A Self-distributing System Framework for the Computing Continuum

Roberto Rodrigues Filho
Institute of Computing
University of Campinas
Campinas, Brazil
robertor@ic.unicamp.br

Renato S. Dias
Institute of Informatics
Federal University of Goiás
Goiânia, Brazil
renato.dias@discente.ufg.br

João Seródio
Institute of Computing
University of Campinas
Campinas, Brazil
j218548@dac.unicamp.br

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

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

Edson Borin
Institute of Computing
University of Campinas
Campinas, Brazil
edson@ic.unicamp.br

Luiz F. Bittencourt
Institute of Computing
University of Campinas
Campinas, Brazil
bit@ic.unicamp.br

***Abstract*—Applications such as autonomous vehicles, virtual reality, augmented reality, and heavy machine learning-based applications are becoming popular and demanding more flexible deployment environments. The computing continuum, a hierarchical hybrid infrastructure comprehending user devices (smartphones, sensors, laptops, *etc*.), edge data centers, and cloud platforms, offers a wide range of deployment possibilities with a full range of varying computing resources. To take full advantage of such infrastructure, application development is faced with many challenges, the most important being the implementation of a transparent and generalized mechanism for code offloading and mobility throughout the continuum. To tackle such issues, this paper presents the Self-Distributing Systems (SDS) framework, a self-distribution framework that supports generalized code-offloading capabilities at the application level with a machine learning agent for deciding where to place components and a component-based model to enable seamless distribution of an application's components at runtime. We describe the framework, show its applicability in different application scenarios, and report our preliminary results. We conclude the paper with a list of challenges and invite the systems community to join the effort to further investigate them.**

***Index Terms*—computing continuum, self-distributing systems, reinforcement learning, energy-efficient computing**

I. Introduction

Computing systems for data processing evolved from centralized, job-based cluster infrastructures to a service-oriented paradigm, culminating in utility computing, nowadays represented by the widespread use of cloud computing [1]. The cloud is able to fulfill a set of requirements for a wide range of applications. However, its centralized nature imposes delays introduced by networking equipment and signal propagation, which can impact the quality of service (QoS) of applications. Moreover, some applications have stringent delay, reliability, and/or throughput requirements [2], constraining their processing to occur geographically close to where the data are needed. Fog and edge computing emerged to reduce latency and improve response time, in which data processing is placed closer to consumers, consequently also reducing network use and contributing to reduce bottlenecks.

The integration of edge devices, the fog, and the cloud is a challenging task: it involves the cooperation of stakeholders with different, sometimes conflicting, objectives. Seamless resource and application management to optimize an application's behavior and fulfill highly heterogeneous application requirements is desirable to compose a single distributed infrastructure that is able to automatically and adaptively handle a plethora of data and application components. The computing continuum emerges from this integration of devices from the edge to the cloud, encompassing sensors and actuators from the Internet of Things (IoT), mobile devices and cellular networks, industrial and agricultural infrastructures, vehicles, networking equipment, and so on [3], as illustrated in Figure 1.

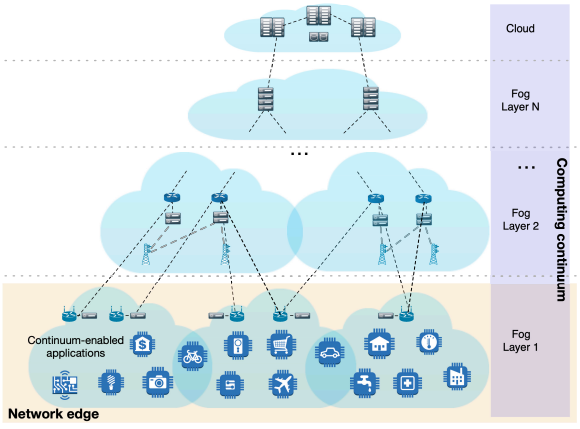


Fig. 1: The computing continuum distributed infrastructure.

The complex management brought by the computing continuum demands new mechanisms to cope with a highly dynamic and heterogeneous environment. Application and data placement decision-making is core for keeping performance

at acceptable levels: where components execute and data are located determines data transfer times and the execution times of application components. Furthermore, the dynamic nature of the problem requires applications to adapt to new scenarios and changes in the environment. In this context, this paper discusses how a self-distributing framework help address decision-making problems in the computing continuum, along with the challenges to achieving effective self-distribution.

The next section discusses related work, while Section III introduces self-distributing systems concepts. Section IV-A presents the use cases we studied for exploring self-distributed systems, discussing results in real systems platforms. Challenges to effectively achieve self-distribution in the computing continuum are discussed in Section V, and Section VI concludes the paper with the final remarks.

II. Related Work

The investigation of techniques to better exploit distributed infrastructures has a long history. In this section, we divide our analysis into three groups: i) adaptation at the service level, ii) runtime code distribution with state management, and iii) approaches that exploit the computing continuum.

