Preprint. Cite as: *Costa, B. P.; Da Silva, H. H.; Morales, A. S.; Bittencourt, L. F.; Panisson, A. R.; Rodrigues-Filho, R. "A Multi-agent Approach to Self-Distributing Systems" In: The 39th International Conference on Advanced Information Networking and Applications (AINA), 2025.* http://dx.doi.org/10.1007/978-3-031-87766-7_12

A Multi-agent Approach to Self-Distributing Systems

Bernardo Pandolfi Costa, Heitor Henrique da Silva, Analucia S. Morales, Luiz F. Bittencourt, Alison R. Panisson, and Roberto Rodrigues-Filho

Abstract As the computing continuum increasingly becomes the default deployment infrastructure for modern systems, the demand for a programming model capable of meeting the requirements for developing software systems on such infrastructures also grows. This model must provide abstractions to manage the high level of dynamism inherent in these environments, particularly addressing the autonomous management of stateful service placement across large-scale edge-cloud continuum infrastructures. To address this issue, we explore the concept of self-distributing systems – a machine-centric approach to deal with the complexity of designing distributed systems. We take a step further and have the system decide on distributed design decisions at runtime as unexpected changes and events occur, leaving the system responsible for reacting quickly and accurately as a response to such changes. This paper presents the application of a multi-agent learning approach to learn how to distribute stateful services across the continuum. We demonstrate the efficiency of such a method in a local testbed. We compare our results against a multi-armed bandits approach, pinpointing the strengths and weaknesses of the two approaches.

1 Introduction

The design and management of distributed systems have been the focus of research on systems communities for many years [3]. Deciding, at design time, how the different modules that compose systems should interact in order to maintain high-

Bernardo Pandolfi Costa · Heitor Henrique da Silva · Analucia S. Morales · Alison R. Panisson · Roberto Rodrigues-Filho
Federal University of Santa Catarina, Araranguá, Santa Catarina, Brazil, e-mail: {bernardo.pandolfi.costa,heitor.h.silva}@grad.ufsc.br,{analucia.morales,alison.panisson,roberto.filho}@ufsc.br

Luiz F. Bittencourt
University of Campinas, Campinas, São Paulo, Brazil e-mail: bit@ic.unicamp.br

performing, reliable, and dependable systems is challenging [21]. This scenario becomes increasingly worse when considering the high levels of volatility we encounter nowadays in the edge-cloud continuum [1], requiring new ideas for handling the new levels of complexity [4].

In this context, the concept of Self-distributing Systems (SDS) [17] has emerged to take on the challenges of pushing the responsibility of assembling distributed systems to the system itself at runtime, as opposed to what is currently done in the industry, where a group of engineers is responsible for carefully considering how the system should operate in many distinct and often unexpected conditions at design time, while at runtime engineers monitor the system to detect changes and manually adapt the system to react accordingly. The development of SDS is not straightforward. Developers are envisioned creating a local version of the software system using small and highly reusable software components. At runtime, SDS experiments with relocating and replicating these components throughout a distributed infrastructure while evaluating distributed architectures in terms of performance. The previous version of SDS [17, 16] explored baseline online greedy and multi-armed bandit approaches, tackling various challenges.

In this paper, we propose and investigate a multi-agent approach for learning distributed compositions for SDS at runtime, aiming to compare performance and address the limitations of multi-armed bandits algorithms when tackling SDS learning problems. The contributions are two-fold: (i) introducing a multi-agent approach to address the learning problem inherent in SDS; and (ii) comparing its performance and limitations with an existing multi-armed bandits solution, providing insights into their effectiveness.

2 Learning in Self-distributing Systems

One of the most challenging aspects of SDS is related to the dynamics of the environment in which constant changes occur. Thus, it is necessary to develop approaches to learn how to split and distribute local components to remote machines (i.e., learn how to make distributed design decisions at runtime) to improve the system's performance when such changes occur.

To define the learning problem we face when designing SDS, we must first consider a system running on a certain operating environment. The operating environment is defined by metrics of computing resources (e.g., CPU, memory, storage) from the infrastructure running the system plus the incoming request pattern characteristics that impact how the system performs. Depending on the characteristics of the operating environment, configuring the system into different compositions has a different impact on the system's performance. Thus, the learning problem consists of starting a single or a set of learning agents (depending on the applied learning strategy) with no predefined information or pre-training and, at runtime, locating which system's composition yields optimal performance for different operating environments. The definition of such a problem is found in the literature in approaches

named Emergent Software Systems (ESS) [13, 18], which share the same characteristics and challenges for the learning problem. To tackle this learning problem, many of the explored ESS approaches employed a technique known as multi-armed bandits, which yielded promising results in this context [14].

