

Multi-armed Bandits for Self-distributing Stateful Services across Networking Infrastructures

Frederico Meletti Rappa

Instituto de Computação

Universidade Estadual de Campinas
Campinas, SP, Brazil

Roberto Rodrigues-Filho

Departamento de Computação

Universidade Federal de Santa Catarina
Araranguá, SC, Brazil
roberto.filho@ufsc.br

Alison R. Panisson

Departamento de Computação

Universidade Federal de Santa Catarina
Araranguá, SC, Brazil
alison.panisson@ufsc.br

Leandro Soriano Marcolino

School of Computing and Communications
Lancaster University

Lancaster, UK
l.marcolino@lancaster.ac.uk

Luiz F. Bittencourt

Instituto de Computação
Universidade Estadual de Campinas
Campinas, SP, Brazil
bit@ic.unicamp.br

Abstract—The investigation of stateful service mobility across networking infrastructures is becoming increasingly important as applications require stateful services capable of migrating from centralized cloud data centers to edge computing infrastructures. State-of-the-art approaches propose either machine learning solutions for *stateless service* placement or stateful service mobility using *static and inflexible* state management strategies. We believe these approaches fall short of addressing the full length of the stateful service mobility problem. In this paper, we revisit an emerging concept named self-distributing systems, where a local executing application manages to detach some of its constituent (often stateful) components and place them in remote machines as a solution for stateful service mobility. In previous work, a machine learning approach to support self-distributing systems has not been thoroughly investigated. We model the distribution of stateful components across networking infrastructures as a multi-armed bandits problem and use the UCB1 algorithm to solve it as a first attempt at a flexible solution for stateful service mobility. We conclude the paper by discussing the main challenges and opportunities in this area.

Index Terms—stateful service mobility, edge-cloud infrastructures, reinforcement learning, self-distributing systems

I. INTRODUCTION

Applications deployed over the edge-cloud continuum [1] are often required to move their services across the infrastructure to exploit the trade-off between resource availability and network latency. Therefore, in order to explore code mobility throughout networking infrastructures without adding the complexity of properly managing services' state consistency, the development of stateless service architectures, such as Function-as-a-service and microservices, became popular.

Designing a stateless service-based system is a convenient way to explore the underlying adaptive platforms that currently support modern applications. Cloud and edge-based infrastructures are supported by containers, container-orchestrators (e.g., Mesos¹, Kubernetes²), and softwarized networks that allow

adaptation of the underlying infrastructure and enable mobility of stateless services wrapped inside containers.

However, avoiding state when developing services is not always possible. Many applications require stateful services to properly function. Nowadays, there is an increase in demand for service mobility capable of migrating from cloud-based platforms to edge computing infrastructures deployed in close proximity to end-user devices. To tackle such issues, some papers have looked into the concept of stateful Function-as-a-service [2], [3], but these solutions often employ static mechanisms to deal with state when moving services across platforms and do not employ any machine learning solutions for service mobility. Moreover, there are many papers in the literature that apply machine learning for service mobility, but they often target stateless services [4], [5].

Self-distributing systems concept [6], on the other hand, enables the flexible distribution of stateful components executing on a local container to other containers executing across infrastructures (e.g., edge to cloud and vice-versa), choosing a state management strategy that better fits the demands of the application. Previous work that employs such a concept, however, either does not explore machine learning in the process of distributing components (using a brute-force online strategy instead), or only targets stateless components. Thus, we currently lack a study of a machine learning approach for the problem of distributing stateful components across networking infrastructures.

In this context, this paper presents the following contributions: *i*) defining the problem of autonomously learning where to place stateful components over distributed platforms, *ii*) showing preliminary results that support the potential of applying a completely autonomous solution for autonomous placement of stateful components, and *iii*) identifying challenges and opportunities in this research field.

