Runtime Microservice Self-distribution for Fine-grain Resource Allocation

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

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

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

***Abstract*—The development of systems using microservices as building blocks have gained major popularity in the industry in the past few years. Widely used services, such as Netflix and Uber, have been built entirely as microservice architectures. Due to the modularity and self-containedness of microservices, coupled with the use of elastic deployment infrastructures, a number of tools to assist the scalability of such systems have been created. However, these tools are limited to act at a fixed granularity, being able to replicate, relocate and provide access to extra resources only at the level of the entire microservice, even when only one of its parts actually demands more resources. In this paper, we propose the use of the concepts of adaptive component models, emergent microservices, and self-distributing systems to explicitly define the internal *micro-architecture* of microservices. In this approach, a microservice is built as a dynamic configuration of components, which can be seamlessly adapted and distributed on top of an elastic cloud infrastructure by the underlying platform. We evaluate the benefits of the approach by exploring different scenarios that entail the use of dynamic adaptation and self-distribution to perform vertical and horizontal scaling of microservices at a fine granularity. We analyze the involved tradeoffs and discuss how the approach can be further explored.**

***Index Terms*—Microservices, Self-distribution**

I. Introduction

Microservice-based systems have gained major popularity in the industry during the past few years. This increasing popularity is due, among other factors, to the high modularity and self-containedness of microservices, which make systems decoupled, highly reusable and relatively easy to adapt to accommodate fluctuations in the incoming workload.

The approach of decoupling services as functions that interact via programming interfaces has been used for some time now. Methods to take advantage of such separation in service-oriented architectures (SOAs) are also not new. In the context of cloud environments, we apply this decoupling to the design and implementation of microservices, maintaining the goals of rapid deployment, interchangeability, adaptability and scalability.

This paper proposes a novel approach for fine-grain adaptation of microservice architectures. The approach is based on the concepts of emergent microservices and self-distributing systems. It enables seamless and autonomic adaptation of microservices by changing their internal composition (such as by replacing and distributing components) at runtime to improve performance and resource usage.

We demonstrate the approach by performing vertical and horizontal scaling of the system at runtime at the service level, showing performance improvements that significantly outweigh the costs of the adaptation mechanisms.

The paper is structured as follows. Section II discusses related work, with a focus on adaptable microservice architectures, while Section III describes the proposed approach, the architecture that realizes it, and the kinds of adaptation that it supports. Section IV presents an evaluation of the performance improvements and adaptation costs in different representative scenarios. Finally, Section V concludes the paper with a discussion of the advantages and limitations of the approach, as well as future work.

II. Related Work

Microservices can be considered the next step from service-oriented architectures (SOAs), focusing on the small granularity and independence of services [3]. Coupled with the elasticity of cloud environments, they now represent a widely used approach to scale systems. Rossi et al. [11] explore such elasticity with hierarchical control policies to manage the adaptation of microservices, showing the advantages over existing tools such as the Kubernetes autoscaler.[1] Other related research [2] explores ways to increase system performance by choosing which microservice to scale in the environment.

Microservices have also been explored as self-reconfiguring systems in the cloud [5]. These systems employ techniques for the automatic deployment of optimized microservices. However, the approach treats microservices as monolithic building blocks. Instead, we focus on the distribution of microservices at a finer-grain level of distribution, using a dynamic component model to implement microservices. We show the possibilities of using this level of granularity to manage the resources utilized by the system.

The use of the "function-as-a-service" (FaaS) model to exploit microservices at a finer granularity is explored in [1]. However, similar to the general paradigm of serverless computing [7], it is limited to single function invocations and

[1]Kubernetes is an open-source system for automating deployment, scaling and management of containerized applications (https://kubernetes.io/)

focuses on adapting the infrastructure rather than the internal implementation of functions. Instead, our approach achieves a similar effect by acting at the service level, providing a way for the system to self-distribute and adapt a microservice's internal functions under the control of the application developer instead of relying solely on the infrastructure provider.