An important aspect of the learning problem is the operating environment, which plays a central role in the runtime self-optimization of systems and presents numerous challenges. If we assume a static operating environment and a single agent, then a learning agent can explore different compositions and will have basis for comparison, considering it is comparing two distinct compositions in the same operating environment. Similarly, if the operating environment changes during the agent exploration phase, the agent will compare two compositions in different environments, making it challenging to determine which composition perform better because the comparison basis becomes invalid. In certain situations, even small changes in the operating environment result in significant differences in performance across different compositions.

Learning what system composition is best for different operating environments in scenarios where the system is exposed to unknown or unexpected environments is a difficult task. Determining the system's operating environment while learning the optimal composition is challenging, primarily because measuring environmental metrics through a specific system composition can distort the results. For example, the implementation of an algorithm may have a larger memory footprint than a different equivalent implementation. Thus, measuring memory usage to characterize the environment may alter the system's perception of which environment it is running, even if nothing else in the environment changes (e.g., the incoming input pattern and available computing resources remains unchanged).

We define *environment* as the set of computing resources' metrics and the system's input (i.e., workload) characteristics to which the system is subjected. Note that the system's composition that has had the best performance in an environment will continue to do so when that environment is encountered again in the future. Thus, identifying and 'remembering' environments allows the system to simply switch to the best-performing composition without having to explore.

Fig. 1 illustrates two scenarios showing the environment changing. Fig. 1 (a) shows a gradual and continuing change in the environment metrics, slowly impact-

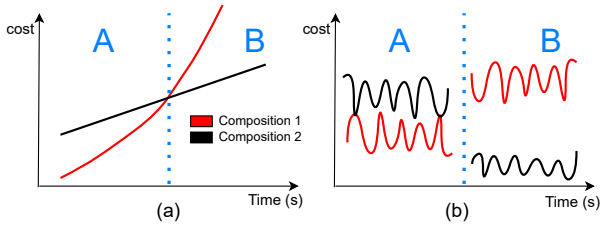

Fig. 1 Two scenarios showing environment changing: (a) shows an environment where their metrics continually and gradually change until the environment is completely changed; (b) shows two distinct environments where changes in their metrics are abrupt and instant.

ing the performance of the systems compositions up to the point where $Composition1$ starts performing worse than $Composition2$. Note that the y-axis is the cost (in this case, we want to minimize cost – i.e., $cost = \frac{1}{reward}$). At that point, represented by the blue dotted line, we consider the system is being exposed to a new operating environment. Fig. 1 (b) shows an environment that maintains its metrics constant until it suddenly changes to different values, completely changing the environment.

Rappa et al. [16] applies UCB1 to implement SDS in scenarios similar to the one depicted in Fig. 1 (b), not considering the environment change mid-execution. In their work, Rappa et al. [16] show that the application of off-the-shelf UCB1 (i.e., with no changes to the original algorithm) makes correct decisions in SDS executing in static environments, but cannot handle scenarios where the environment gradually changes over time, like the one shown in Fig. 1 (a). In this paper, however, we explore a multi-agent approach as an alternative solution for learning in SDS.

3 Multi-agent Approach

3.1 Learning Strategy

To address the challenges of dynamic environments in SDS, we propose a multi-agent approach that enables real-time learning and adaptation. Our method involves deploying multiple learning agents without predefined information or pre-training, allowing the system to continuously learn the dynamics of the operating environment and determine the optimal system compositions as conditions change. In our approach, each agent within the MAS is responsible for one possible system composition and is tasked with evaluating the performance of that particular composition when it is assigned to the system. Consequently, agents need to communicate to exchange metrics for their respective compositions and agree on the best composition. We opted for a one-agent-per-composition approach, as the number of compositions is known, in contrast to the uncertain number of different environments.