The remainder of this paper is organized as follows: Sec. II surveys the most relevant related work; Sec. III revisits the

¹<https://mesos.apache.org/>

²<https://kubernetes.io/>

Self-distributing Systems concept and how it can be used to explore stateful service placement. Moreover, this section also defines self-distribution of stateful components as a multi-armed bandits problem; Sec. IV describes our preliminary evaluation and results; Sec. V discusses challenges and opportunities; and finally, Sec. VI concludes the paper.

II. RELATED WORK

This section surveys related on three main topics: *i)* stateful service mobility, *ii)* machine learning for placement and mobility of services over networking infrastructures, and *iii)* previous work on self-distributing systems.

We first survey work on stateful service mobility. In this topic, Function-as-a-Service (FaaS) presents itself as the state-of-the-art solution for enabling the mobility of stateful services over edge-cloud computing infrastructures [2], [3], [7], [8].

As it is well-known, current FaaS programming models provide mechanisms to replicate and move functions across edge-cloud platforms. However, these platforms only provide support for stateless functions, not offering any support for stateful functions. On the other hand, recent papers in the literature [2], [3], [7], [8] present mechanisms to add a replicated key-value (KV) store close to the executing function, along with consistency strategies to avoid replicated inconsistent data. By having a KV store and a consistency management strategy close to functions, these approaches enable the use of the FaaS abstractions to support stateful functions mobility.

Our approach differs from the stateful FaaS work in two major aspects. First, current approaches for supporting replicated data consistency in stateful FaaS are predefined and fixed, not considering the details of how the function handles the data nor the characteristics of the operating environment. Second, the stateful FaaS do not leverage any machine learning solution for autonomously deciding where to place each function.

Our approach, on the other hand, offers a set of state consistency strategies that better fit with the way the state is handled by the service or the characteristics of the executing environment. For instance, if state is represented as a linked list, we could either replicate such lists and provide a strict consistency model when read operations (*i.e.*, operations that only read the list elements, but not change them) make up the majority of the interaction between the service and the state. Similarly, a different approach could be employed, such as sharding a list (*i.e.*, break the list into smaller fragments). Sharding consists of placing each list fragment in different replicas of the service. This can improve the system's performance when, for instance, the service's state holds different information depending on the geographical location of the service. Furthermore, our approach employs a reinforcement learning algorithm to decide which state management strategy to use and where to place services.

The application of machine learning techniques for service placement is not new. There is a set of works that target the service placement problem using machine learning techniques [4], [5], [9], [10]. These papers model service placement as an optimization problem and provide a machine

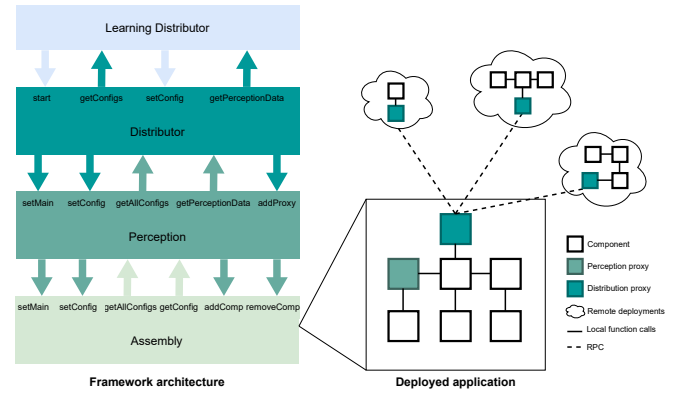


Fig. 1. Self-distributing systems architecture. The framework is depicted on the left side of the figure, and the target application on the right side.

learning technique to solve such a problem considering networking metrics and goals. These papers are different from our approach in some aspects and complementary in others. The surveyed papers differ from ours in one key aspect: they do not target stateful services. Handling state consistency in replicated services is not easy and has interesting side effects when learning the placement of such services at runtime. On the other hand, these papers are similar because they aim to autonomously solve the service placement problem and could serve as complement to our solution in future work.

