Exploiting the Potential of the Edge-Cloud Continuum with Self-distributing Systems

Roberto Rodrigues Filho, Luiz F. Bittencourt
University of Campinas
Campinas-SP, Brazil
{robertor, bit}@ic.unicamp.br

Barry Porter
Lancaster University
Lancaster, UK
b.f.porter@lancaster.ac.uk

Fábio M. Costa
Federal University of Goiás
Goiânia-GO, Brazil
fmc@inf.ufg.br

***Abstract*—The Edge-Cloud Continuum offers a wide range of adaptive deployment settings for modern applications. However, in order to exploit the full potential of the edge-cloud infrastructure and platforms, applications have to be carefully crafted to be stateless and self-contained in small services or functions, *i.e.*, the opposite of the classic stateful monolithic applications. In this paper, we explore an alternative approach that allows stateful single applications to also exploit the full potential of the edge-cloud continuum. We explore the concept of *Self-distributing Systems* (SDS) as a general approach for code offloading and as an elastic application-level mechanism for performance scale-out on the edge-cloud continuum. Our preliminary results indicate that SDS enables enough flexibility for applications to fully explore the edge-cloud resource mixture. Particularly, we describe our state management strategies for stateful code mobility; explore SDS as a general mechanism to exploit horizontal scaling on the cloud; and examine SDS as a general code offloading mechanism to move code from edge to cloud, showing the scenarios where our approach enables applications to positively exploit the edge-cloud continuum for better performance.**

***Index Terms*—self-distributing systems, state management, edge-cloud continuum**

I. Introduction

Blair [1] argues that volatility is one of the main causes of complexity in modern distributed systems. The constant changes that occur in a system's operating environment, in every layer (*i.e.*, infrastructure, platform and application/services), require systems that are equipped to adapt themselves to accommodate new operating conditions as and when they occur. A wide range of technologies have been design to help build adaptable and flexible infrastructures and platforms to handle the volatility issue.

Technologies such as cloud computing and edge computing offer on-demand provision of computing resources and user-proximate service execution respectively, allowing a wide range of deployment variants. At the platform level, containers (*e.g.*, Docker, containerd) and container-orchestrator technologies (*e.g.*, Kubernetes, Mesos) allow self-contained services to be deployed, replicated and moved across the infrastructure. These technologies enable a wide range of flexible deployment settings that could be drastically changed to accommodate new operating conditions at runtime for all sorts of services and applications (*e.g.*, autonomous vehicles [2], deep learning [3], augmented reality [4], etc.).

These infrastructure and platform-level technologies, however, require the application to be carefully crafted. The main barriers in traditional software development that prevent exploitation of these cloud technologies are i) monolithic applications, and ii) stateful applications. Monolithic applications are large and tightly-coupled pieces of software that are difficult to migrate and replicate unless they are carefully designed for that purpose. Stateful applications are a further challenge to the migration and replication of compute as the state has to be properly managed for assured consistency.

In response to those issues, new architectural styles for developing applications have been widely adopted, particularly around Microservices [5] and Serverless Computing [6]. These architectures consist of a collection of *stateless*, self-contained, highly reusable *small* services/functions, which allow the full exploitation of adaptive platforms and infrastructures in the cloud-edge continuum. These approaches provide architectural guidelines of how to develop cloud-capable systems, but still require developers to carefully craft (or refactor) systems to match this architectural style. Furthermore, these approaches require the strict separation of state from services/functions, where all application state is typically pushed into a database, and also require up-front decision-making on how large or small each microservice or function should be.

In this paper, we explore an alternative approach to exploit the full potential of the edge-cloud continuum, in which parts of regular local applications can be automatically distributed and scaled out. We explore and expand the Self-distributing Systems (SDS) concept [7], [8] to allow the development of local, stateful applications that are replicated or moved across a distributed infrastructure at runtime, facilitating horizontal scaling and code offloading on the edge-cloud continuum. SDS employs a lightweight component model (*e.g.*, Dana [9], OpenCom [10], etc.) to create componentised local applications that are able to replace components at runtime to change their behaviour. In detail, locally-running components are swapped for proxy components that implement an RPC layer to forward function calls to a relocated remote component. By relying on the relocation mechanism enabled by SDS, we construct general applications that relocate parts of itself across any infrastructure. This paper's main contributions are:

i) State management strategies for transparently relocating stateful components to remote hosts;

ii) SDS extension for performing stateful code mobility and horizontal scale out on the edge-cloud continuum.