Considering the system starts without any predefined information or pre-training, similar to the UCB algorithm described in [2], our approach requires an exploration phase during which the system evaluates each composition and collects performance metrics. This means each agent assigns its respective composition to the system, collects metrics for a predefined period of time, and all agents take turns assigning their respective compositions during the exploration phase. In particular, in this section, we focus on metrics related to the average response time and how the whole system, i.e., the MAS, addresses the learning problem in SDS. Later, in Section 3.2, we describe how the proposed approach is realized, considering the distributed nature of MAS, using consensus mechanisms. After the exploration phase, the system has access to the composition metrics.

To introduce the parameters and equations that govern the system's learning aspects, we use the following notation:

- **Average Time** (Avg_i^t): The observed average response time for composition i at time t, corresponding to one observation interval from $t-1$ to t.
- **Number of Observations** ($\mathcal{N}$): The total number of observations.
- **Penalty** (ρ): A dynamic learning parameter representing a penalty received when a system composition perspective of performance appears inferior to other compositions.
- **Performance Metric** (v): The resulting performance metric, indicating the performance of a composition relative to all other considered compositions.
- **Sensitivity** (α): A learning parameter indicating how sensitive the system is to changes in response times.

Each time a composition is assigned to the system, its corresponding agent i has access to the metric of average response time, Avg_i^t. Further, agents can infer their respective composition's performance over time concerning the average response time when the same composition, refereed to as its corresponding agent i, is assigned to the system multiple times.

$$\Delta_i^t = \frac{1}{\mathcal{N}-1} \sum_{k=1}^{\mathcal{N}} \left(Avg_i^{(t-k-1)} - Avg_i^{(t-k)} \right) \tag{1}$$

Equation (1) implements this inference, where Δ_i^t is the average of differences between multiple observations of the average response time for composition i at time t. The summation runs from $k=1$ to $\mathcal{N}$, summing the differences $(Avg_i^{(t-k-1)} - Avg_i^{(t-k)})$. The division by $\mathcal{N}-1$ accounts for the number of differences being averaged. The basic idea behind this equation is that it enables each agent to understand how the environment evolves, i.e., the scenarios illustrated in Fig. 1. Later, the system can infer which composition has the best perspective of performance at time t, by comparing Δ^t of all compositions. This is determined by finding the minimum value among the Δ_i^t values for all compositions i. $\Delta^t MIN$ indicates the best perspective of performance among all compositions at time t and it is calculated as follows: $\Delta^t MIN = \min(\Delta_i^t, \ldots, \Delta_n^t)$.

However, even with a better perspective on performance, that composition may only present better performance in the distant future; i.e., it presents a better perspective on performance, but currently, it has a worse average response time. This scenario is illustrated in Fig. 1 (a), where $Composition2$ has a better performance perspective but only begins to outperform $Composition1$ after one event (blue dotted line). Thus, it is worthy to understand how the different perspectives of performance of all compositions are distributed to make informed decisions regarding system optimization and composition selection.

$$\rho_i^t = \left| \frac{\Delta_i^t - \Delta^t MIN}{\left(\frac{\Delta_i^t + \Delta^t MIN}{2} \right)} \right| \tag{2}$$

Equation (2) calculates the relative difference between the perspective of performance of composition i at time t, Δ_i^t, and the best perspective of performance

among all compositions, $\Delta^t MIN$. It provides insights into how much the composition i's perspective of performance deviates from the best perspective of performance among all compositions, as given by ρ_i^t. A higher value of ρ_i^t indicates that the composition i is further away from the best perspective of performance. Then, ρ_i^t and the learning sensitivity parameter α are used to calculate an indicator of the performance of each composition, weighting its current average response time, Avg_i^t, by penalizing it according to how much its perspective of performance deviates from the best perspective among all compositions. Equation (3) provides this performance metric v_i^t for composition i at time t.

$$v_i^t = Avg_i^t + (Avg_i^t \cdot \rho_i^t \cdot \alpha) \tag{3}$$

Then, the system can determine the best composition at the current time by looking for the minimum value of v^t. However, v^t and $\Delta^t MIN$ only can be calculated when agents share their private metrics, i.e., Δ_i and v_i, considering each agent collects metrics for its composition. Thus, agents will share their private metrics to reach a consensus about $\Delta^t MIN$ and v^t, and consequently understand which composition is the best. The next section introduces the consensus mechanism used.

3.2 Multi-agent Consensus Mechanism

Given the distributed nature of MAS, consensus mechanisms are the basis for coordinated control of the system [10]. Consensus mechanisms are defined by information flow or protocols, which specify how and when agents exchange information in order to reach an agreement regarding a certain piece of information [6, 12].