III. Approach

A. Background and overview

Our approach traces its origins to the concepts of emergent microservices (EM) [4] and self-distributing systems [10]. EM enables microservices to dynamically self-adapt in the face of changes in the workload, aiming at performance improvement and resource efficiency. The concept is based on the autonomic adaptation loop of emergent software systems (ESS) [9] and on the design and implementation of microservices using a dynamic component model that enables safe hot-swapping of components at runtime [8], such as provided by the Dana programming language.[2]

Self-distributing systems, in turn, refer to the ability to transform a local system into a distributed one by seamlessly distributing its components at runtime. The concept relies on transparent RPC (Remote Procedure Call) to enable the relocation of local components to remote hosts.

We employ these concepts in the context of an elastic cloud infrastructure in order to facilitate the required dynamic allocation of resources. To take advantage of the elastic environment, EMs are hosted within containers, which in turn are grouped in *pods* to facilitate their management. A container orchestrator (such as Kubernetes) is then used to enable the dynamic creation of pods to host new instances of an EM (or indeed of any its parts in the case of self-distribution).

The overall result is a self-adaptive *micro-architecture* model for microservices, capable of seamless reconfiguration and distribution of a microservice's components at runtime. It enables taking full advantage of vertical and horizontal scaling and distribution in elastic infrastructures, including component relocation and replication, which in turn provides a means for dynamic resource allocation.

Being able to break a microservice into smaller components for replication and relocation enables a fine-grain approach to handle situations in which the system finds itself competing for resources, such as CPU and memory, without the need to replicate or relocate the microservice as a whole. We can identify which part of the microservice is consuming most resources and apply different solutions to solve resource competition. For instance, we can relocate a resource-intensive component, giving it its own pod and resources, so that it does not have to compete with the other components for the same resources. Alternatively, we can relocate a rarely used component, so that the remaining components can perform better with more resources available at their current pod.

[2]https://projectdana.com

B. System architecture

The architecture to support the self-distribution of microservices can be described in terms of its three main modules, which are located in different pods in the system, as shown in Figure 1: *Distributor*, *RemoteDist*, and *ServiceCTL*. Their respective roles in realizing the approach are described next.

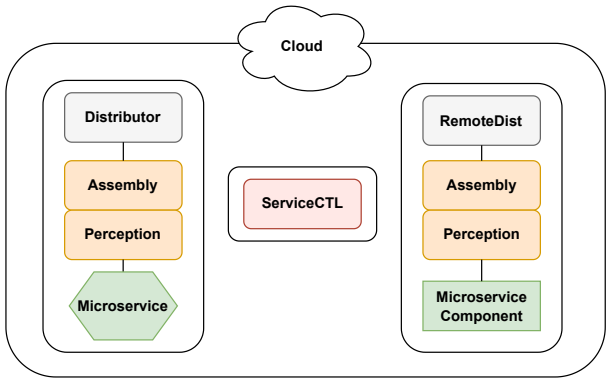

Fig. 1. The three types of pod in the supporting architecture of self-distributing microservices.

As shown in Figure 1, the system has a layered architecture composed of three modules located in the pods that run the microservice and its (distributed) components. The *Assembly* and *Perception* modules are common to all those pods and perform the two core functions of emergent software systems that are relevant to the approach, namely dynamic composition and monitoring [9]. *Assembly* is responsible for loading components from the file system into memory and connecting them to form a functioning system. It is also in charge of the runtime adaptation of components, by enumerating all valid compositions, thus making it possible to explore a variety of components to adapt the system at the service level. The *Perception* module in turn is responsible for collecting information from the running system. It is used to monitor the performance of the microservice while in each of its possible compositions and according to a predefined set of metrics and events, in order to establish the best composition for the microservice under different environment conditions.

