An Interface Definition Language for Supporting Stub Generation in Self-distributing Systems

Arthur Barata*, Roberto Rodrigues-Filho†, Carlos A. Astudillo* and Luiz F. Bittencourt*
*Instituto de Computação (IC), Universidade Estadual de Campinas (UNICAMP), Campinas, SP, Brasil
† Departamento de Ciência da Computação (CIC), Universidade de Brasília (UnB), Brasília, DF, Brasil
a221738@dac.unicamp.br, roberto.filho@unb.br, {castudillo, bit}@unicamp.br

***Abstract*—Programming software systems for the computing continuum is not an easy task. Engineers have to account for operating environment volatility and high levels of heterogeneity while optimally exploiting the underlying computing resources. As a response to that, the concept of Self-distributing System (SDS) was introduced. The SDS enables the creation of local systems using a component-based approach, with intelligent algorithms responsible for dynamically distributing components across a distributed infrastructure at runtime and with minimum human intervention, using a distribution technique that involves replacing local components with proxies that act as Remote Procedure Calls (RPC) stubs, forwarding incoming calls to relocated components running on external machines. However, this programming paradigm requires developers to manually create distribution proxies after system components are developed, adding an extra burden for software engineers by requiring development time and knowledge about distribution techniques/infrastructure. To address this challenge, this work introduces an Interface Definition Language (IDL) that supports the generation of stubs for self-distributing systems. We evaluate our approach through a matrix multiplication use case, where a single local instance of a matrix multiplication program autonomously distributes itself via automatically created proxies and dynamically scales the system in response to an increase in the volume of requests.**

***Index Terms*—self-distributed systems, computing continuum, interface definition language.**

I. Introduction

The computing continuum [1] refers to the infrastructure composed of multiple, distinct computing resources, ranging from edge devices and fog computing servers to cloud computing platforms. This infrastructure is increasingly serving as the foundation for deploying modern software systems such as edge AI applications [2], with its devices displaying different characteristics: edge devices have limited resources and lower dependability but offer low network latency, while cloud servers provide higher reliability and abundant resources at the cost of increased network latency. Given these trade-offs, it is crucial to design adaptive systems capable of dynamically migrating execution across different layers of the continuum to optimize performance and resource utilization.

Building software for the computing continuum presents several challenges [1], with the most significant being the dynamic migration of services across different computing layers. In order to exploit the continuum infrastructure, services should be able to relocate execution from cloud to servers positioned closer to end-users, enabling the services to leverage the available computing resources more efficiently. This implies that developing such software requires addressing device heterogeneity, service state management, and—most critically—the inherent complexities of distributed computing, such as consistency, fault tolerance, and latency constraints [1]–[5].

At the platform level, software is often deployed on container-orchestrated platforms such as Kubernetes, Apache Mesos, and OpenFaaS, which automate the container lifecycle management. These platforms handle the creation, execution, scaling, and termination of containers while dynamically allocating resources across cloud, fog, and edge nodes. Furthermore, advanced scheduling mechanisms enable adaptive workload placement, ensuring that computation occurs closer to the data source when low latency is needed or in the cloud when more computational power is required. Additionally, service migration capabilities allow seamless container relocation between devices, enabling code mobility across the computing continuum.

At the service level, different software architectures and programming models have been adopted to develop systems that efficiently operate within the edge-cloud continuum, with stateless service architectures, microservices and Function-as-a-Service (FaaS) paradigms among the most used technologies to explore this infrastructure [6]. Most architectural patterns in the continuum consist of components that are distributed and managed across computational nodes. By building these components as functions, in the case of FaaS, or as services, in the case of microservices, developers are allowed to design applications that automatically scale, execute, and relocate as needed, reducing infrastructure overhead. However, the inherent stateless nature of these services requires external state management, typically via distributed databases (e.g., Redis, Cassandra) or object storage (e.g., MinIO, S3). In this context, isolating the state allows services to be dynamically deployed, scaled, and migrated across the continuum, leveraging container-managed platforms for optimized service placement and efficient resource utilization.