Definition 1 (Consensus). Given a group of agents Π, and given a subset of the agents' belief base s, i.e., $s_i \subseteq \mathscr{B}_i$ for agent i, a consensus over s is defined as $\lim_{t\to\infty} |s_i^t - s_j^t| = 0$ for all pairs of agents i and $j \in \Pi$, meaning that the state of the subset s in the belief base of all agents in Π converges to common values as time progresses.

The literature shows that agents can reach consensus through constant communication in both discrete-time and continuous-time MAS. However, this approach is resource-consuming, requiring significant energy and network bandwidth [10]. To avoid excessive and sometimes unnecessary use of system resources, the consensus problem in MAS is commonly approached using event-based consensus mechanisms [15], also known as consensus based on triggering mechanisms [10]. Event-based consensus mechanisms reduce the use of a MAS's resources by minimizing the information exchanged between agents through a strategy where events trigger data transmission and control updates [15]. Additionally, when periodic consensus checks are needed, time-triggered events are typically used [15].

We follow this literature, in which an event e is associated with a set of agents' beliefs s. When the event occurs, i.e., $\langle e, s\rangle \in \mathscr{E}_i$, it triggers a consensus mechanism, in which agents exchange information to reach an agreement about the beliefs in s.

The consensus mechanism is implemented according to the following belief update and protocol:

1. An event $\langle e,s \rangle$ occurs, triggering the consensus mechanism. It may be an event triggered by receiving a message from one agent of the group, or triggered by a clock, for example. The protocol goes to (2).
2. All agents in the group share their own values for the beliefs in s, communicating it to all other agents of the group. The protocol goes to (3).
3. All agents need to update the beliefs with the received information according to some common belief update function, i.e., $\mathscr{B}_i' = update(\mathscr{B}_i, \{s_j, \ldots, s_n\})$ for all agents in the group $\{i, j, \ldots, n\}$. The protocol goes to (4).
4. All agents reach a consensus about s, according to Definition 1.

There are two essential information from Section 3.1 that agents require to reach a consensus in order to make decisions about which system composition should be used: $\Delta^t MIN$ and v^t. To reach a consensus about $\Delta^t MIN$ and v^t, we used the same event-triggering consensus strategy. When an agent i perceives a significant change in either Δ_i^t or v_i^t (the event), it sends a message to all agents informing them of the new value. Subsequently, the system executes an exploration phase, and then all agents share their new metrics. For v^t, they compare the received v_n^t values with their own v_i^t values to determine the best composition for the system at that time according to equation (4).

$$v^t = \min(v_i^t, \ldots, v_n^t) \tag{4}$$

This unified strategy ensures that all agents reach a consensus for both parameters. Thus, when an agent i infers that its corresponding composition is the best, i.e., v^t is equal to v_i^t, it assigns the composition for the system.

4 Evaluation

4.1 Self-distributing Web Service

In our experiments, we use a self-distributing web service that implements two main HTTP functions: one that retrieves data from a list and another that adds data to a list. The service also implements a prime function that is activated when clients request data to be retrieved. This is to simulate a CPU-intensive function for data retrieval. The prime function works according to the following equation $\sum_{i=1}^{n} \pi(i \cdot k)$. The term π represents a prime verification function; given a number, the function verifies whether that number is prime or not. Thus, for every index i in the list, the CPU-intensive function verifies whether $(i \cdot k)$ is prime.

The factor k intensifies how CPU-intensive the function gets, considering the number of items stored in the list. A small k (e.g., $k = 1$) requires a large list to make it very CPU-intensive, whereas a large k quickly increases CPU utilization with a few items stored in the list.

For the experiments, there is a single distribution proxy for this web application. Thus, the SDS framework comprises two distinct system compositions: one that executes locally (named local composition) and another that splits the list into two shards and places them in two external machines (sharding composition). Furthermore, two client workload generators produce different workload patterns for the application. The first client generates a set of requests to continually fetch data, whereas the second client continually fetches data while adding items to the list every step (i.e., every $\approx$ 10s).

Our multi-agent approach was implemented using the JaCaMo Framework [5], and all code used for the experiments is open source and is freely available on GitHub[1], along with detailed instructions to replicate the results.