Microservices and serverless models are popular technologies that have been used for creating large-scale applications. Breaking monolithic applications into small, highly reusable, and stateless services helps applications exploit elastic platforms such as cloud computing. Recent examples of these technologies for scaling applications are presented in [4], [5], and [6], [7]. In particular, Rossi *et al.* [4] describe hierarchical control policies to control replication using horizontal autoscalers, while Coulson *et al.* [5] concentrate on the identification of which microservices to scale. Boucher *et al.* [6] and McGrath *et al.* [7], on the other hand, present an approach for scaling function-as-a-service applications by performing adaptation at the platform level (*i.e.*, creating replicas of the services). Our approach differs from the above as we propose an intelligent agent that performs self-distribution at the service level based on information collected from the services themselves. Furthermore, services must purposefully have a stateless design, with no explicit mechanisms to handle state. Our approach works by distributing stateful components throughout the infrastructure while transparently maintaining state consistency across replicas.

Another important part of SDS research and practice is the exploration of mechanisms for code mobility throughout the continuum. Code offloading [8] has been widely used in mobile devices to better exploit computing resources. In the context of the computing continuum, Wright *et al.* [9] describe the application of code offloading in the edge-cloud infrastructure. Moreover, architectures based on Object Request Brokers (ORB) also implement mechanisms for distributing objects across infrastructures while handling state, as presented in [10], [11]. Traditional code offloading techniques, such as the above, differ from our approach as they are application specific and are built to exploit particular tradeoffs in specific operating environments. Our approach, on the hand, is aimed at being generalized, considering applications built using component-based models. Regarding state management, SDS draws inspiration from the mechanisms employed by ORB-based architectures.

Systems that are able to change their architecture to cope with changes in their operating environment have been widely used to exploit resources in distributed infrastructures. Cattermole *et al.* [12] and Ju *et al.* [13] describe relevant examples of self-adaptive systems to exploit edge-cloud continuum resources. The SDS approach aims at self-adaptive systems that are able to make decisions about their distributed architecture at runtime with minimum human interference. Our approach aims to provide a framework to facilitate the development of everyday software to exploit computing continuum infrastructures. In this sense, our work is aligned with other initiatives in the literature, such as [14].

III. Self-distributing Systems

Self-distributing systems (SDS) aim to create software systems able to autonomously (*i.e.,*, with minimum or no human interference) learn a distributed software design at runtime considering the dynamic operating conditions on which the system is executing, with the goal of improving efficiency, mitigating the likelihood of system errors, and adapting to changing environments in real-time [15]–[17]. The distributed software design decisions are pushed to the system itself at runtime as it gathers performing metrics from the system and its operating environment. The realization of SDS involves the integration of two core concepts: a component-based model and an intelligent agent.

The goal of the component-based model is to allow the autonomous composition of software out of small and highly reusable software components. The role of the intelligent agent, on the other hand, is to learn which distributed design (*i.e.*, how to distribute local components to remote machines) optimizes systems performance and efficiency when the system is subjected to a given operating environment.

A. Component-based Model

One of the bedrock concepts of our approach is the lightweight component-based model. These models are discussed and evaluated in the literature [18]–[20]. A lightweight component-based model allows the construction of adaptive systems by separating the logic of the components from the way the components are connected. In the object-oriented programming paradigm, this separation is not always reinforced, and the logic of a class can determine to what other classes the implementation depends. In the component-based model, dependencies are explicitly determined by interfaces, and at runtime, the connection between components is determined by the components that require such interfaces with components that provide the interfaces. This provide-require interface policy is key to enabling runtime adaptation by replacing a component with another that provides the same interface.

In the SDS, the component-based model allows the replacement of a component to a Remote Procedure Call (RPC)

proxy, *i.e.,* a component that implements the same interface as the soon-to-be-replaced component, but each method only has code that forwards the call from the proxy to a remote instance of the component. By replacing a component with a proxy, the original local component can be relocated and replicated throughout the infrastructure. The proxy has the responsibility for handling state management among the component replicas to ensure consistency and to apply some load balancing policy to split the incoming workload among the distributed component replicas.

B. Intelligent Agent

An important part of the realization of SDS is the intelligent agent, *i.e.,* the agent that monitors the environment and employs the component-based model runtime adaptation mechanism to optimize the system performance taking into account the different available distributed systems designs. For this, we envision the application of reinforcement learning, where the actions of the agent change the composition of the executing software. The learning algorithm guides the decision-making of the distributed software composition and is responsible for deciding the software distributed design at runtime with minimum human interference.

In detail, the agent receives a list of possible software compositions as actions at runtime. The agent then selects an action (*i.e.*, changes the system to a particular composition), observes the system executing for a predefined time frame in the selected composition, and afterward collects metrics from the system to establish the reward of the selected action. The main problem consists of learning which action yields maximum reward as the environment changes. Many reinforcement learning algorithms could be used to guide the system distributions decisions. In a later section, we describe our baseline approach used to produce the results we present in this paper.

C. State Management

Autonomous state management is crucial to the realization of SDS. The development of applications using component-based models requires the development of stateful components as in any other programming paradigm.

State management is widely studied, and in our approach, we envision the application of consistency models implemented in the distributing proxy (the proxy responsible for forwarding requests to remote components). The goal is to keep the state consistent across replicas, using a consistency model appropriate for the state type and application consistency tolerance levels.