Developing software for the edge-cloud continuum requires an engineering team capable of addressing both platform- and service-level concerns, raising the application complexity when it comes to designing systems that can dynamically exploit service mobility, in order to adapt to workload fluctuations while making efficient use of computing and network resources. To address the challenging and cumbersome task of creating such systems, the concept of Self-Distributing Systems (SDS) [3], [7] arise, bringing a new architectural paradigm that enables software initially designed for single instance execution—i.e., within a single process—to autonomously distribute its components across a distributed infrastructure at runtime as resource demands increase. By abstracting away the complexity of distributed system design, this paradigm allows engineers to focus solely on developing locally interacting components, without the burden of manually defining effective distribution strategies.

SDS relies on component-based models [8]–[10], where software is structured into modular components that can be hot-swapped at runtime. When additional resources are required, SDS replaces certain components with RPC stubs that intercept local function calls and redirect them to remotely instantiated versions of the original components on external machines [7], [11], allowing the software to dynamically evolve from a single instance execution model to a distributed architecture, leveraging available infrastructure without predefined distribution rules at design time. In summary, the SDS explores and adapts its architecture at runtime by integrating component stubs, effectively transforming into a self-organizing distributed system.

Although the SDS paradigm aligns well with the demands of software development for the computing continuum, its current implementation still requires engineers to not only design single instance executing components but also manually create distribution stub-proxies. This process introduces an additional burden, as developers must explicitly implement proxies that forward incoming local function calls to remote instances, requiring developers to increase development time, implement distribution techniques, and exploit information about the infrastructure.

To address this challenge, we propose an *Interface Definition Language* (IDL) to support the generation of stubs, allowing developers to focus solely on implementing the single instance system without dealing with the complexities of manual stub creation. To demonstrate the IDL, we present a Matrix Multiplication (MatMul) example, where the developer, concentrating only on the single instance implementation, can still produce software capable of dynamic self-organization in a distributed environment. By simply annotating the MatMUl interface using our IDL, the system can autonomously scale and distribute computations to accommodate increasing request volumes.

II. Background

This section introduces the fundamental concepts underlying our proposed approach. We begin with an overview of the Edge-Cloud Continuum, highlighting its key characteristics and its role in enabling dynamic and distributed execution. Next, we present the Dana component-based model, which serves as a foundation for building highly adaptive systems. Finally, we introduce the Self-Distributing Systems (SDS) paradigm, which we adopt as the programming model for software development for the computing continuum.

A. The Edge-Cloud Continuum

Modern applications such as mobile code offloading [12], [13] are evolving toward hybrid environments, placing computing nodes between the user and the server to process data as it travels between the two ends of the network, and leveraging geographical proximity to provide low latency and processing flexibility [3] as illustrated in Figure 1, which portrays a layered architecture with the edge closer to the user, the cloud further away, and the fog composed of the intermediate layers in between. In this way, the infrastructure provides multiple devices to compute applications between the client and the cloud, with each device acting as a computational node exhibiting different characteristics with varying processing capabilities, making it computationally heterogeneous.

With advances in computational capacity, the nodes in the computing continuum environment can provide the necessary processing power to run applications or their components, expanding the possibilities for system arrangements to support different types of computational demands. The experiment of this article uses a continuum infrastructure simulation to measure the efficiency of the generated stubs, in order to provide more information about the benefits of this paper approach on the continuum usage.

B. Self-distributing Systems (SDS)

Self-adaptive systems can modify their behavior and/or structure in response to their perception of the operating environment, the system's state, and requirements [4]. This definition includes the system's ability to modify its components and architecture at runtime to improve performance in the environment where it is running. Thus, self-distributed systems are those capable of modifying their own architecture, adapting to their environment and context by distributing their parts, without being programmed to do so, generating distribution transparency at the application level.