4.2 Experiments and Results

We conducted experiments to evaluate our proposed approach by deploying a self-distributing web service in a local cluster. We compared the performance of our multi-agent approach with the multi-armed bandits' algorithm to investigate their effectiveness and limitations in dynamic operating environments. Our experimental setup consists of three machines. The first machine hosts the SDS framework with the web service and learning module, using either the multi-agent approach or the UCB1 algorithm. The second and third machines run the SDS framework in separate process. The Distributor module waits for the list's shards from the web application when the system chooses the sharding composition.

The experiments involve exposing the SDS to three different environments defined by the client's workload and the k factor that determines the CPU usage by the service. In the first scenario, k is set to 2 (low CPU usage), and the workload consists of a series of requests to retrieve data from the service's list. Fig. 2 shows the results of subjecting SDS to the first scenario. In both graphs, the local composition

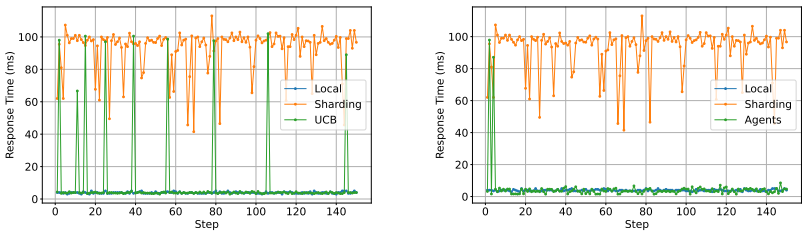

Fig. 2 SDS (green) converging towards the best-performing composition, and the performance of two fixed compositions: local (blue) and sharding (orange). (left) SDS with UCB1; (right) SDS with multi-agent approach. Scenario: $k = 2$, fixed list size = 2.

[1] Repository: `https://github.com/heitorhenriques/mas_ucb1`

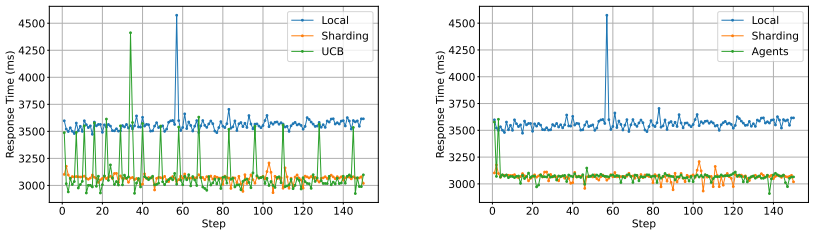

Fig. 3 SDS (green) converging towards the best-performing composition, and the performance of two fixed compositions: local (blue) and sharding (orange). (left) SDS with UCB1; (right) SDS with multi-agent approach. Scenario: $k = 8$, fixed list size = 38.

clearly outperforms the sharding composition, as indicated by its lower response time. Moreover, the SDS converged to the correct composition. However, UCB1 presented some spikes due to its occasional exploration, whereas the multi-agent approach, which detects the trend of the response time curve, converges and never returns to exploration for the rest of the experiment.

Fig. 3 shows the results of exposing SDS to a scenario with high CPU usage, with $k = 8$ and a static list size of 38 items. The client's workload consists of sequential requests to retrieve data from the list. Intuitively, in situations where CPU becomes a bottleneck, load balancing the incoming workload amongst replicas of the server would allow the system to scale. Therefore, a composition that balances the load of incoming requests is preferable. This expected result is confirmed in our experiments (Fig. 3, as the sharding composition presents lower response time than the local composition. Similarly to previous results, both learning strategies converges towards the best composition. However, the proposed multi-agent approach converges faster, with no exploitation time after convergence, resulting a better overall average response time than the UCB1 solution.

Finally, we expose the SDS to a scenario where $k = 2$ (low CPU usage) but the list size gradually increases in size by one element every step. As the list size increases, the demand for CPU also gradually increases. This creates an interesting effect on

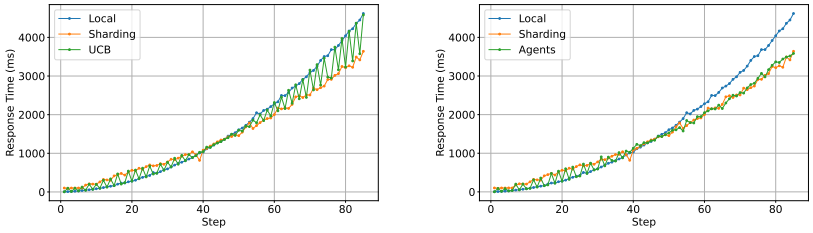

Fig. 4 SDS (green) converging towards the best-performing composition, and two fixed compositions: local (blue) and sharding (orange). (left) SDS with UCB1; (right) SDS with multi-agent approach. Scenario: $k = 2$, list size increases by 1 every step.