Finally, we present previous work that introduces and explores the concept of self-distributing systems [6], [11], [12]. The self-distributing system concept entails the development of software from a collection of small software components, and at runtime, relocate these components throughout a networking infrastructure. Component adaptation is performed at runtime with the assistance of a component-based model runtime [13], [14] (Dana³). Previous work has introduced the concept of relocating stateful components at runtime with different state consistency strategies. However, previous work does not explore a machine learning algorithm for distributing stateful components across a networking infrastructure. This paper, on the other hand, investigates multi-armed bandits algorithms [15] to solve the stateful service mobility.

III. SELF-DISTRIBUTING SYSTEMS

The concept of self-distributing systems has been previously explored in the literature. Rodrigues-Filho et al. [12], and Rodrigues-Filho and Porter [6] have introduced the concept and explored it within the context of web-based services.

To realize the concept of self-distributing systems, we employ a framework consisting of four main modules: Learning, Distributor, Perception, and Assembly, depicted on the left-hand side of Fig. 1. This framework runs on a single process at the OS-level, and it is responsible for assembling and executing the actual application (hereafter referred to as target application), depicted on the right-hand side of Fig. 1. In summary, this framework is responsible for assembling, executing,

³Dana Programming Language: <https://www.projectdana.com/>

monitoring, and deciding where to relocate or replicate the components that make up the executing application.

The distribution of components at runtime made by the self-distributing system framework is done by leveraging the component-based model we employ. When we create the target application, we use the Dana⁴ programming language to code each individual component that makes up the application. Note that some of these constituent components hold state. These components are similar to objects in object-oriented programming languages because they are small and encapsulate state (*i.e.*, attributes) and methods. Furthermore, we make available a set of proxy components that the Distributor inserts in the application's architecture to replace any constituent component. These proxy components act as the regular components, providing the same functionality but they execute as a Remote Procedure Call (RPC) stub. Once a method is invoked in the proxy, it forwards incoming method invocation to an instance of the component that is now executing in a remote machine. Each available proxy component implements a different state management strategy, depending on the state the original component holds. For more details, see [12].

The distribution of stateful components consists of creating new instances of a certain stateful component that is part of the original target application across different machines and replacing the local component instance with a proxy component responsible for forwarding requests to the newly created remote replicas of the component while maintaining the replicas state consistently.

A. Learning with Multi-armed Bandits

Given the ability of the self-distributing systems framework to distribute stateful components across an infrastructure, we end up with the problem of deciding where to place the stateful components of a target application. For this, we model the placement problem as the multi-armed bandits' problem [15].

The multi-armed bandits' problem is a classic problem in statistics where an agent must select actions to maximize its reward. Most algorithms can be defined as an agent that has K possible actions and T rounds; in each round, the agent must select one action that has an associated reward. Therefore, such algorithms must be able to balance exploitation by choosing the apparent best action for that scenario and exploration, by choosing different actions.

In our context, actions become different ways to compose the system (*i.e.*, the different ways to distribute stateful components across the infrastructure), whereas the rewards are the calculated response time for the target application executing in a specific composition. The Distributor module of the self-distributing framework receives a set of proxy components and a set of machines available in the infrastructure, it then generates a list of actions based on the resulting set of possible ways in which to distribute the target application over a given infrastructure. The Perception module, in turn, is responsible for calculating the average response time for the chosen target

application composition. This response time is later used to calculate the machine learning algorithm's reward.

We employ the UCB1 [15] algorithm to solve the multi-armed bandit problem we just defined. UCB1 utilizes the Upper Confidence Bound to choose its action. After choosing each action at least once, which is usually done when the model is initialized, the choice of an action in round n can be expressed as follows:

$$A_n = \operatorname{argmax} \left(\bar{x}_j + \sqrt{\frac{2\ln(n)}{n_j}} \right) \quad (1)$$

where \bar{x}_j represents the average reward of an action, n_j the number of times action j was chosen. Commonly, it is expected that the average reward and the right-size term are scaled to $[0, 1]$. By employing Equation 1, the algorithm is able to balance exploration and exploitation phases and converge towards a specific distributed composition for the target application with no training nor human interference.