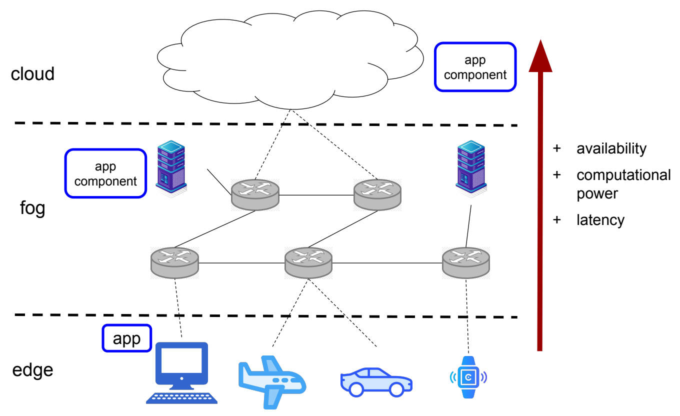

Fig. 1: Computing continuum infrastructure.

In this paper, SDS are used in order to show the benefits of this technology in environments such as the computing continuum, as it can be seen on Figure 1, where the application's components are placed in different layers of the infrastructure, with the application exploiting them to improve its performance, and, in the end, this paper provide a demonstration of the benefits of distributing systems at runtime.

C. Dana Component-based Model

The Dana programming language [8] emerges as a component based programming model, focusing on seamless adaptation by embedding the component with its definition directly into its programming model, leveraging abstractions to build solutions where components can be programmatically replaced by others that implement the same definition, ensuring runtime adaptation, and allowing the creation of new types of applications such as the SDS [11]. In Dana's programming model, the definitions are provided by the interface while the implementations are provided by the component. Interfaces define data types, methods, constants, and states, while components implement the methods and provide the code that will create the component's rules.

To generate SDS in Dana it is needed to manually create a new component that will act like a stub-proxy and distributed an applications part, to do this the new component need to implement the same definition of its counterpart, also worrying about distribution methods, and handle state distribution- if needed-, introducing an additional burden to developers since they must know information about the infrastructure and how they will use distribution on it. To address this problem, this article proposes an extension to Dana's definitions by creating a new definition language, in order to generate stub components effortlessly.

III. Related work

In this section, we review the most relevant related work, focusing on three key research directions: (i) studies that employ Self-Distributing Systems (SDS) as a programming model for the computing continuum, (ii) research on programming services to address the demands of the computing continuum, and (iii) work on Interface Definition Languages (IDLs) designed to support Remote Procedure Calls (RPCs) in distributed object models.

Previous research on SDS has introduced and explored the concept in different contexts, including data centers [7] and the edge-cloud continuum [3]. The concept has gradually evolved in the literature, with studies examining distinct aspects of its construction [4], [5]. For instance, [7] introduces the concept, provides an overview of SDS, and investigates its application in data center software, using a self-distributing web server as a case study. In [3], the authors extend the concept to the computing continuum, presenting an early SDS prototype evaluated in multiple scenarios, such as energy optimization, performance optimization, and a comparative analysis of SDS and FaaS.

Continuing the progressive development of SDS, this work takes an incremental approach by advancing a specific, yet unexplored, aspect of SDS. While previous studies have focused on SDS applications and optimizations, this work specifically addresses the automation of distribution stub generation, a critical component that has remained manually constructed, thus generating extra complexity in system development. By eliminating the need for developers to manually implement distribution proxies, our approach reduces complexity in the development process, making SDS more accessible and easier to design for the computing continuum.

A key aspect of programming the continuum is the creation of services that are able to move across the heterogeneous infrastructure. With that in mind, code offloading techniques present themselves as a useful solution, since they distribute data and processing across edge devices, providing manually programmed solutions to exploit the continuum [12]. Despite code offloading techniques, cloud-based architectures such as FaaS and microservices demonstrated a great fit, since they were built to work in distributed and heterogeneous infrastructures [6], working as stateless applications or relying on other services to persist applications' state, which diminishes applications' performance. But all presented solutions require the programmer to design systems based on the infrastructure provided, demanding knowledge about system distribution, and demanding different experiments to improve performance. Our approach stands out due to the effortless creation of applications that are capable of adapting to the infrastructure, and change their distribution at runtime without affecting the execution.