The algorithm responsible for state management is implemented in the RPC proxy. State has to be carefully handled both during the component distribution process and after the components are distributed. During the distribution process, the soon-to-be relocated component is replaced by the RPC proxy component. The proxy, in turn, interacts with the Distributor layer running on other machines to load instances of the relocated component remotely. It then carefully ships

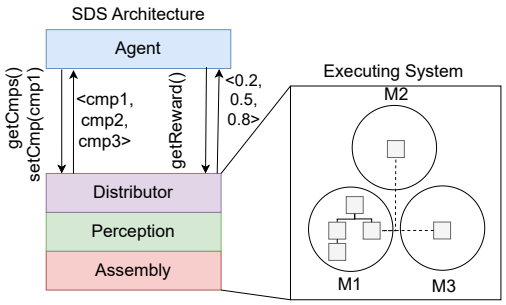


Fig. 2: Self-distributing Systems framework overview.

the state from the local machine to the remote machines and finishes the distribution process. Note that to maintain the state consistency, during the distribution process, incoming requests to the relocated component are paused, preventing state changes mid-distribution and maintaining state consistency.

After the distribution process is finished, the system is in a new distributed composition. From this point on, the proxy forwards the incoming requests to the remote instances of the component. The request forwarding process has to take into account the state consistency among the multiple component replicas. Depending on the state type (*e.g.,*, a single integer value, a list of strings, a hashmap, *etc*.) a different approach or algorithm can be more appropriate. For instance, a list can be sharded and each part of the list can be placed in a different replica across the infrastructure and the incoming workload can be split among the exiting replicas.

We discuss different strategies to maintain state consistency in a later section, and discuss the challenges involved in ensuring state consistency for a variety of RPC proxies.

D. Self-distributing Systems Architecture

This section presents the proposed Self-distributing Systems architecture. We describe each one of the layers, their services, and their interaction to realize SDS. We highlight the interaction between the intelligent agent and the remaining layers of the architecture, describing the services each layer offers and how they are used to implement SDS for general applications.

An overview of the SDS architecture is depicted in Fig. 2. On the right-hand side of the figure is a four-layer architecture of the SDS concept. On the top, we have the intelligent agent layer that implements the machine learning algorithm. The agent uses the underlying layers to choose actions (*i.e.*, system compositions) and get metrics from the system (*e.g.*, response time) to calculate the selected action's reward. The agent makes distributed systems design decisions by interacting with the system by choosing compositions, calculating their reward, and deciding on the best-performing composition at runtime.

The second layer is the Distributor, which implements the distribution process and provides an API (*e.g.*, REST API) for the agent to interact with the system. The distributor provides functions that allow the agent to get a list of possible distributed systems designs, a function that allows the agent to

change the system from one composition to another seamlessly at runtime (with no systems downtime), and a function that allows the agent to gather performing metrics from the system and metrics to characterize the operating environment. The distributor uses the layers below to implement the distribution process and to gather live metrics from the executing system.

The Perception layer implements all mechanisms to extract metrics from the executing system and its operating environment [21]. The main mechanism implemented by the perception is the insertion of monitoring proxies into the system architecture to extract metrics from a target executing component. The monitoring proxy intercepts function calls aimed at the target component and extracts performance metrics from it, and makes those extracted metrics available to the upper layers. The perception also accommodates ways to collect metrics from infrastructure by pulling metrics from services (*e.g.*, Prometheus[1], or tailored services) and make them available to the upper layers.

Finally, the $Assembly$ is the bottom layer and is responsible for composing the system using small components following the applied component-based model [21]. It is responsible for supporting both the $Perception$ and $Distributor$ layers by providing a list of possible compositions for the system, changing the system composition at runtime, and enabling the insertion of proxy components (both monitoring and RPC proxies) into the executing system. Note that the interaction among the $Distributor$, $Perception$, and $Assembly$ are through local function calls, whereas the interaction between the Agent and Distributor is done through a remote API.

Besides the SDS architecture, Fig. 2 also shows an executing distributed composition of the system running on a distributed infrastructure. To execute the system, we assume that each of the machines is visible and accessible from $M1$, the entry-point of the system and where the local version of the application executes. Machines $M2$ and $M3$ are other nodes in the infrastructure. We also assume that the Distributor, Perception, and Assembly processes are running on all machines. The distribution process starts on the Distributor running on $M1$ that replaces a component running on the local application in $M1$ with an RPC proxy. The proxy in turn interacts with the remote instance of a Distributor running in both $M2$ and $M3$ to load a version of the local component on the remote machines. After the components are loaded, the proxy running on $M1$ sends the component state to the component on $M2$ and $M3$, concluding the distribution process. Afterward, all incoming requests to the RPC proxy are then forwarded to the $M2$ and $M3$ following an algorithm to ensure state consistency among the replicas.

Assuming we have access from $M1$ to the remaining nodes in the infrastructure ($M2$ and $M3$) and the execution of $Distributor$, $Perception$, and $Assembly$ in all nodes, components could execute anywhere in the cloud-edge computing continuum. We demonstrate that in the next section in a series of experiments using different use cases.

[1]Prometheus is an open-source monitoring tool (https://prometheus.io/).

E. Interfacing the Computing Continuum

We argue that the concept of SDS is ideal for exploiting resources on the computing continuum at the service level. The ability of SDS to relocate and replicate components running on a process to different processes throughout an infrastructure allows self-distributing systems to autonomously exploit service mobility by placing them throughout the infrastructure, and to exploit, for instance, the trade-off between network latency and computing resources availability.