IV. EVALUATION

A. Methodology and scenarios

We evaluate the performance of UCB1 to learn the optimal composition for a web-based stateful service in four distinct scenarios. Each defined scenario was designed to observe how UCB1 performs when the application state varies in size and response time. In our experiments, optimal composition means the composition of the target application that yields the lowest measured response time over a series of requests.

In detail, we create a web service that handles incoming HTTP requests to retrieve elements stored in a linked list (reading operation) or add/remove elements to the list (writing operations). Moreover, for each retrieval request, the service executes a prime number calculation function to simulate a CPU-bound operation that impacts the service's response time, depending on the number of elements the list contains. As the number of elements in the list increases, the service response time increases according to a cost function (see Equation 2).

Let N be the length of the list that represents the state, and k a positive integer value, the cost of an operation with a state of size N is:

$$T = \sum_{i=1}^N \pi(i * k) \quad (2)$$

where $\pi(n)$ is a function that counts the prime numbers less than or equal to n . Therefore, both reading and writing operations have $O(N^3 k^2)$ complexity, with k defined empirically.

We also make available two distinct compositions for the self-distributed systems framework to explore; the web service can either execute in a single instance on a single machine (local composition) or be distributed, having its list sharded in half for two executing replicas of the service in two remote machines (sharding composition). The self-distributing system framework can seamlessly adapt the web service from one composition to the other with no downtime.

⁴<https://projectdana.com>

B. Preliminary results

We used two machines for the evaluation setup: the first machine managed the HTTP server, distribution, and processes responsible for learning and coordinating distribution when in use. Its specifications included an i7-8700 CPU 3.20 GHz CPU, and 16 GB of DDR4 RAM. The second machine hosted two instances of the application to which the state was distributed, and its specifications are an i5-8265 1.6 GHz GPU and an 8 GB DDR4 RAM. Both machines are connected to the same local network.

The results depicted in this section were measured by executing fully functioning services on a real-world setup on a small scale testbed. We did not simulate any part of the experiment.

Every graph presented in this section depicts three versions of the web service. A fixed/static version of the web service executing as a single instance on a single machine (Local – blue line), a static version of the web service executing in a distributed composition, where the list is sharded and distributed between two replicas of the web service executing on two remote machines (Sharding – orange line), and the self-distributing system seamlessly adapting between the two available compositions of the web service (local and sharding) to decide which yields the best response time at runtime (green line). The y-axis is the measured response time, and the x-axis represents the requests handled by the services over time.

The first scenario consists of a state represented by a two-element list, with a cost factor $k = 2$. In this scenario, the list's size remains constant, and the clients make consecutive requests to retrieve the elements from the list. As seen in Figure 2, the sharding composition has, on average, higher response time and higher variance of values than the strictly local composition, which might be explained by the network latency to access the remote replicas of the service. It is clear that the local composition is more advantageous in this scenario, and the self-distributing system, shown in green, clearly converges to this composition.

The second scenario consists of a state represented by a 12-element list, with a cost factor $k = 5$. Similar to the previous scenario, the list's size remains constant, and the clients make multiple requests to retrieve the elements from the list. As seen in Figure 3, the sharding composition has an average response time of 392 ms, while the strictly local composition has 640 ms on average. As expected, after exploring both compositions, the self-distributing system converges to the distributed setup (sharding composition).

In the third scenario, the system is initialized with an empty state, and one element is added to the list every 2.5 seconds. The purpose of this experiment is to assess if UCB1 is capable of converging when the size of the state increases. In this case, UCB1 should prioritize keeping the state local for small lists and shard the state as the size increases. As seen in Figure 4, the remote composition becomes advantageous from around 10 elements in the list. The graph shows, however, that the system is unable to converge to the composition with a lower response