Our results[1] demonstrate that the concept of SDS with transparent state management enables sufficient flexibility for componentised local applications to allow them to fully explore the potential of the edge-cloud continuum. Particularly, we explore the advantages and limitations of transparent state management strategies; we also examine the transparent relocation and *replication* of software components to scale web-based applications on the cloud; and finally, we experiment with stateful component relocation as an alternative to classic code offloading [11] from edge to cloud.

II. Background

This section presents the two main concepts on which our approach was built. We first present the concept of component-based models, focusing the discussion on the adaptation algorithm implemented by Dana [9], which is key to realise our proposed transparent state management framework. Then, we visit the concept of Self-distributing Systems [7], which we expand to add a state management framework to explore the self-distribution of stateful components.

A. Software Adaptation on Component-based Models

The component-based model is a programming model that allows software systems to be built as a result of the composition of small, reusable, and swappable components. The most relevant examples of such models are OpenCom [10], OSGI[2], and most recently, Dana [9].

A component-based software system is composed of small components that are connected following the required-provided interface policy, *i.e.,* given a specific interface I, a component that *provides* interface I implements its functions, whereas the components that *require* interface I use its defined functions to implement their own. The component that provides interface I is connected to the component that requires I. Furthermore, the component-based runtime enables the replacement of a component that provides interface I with another that also provides I. By enabling this component replacement, software written using such models allows the system to change its behaviour at runtime.

Note that some of the components hold state, *i.e.,* a set of global variables defined in the component. The component-based model runtime provides mechanisms that support seamless (*i.e.*, no software downtime) adaptation of stateful components. Fig. 1 illustrates the three main steps of the algorithm that performs adaptation in Dana. The first part of Fig. 1 shows component $CompA$ connected to $CompB$ through some interface I. Following the component-based model, $CompA$ requires interface I to work, and thus, any component that implements I can be connected to $CompA$ to satisfy its dependency. In the described example, both $CompB$ and $CompC$ implement I, allowing either one of these components to be connected to $CompA$.

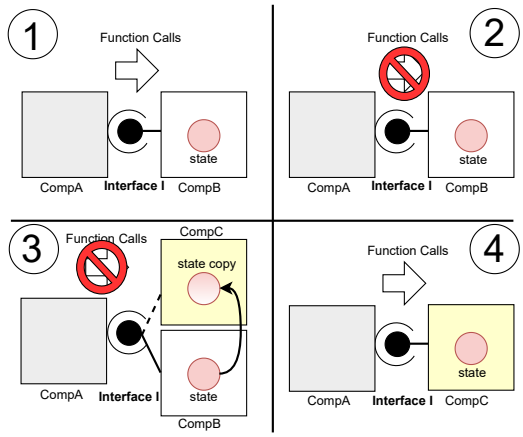

Fig. 1. The main stages of swapping a stateful component executing on a local process: (1) the initial system with its components and state; (2) pause incoming function calls; (3) copy state into the newly instantiated component; (4) connect the new component and resume incoming function calls.

Fig. 1 shows the replacement of the stateful component $CompB$ that satisfies $CompA$ dependency to $CompC$, a different stateful component that also implements interface I and also satisfies $CompA$. The second part of Fig. 1 (step 2) shows the first step of the adaptation algorithm, which consists of stopping function calls from $CompA$ to $CompB$. This step is crucial when the component to be replaced holds state. Pausing incoming function calls guarantees that the state remains unchanged during the adaptation process. In step 3, the new component is loaded to memory, and a copy of $CompB$'s state is transferred to $CompC$. Finally, in the final step (step 4), $CompA$ is connected to $CompC$, and the incoming function calls are resumed and handled by $CompC$.