Finally, despite the idea of joining IDL with RPC mechanisms emerged around the year 2000 [14], part of recent studies focus on defining configuration without necessarily creating a new definition language [15], [16]. In this work, the definition language for creating RPC proxies is used as solution, but its principles and fundamentals are easily expanded to other annotation methods.

IV. Dana Interface Definition Language (DIDL)

The DIDL is designed to describe components that can be replaced to distribute the processing and/or the data, and to accomplish this task, it is crucial to have knowledge about the components' context and metadata related to their implementation. We achieve this by annotating the metadata in the application [1], and we chose to follow common industry standards by providing the annotation language in *JSON* format. Since Dana already has a place to describe component definitions, we placed annotation files alongside the interface definition files.

With the aim of acquiring the metadata needed, we propose a stub generator that scans the folder with the interface definitions [8] to load all the files that have the *.didl* extension, then it generates each new component independently, building a stub based on five main criteria as the minimum metadata

[1]Repository: https://github.com/baratarthur/distributed-matmul

needed: the output folder of the stub component, the component implementation file, the component dependencies, the address of its remote parts, and the component methods.

```
 1 {
 2     "outputFolder": "./matmul",
 3     "componentFile": "./matmul/Matmul.dn",
 4     "dependencies": [
 5         {
 6             "lib": "rpc.RPCUtil",
 7             "alias": "connection"
 8         }
 9     ],
10     "remotes": [
11         {
12             "address": "10.5.0.3",
13             "port": 8081
14         },
15         {
16             "address": "10.5.0.4",
17             "port": 8082
18         }
19     ],
20     "methods": {
21         "multiply": {
22             "returnType": "Matrix",
23             "strategy": "distribute",
24             "returnParser": "parser({})",
25             "parameters": [
26                 {
27                     "name": "A",
28                     "type": "Matrix"
29                 },
30                 {
31                     "name": "B",
32                     "type": "Matrix"
33                 }
34             ]
35         }
36     },
37 }
```

Fig. 2: JSON-based syntax of DIDL for defining a Matrix Multiplication interface, which is subsequently used as a use case in the evaluation.

Figure 2 illustrates a DIDL file with two types of metadata: lines 20 to 36 show the metadata related to the component implementation, such as the method type or method name; and lines 2 to 19 show the metadata related to the stub implementation, such as the stub component dependencies, the remote component addresses, and the output file of the component stub. This metadata is used by the stub to generate distribution methods that balance the workload among the remote component implementations. The stub generator can be extended to support different distribution algorithms in the same code piece, with the aim of generating different types of stub components that should fit into different scenarios, turning the final systems into an aggregation of smaller and independent parts.

After collecting all the metadata, the stub generator creates new component implementation files following the interface definitions (see Section II-C) and based on the principles depicted in Figure 3. The stub component workflow shown in Figure 3 presents 4 layers that illustrate the stub architecture, a package layer, two transmission layers, and a processing layer; notice that steps 1 and 8 refer to the application interaction with the stub component, where they represent the application method call – with or without parameters – and method data return, respectively. Steps 2 and 7 show the packing and unpacking of the data in the stub, which may include packing together the function parameters of step 1 with the metadata needed to execute the RPC call. Notice that, differently from steps 7 and 8, the layer 2 cannot be skipped because of the metadata that is added to the request bundle, turning it into an essential layer even if the method has no parameters to pack. Steps 3 and 6 present the data exchange between the stub and the remote component, where the stub component can implement different distribution methods to balance the workload among the available remote components. Finally, steps 4 and 5 represent the method execution on the remote component, also packaging the information before the transmission step.

In strongly typed languages, such as Dana, steps 2, 4, 5, and 7 need to provide data types for data bundling, where these data types will be the same in both remote and stub components, due to the fact that they execute the same code pieces. In these layers, the stub generation engine searches for the method name followed by a specific keyword format, such as *ParamsFormat*, relying on data parsing methods provided on the DIDL to successfully complete the packaging.