The *Distributor* module, which is located in the original pod running the microservice, is in charge of directing the adaptation and self-distribution of the microservice's components, being also the starting point of the system. It calls the *Perception* module to monitor each of the compositions that the *Assembly* module was able to form and establishes the composition that is more appropriate (i.e., has the best performance) at each given point in time. It also connects to the *ServiceCTL* module to direct the relocation or replication of any component of the microservice when necessary. *ServiceCTL* in turn interacts with the container orchestrator (Kubernetes) in order to create a new pod to actually host the relocated/replicated component. Note that *ServiceCTL* runs on a separate pod as it may serve different microservices.

The *Distributor* is also responsible for establishing the connection between the different parts of a microservice that were distributed across the environment. Once it initiates the relocation (or replication) of a component, it replaces it (in its original pod) with a proxy that uses RPC to communicate with the new (remote) instance of the component. Thus, clients of the relocated component can continue interacting with it in a transparent way.

In order to complete the process of component relocation, the *RemoteDist* module, located in the destination pod, creates a local proxy and connects it to the corresponding proxy in the original pod. It then creates and activates the new instance of the relocated component. The two proxies then transparently forward the requests from the original pod to the component in its new location. Note that *RemoteDist* also uses the *Assembly* and *Perception* modules to adapt and monitor the relocated microservice components.

C. Self-distribution

The main motivation for self-distributing systems is to exploit the ability to relocate and replicate a system's components. As we adopt a fine-grain programming model for microservices, where each internal function of the microservice is presented as a separate component, we can use self-distribution to rearrange these components into distributed compositions to better utilize the environment provided by the cloud.

Relocation refers to the transfer of a component and all its dependencies to a remote host. It relies on the ability to seamlessly hot-swap components, as provided by the Dana programming language. Dana uses transparent proxies and, when an object is instantiated, it creates a proxy version of the object at runtime, which has an internal reference to the actual object. The proxy can then be used as a remote reference and can be passed to other objects. To relocate a component of a microservice to a remote host, we select the interface of the component and replace it with a proxy that forwards function calls to the component in its new (remote) location.

In this paper, we use relocation of components to perform resource management. Relocating an intensively used component provides a simple means to isolate resource usage within a microservice and to use the elastic environment of the cloud to give exclusive access to a pod's resources to the relocated component. We may also explore the relocation of microservice components that are not being frequently used in order to free resources (e.g., memory) for the other components. This allows busy components to have more resources without the need to interrupt them for their own relocation.

Another use of self-distribution is for the replication of components. This enables horizontal scaling at an intra-service level, where we replicate copies of the desired component to different pods. This process is carried out by the *Distributor* module with the help of *ServiceCTL*. The latter uses Kubernetes to create the desired number of replicas. Once the replicas are up and running, the addresses of their pods are given to *Distributor* so that it can connect to them.

D. Local adaptation

Resource management can also be exploited in a local composition without necessarily distributing the components. Using the concept of emergent microservices, we can explore different configurations of the system by changing which components are currently being used. A microservice may have different components that can act as handles for resource management, such as caches. The *Assembly* and *Perception* modules use these components to establish different microservice compositions with different resource usage profiles. Vertical scaling at the service level may then be realized by changing from a composition that uses a lightweight component to a different one that uses a component with more resources. At system start, all the functional components of the microservice have an attached non-functional component with this purpose. Such a "resource component" can then be replaced at runtime, granting the corresponding microservice component more (or less) resources to perform its task.

IV. Evaluation

In this section, we present an evaluation of the proposed approach for microservice adaptation, aiming to show its impact on application performance in different scenarios. First, we characterize the different compositions which the system can be adapted to, showing that some compositions may result in better performance, thus justifying adaptation from one composition to another. We also explore the cost of adaptation, showing the impact on the overall system performance. Our goal is to discuss and provide first answers to the following questions:

- Is our approach able to make adaptations at runtime that have a positive impact on the system, and if so, under which circumstances?
- What are the limitations of the approach?