The adaptation algorithm is a generic algorithm employed by different models that enables the adaptation of stateful components. The Dana model, in particular, offers a way to determine the state that is shared among different component variants (*i.e.,* different components that implement the same interface). Dana offers the $transfer$ keyword used to define global variables in the **interface**. Thus, every component that implements said interface inherits all variables and their types, ensuring that all component variants hold the same set of variables as their local state. This facilitates the identification of the state in a component and the action of transferring state between component variants during adaptation, since they all hold the same set of variables of the same exact type.

B. Self-distributing Systems

Self-distributing Systems (SDS) [7] enable local software, *i.e.*, applications built to execute on a single process, developed using a component-based model to relocate and replicate local components at runtime. Employing Reinforcement Learning (RL) [12] algorithms, these systems are able to learn which local components to relocate/replicate across an infrastructure to optimise their performance (*e.g.,* decrease response time) under different operating environments.

[1]Code repository with detailed tutorial to replicate our results: https://github.com/robertovrf/PFGElasticity

[2]OSGI Working Group: https://www.osgi.org/

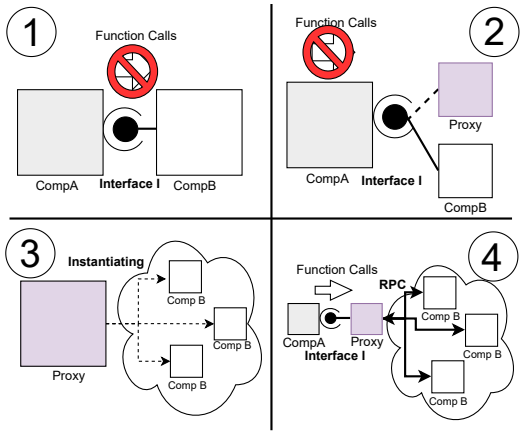

Fig. 2. The main stages of distributing replicas of local components. (1) pause incoming calls; (2) instantiate the distribution proxy component; (3) the proxy instantiates multiple instances of the replicated component; (4) replace component to the proxy and resume incoming function calls.

SDS apply proxy components as the basis for relocating and replicating local components. Leveraging the adaptation mechanisms supported by component-based models (see Sec. II-A), SDS relocate local components by replacing them with proxy components, as illustrated in Fig. 2. The proxy then implements Remote Procedure Calls (RPC), forwarding incoming function calls to the relocated component.

Fig. 2 illustrates the steps to realise the replication of stateless components. The first step of the adaptation algorithm is to stop incoming function calls to the soon-to-be-replaced component (step 1). Then, the proxy component is loaded into the memory (step 2). Besides acting as a regular proxy, the proxy component is also responsible for loading new instances of the relocated component to external processes (step 3). After the relocated instances are loaded, the proxy component is connected to $CompA$, and the application resumes (step 4). Note that the adaptation algorithm treats the proxy component like any other component during adaptation. Before connecting any component to the rest of the application, the adaptation algorithm invokes a special function implemented by the component. This function is often responsible for preparing the component to start execution. Distribution proxy components, however, use this function to create new instances of the relocated component. After the application is resumed, all function calls made to the proxy are forwarded to the relocated component through Remote Procedure Calls (RPC). On the external process, where the relocated component is executing, there is a reverse proxy component that implements the remote end of the RPC, handling incoming function calls, forwarding the calls to the right function, and returning the function return values to the distribution proxy component.

This work presents an extension of the SDS concept. We extend the SDS to enable distribution proxy components to handle relocation and replication of stateful components. We also explore the local component relocation/replication mechanism to exploit code offloading and horizontal scale out in the edge-cloud continuum.

III. Managing State in Self-distributing Systems

This section introduces the state management framework and describes the extension to the adaptation algorithm in the Dana component-based model. It also describes the integration of the extended version of SDS to a container-orchestrator that allows the exploitation of application code mobility and horizontal scale out in the edge-cloud continuum.

A. State Management Approach

In some scenarios, local component distribution across the infrastructure increases the system's performance as incoming calls are spread across multiple instances of the replicated component. For stateful components, however, replication (when not properly handled) leads to state consistency issues and, ultimately, to a system's malfunction. The state management approach aims to provide a framework to support state consistency when replicating stateful components. The framework consists of a collection of distribution proxy components that manage state when replicating stateful components.