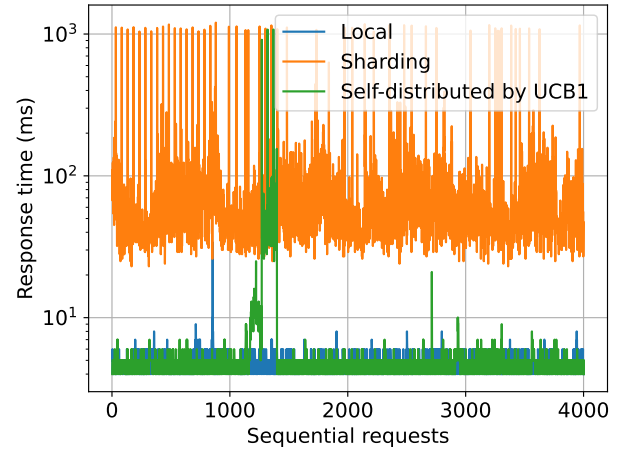


Fig. 2. The list's size remains fixed with size 2, and the cost factor is defined as $k = 2$. The local composition yields the best overall response time. The self-distributing system converges to the local composition.

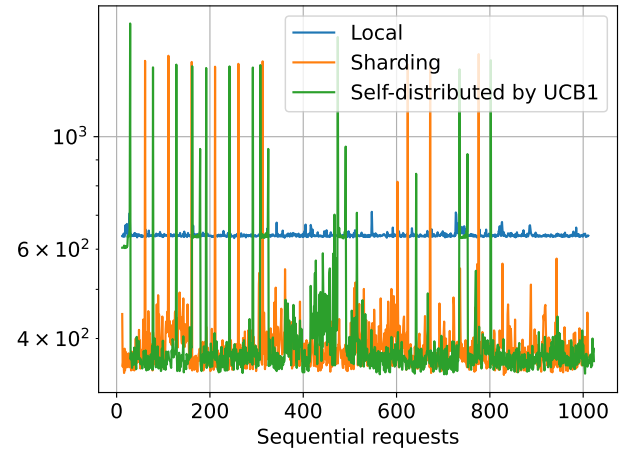


Fig. 3. The list's size remains fixed with size 12, and the cost factor is defined as $k = 5$. The sharding composition yields the best overall response time. The self-distributing system converges to the sharding composition.

time for large states. One reason why this behavior could occur is, since the range of response time value is unknown a priori, it is not possible to establish a function that restricts the values obtained to the necessary interval $[0, 1]$: as the system state increases in size, so does the processing time required to add elements to the list. This makes UCB1 inefficient in converging towards a composition within a reasonable time.

Furthermore, as the size of the list increases, the adaptation cost becomes more significant and exceeds the response time shown in both fixed compositions (local and sharding). In many instances, when the self-distributing system adapts from one composition to another, the size of the state has a negative effect on its performance, as it demands more time to properly handle the state to avoid creating inconsistencies during the adaptation process.

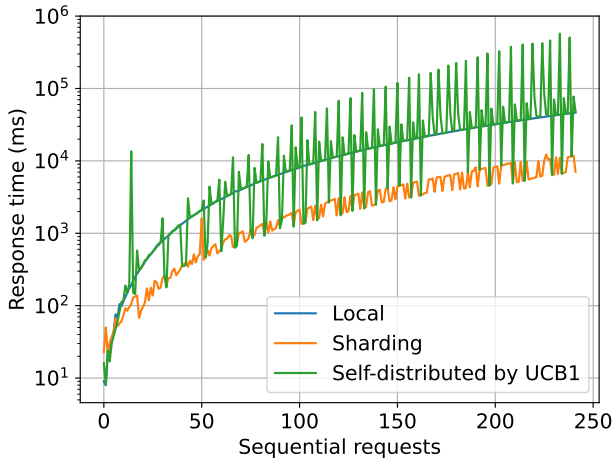


Fig. 4. The list's size increases over time (factor $k = 2$). The sharding composition yields the best overall response time after the list reaches a certain size (after 0 on the x-axis). The self-distributing system does not converge.