All experiments were conducted on the Google Cloud. We created a cluster located in the US-Central region, with 3 nodes managed by the Google Kubernetes Engine (GKE). Each node was running the GKE standard Ubuntu image, with 2 vCPUs, 4GB of memory and 100GB of storage.

A. Case Study

In order to explore our approach in scenarios with real microservices, we take advantage of the InterSCity platform [6]. With its microservice architecture, the platform's goal is to provide an open-source environment that has all major building blocks for the development of smart city applications and services. Its microservices facilitate interaction with, and control of, the devices (sensors and actuators) that make up the cyberinfrastructure of a city, providing high level APIs to application and service developers. The InterSCity platform has six microservices with distinct functionalities: the *Resource Adaptor* microservice is responsible for interacting with sensors and providing their input to the other microservices; *Resource Catalog* and *Resource Discovery* enable the description and discovery of specific devices; *Data Collector* is responsible for providing applications with access to data

collected from city devices; *Actuator Controller*, as its name suggests, facilitates control of actuators; and *Resource Viewer* is in charge of the visual representation of city resources.

As a case study, we focus on the *Data Collector* (DC) microservice, as it provides a suitable example to explore self-distribution and fine-grain resource allocation in the context of microservices. We use DC to demonstrate the approach and evaluate the performance improvements that it enables as well as the associated costs.

For this purpose, we re-implemented DC using the dynamic component model of the Dana programming language. This new version has the same features of the original DC, which are provided in terms of four main functionalities:

- Historical Data (HD) – provides access to all historical data stored in the database;
- Historical Resource Data (HRD) – provides access to historical data of a specific resource;
- Most Recent Data (MR) – provides access to the latest data collected from all the existing resources; and
- Most Recent Resource Data (MRR) – provides access to the latest data collected from a specific resource.

Each of these functionalities is realized as a separate component, which can be relocated, replicated and scaled, thus enabling several different composition variants of the system.

In addition to functional components, a microservice composition may also employ components that implement non-functional properties. For DC, we define non-functional components that implement two different caching strategies, *CacheLRUbasic* and *CacheLRUextended*, which differ in terms of the amount of memory provided to the component, thus enabling tuning of resource allocation at the service level. The composition of a microservice can thus be adapted in a straightforward way by adding, replacing or removing these components at runtime as the environment of the microservice (e.g., workload and resource availability) changes.

We explore three types of microservice composition in the evaluation, as shown in Figure 2.

The *Local* composition is the one deployed when the system starts. It consists in all the components of the microservice residing in the same pod, thus sharing the same resources. Each of the four microservice components has an attached *cache* component, which can cache a certain amount of request responses to avoid the cost of remote database queries.

The second composition, called *Extended*, differs from the first one by changing the cache component attached to each functional component of the microservice. The new cache provides more memory, thus being able to store a larger amount of request responses in memory, boosting system performance by further avoiding remote calls to the database.

Finally, the *Distributed* composition results from the relocation of one of the microservice components to another pod. This allows us to experiment with two different cases: (a) relocation of a heavily used component, giving it its own exclusive resources and relieving the system during heavy workload periods; and (b) relocation of the least used microservice component, moving it to another pod and thus freeing up resources for the remaining components.