The distribution proxy implements four main tasks: i) the creation of the replicated component instances in external processes; ii) transferring of the state from the original local component to the replicas; iii) the implementation of the RPC that forwards incoming function calls to the replicas; and, finally, iv) the algorithm for maintaining state consistency amongst the relocated component replicas. The first two tasks are carried out when the adaptation algorithm is executing. Fig. 3 illustrates the adaptation algorithm for replicating stateful components across infrastructures.

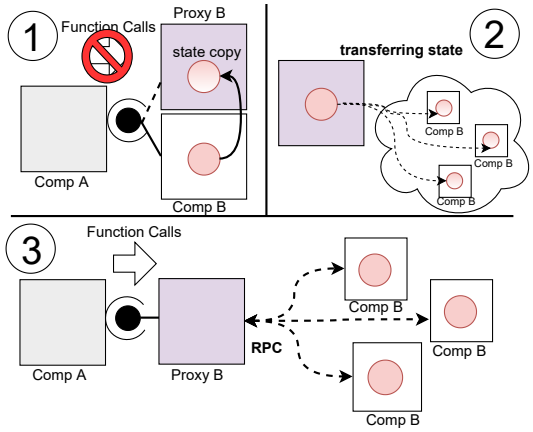

Fig. 3. The steps to replicate stateful components across a distributed infrastructure: (1) copy local state to a proxy component; (2) instantiate and transfer state to newly created remote replicas of the component; and (3) resume incoming function calls.

The adaptation algorithm for replicating stateful components is similar to replacing stateful components with local ones. The first step is to stop function calls to reach the soon-to-be-replaced component, then create an instance of the new component (in this case, a distribution proxy). After creating the proxy instance, a copy of the component state is transferred to the proxy (step 1 in Fig. 3). Up until this point, the adaptation process remained unchanged. After the proxy

receives a copy of the state, the proxy creates the instances of the replicated component in the remote processes (this process is described in Sec. III-B). Once the remote instances are created, the proxy transfers a copy of the state to the remote instances (step 2). Finally, the proxy component is connected to $CompA$, and the application is resumed (step 3).

State transferring is a task performed by the distribution proxy in two situations: when distributing (replicating or relocating) stateful components and when the system is adapting from a distributed composition with replicas of stateful components to a local composition (where all components execute in the same initial process). The first situation is illustrated in Fig. 3. In the second situation, where the application changes from a distributed composition to a local one, the new state that the replicas hold are copied back and merged in the proxy component before copying the state from the proxy to the new component. Besides these two situations, the state can either be entirely copied into the remote instances (full replication) or divided amongst the replicas (sharding), so each replica holds a piece of the original state. These are the two distribution proxy we evaluate in this work (see Sec. IV).

After the application is resumed and the proxy forwards incoming function calls to the replicas, the proxy state executes its state consistency algorithm. A proxy-provided state consistency is essential because the replicated stateful component is not initially designed for replication. The proxy's job is to ensure that a state change in a specific replica is propagated to the other replicas so that the system state remains consistent. As there are no one-solution-fits-all cases for state consistency, different proxy components may implement radically different approaches. We note that there are two concerns all state consistency approaches have to consider: i) the tolerance level for inconsistency the application accepts, and ii) the state's characteristics (data type format, values they hold, etc.).

This paper evaluates two different distribution proxy implementation: full replication and sharding. Our case study is a simple web application that handles requests to add or retrieve numbers from a list. The full replication proxy creates multiple replicas of the list and copies the local list content to all replicas. As new requests are handled by the server, the proxy forwards the incoming requests to all instances of the list maintaining all replicas with the same state. The sharding implementation, on the other hand, splits the list content amongst the available replicas using a hashing algorithm similar to a Distributed Hash Table (DHT). Each list shard has a range of values they host, so that when a number is added or searched, the proxy applies a hash function to the number and knows which replica to forward requests.

B. Container-Orchestration Integration

A crucial part of the adaptation algorithm is the creation of component replicas across an infrastructure. After copying the state to the proxy, the process of creating replicas of the component across the infrastructure begins. Here we describe all the elements involved in creating replicas of components across elastic environments.

The mechanism we explored was developed and explored by Dias *et al.* [13]. The elastic environment is essentially a cluster of virtual machines on a cloud infrastructure managed by a container-orchestrator (in this case, Kubernetes[3]). Kubernetes provides an API service that allows applications to deploy and manage containers life cycle.