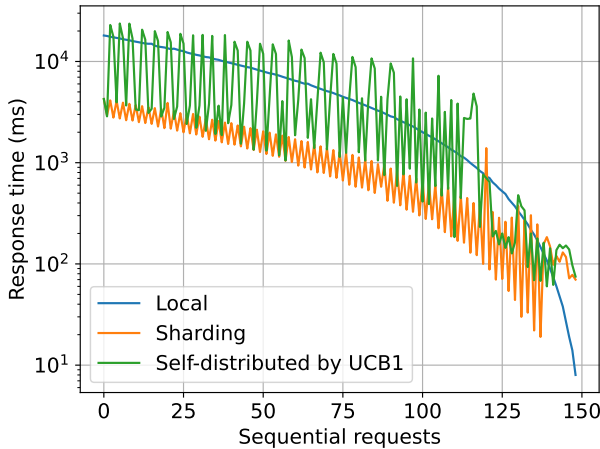


Fig. 5. The list's size decreases over time (factor $k = 2$). The sharding composition yields the best response time while the list maintains a certain size (before 150 on the x-axis). The self-distributing system does not converge.

Finally, the fourth scenario is the opposite of the previous one: the list is initialized with 150 elements, while elements are removed at every 2.5-second interval throughout the experiment. Similar to the previous scenario, this experiment aims to determine if UCB1 can converge when the size of the state decreases; the system should adopt the sharding composition for large lists and swap to the local composition when it reaches a small size, as elements are removed.

Figure 5 illustrates the UCB1's behavior in this final scenario. While both static compositions behave as expected, the self-distributing system clearly does not converge to any composition: the chosen action changes constantly, and, as in the previous scenario, the response time perceived by the client is higher than any of the static compositions. As outlined above, possible reasons for this behavior are the lack of a well-

defined cost function and the varying overhead caused to the system during adaptation when the size of the state fluctuates.

V. CHALLENGES AND OPPORTUNITIES

This section describes the identified challenges in providing a machine learning approach for handling stateful services based on the preliminary results we reported. We also describe opportunities for future work in this area.

As discussed in the evaluation section (see Sec. IV), our preliminary results showed some limitations of applying an off-the-shelf version of the UCB1 algorithm. First, we see that the system's runtime adaptation has a cost associated with the system's state size, which greatly impacts the system's ability to learn from its current context and correctly choose an action.

Due to the unpredictability of the response time, as a consequence of the adaptation cost in relation to the state's size, the lack of a function that limits the cost to the $[0, 1]$ interval expected by UCB1 directly impacts its ability to converge. Therefore, it is essential that the learning algorithm is modified to consider fluctuations in the reward signal. One possible alternative is to apply Contextual Bandits algorithms [16], in which each choice made by the algorithm considers both the action's characteristics (features) and the reward, instead of just considering the reward.

Furthermore, the UCB1 was unable to identify scenarios in which the ideal service composition changed over time, as a consequence of increasing or decreasing the state size. This limitation is possibly exacerbated by the fact UCB1 considers only one metric (response time, in our experiment) during the learning process. Thus, the introduction of other metrics may be necessary to better characterize contexts that would help the convergence towards the ideal composition.

The discussed limitations were identified by applying a popular multi-armed bandits algorithm with no modifications to consider the unique aspects of the proposed learning problem. Therefore, we envision the exploration of this problem following two paths: first by altering UCB1 algorithm to consider new metrics and adjust its cost function, and second by exploring new types of algorithms (*e.g.*, Contextual Bandits [16]) to see which approach enables proper convergence of the self-distributing system. Moreover, we believe that the investigation of different types of state and different stateful services (*e.g.*, virtual network functions, instead of web-based service) are also interesting research opportunities.

Finally, the exploration of larger search spaces resulting from the possible ways in which a stateful service can be distributed is a crucial line of research. In our preliminary experiments, our approach explored a search space composed of two compositions (local and sharding) over a single machine or replicated across two machines. In real settings, the infrastructure may have hundreds of nodes, which would dramatically increase the possibilities of distributing the service's components. Approaches such as the one introduced by Ontanón [17] are interesting candidates to explore.