The interpretation of SDS and the computing continuum infrastructure is realized through the interaction of the $Distributor$ layer with the API of a container-orchestrator. We assume that the entire continuum infrastructure is managed by a container-orchestrator (*e.g.*, Kubernetes[2] or Mesos[3]). The $Distributor$ is extended to interact with the container-orchestrator's API to deploy container images running the $Distributor$, $Perception$, and $Assembly$ images during the distribution process.

In detail, when the $Distributor$ layer receives requests to change the system from a local to a distributed composition, it receives a list of nodes addressed to which the components will be relocated, the component to be relocated, and the RPC proxy that will be used in the distribution process. The $Distributor$ then interacts with the container-orchestrator's API to deploy the container image of the $Distributor$, $Perception$, and $Assembly$. Once the image is bootstrapped, the $Distributor$ proceeds to the next steps of the process. Dias *et al.* [16] describe in detail the full integration of SDS and a Kubernetes-managed cluster on the cloud. The same approach could be used to run SDS across the entire continuum.

IV. Use-cases and Evaluation

This section presents the main use cases that we have been exploring to demonstrate the potential of SDS to exploit the computing continuum resources. We describe different use cases for applications that run on the cloud platform, on edge-cloud platforms, and on a heterogeneous computer cluster. Particularly, we explore SDS to provide horizontal scaling mechanisms on cloud platforms; code offloading in edge-cloud mixed platforms; a comparison between SDS and other currently popular service models, showing the clear advantages of SDS over the state-of-the-art technology; and the use of different metrics for decision-making in optimizing SDS.

A. Web-based Stateful Application

We explored SDS in the computing continuum using a web-based stateful application as the main use case. This application was also explored in the context of previous work that investigates SDS as a general mechanism for horizontal scaling and code offloading [17]. In addition to revisiting the results of previous work to explore horizontal scaling and code-offloading, we use this use case to explore energy consumption as a metric for optimizing SDS.

[2]https://kubernetes.io/

[3]https://mesos.apache.org/

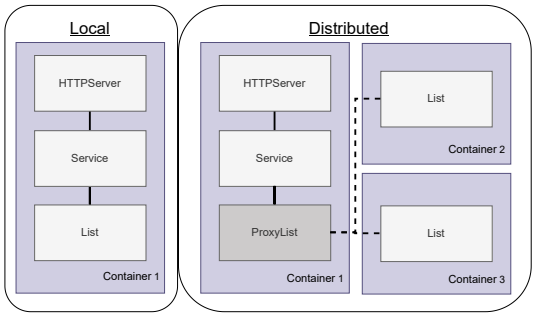


Fig. 3: Web-based application architecture in two distinct compositions. On the right-hand side, a local (*i.e*, all components in a single container), and on the left-hand side, a distributed composition with the list component replicated in two remote containers. Note that the depicted components can be executed anywhere in the computing continuum.

The use case application illustrated in Fig. 3 consists of a web-based application that offers services to post and retrieve information on a list over HTTP. The application has three main components: $HTTPServer$, $Service$, and $List$. The $HTTPServer$ component is responsible for handling incoming HTTP 1.0 requests and handover the request to the appropriate function defined in the $Service$ component. The $Service$ component, in turn, implements the main logic of the application, which basically enables users to add items to or retrieve items from the list. The function that retrieves items from the list has an extra processing time as it sorts the list before returning information, and as the list size increases, it demands more processing time.

Apart from the main illustrated components in Fig. 3, we also have the RPC proxy component. In this particular use case, the RPC proxy implementation creates shards (fragments) of the list during the distribution process and inserts the different shards of the list into the different created replicas. Given the number of replicas to be created, the proxy iterates through the list's items and applies a hash function to uniformly distribute the items across the replicas. This approach is inspired by Distributed Hash Tables (*e.g.,* [22]).

After the distribution process ends, requests for the list's functions to add or search items are forwarded to the replicas following the hash function. The main challenge is to choose a good enough hash function to distribute items evenly across the replicas. On the other hand, cross-sharding operations, *i.e.,* operations that require more than one shard to execute (*e.g.,* searching the smallest item on the entire list) requires coordination of the proxy with all shards to properly implement the operation.

Note that for the functions that affect only one shard, this implementation works as load balance, whereas the cross-sharding operations have a high-performance cost as they require interactions over the network with the existing replicas.

This use case is a fully functioning HTTP service that handles requests from web browsers and other HTTP-based clients and implements a stateful service with added processing time as the list (*i.e.,* the component's state) becomes larger. This enables the distribution of stateful services to exploit the continuum of resources as client workload changes.

B. Horizontal Scaling

This use case discusses the implementation of horizontal scaling through SDS. The ability of SDS to replicate part of their composing components allows these systems to exploit the elasticity of cloud computing platforms at the service level with stateful components. In current systems, however, in order for applications to exploit the elasticity of cloud platforms, they have to be built considering mobility and replication and thus stateless services have become popular.