Kubernetes manages containers on cloud-based and edge infrastructures (*e.g.,* KubeEdge[4]). In a Kubernetes-managed infrastructure, we can explore horizontal application scaling by creating multiple copies of a container. We can also explore code mobility by transferring a container running on the edge to the cloud. We explore horizontal scale-out and code offloading using Kubernetes at the platform level.

The proxy component interacts with Kubernetes service API to create multiple containers running a Dana-based application and a repository full of the application components on the container's file system. The Dana-based application is a service that essentially implements two functions: i) a function that receives a component's name as a parameter and loads the component and all its dependencies and executes it; and ii) a function that receives a JSON-formatted state, translates it to a Dana data type and inserts it into the loaded component.

Once the containers are running, the proxy sends a request to the Dana application to load the component remotely. After the component is loaded, the proxy sends a copy of the state to be inserted into the component. The proxy component performs the state transferring for every replica created (the number of replicas to create is hard-coded into each proxy).

IV. Evaluation

This section reports the main findings of our preliminary exploration of SDS to exploit the edge-cloud continuum. First, we experiment with the self-distribution of the web application as a generic alternative for code offloading from the edge to the cloud. Then, we use SDS as a mechanism for cloud horizontal scaling. We used both the cloud and a device on the edge. For the cloud, we used a Google Cloud Kubernetes-managed cluster (GKE) with 5 virtual machines. Each VM is equipped with 2 vCPUs and 8GB of memory. For the edge, we used a laptop (1,4 GHz Quad-Core CPU and 8GB of memory) placed on the same local network as the client scripts.

We first evaluate self-distribution as a code offloading alternative for the application's performance enhancement. Fig. 4 shows the results of our first experiment. We evaluate the performance (in terms of response time) of the web application in three configurations. We executed the web application entirely on the edge (blue line); the application's entry-point on the edge and the list component on the cloud (red line); and finally, the self-distributing version of the application, capable of switching at runtime between the edge and cloud.

As part of the experiment, we executed the same client workload for all evaluated configurations. The workload consisted of issuing sequential requests that added items to the

[3]Kubernetes – Container-orchestration: https://kubernetes.io/

[4]KubeEdge – Edge computing framework: https://kubeedge.io/en/

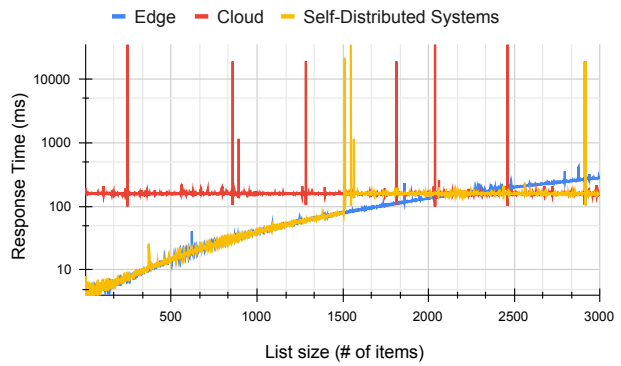

Fig. 4. Response time of the web application running on the edge (blue line); running parts on the edge and parts on the cloud (red line), and a self-distributing version that starts on the edge and moves (at 1500 size on the x-axis) parts of its components to the cloud (yellow line).

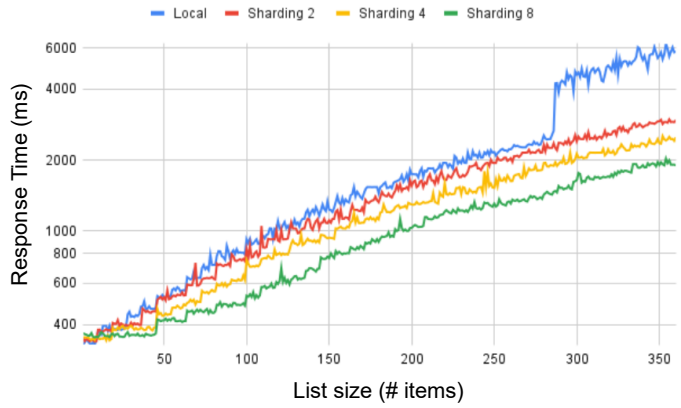

Fig. 5. Response time of the web application in different compositions exploring elasticity (local, 2-shards, 4-shards and 8-shards) as the size of the list increases overtime.