VI. FINAL REMARKS

This paper explored the problem of autonomously learning how to distribute stateful components across networking infrastructures. We implemented a self-distributing stateful web service that can autonomously adapt and converge towards specific compositions using the UCB1 algorithm, a Multi-armed bandits algorithm.

By measuring the service's response time, the system can correctly converge to the best-performing composition when the state's size remains static. However, UCB1 was unable to successfully converge in situations where the size of the state changes. Regardless of the presented limitation, this paper shows the potential of using multi-armed bandits algorithms to solve the stateful service placement problem.

In future work, we expect to further investigate the application of other algorithms, and explore different services with varying state characteristics.

ACKNOWLEDGMENT

This work was partially supported by the São Paulo Research Foundation (FAPESP), grant 2021/00199-8, CPE SMARTNESS, and by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

REFERENCES

- [1] S. Dustdar, V. C. Pujol, and P. K. Donta, "On distributed computing continuum systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, 2023.
- [2] T. Pfandzelter and D. Bermbach, "Enoki: Stateful distributed faas from edge to cloud," in *Proceedings of the 2nd International Workshop on Middleware for the Edge*, ser. MiddleWedge '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 19–24. [Online]. Available: <https://doi.org/10.1145/3630180.3631203>
- [3] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: stateful functions-as-a-service," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, p. 2438–2452, Aug. 2020. [Online]. Available: <http://dx.doi.org/10.14778/3407790.3407836>
- [4] T. Subramanya, D. Harutyunyan, and R. Riggio, "Machine learning-driven service function chain placement and scaling in mec-enabled 5g networks," *Computer Networks*, vol. 166, p. 106980, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128619310254>
- [5] T. Subramanya and R. Riggio, "Machine learning-driven scaling and placement of virtual network functions at the network edges," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 414–422.
- [6] D. M. Manias, M. Jammal, H. Hawilo, A. Shami, P. Heidari, A. Larabi, and R. Brunner, "Machine learning for performance-aware virtual network function placement," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.
- [7] R. Rodrigues Filho and B. Porter, "Hatch: Self-distributing systems for data centers," *Future Generation Computer Systems*, vol. 132, pp. 80–92, 2022.
- [8] A. Akhter, M. Fragkoulis, and A. Katsifodimos, "Stateful functions as a service in action," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 1890–1893, aug. 2019. [Online]. Available: <https://doi.org/10.14778/3352063.3352092>
- [9] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos, "Distributed transactions on serverless stateful functions," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 31–42. [Online]. Available: <https://doi.org/10.1145/3465480.3466920>
- [10] Z. Zhang, L. Ma, K. K. Leung, L. Tassiulas, and J. Tucker, "Q-placement: Reinforcement-learning-based service placement in software-defined networks," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1527–1532.
- [11] R. R. Filho and B. Porter, "Autonomous state-management support in distributed self-adaptive systems," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, 2020, pp. 176–181.
- [12] R. R. Filho, R. S. Dias, J. Seródio, B. Porter, F. M. Costa, E. Borin, and L. F. Bittencourt, "A self-distributing system framework for the computing continuum," in *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, 2023, pp. 1–10.
- [13] B. Porter and R. R. Filho, "A programming language for sound self-adaptive systems," in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2021, pp. 145–150.
- [14] B. Porter, "Runtime modularity in complex structures: a component model for fine grained runtime adaptation," in *Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*, ser. CBSE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 29–34. [Online]. Available: <https://doi.org/10.1145/2602458.2602471>
- [15] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002. [Online]. Available: <https://doi.org/10.1023/A:1013689704352>
- [16] W. Chu, L. Li, L. Reyzin, and R. Schapire, "Contextual bandits with linear payoff functions," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. JMLR Workshop and Conference Proceedings*, 2011, pp. 208–214.
- [17] S. Ontanón, "Combinatorial multi-armed bandits for real-time strategy games," *Journal of Artificial Intelligence Research*, vol. 58, pp. 665–702, 2017.