation will benefit from more accurate metrics, one should consider if finding these metrics is worth the benefit, as determining these can be costly. Determining the throughput of a link can be determined using *iperf*, but this congests the entire link and uses precious hardware resources to generate the packets. Our approach has been validated against a virtual testbed, showing the strength of autonomous task allocation.

VII. Future Work

Every step of the control loop can be further optimized to satisfy multiple aspects of varying application domains. Monitoring can be optimized by using network controllers to measure the link capabilities for the coordinator. This removes the overhead generated while measuring the link capabilities.

Additional research can be done into comparing device capabilities. Using the processor load is a rough estimate, and using the task-set generator to estimate WCET improves the results. However, better results and placements can be achieved by comparing several different devices on additional metrics, such as on processor speed, number of cores and architecture.

The analysis can be improved with further research into placement algorithms, such as automatic determination of the problem type to do search space reductions. Improving the Planning phase is dependent on the middleware used. The higher the configurability of the middleware, the more options the planner has to improve application performance [20].

Acknowledgment

This research received funding from the Flemish Government (AI Research Program). This article describes work in the context of the DEDICAT 6G project under the European Union (EU) H2020 research and innovation programme (Grant Agreement No. 101016499). The contents of this publication are the sole responsibility of the authors and do not in any way reflect the views of the EU.

References

[1] M. Aazam *et al.*, "Fog Computing Architecture, Evaluation, and Future Research Directions," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 46–52, 2018.

[2] T. Taleb *et al.*, "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration," *IEEE Communications Surveys and Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[3] R. Mahmud *et al.*, "Context-aware Placement of Industry 4.0 Applications in Fog Computing Environments," *IEEE Transactions on Industrial Informatics*, vol. 3203, no. c, pp. 1–1, 2019.

[4] S. Wang *et al.*, "Dynamic Service Migration in Mobile Edge Computing Based on Markov Decision Process," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1272–1288, 2019.

[5] R. Eyckerman *et al.*, "Requirements for distributed task placement in the fog," *Internet of Things*, vol. 12, p. 100237, dec 2020.

[6] R. Eyckerman *et al.*, "Evaluation of objective function descriptions and optimization methodologies for task allocation in a dynamic fog environment," in *2020 7th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2020, pp. 1–8.

[7] S. Jiménez Gil *et al.*, "Open challenges for probabilistic measurement-based worst-case execution time," *IEEE Embedded Systems Letters*, vol. 9, no. 3, pp. 69–72, 2017.

[8] J. Kephart *et al.*, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[9] A. Brogi *et al.*, "Measuring the Fog, Gently," in *Service-Oriented Computing*, S. Yangui *et al.*, Eds. Cham: Springer International Publishing, 2019, vol. 11895, pp. 523–538.

[10] OSHI, "OSHI," https://github.com/oshi/oshi.

[11] T. Huybrechts *et al.*, "A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning," no. 5, pp. 1–5, 2018.

[12] "Ping(8) - Linux man page," https://linux.die.net/man/8/ping.

[13] "iPerf - The TCP, UDP and SCTP network bandwidth measurement tool," https://iperf.fr/.

[14] J. Haxhibeqiri *et al.*, "Low Overhead, Fine-grained End-to-end Monitoring of Wireless Networks using In-band Telemetry," Tech. Rep.

[15] K. Deb *et al.*, "A fast and elitist multi-objective genetic algorithm:NSGAII," vol. 6, no. 2, pp. 182–197, 2002.

[16] R. Rădulescu *et al.*, "Multi-objective multi-agent decision making: a utility-based analysis and survey," *Autonomous Agents and Multi-Agent Systems*, vol. 34, no. 1, p. 10, apr 2020.

[17] R. Marler *et al.*, "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, vol. 26, no. 6, pp. 369–395, Apr. 2004.

[18] T. G. Rodrigues *et al.*, "Cloudlets Activation Scheme for Scalable Mobile Edge Computing with Transmission Power Control and Virtual Machine Migration," *IEEE Transactions on Computers*, vol. 67, no. 9, pp. 1287–1300, 2018.

[19] S. Vanneste *et al.*, "Distributed Uniform Streaming Framework: Towards an Elastic Fog Computing Platform for Event Stream Processing: Proceedings of the 13th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2018)," pp. 426–436, 2019.

[20] R. Eyckerman *et al.*, "Towards the Generalization of Distributed Software Communication." Springer International Publishing, 2021, pp. 261–270.

[21] "stress-ng." [Online]. Available: https://kernel.ubuntu.com/ cking/stress-ng/

[22] "DUST: The Distributed Uniform Streaming Framework." [Online]. Available: https://dust.idlab.uantwerpen.be/dust/docs