list, and after each item was added, the application sorted all elements in the list. Due to the increasing number of inserted items, the processing time for each request increases. Fig. 4 shows that response time increases faster when the web application runs on the edge. This effect is because the application has a single list running on the edge. In contrast, the cloud configuration takes advantage of the cloud resources and splits the list into two shards, splitting the incoming workload between them.

In this scenario, after 2000 items on the list, running the application on the cloud is best, despite the high latency added to forward requests to the cloud. We can also note that the SDS version of the application can take advantage of both the edge and cloud, as it can seamlessly relocate its list to execute on the cloud as the application runs. Also, note that the yellow line performs similarly to the application running on the edge. After the 1500 mark on the x-axis, the application relocates its list component to the cloud, performing similarly to the fixed cloud configuration. We also show that self-distribution has a high cost for state transfer, which can be noted as a series of sequential spikes on the yellow line right after the application is adapted (1500 mark on the x-axis).

Finally, we experimented with the cloud horizontal scaling (Fig. 5). Again, we subject the web application to the client's workload, which consists of sequential requests to the application to add items and sort the list. We executed the SDS web application in 4 configurations: the web application in a single process (local – blue line), the SDS version of the web application with two shards (red line), four shards (yellow line), and eight shards (green line). The graph shows that as we use SDS to split the list into shards, the application can handle the incoming workload with a smaller response time.

V. Related Work

This section surveys the most relevant related work in code offloading to edge-cloud infrastructures, adaptive systems that exploit the edge-cloud continuum, and state management in Object Request Brokers (ORB) architectures.

Code offloading is vastly explored in mobile devices [11]. More recently, other devices were also leveraging the advantages of code offloading from device to edge-cloud infrastructures [14]. These code offloading approaches require software engineers to carefully design ad-hoc solutions for code offloading, considering both code mobility mechanisms and state management (when applicable). On the other hand, our approach leverages a generalised programming model and a proxy-based strategy that simplifies and generalises code mobility. It also establishes a predefined proxy structure where engineers provide state management code accounting for the state consistency tolerance for their application.

Self-adaptive systems, i.e., systems that change their behaviour or structure to accommodate changes in the operating environment, also exploit the edge-cloud continuum [15], [16]. In such approaches, systems have an adaptation logic that guides adaptation to better exploit edge-cloud infrastructure resources. Although we have not explored self-adaptation in this study, our approach is inspired by the Self-distributing Systems concept that employs Reinforcement Learning algorithms to learn, at runtime, optimal distributed compositions. Furthermore, our approach uses a component-based model to abstract software compositions to actions for the learning algorithm. Due to the component model, the execution of the learning approach is generic, *i.e.*, application independent, and can be used in different SDS-based applications without any change to the learning algorithm.

Finally, the proxy implementation of state management consistency is the most crucial part of the presented approach. Our described approach was heavily inspired by previous work on state consistency on ORB-based architectures [17], [18]. These approaches are complementary to our own and provide the basis for state management for SDS.

VI. Final Remarks

This section concludes the paper by listing a set of further challenges and new lines of research to pursue as future work.

This work presented a preliminary study on the first version of the extended SDS to support state management. As part of

this first attempt at this extended concept, we have identified a set of challenges to be addressed in the future.

One of the most crucial challenges is to further generalise state management within Self-distributing Systems. We developed the proxy components for the list example, knowing that the stored items were numbers. In reality, however, any item could be stored in a list, and a more generic proxy extension would be necessary to handle generic typed lists properly.

Another critical challenge is to reduce the added burden placed on the application developer, which is now required to develop the application **and** the distribution proxy components. We envision the creation of a repository of distribution proxy components for the widely used abstract data types (*e.g.*, list, maps, dictionary, sets, trees, etc.). This repository would allow maximum reusability of distribution proxy components for applications, as new proxy components are added to the repository for every designed new interface.