In this scenario, SDS is explored to replicate stateful components throughout a cloud computing platform as the incoming workload increases. In this set of experiments, we use the web-based application described in Sec. IV-A. The application starts in a local composition, *i.e.*, all components are executing on the same container on the cloud, and as the incoming workload increases, the SDS autonomously distributes the $List$ component to exploit the cloud horizontal scale-out.

Rodrigues-Filho *et al.* [17] show the web-based application described in Sec. IV-A performing horizontal scale-out on the cloud as the size of the list increases, showing that a local implementation of a web-based stateful service can exploit the cloud elasticity at runtime, with no service downtime and minimum service disruption. The paper only explored the distribution mechanism supported by SDS but did not implement or experimented with any decision-making. In a later section, we explore horizontal scale-out on the cloud with decision-making support and compare it with serverless models, which are currently state-of-the-art service technology.

C. Code Offloading

SDS has also been used to demonstrate how code-offloading can be performed in the cloud-edge continuum infrastructure. Rodrigues-Filho *et al.* [17] demonstrate code-offloading using the web-based application use case described in Sec. IV-A.

In the experiment, the web-based application was fully executed on an edge node and served requests to a client node in the same network. This first system composition was ideal for performance because the list had a small number of items, and thus processing time was low. Furthermore, the edge node was in the same network as the client and presented a very low network latency. However, as the number of items is continuously added to the list, the processing time increases and starts affecting the system's performance to a point where it is best to create shards of the list on the cloud and split the workload among the list shards, suffering from the high network latency than maintaining the entire system on the edge node and suffering from the increasingly high processing time. Fig. 4 illustrates the results of the experiment.

In detail, Fig. 4 shows a bar graph of three versions of the executing system. The "Edge" label (blue bar) represents the system statically executing on the edge node. The "Cloud"

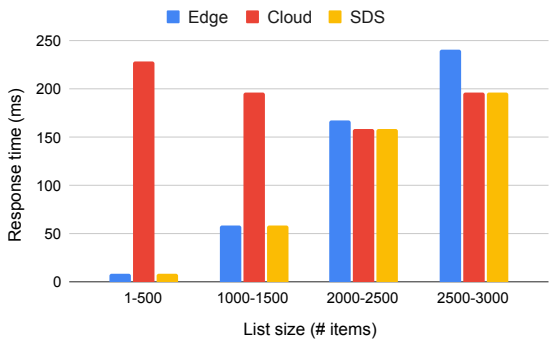


Fig. 4: Graph showing the performance of two static compositions (Edge and Cloud) of the system, and the SDS performing code off-loading from the edge to the cloud. The Edge and Cloud static compositions have optimal performance depending on the size of the list. SDS adapts between the two to better exploit the infrastructure.

label (red bar), on the other hand, shows the performance of the system executing on the cloud with the list divided into two shards. Finally, the "SDS" label shows the system performance, switching the system composition from Edge to Cloud as the list size increases and thus better exploiting the infrastructure resources, maintaining good performance throughout the entire experiment.

D. Energy-consumption as Optimization Criterion

In previous works [15]–[17], the main chosen metric to decide the best composition for the executing SDS application was performance based on response time. In this section, we experiment with energy consumption as a different and currently very important metric. The goal of this use case is to show SDS works regardless of metrics and to demonstrate a baseline algorithm that allows SDS to autonomously select the distributed composition that yields minimum energy consumption. To execute the experiments, we use a heterogeneous cluster with machines with different power consumption specs.

For this use case, we also use the web-based application described in Sec. IV-A and two laptops with different power profile: Dell g15 and Dell XPS. We executed the application on the g15 laptop and measured the energy consumption and operation time (*i.e.,* the time it takes to complete the client-requested operation – sorting the list and returning the items). Afterward, we executed the service on the XPS laptop and also measured both energy consumption and operation time. Based on the information collected in the measurements, we found that g15 has the highest energy consumption but the lowest operation time, whereas XPS is the opposite and has the lowest energy consumption and the highest operation time.

After executing the measurements and profiling energy consumption and operation time by executing the web application on both machines, we executed the decision-making algorithm twice, each time using a different metric for optimization. We implemented a baseline algorithm that consists of two phases: a fixed exploration and a fixed exploitation phase. In the exploration phase, the algorithm iterates through all available compositions, executing them for a predefined time span (observation window). After sampling one metric from each composition, the algorithm enters the exploitation phase and changes the system components to the one that generated the lowest metric, maintaining the same composition while it continues to monitor the metric.

In this set of experiments, the SDS has two compositions. The first is to execute the list component on the g15 laptop, and the second is to distribute (relocate) the list component and run it on XPS one. For the distributed composition, the RPC proxy does not need to handle state consistency, as the component is simply relocated to a different node along with its entire state, having no other replica on the infrastructure. SDS can, then, at runtime, and with no service disruption, identify what composition yields the minimum value for the observed metric.

Fig. 5a and Fig. 5b show the results of executing the SDS baseline decision-making algorithm for energy consumption and operation time metric. The blue window in both graphs shows the time the algorithm spends exploring. Due to the fixed exploration phase, the system explores the available composition only once, so the exploration time depends only on the observation window size and the number of compositions, and hence they remain the same in both experiments.