the systems perception of the environment. Initially, the CPU demand is low and the environment is similar to the one in the first scenario, where the local composition has the lowest response time. As the list increases and the CPU demand increases, the environment transitions to a state of high CPU demand, similar to the second scenario.

In situations where the environment gradually changes, we observe in Fig. 4 that UCB1 never converges. This has been reported in the literature by Rappa et al. [16]. The reason is that the gradual change in the environment constantly worsens the measured cost, forcing the UCB1 to keep checking the cost to determine the best option. Since the cost of all arms continually deteriorates, the algorithm never converges. In our approach, on the other hand, when the cost are too small and the curve is too steep (curve trend from step 0 to 40), our approach struggles to identify the trend and keeps sampling the system's cost in both compositions. Once the system reaches a certain cost value, our approach manages to identify the trend and converges towards the best composition (step 40 onwards). In this scenario, our multi-agent approach also outperforms UCB1 and has its average performance closer to the best composition.

5 Related Work

Multi-Armed Bandit (MAB) problems have become a powerful tool for optimizing operations in distributed systems where dynamic decisions are required in uncertain environments. For example, [8] demonstrates that MAB solvers enable adaptive power allocation, optimizing resource allocation, and maintaining robust communication links between devices. In [11], MAB is used for adaptive configuration optimization of recurring data-intensive applications in cloud computing, improving performance and reducing costs. In [20], MAB-based runtime adaptation of cache replacement policies optimizes performance by adjusting to changing access patterns in content management systems. In [19], MAB and deep reinforcement learning are used for adaptive caching in hierarchical content delivery networks, enhancing content delivery efficiency.

MAS are recognized for their self-adaptive and self-distributing capabilities, making them ideal for implementing control mechanisms in self-distributing systems. Similar approaches have been explored in related literature. For example, in [7], an approach using experience exchange among agents is proposed to dynamically influence environmental behavior patterns. In [9], a framework is introduced for self-adaptive management, integrating organizational aspects and cooperative behaviors of agents to automate management actions across distributed environments. Additionally, insights in [22] highlight how MAS manage complexity in self-adaptive systems, distinguishing MAS from other technologies.

To the best of our knowledge, our work is the first to introduce a multi-agent approach for the Learning module in Self-distributing Systems.

6 Conclusion

We have presented a multi-agent approach to realizing the concept of Self-Distributing Systems. These systems are designed to execute on a single process, but at runtime, they autonomously experiment and decide on distributed composition for the system to maximize performance.

There are many benefits to using a MAS approach, with the most important being its ability to enhance adaptability at the control level. MAS can dynamically adapt to changing environments by leveraging the collective intelligence and learning capabilities of individual agents. This adaptability ensures that the system can quickly respond to environmental changes, maintaining optimal performance even under varying conditions. Although this paper focuses on a simple instance of the agents' learning capabilities, our approach supports the integration of specialized learning components for individual agents. These specialized capabilities can later be combined through a consensus mechanism to form a collective intelligence.

We evaluated our approach in a real small-scale testbed composed of 3-nodes connected in a local network. For our experiment, we explored the self-distribution capabilities of a web service. We exposed the SDS to three distinct scenarios and compared the UCB1 algorithm against our proposed multi-agent approach, which is a different learning strategy. Based on our experiments, we conclude that our approach generally outperforms UCB1 in the selected scenarios, converging in cases where UCB1 continues to explore. This indicates that approaches that observe and predict curve trends have an advantage over approaches based on confidence levels. On the other hand, in scenarios where the cost function of multiple compositions is too close, UCB1 might perform better by continually sampling costs until confident about the composition with the lower cost. In such cases, our approach, which samples each composition only once, could risk converging to a suboptimal solution when costs are too similar, and the signal is noisy.