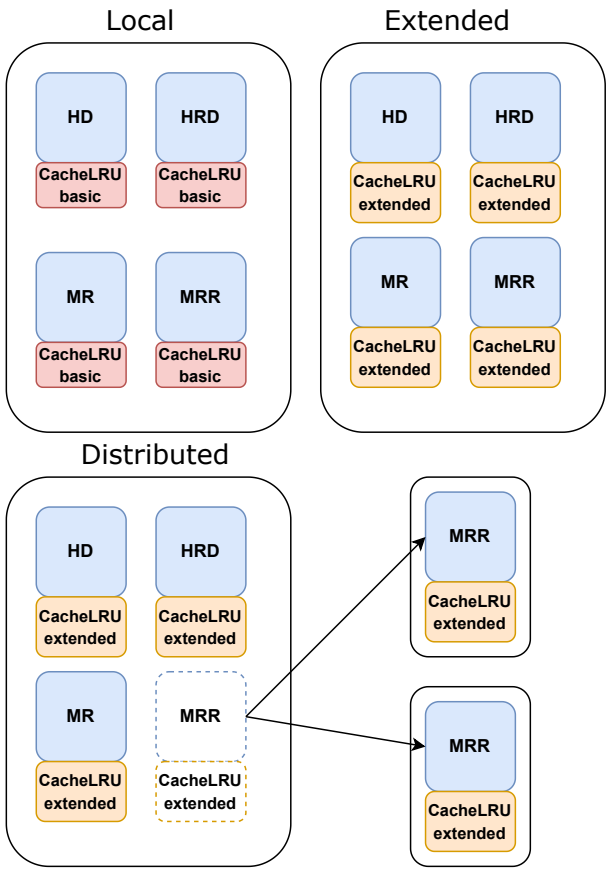

Fig. 2. Three different compositions of the Data Collector microservice

B. Workload and evaluation metric

We created a synthetic workload to evaluate the proposed microservice compositions and the performance impact of adapting between them. The workload consists of a sequence of requests to one of the components of the DC microservice, in this case MRR, to query the resources of a simulated smart city infrastructure. It simulates a *high-entropy-low-volume* workload pattern (i.e., small and rarely repeated requests) in order to highlight the need for a larger cache for the component.

We measure the response time of each request as the microservice is exposed to the workload. The measurements are taken at three different times: before, during and after adaptation. The aim is to evaluate the performance of both the current and the new composition (thus showing the end result of adaptation), as well as the (temporary) performance impact while adaptation itself is taking place. Using this evaluation approach, we run separate experiments, considering the different microservice compositions shown in Figure 2.

C. Local-to-Extended adaptation

In this section we analyze the transition from the *Local* to the *Extended* composition. In the *Local* composition, the Cache components have a cache size of 1KB, which is enough to store the response from a single database request made by each of the microservice's functional components.

The microservice then undergoes an adaptation that replaces the cache component with a larger one, which provides the functional components with 150KB of caching memory each. We explore this specific adaptation to show the ability of the system to perform vertical scaling within a microservice. Through the replacement of some of its components, we can manage the resources available to the microservice without having to replace the microservice itself. Although more conventional approaches exist for microservice resource management, they generally entail the addition or removal of resources for the entire microservice (e.g., at the container level). Our approach, on the other hand, enables greater precision, by handling resource allocation at a finer granularity.

Figure 3 depicts the response times obtained when running the workload through the microservice before, during, and after the adaptation. As expected, there is a performance gain when running the microservice with the new composition. The extra resources allocated to the component that handles the database requests allows more responses to be cached, thus requiring less remote calls to the database.

The graph in Figure 3 also brings attention to the cost of the adaptation process. While the system is adapting, which takes nearly 5 seconds, there is a marked increase in response time for all requests made to the microservice. Thus, despite the fact that vertical scaling of the microservice brings advantages of itself, there's a trade off to be expected, meaning that performing adaptations too often may actually compromise the overall performance. Note, however, that despite the temporary decrease in performance, the microservice remains fully functional during this kind of adaptation.

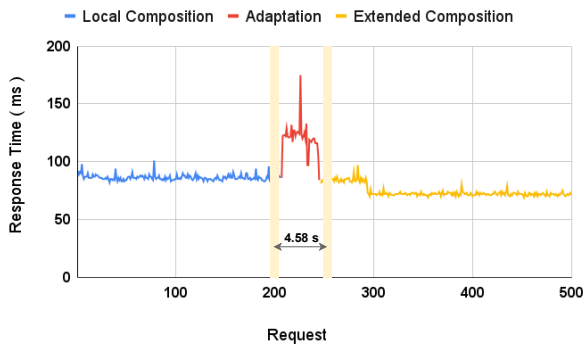

Fig. 3. Client-side response time of the DC microservice, showing the performance of the *Local* and *Extended* compositions, as well as during adaptation from the former to the latter.