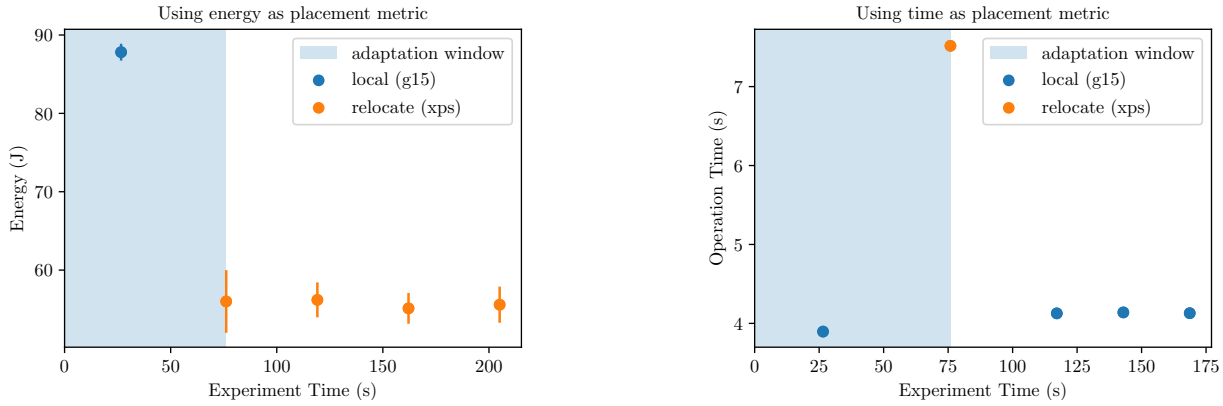
The algorithm executes and explores both compositions (local and relocated) for a time span (observation window) and collects the system's metrics after sampling the metric exactly once. Next, the algorithm decides on the composition that yielded the minimum metric value. For the energy consumption metric shown in Fig. 5a, SDS correctly selects the least energy-consuming node to execute the service (XPS). Also, when employing the operation time metric, as illustrated in Fig. 5b, SDS converges towards the node that has the lowest operation time (g15).

We conclude that our baseline algorithm works to demonstrate the potential of SDS approach. By coding a single stateful application and choosing a metric, our algorithm is able to determine which composition best exploits the available resources at runtime, with no predefined information nor human involvement.

E. Comparing Against State-of-the-Art Service Technology

Finally, for our last discussed use case, we experiment with a couple of service benchmark applications. We use two popular applications to compare SDS performing horizontal scale-out on the cloud against the application running on serverless model platforms (*e.g.*, Google Functions). The goal of these experiments is to show how SDS differs from popular service technology and the advantages of using SDS.

Serverless computing and microservices have been largely adopted as architectural styles to create large-scale modern applications. Serverless computing applications are composed of interacting stateless functions running on top of an autonomously managed infrastructure. In this model, developers



(a) Energy consumption minimization.

(b) Time minimization.

Fig. 5: SDS decision-making algorithm exploring two compositions and deciding on each node to execute the service aiming to minimize two different metrics.

focus on coding stateless and highly reusable functions. Then, they define a workflow determining how the executing functions interact amongst themselves to form an application. As the Services handle the incoming requests, the platform determines when to scale them out by creating replicas and when to kill the instances of idle functions, leaving all management of the lifecycle of the function to the platform itself.

SDS, on the other hand, runs on container orchestrators-managed platforms and has full control over the creation of new containers on the platform and the choice of which local component to relocate or replicate across the infrastructure. The decision-making that determines the systems resulting distributed design takes into consideration the system composition at the service level and metrics collected directly from the executing application. As opposed to the decision-making of when to scale out in serverless platforms, which do not consider any service-specific metric or information.

Furthermore, SDS allows great flexibility to the system's composition, as it allows the rearrangement of local application components to explore computing resources at runtime, adding components to the same container and having them interact with each other through local function calls, or relocating components to different containers and having them interact via RPC over the network. Serverless functions, on the other hand, often execute in isolated containers and always interact through a REST API over the network, limiting the composition flexibility to assemble the application in a way it can best exploit the available computing resources.

For the comparison between SDS and serverless computing models, we used two applications: a prime number calculating application and an array sorting and searching application. Both applications are CPU intensive, and we use response time as our main metric for performance comparison. Next, we describe how each of the applications is implemented both using the SDS framework and following the serverless model.

The prime service application consists of a single function that receives a number n that represents an interval from 0 to n used by the function to calculate the number of prime numbers that exist in the given range. The service then receives requests from clients, which queries for the number of prime numbers existing in a given range, and, as the range increases, the processing time of the service also increases. For the serverless platform, we created only one function that receives the range and calculates the number of primes. For the SDS, on the other hand, we have two components; one that implements an HTTP server responsible for handling incoming HTTP requests and forwarding them to the prime function component that calculates the number of primes in the given range.

On the serverless platform, the prime service application is a single stateless function, and the number of replicas is created by the platform as the number range increases and the service demands more CPU. We configured the platform to have a minimum of one instance and a maximum of ten replicas and leave the platform to coordinate replica creation as we increase the range. The SDS version, on the other hand, starts with all components (the HTTP server and the prime components) running in the same container with only one replica. As we increase the range, we create a composition with two replicas of the prime service component and another with three replicas of the prime component.