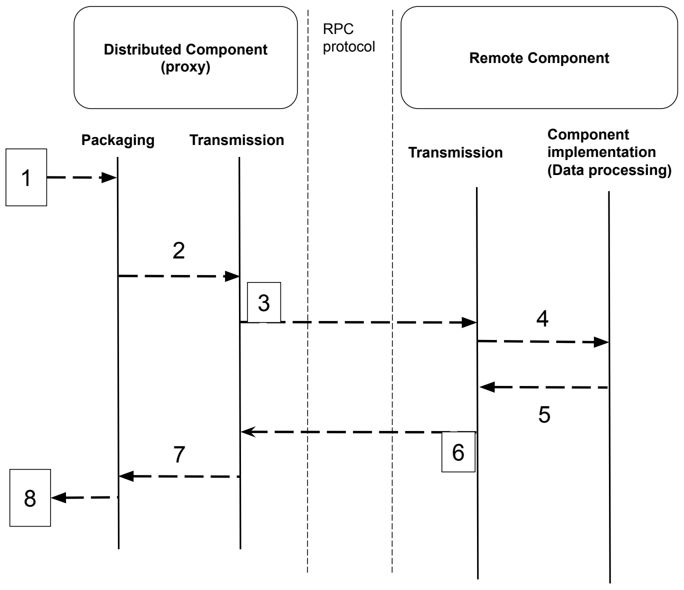
Distributed Component (proxy)
RPC protocol
Remote Component
Packaging
Transmission
Transmission
Component implementation (Data processing)
1
2
3
4
5
6
7
8

Fig. 3: Stub component workflow.

V. Evaluation

We aim to demonstrate the advantages of our approach by comparing two versions of a stateless Matrix Multiplication (MatMul) application. Both versions consist of the same implementation of MatMul coded to execute within a single process. In the first version, we implement MatMul using the SDS stack, while in the second version, we implement it as a traditional monolithic (single-process) service. Our objective is to measure the benefits of autonomously distributing the SDS-based MatMul across a distributed infrastructure and compare its performance against the conventional monolithic implementation.

Both versions of MatMul were implemented in Dana [8], the foundational language of the SDS approach. By using the same language for both the SDS-based and monolithic implementations, we ensure a fair performance comparison, eliminating any bias introduced by language-specific optimizations. Additionally, by conducting this experiment, we verify that the SDS stub, generated automatically from DIDL, functions correctly, enabling the SDS-based MatMul to distribute itself across an infrastructure, even when the system was not explicitly programmed to do so.

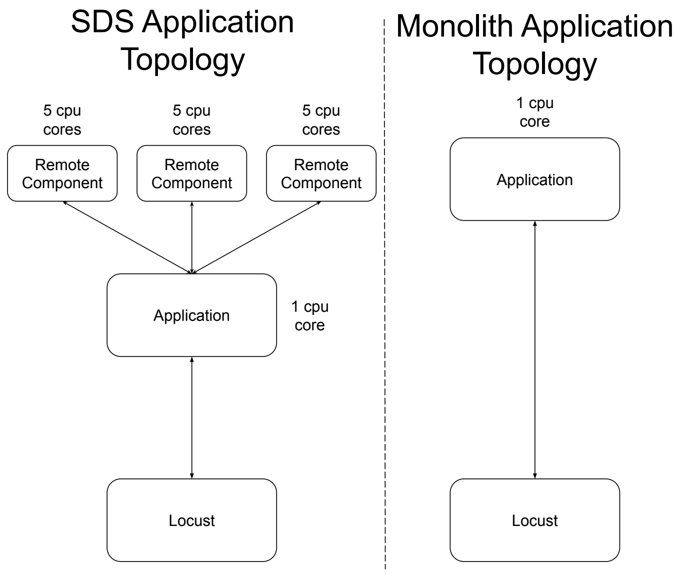

Fig. 4: Experiment topology for the MatMul application. The left side depicts the distributed composition, while the right side shows the single-process deployment.

The experiment involves a client and three machines, as illustrated in Figure 4. The left side of the figure depicts the distributed MatMul, achievable only at runtime using our SDS-based approach, while the right side shows the single-process MatMul, which both our implementations support.