D. Local to Distributed reconfiguration

The second kind of adaptation that we explore is the transition from the *Local* to the *Distributed* composition. As we start the system with all the components of the microservice sharing resources in the same pod, we explore the concept of horizontal scaling by taking advantage of the elastic environment of the cloud. While there are existing tools to perform this task, like the Horizontal Pod Autoscaler (HPA), they work by replicating the entire microservice that is located in the pod. We explore a different approach to horizontal scaling, by managing the microservice as several components instead of a single block. As the workload is focused on requests for individual components of the microservice, we may adapt the microservice by taking a less used component and relocating it to another pod. This leaves the resources of the pod for the components that are currently the most used.

The graph in Figure 4 shows the performance of the microservice before, during, and after such adaptation. Similarly to the previous experiment, we see an expected improvement in performance after the transition to the *Distributed* composition. Relocating a component frees up resources for the components that remain in the pod. We also see an increase in the duration of the adaptation phase to almost 40 seconds. This is due to the remote interactions required to perform this kind of adaptation. In particular, the *Distributor* module has to connect to the GKE, through the ServiceCTL API, and make requests for the creation of the required pods in the cloud. Thus, the time that Kubernetes takes to create new pods, as well as the communication time between the modules of the architecture, add up to the adaptation time.

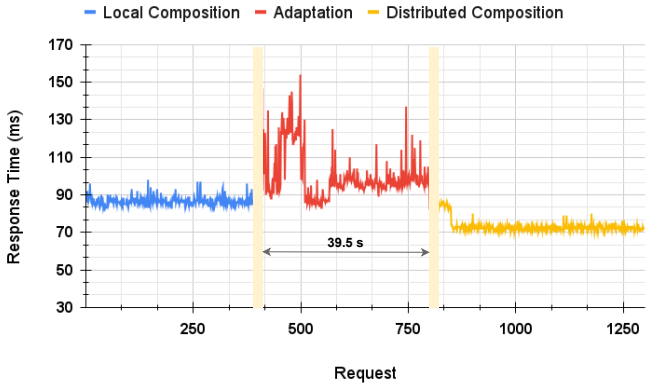

Fig. 4. Client-side response time during the transition from the *Local* to the *Distributed* by relocating the least-used component of the microservice.

We also explore a second case of adaptation from the *Local* to the *Distributed* composition. As queries to the DC microservice are undemanding, we artificially increase the database response time, so that we can better compare compositions that involve the distribution of a component handling high request loads. In this experiment we relocate a busy component of the microservice. While the workload is being handled by the MRR component, the *Distributor* module requests the

creation of an additional pod to relocate the component to. As Figure 5 shows, a marked difference from the previous experiment is that during the adaptation process the system now experiences a performance disruption. While the overall process takes approximately the same time, we see that the component processing the requests is put on hold during most of the time, i.e., from the moment the extra pod is created until the component is properly instantiated and restarted in the new location. Note that the x-axis refers to requests (instead of time), and that the peak at the end of the adaptation process, which refers to a single request, actually takes most of the total adaptation time (approximately 30s). Nevertheless, there is a significant performance improvement after the adaptation.