Fig. 6 shows the ground truth of the prime number application experiment. We executed three versions of the prime application; the first is the serverless version of the application running on the Google Functions platform. This version is illustrated by the blue bar in the graph. The remaining two versions of the application are executed on the Google Kubernetes Engine (GKE) using SDS in two static compositions; the red bar is the application running with two replicas of the prime service, and the yellow bar is the application running with three instances of the prime service. The graph shows that as the range increases, the response time increases for all compositions. It also shows that the serverless model has better performance than the SDS with two replicas, but it performs worse than the SDS with three replicas.

Fig. 6: Graph showing the response time degradation of the prime number application when receiving input ranges that vary from 1000 to 2000. The blue bar shows the performance degradation of the application running on the serverless platform. The red and yellow bars show the performance degradation of the application running as a SDS with two (red bars) and three (yellow) replicas.

Fig. 7 shows the execution of serverless and SDS being subjected to range [0:1000], *i.e.*, n=1000. This experiment applies the decision-making algorithm with fixed exploration and exploitation phases used in the experiments in Sec. IV-D. The serverless version runs on the serverless platform and has its number of replicas determined by the platform, whereas the SDS has two available compositions: the local one, where all components run on the same container, and a distributed composition where the prime service component has two replicas. The SDS version starts on the local composition and has a high response time. Then it changes its composition to the distributed, collects the response time metric, and makes a decision to continue exploiting the distributed composition for the remainder of the experiment.

Fig. 7: Prime application running on the serverless platform (blue line) and as SDS (red line), both subjected to n=1000. The SDS decision-making algorithm explores both local and distributed compositions (with two replicas) and converges toward a distributed composition with a low response time.

The prime application is a good example of an application that has good performance in the serverless model, as the creation of individual replicas improves the performance of the system. We show that SDS has better control over the resources in the infrastructure and can decide on the number of replicas it creates by measuring the response time at the application level. We do not claim that SDS has better performance than serverless for this application. Rather, we demonstrate that SDS takes a local application and can converge towards a distributed composition similar to serverless. Moreover, SDS can act on metrics collected at the service level and thus make adaptation decisions to better use the available resources.

Next, we present another example of an application and implement it using both serverless functions and a SDS to explore the horizontal scale-out on cloud platforms. This application implements sorting and searching on an array. The application receives as input the name of the file and a number to search. The service then proceeds to load the file content to memory (in an array), sort the array, and search for the given number in the sorted array, returning $false$ when the number is not in the file and $true$ when it is.

The serverless implementation has three main functions: an IO function responsible for fetching information from a file to memory, a function that receives an array as input and sorts the array, and a function that searches a number on a sorted array. To execute this application on the serverless platform, we define a workflow that determines the path among the functions all incoming requests has to go through. Each function runs on an independent container, and they are all managed by the serverless platform.

The SDS application has four main components interacting with each other through local function calls. The first component is the HTTP server which allows the application to handle incoming requests. The second component implements the IO function that loads information from a file to an array on the main memory. The third component receives the loaded array and sorts its content. Finally, the fourth component receives the sorted array and the number and searches for the number in the array, returning $true$ in case the number is found in the array and $false$ otherwise.

Fig. 8 shows the response time of the sorting and searching application given the size of the array loaded to memory (n). As the size of the array increases, the response time of the service also increases. The graph shows a particular disadvantage in terms of the response time in the serverless application version as compared to the SDS version. This significant difference occurs because the array has to be transferred to the nodes executing the functions through the network in the serverless platform, as in the SDS application interacting components happen through local function calls, and the array is passed as reference.

Note that the SDS performance shown in Fig. 8 creates two replicas of the three main components (IO, sorting array, and the searching components) grouped together in the same container making local function calls. The serverless version, on the other hand, can only scale the application by creating replicas of the individual functions which continue to exchange large data over the network. This is a scenario where SDS has clear advantages over serverless as it can either divide and run multiple replicas of individual components or group

Fig. 8: Response time of the sorting and searching application running both on the serverless platform (blue bars) and as SDS (red bars).

components into the same container and allow them to interact with each other through local function calls.

Fig. 9 shows the SDS running the decision-making algorithm in the context of the sort and searching application. The blue line is the application on the serverless platform subjected to requests for n=1200. The SDS starts in a local composition, *i.e.*, where all components are executed on a single instance of the application in a single container, and then it explores the other alternative design, which is to create two replicas of all three main components and have them execute in two containers and split the incoming request load between the two. As the distributed composition yields the lowest response time, the algorithm decides to maintain the system on the distributed composition for the remainder of the experiment.

Fig. 9: Sorting and searching application running on the serverless platform (blue line) and as SDS (red line), both subjected to n=1200. The SDS decision-making algorithm explores both local and distributed compositions and selects the distributed composition with a low response time.

V. Challenges in Self-Distribution in the Computing Continuum

Although our preliminary exploration of the concept of SDS in the cloud continuum resources is promising, there are many issues and challenges that need to be further investigated.

The intelligent agent is an important part of realizing the concept of SDS. The presented baseline decision-making algorithm (see Sec. IV) works in simple scenarios but does not address the entire learning problem. The presented algorithm, however, can still be used as a baseline for future comparison with more elaborated approaches. Learning which distributed composition the system must select to maximize the reward (or minimize costs) of an executing system has many issues, such as large search space, fluctuating rewards, classifying operating environments, and distribution cost.