To reflect the computing continuum, we deployed the left-side topology with the application on the node closer to the client (simulating the edge), forwarding requests to the remote nodes (illustrating the cloud) for execution. While this introduces network latency due to the cloud-edge distance, it leverages the cloud's computational resources more effectively. This architecture is ideal for scenarios with high request intensity, where parallel workloads demand scalable resources.

Conversely, the right-side topology (single-process) is optimized for low-volume requests, avoiding unnecessary cloud latency when minimal computing resources suffice. Our experiment shows that the SDS-based MatMul dynamically adapts between these topologies, selecting the optimal deployment—edge-only or edge-cloud-based, depending on request intensity and resource demands.

We used Mininet [2] to emulate the network infrastructure [17], with Locust [3] for stress testing. During the experiment, simulated users sent requests to the MatMul application, starting at 5 concurrent requests and incrementally increasing by 5 until reaching 50 simultaneous requests per second. We collected the volume of requests each version of the MatMul was able to handle and the average response time during the experiment. The results are depicted in Figure 5, Figure 6, and Figure 7. All experiments were executed on an Intel Xeon Silver 4210R CPU (2.40GHz) with 257GB of RAM.

The experiment depicted in Figure 5 simulated 5 requests per second and measured the response rate, tracking how many requests received replies within the test script's timeout period. As shown in Figure 5, the distributed application dynamically scaled its capacity to handle increasing loads. Although both implementations initially deployed MatMul as a single-process service, the SDS-based version autonomously adapted at runtime to leverage additional cloud resources. This optimization nearly doubled its throughput, demonstrating superior scalability under load compared to the static monolithic version.

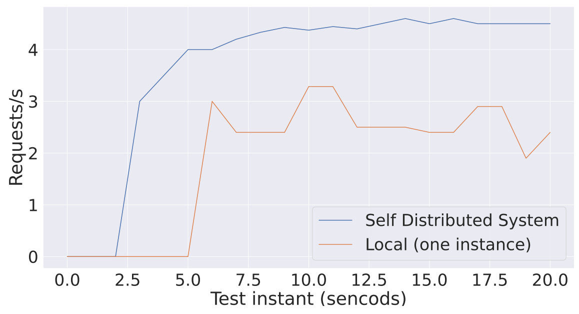

Fig. 5: Volume of requests (# of reqs/s) handled by the monolithic application (orange line) and the SDS (blue line) during the experiment.

Figure 6 shows the adaptation impact on the response time of each request, where, in the beginning of the test – around the instant 0 to 10 –, the adaptation reduced the latency of the responses; later, as the number of requests increases, the adaptation of the system decreased response latency while improved the ability of the application to handle more parallel requests over time, showing a peak with still lower response time than monolithic version of MatMul.

Finally, we look into the median response time of each second of the experiment (Figure 7). We show that SDS has lower response time and a smoother curve evolution when compared to the monolithic MatMul, demonstrating SDS's ability to handle higher request volumes when adapting from single-process to a distributed deployment.

[2]http://mininet.org/
[3]https://locust.io/

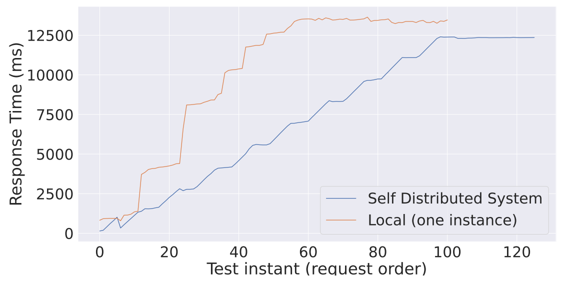

Fig. 6: Average response time (ms) for the monolithic application (orange line) and SDS (blue line) considering each request.