Another challenge is the provision of fault tolerance mechanisms for the distributed stateful components. As the application is initially designed to execute in a single process, the distribution of local components may introduce errors the application is not expecting. The distribution proxy has to provide fault tolerance mechanisms to prevent such situations.

Moreover, as part of the adaptation algorithm supported by the component model runtime, the state transferring step may cause the system some disruption due to the time it may take to complete the task. The duration of the state transferring step is directly connected to the size of the state. Part of this challenge is to consider a lazy approach for state transfer.

To conclude, the state management overhead added by the distribution proxy may also be prohibited depending on the application and scenario. Thus, an important challenge is to consider the proxy implementation's performance overhead.

This paper introduced an extension to the Self-distributing Systems (SDS) concept to enable the distribution of stateful components across infrastructures. As a result, the extended SDS presented itself as an alternative to facilitate code mobility and horizontal scaling on edge-cloud platforms. To demonstrate SDS's potential in exploiting the edge-cloud continuum, we designed a stateful single web application, showing that it can self-distribute its internal components across the edge and cloud to enhance its performance in different scenarios.

In future work, we expect to explore Reinforcement Learning (RL) algorithms to learn optimal distributed compositions that exploit the edge-cloud continuum resources at runtime. A solution similar to the one described by Rodrigues-Filho, et al. [7]. We also aim to explore different state management approaches, data structures, and applications (*e.g.*,, autonomous vehicles, video streaming, or augmented reality) to further explore self-distribution on the edge-cloud continuum.

Acknowledgment

The authors thank the Computer Engineering students from University of Campinas: André Papoti, Bruno Berbare, Felipe Guardão, Gabriel Oswaldo, and Ricardo Koaro for developing some of the ideas presented in this paper. Roberto Rodrigues-Filho thanks FAPESP for supporting his postdoctoral research at INF/UFG during the time this work was developed (2020/07193-2). This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc.465446/2014-0, CAPES -– Finance Code 001, FAPESP procs.14/50937-1 and 15/24485-9. Finally, this work was also funded by FAPESP under the grant 2019/26702-8.

References

[1] G. Blair, "Complex distributed systems: The need for fresh perspectives," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1410–1421.

[2] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[3] F. Wang, M. Zhang, X. Wang, X. Ma, and J. Liu, "Deep learning for edge computing applications: A state-of-the-art survey," *IEEE Access*, vol. 8, pp. 58 322–58 336, 2020.

[4] P. Ren, X. Qiao, J. Chen, and S. Dustdar, "Mobile edge computing–a booster for the practical provisioning approach of web-based augmented reality," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 349–350.

[5] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[7] R. Rodrigues-Filho and B. Porter, "Hatch: Self-distributing systems for data centers," *Future Generation Computer Systems*, vol. 132, p. 80, 2022.

[8] R. Rodrigues 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)*. IEEE, 2020, pp. 176–181.

[9] B. Porter and R. Rodrigues Filho, "A programming language for sound self-adaptive systems," in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2021, pp. 145–150.

[10] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama, "A component model for building systems software," in *Proceedings of the Eighth IASTED International Conference on Software Engineering and Applications (SEA '04)*. Acta Press, 2004.

[11] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[13] R. Silva Dias, R. Rodrigues-Filho, L. F. Bittencourt, and F. M. Costa, "Runtime microservice self-distribution for fine-grained resource allocation," in *15th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2022.

[14] K.-L. Wright, A. Sivakumar, P. Steenkiste, B. Yu, and F. Bai, "Cloudslam: Edge offloading of stateful vehicular applications," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020.

[15] A. Cattermole, J. Dowland, and P. Watson, "Run-time adaptation of stream processing spanning the cloud and the edge," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2021, pp. 1–7.

[16] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with kubernetes," in *14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2021.

[17] P. Narashimhan, L. E. Moser, and P. M. Melliar-Smith, "State synchronization and recovery for strongly consistent replicated corba objects," in *2001 International Conference on Dependable Systems and Networks*. IEEE, 2001, pp. 261–270.

[18] P. Narasimhan, L. Moser, and P. Melliar-Smith, "Consistency of partitionable object groups in a corba framework," in *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, vol. 1, 1997, pp. 120–129 vol.1.