As a consequence of deciding how to distribute systems throughout the continuum at runtime, the search space of possible distributed designs becomes extremely large. In the decision-making algorithm we experimented with in this paper, we did not show how large the size of the search space can become. The search space for SDS in the computing continuum must take into consideration the number of components a service/application has, consider how many RPC proxies for each component are available for exploration, the number of nodes in the entire infrastructure, and the performance of each node when dealing with heterogeneous settings. As the cloud continuum is a large infrastructure with the potential of having millions of available nodes, the number of possible ways to distribute components across the nodes is very large.

Another learning issue is the fluctuating rewards. This means that subsequent measures of system-level metrics from the same executing system composition are not static and may fluctuate, impacting the calculated value of rewards. Therefore, a decision-making algorithm like the one experimented with in this paper may not be sufficient in some situations, as it makes decisions after sampling metrics only once. It is not uncommon that the effects of a certain composition take time to stabilize and yield metrics that fluctuate in a small range. Caching is a notable example, as the cache needs to warm up (*i.e.*, store items) before it can have a positive impact on the system's performance metrics. SDS requires decision-making strategies that take these metric fluctuations into account.

The decision-making algorithm explored in the SDS agent does not explicitly accounts for the operating environment and thus can not remember past decisions. As a consequence, every change in the environment triggers exploration in the decision-making algorithm, requiring the algorithm to explore the systems compositions, even when the system has been exposed to the same environment. To solve this issue, strategies for classifying the operating environment while learning which distributed composition yields maximum rewards is an important challenge to address in SDS.

Online decision-making requires the agent to frequently explore the available compositions of the system. However, changing the system composition at runtime currently has a big impact on the system's performance. This is illustrated in Sec. IV, particularly in Fig. 7 and Fig. 9 in the form of a peak in the SDS performance at the time distribution takes place. The leading cause of performance degradation occurs in maintaining state consistency during the distribution process. The current implementation pauses incoming requests while the state is copied to remote processes. Lazy strategies to mitigate this performance impact could be explored and

implemented in RPC proxies.

State management is key to realizing SDS as a general approach for exploiting the continuum. We believe transparent state management when distributing components across infrastructure is one of the main advantages of SDS over state-of-the-art service technology. However, generic strategies for state consistency applied to all cases have a negative performance impact on the system as state management is highly dependent on the type of state and the way components interact with their state. In our SDS framework, we address this issue by having component developers write RPC proxy specific to the application components' needs to allow component distribution at runtime. The component developer is also the proxy developer; they carefully design the proxy to maintain state consistency during adaptation. This adds a burden to the system developers, that also have to create RPC proxy. Further investigation to minimize the developers' burden to developing RPC proxy is also necessary.

VI. Conclusion

This paper discusses how a Self-distributing System framework can explore code mobility and horizontal scaling of stateful components in the computing continuum. We demonstrate, with examples of previous work and new ones, the potential of the proposed approach. Particularly, we showed the potential of SDS to perform horizontal scaling of stateful components in cloud platforms, the code offloading of stateful components between edge and cloud, and the ability of SDS to autonomously explore the available space of possible distributed compositions and choose the one that yields maximum performance. We demonstrate the potential of a baseline decision-making algorithm using different metrics like response time and energy consumption. We perform preliminary experiments to compare SDS with state-of-the-art popular serverless models to show that SDS supports more flexibility in (re)arranging systems components through the continuum. Finally, we listed further challenges and future research directions to guide future work in this domain and invite the research community to work on mechanisms to raise abstraction to facilitate the development of applications for the computing continuum.

Acknowledgements

The authors thank São Paulo Research Foundation (FAPESP) for the grants 2019/26702-8 and 2020/16763-7. This research was also partially funded by the CNPq proc. 465446/2014-0, CAPES, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[2] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2557–2589, 2021.

[3] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3-4, pp. 134–155, 2018.

[4] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 28–37.

[5] N. Cruz Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4195–4202, 2020.

[6] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "micro" back in microservice," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 645–650.

[7] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 405–410.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] M. Jansen, A. Al-Dulaimy, A. V. Papadopoulos, A. Trivedi, and A. Iosup, "The spec-rg reference architecture for the edge continuum," 2022.

[15] R. Rodrigues Filho and B. Porter, "Hatch: Self-distributing systems for data centers," *Future Generation Computer Systems*, vol. 132, pp. 80–92, 2022.

[16] R. S. Dias, R. Rodrigues Filho, L. F. Bittencourt, and F. M. Costa, "Runtime microservice self-distribution for fine-grain resource allocation," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022, pp. 234–239.

[17] R. Rodrigues Filho, L. F. Bittencourt, B. Porter, and F. M. Costa, "Exploiting the potential of the edge-cloud continuum with self-distributing systems," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022, pp. 255–260.

[18] 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)*, 2021, pp. 145–150.

[19] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[20] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, mar 2008. [Online]. Available: https://doi.org/10.1145/1328671.1328672

[21] B. Porter, M. Grieves, R. Rodrigues-Filho, and D. Leslie, "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)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 333–348.

[22] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.