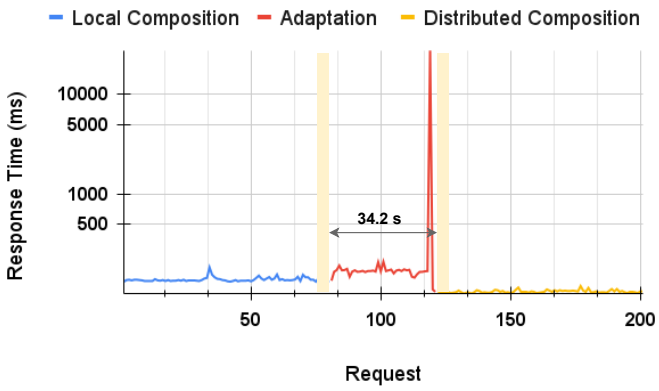

Fig. 5. Client-side response time for changing from the *Local* to the *Distributed* composition, with relocation of the most used component.

V. Conclusion

In this paper we aimed to demonstrate the potential of combining the concepts of emergent microservices and self-distributing systems. Based on the use of a dynamic component model to break a microservice down into smaller parts, our approach supports runtime adaptation of microservices at a fine-grain level. By changing the internal composition of a microservice and exploiting both local and distributed compositions in an elastic cloud environment, the approach enables performance improvements and the management of resource usage among the components.

We demonstrate and evaluate the approach using a case study and experiments to explore different adaptation scenarios. Although the results show the benefits of runtime microservice adaptation at a fine granularity, they also highlight some limitations. While the system is capable of adapting itself at runtime by seamlessly changing the internal composition of microservices, each adaptation has a performance cost, which must be taken into account in relation to the potential performance improvements. In particular, compositions that require the relocation of components that are in active use have a higher cost in terms of performance, as the component is put on hold during most of the adaptation process.

In future work, we will explore adaptations that involve more general composition patterns, such as by relocating and/or replicating different combinations of parts of the microservice. We also aim to explore the learning capabilities of the ESS framework [9] to enable exploration of the search space and selection of the best composition in a fully autonomic way. Lastly, we aim to explore different types of microservices (e.g., CPU-intensive, memory intensive) and scenarios in which the approach can have the highest positive impact.

VI. Acknowledgments

Renato S. Dias and Roberto Rodrigues-Filho would like to thank FAPESP for supporting their work under grants 2021/06425-0 and 2020/07193-2, respectively. During the development of this work, Roberto Rodrigues-Filho was a postdoctoral researcher in the Institute of Informatics at the Federal University of Goiás (INF/UFG). Finally, this research is part of the INCT of the Future Internet for Smart Cities funded by CNPq grant 465446/2014-0, CAPES grant 88887.136422/2017-00, and FAPESP grants 14/50937-1 and 15/24485-9.

References

[1] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 645–650, USA, 2018. USENIX Association.

[2] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis. Adaptive microservice scaling for elastic applications. *IEEE Internet of Things Journal*, 7(5):4195–4202, 2020.

[3] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: How to make your application scale. In Alexander K. Petrenko and Andrei Voronkov, editors, *Perspectives of System Informatics*, pages 95–104, Cham, 2018. Springer International Publishing.

[4] Roberto Rodrigues Filho, Marcio Pereira de Sá, Barry Porter, and Fábio M. Costa. Towards emergent microservices for client-tailored design. In *Proceedings of the 19th Workshop on Adaptive and Reflexive Middleware*, ARM '18, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. *Self-Reconfiguring Microservices*, pages 194–210. Springer International Publishing, Cham, 2016.

[6] Arthur M. Del Esposte., Fabio Kon., Fabio M. Costa., and Nelson Lago. Interscity: A scalable microservice-based open source platform for smart cities. In *Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems - SMARTGREENS,*, pages 35–46. INSTICC, SciTePress, 2017.

[7] Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, June 2017.

[8] Barry Porter. Runtime modularity in complex structures: A component model for fine grained runtime adaptation. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*, CBSE '14, page 29–34, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David 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)*, pages 333–348, Savannah, GA, November 2016. USENIX Association.

[10] Roberto Rodrigues-Filho and Barry Porter. Hatch: Self-distributing systems for data centers. *Future Generation Computer Systems*, 132:80–92, 2022.

[11] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Hierarchical scaling of microservices in kubernetes. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 28–37, 2020.