Future work will focus on conducting larger-scale experiments with an extensive network of nodes to validate the robustness and scalability of our multi-agent approach across diverse environments. Additionally, we aim to integrate different learning techniques to enhance the adaptability and decision-making capabilities of the agents while comparing them with our current proposal. Another direction involves exploring hybrid models that combine our approach with existing algorithms, leveraging their strengths.

Acknowledgement

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. Al-Dulaimy, A., Jansen, M., Johansson, B., Trivedi, A., Iosup, A., Ashjaei, M., Galletta, A., Kimovski, D., Prodan, R., Tserpes, K., Kousiouris, G., Giannakos, C., Brandic, I., Ali, N., Bondi, A.B., Papadopoulos, A.V.: The computing continuum: From iot to the cloud. Internet of Things **27**, 101,272 (2024).
2. Auer, P.: Using confidence bounds for exploitation-exploration trade-offs. Journal of Machine Learning Research **3**(Nov), 397–422 (2002).
3. Bal, H.E.: The shared data-object model as a paradigm for programming distributed systems. Ph.D. thesis, Vrije Universiteit, Amsterdam (1989).
4. Blair, G.: Complex distributed systems: The need for fresh perspectives. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1410–1421 (2018).
5. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Science of Computer Programming **78**(6), 747–761 (2013).
6. DeGroot, M.H.: Reaching a consensus. Journal of the American Statistical association **69**(345), 118–121 (1974).
7. Jiao, W., Sun, Y.: Self-adaptation of multi-agent systems in dynamic environments based on experience exchanges. Journal of Systems and Software **122**, 165–179 (2016).
8. Khan, M.I., Alam, M.M., Le Moullec, Y.: A multi-armed bandit solver method for adaptive power allocation in device-to-device communication. Procedia computer science **130**, 1069–1076 (2018).
9. Lavinal, E., Desprats, T., Raynaud, Y.: A multi-agent self-adaptative management framework. International Journal of Network Management **19**(3), 217–235 (2009).
10. Li, Y., Tan, C.: A survey of the consensus for multi-agent systems. Systems Science & Control Engineering **7**(1), 468–482 (2019).
11. Liu, Y., Xu, H., Lau, W.C.: Accordia: Adaptive cloud configuration optimization for recurring data-intensive applications. pp. 831–841 (2020).
12. Olfati-Saber, R., Murray, R.M.: Consensus problems in networks of agents with switching topology and time-delays. IEEE Transactions on automatic control **49**(9), 1520–1533 (2004).
13. Porter, B., Faulkner Rainford, P., Rodrigues-Filho, R.: Self-designing software. Communications of the ACM **68**(1), 50–59 (2024).
14. Porter, B., Grieves, M., Rodrigues Filho, R., Leslie, D.: {REX}: A development platform and online learning approach for runtime emergent software systems. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 333–348 (2016).
15. Qin, J., Ma, Q., Shi, Y., Wang, L.: Recent advances in consensus of multi-agent systems: A brief survey. IEEE Transactions on Industrial Electronics **64**(6), 4972–4983 (2016).
16. Rappa, F.M., Rodrigues-Filho, R., Panisson, A.R., Soriano Marcolino, L., Bittencourt, L.: Multi-armed bandits for self-distributing stateful services across networking infrastructures. In: NOMS 2024 IEEE/IFIP Network Operations and Management Symposium (2024).
17. Rodrigues Filho, R., Dias, R.S., Seródio, J., Porter, B., Costa, F.M., Borin, E., Bittencourt, L.F.: A self-distributing system framework for the computing continuum. In: 32nd International Conference on Computer Communications and Networks (ICCCN), pp. 1–10. IEEE (2023).
18. Rodrigues Filho, R., Porter, B.: Defining emergent software using continuous self-assembly, perception, and learning. ACM Transactions Autonomic Adaptive Systems **12**(3), 1-25 (2017).
19. Sadeghi, A., Wang, G., Giannakis, G.B.: Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. IEEE Transactions on Cognitive Communications and Networking **5**, 1024–1033 (2019).
20. Sridharan, S., Gaonkar, S., Fortes, J.A.B.: Bandit-based run-time adaptation of cache replacement policies in content management systems. pp. 79–88 (2020).
21. Tanenbaum, A.S., Van Steen, M.: Distributed systems principles and paradigms. Prentice Hall PTR, Upper Saddle River, NJ, USA (2002).
22. Weyns, D., Georgeff, M.: Self-adaptation using multiagent systems. IEEE software **27**(1), 86–91 (2009).