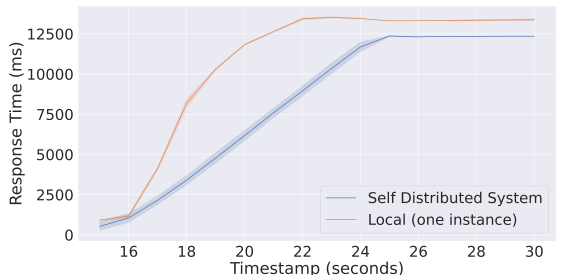

Fig. 7: Average response time in milliseconds, sampled every second, for the monolithic application (orange) and SDS (blue)

VI. Final Remarks

Creating distributed applications requires significant effort [15], [18], specially considering heterogeneous and dynamic environments such as the computing continuum. Therefore, having a description language that distribute the application automatically reduces this effort, taking away the programmer's responsibility of thinking about how to distribute the application, improving development productivity and enabling the advancement of new types of self-managed applications.

Preliminary results presented in this paper suggest that SDS can outperform single instance applications demonstrating more computational capability and lower latency, specially when considering a computing continuum environment.

Despite all the benefits of this new technique, there are future research avenues that can improve the process described on this paper. We consider as future work expanding the collection of distribution algorithms of the stub component. In order to cover different distribution scenarios, it is also possible to enhance the application performance by improving the adaptation trigger of the SDS. Extend the evaluation scenarios to comprise other setups, with heterogeneous machines capabilities and different use of remote components, its also considered as future work.

VII. Acknowledgements

This research is part of the INCT of Intelligent Communications Networks and the Internet of Things (ICoNIoT) funded by CNPq (proc. 405940/2022-0) and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) Finance Code 88887.954253/2024-00.

References

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

[2] D. Rosendo, A. Costan, P. Valduriez, and G. Antoniu, Journal of Parallel and Distributed Computing," "Distributed intelligence on the edge-to-cloud continuum: A systematic literature review, vol. 166, pp. 71–94, 2022.

[3] R. Rodrigues Filho, R. S. Dias, J. Seródio, B. Porter, F. M. Costa, E. Borin, and L. F. Bittencourt, 2023 32nd International Conference on Computer Communications and Networks (ICCCN)," in "A self-distributing system framework for the computing continuum. IEEE, 2023, pp. 1–10.

[4] et al.R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel , Software Engineering for Self-Adaptive Systems II," in "Software engineering for self-adaptive systems: A second research roadmap. Springer, 2013, pp. 1–32.

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

[6] F. Tusa, S. Clayman, A. Buzachis, and M. Fazio, Future Generation Computer Systems," "Microservices and serverless functions—lifecycle, performance, and resource utilisation of edge based real-time iot analytics, vol. 155, pp. 204–218, 2024.

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

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

[9] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, ACM Transactions on Computer Systems (TOCS)," "A generic component model for building systems software, vol. 26, no. 1, pp. 1–42, 2008.

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

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

[12] A. M. Rahmani, M. Mohammadi, A. H. Mohammed, S. H. T. Karim, M. K. Majeed, M. Masdari, and M. Hosseinzadeh, Wireless Personal Communications," "Towards data and computation offloading in mobile cloud computing: taxonomy, overview, and future directions, vol. 119, pp. 147–185, 2021.

[13] O. Gheibi, D. Weyns, and F. Quin, ACM Transactions on Autonomous and Adaptive Systems (TAAS)," "Applying machine learning in self-adaptive systems: A systematic literature review, vol. 15, no. 3, pp. 1–37, 2021.

[14] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, ACM SIGCOMM Computer Communication Review," "A survey of programmable networks, vol. 29, no. 2, pp. 7–23, 1999.

[15] A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano, arXiv preprint arXiv:2101.01159," "New directions in cloud programming, 2021.

[16] L. De Martini, D. d'Abate, A. Margara, and G. Cugola, arXiv preprint arXiv:2503.15199," "Radon: a programming model and platform for computing continuum systems, 2025.

[17] B. Lantz, B. Heller, and N. McKeown, Proceedings of the 9th ACM Workshop on Hot Topics in Networks (HotNets-IX)," in "A network in a laptop: Rapid prototyping for software-defined networks, 2010.

[18] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, arXiv preprint arXiv:1812.03651," "Serverless computing: One step forward, two steps back, 2018.