IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 16, 2021, accepted August 5, 2021, date of publication August 12, 2021, date of current version August 23, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3104475

Towards Edge Computing as a Service: Dynamic Formation of the Micro Data-Centers

MILOŠ SIMIĆ, IVAN PROKIĆ, JOVANA DEDEIĆ, GORAN SLADIĆ, AND BRANKO MILOSAVLJEVIĆ

Faculty of Technical Sciences, University of Novi Sad, 21000 Novi Sad, Serbia

Corresponding author: Miloš Simić (milos.simic@uns.ac.rs)

ABSTRACT Edge computing brings cloud services closer to the edge of the network, where data originates, and dramatically reduces the network latency of the cloud. It is a bridge linking clouds and users making the foundation for novel interconnected applications. However, edge computing still faces many challenges like remote configuration, well-defined native applications model, and limited node capacity. It lacks geo-organization and a clear separation of concerns. As such edge computing is hard to be offered as a service for future real-time user-centric applications. This paper presents the dynamic organization of geo-distributed edge nodes into micro data-centers to cover any arbitrary area and expand capacity, availability, and reliability. A cloud organization is used as an influence with adaptations for a different environment, and a model for edge applications utilizing these adaptations is presented. It is argued that the presented model can be integrated into existing solutions or used as a base for the development of future systems. Furthermore, a clear separation of concerns is given for the proposed model. With the separation of concerns setup, edge-native applications model, and a unified node organization, we are moving towards the idea of edge computing as a service, like any other utility in cloud computing.

INDEX TERMS Cloud computing, distributed systems, edge computing, formal specifications, infrastructure as software, platform.

I. INTRODUCTION

Over the past decade, computation and data volumes have increased significantly [1]. Augmented reality, online gaming, autonomous vehicles, or the Internet of Things (IoT) applications produce huge volumes of data. Such workloads require latency below a few tens of milliseconds, which a centralized model like the cloud cannot offer [1]. Cloud computing (CC) can be defined as the aggregation of computing resources as a utility and software as a service [2]. Hardware and software in big data-centers (DCs) provide services for user consumption over the internet [3]. Resources like CPU, GPU, storage, and network can be used as well as released on-demand as utility [4]. The key strength of the CC is offered services [2]. The traditional CC model provides enormous computing and storage resources elastically, to support the various applications needs. This property refers to the ability of the cloud to allow services allocation of additional resources or release unused ones to match the application workloads on-demand [5].

The associate editor coordinating the review of this manuscript and approving it for publication was Wenbing Zhao.

Data is required to be moved to the cloud from data sources, which introduces a high latency in the system [6]. For example, Boeing 787s generates half a terabyte of data per single flight, while a self-driving car generates two petabytes of data per single drive. Bandwidth is not large enough to support such requirements [7]. Data transfer is not the only problem: applications like self-driving cars, delivery drones, or power balancing in electric grids require real-time processing for proper decision-making [7]. Serious issues might be faced if a cloud service becomes unavailable due to a denial-of-service attack, network, or cloud failure [8].

Cloud centralized architecture with enormous DCs capacities creates an effective economy of scale to lower administration cost [9]. However, when such a system grows to its limits, centralization brings more problems than solutions [8], [10]. Despite all the CC benefits, applications and services face serious degradation over time due to the high bandwidth and latency [11]. This can have an enormous consequence on the business and potentially human lives as well. Organizations use cloud services to avoid huge investments [12], like creating and maintaining their own DCs. They consume

resources created by cloud providers [13] and pay for usage time – the ''pay as you go'', model.

To overcome cloud latency, research led to new computing areas like edge computing (EC). EC is a model in which computing and storage utilities are in proximity to data sources [13]. The cloud is enhanced with new ideas for future generation applications [14]. Over the years, designs like fog [15], cloudlets [12], and mobile edge computing (MEC) [16] emerged. In this paper, we refer to all these models as edge nodes. They all use the concept of data and computation offloading from the cloud closer to the ground [17], while heavy computation remains in the cloud because of resource availability [14]. EC models introduce small-scale servers (i.e., EC nodes), operating between data sources and the cloud. Typically, these EC nodes have much fewer capabilities compared to their cloud counterparts [18]. Aroca *et al.* investigated the capabilities of small devices, and they noticed that small ARM-based devices have a good performance for building servers and clusters, considering their performance per Watt relation [19]. These servers can be spread in base stations [16], coffee shops, or over geographic regions to avoid latency, as well as huge bandwidth [12]. They can serve as firewalls [20] and pre-processing tier, while users get a unique ability to dynamically and selectively control the information sent to the cloud.

EC models, on the other hand, lack dynamic geo-organization, well-defined native applications, and a clear separation of concerns (SoC) (i.e., a formal separation of the algorithms from special-purpose concerns [21]). SoC allows for modularity: a part of a system can be safely implemented or changed without need for detailed knowledge of the overall system and without affecting other parts of the system. All the above-named issues imply that the existing EC models cannot be offered as a service to the users. They usually exist independently from one another, scattered without communication between them, offered by providers who mostly lock users in their ecosystem. Co-located edge nodes should be organized locally, as micro [22], community or edge clouds [23] to help power-hungry servers reduce traffic. This cloud-like extension makes the whole system and applications more available and reliable, but also extend resources beyond the single node or group of nodes, maintaining good performance to build servers and clusters [19]. Not all companies and organizations will be able to deploy edge nodes [12]. EC nodes could be deployed by government authorities or large cloud companies [24] for their own needs, and the general public can use them with the familiar ''pay as you go'' model.

This paper contributes to CC with unique results in the literature that deepen and strengthen our understanding of CC as a whole. It proposes a formal model that can serve as a base of EC as a service model that will organize EC nodes into micro clouds dynamically, abstracting infrastructure to the level of software — infrastructure as software [25]. The presented model is inspired by the cloud architecture, with adaptations for a different environment. Two possible scenarios of model implementation could coexist: (*i*) a stand-alone implementation or (*ii*) an extension integration within existing tools, as a node organizer and register. Both implementations will enable reorganization of EC nodes dynamically as needed, allowing disposable micro cloud infrastructure due to the proposed model.

Our model expands peer-to-peer systems into new directions and blends them with the cloud to allow novel human-centered, cloud-like applications. This extension yields a model for EC applications utilizing these adaptations, and clear SoC for extended cloud model as a direct **implication**.

The main contributions of the paper are as follows:

(i) A formally correct and validated model able to maintain a record of available EC nodes registered in the system and their employment in some micro cloud, allowing recruiting more free nodes if required.

(ii) A formally correct and validated model that can organize EC nodes dynamically in a standard way based on cloud architecture, with adaptation for an EC geo-distributed environment. Give users the ability to reorganize nodes in the best possible way in some geographic areas to serve only the local population nearby. Such a model allows treating micro clouds disposable, allowing infrastructure to be abstracted to the level of software, creating infrastructure as software system.

(iii) A set of clear technical requirements that nodes must fulfill to join the platform. With these requirements, we can unify heterogeneous EC nodes in the same way, even allowing the inclusion of volunteer nodes if available infrastructure cannot support newly created load.

(iv) Describe how to offer EC as a service, like any other utility in the cloud. EC as a service could be offered to researchers and developers to create new human-centered applications, giving them the ability to do resource management. If the users need more resources on one side, they can take them from some other resource pool and move them to the place they need. Or, they can organize their resources in any other way that suits them best.

(v) Present bright SoC for the future system that will extend the standard CC model, in which every part will have an intuitive role and can complete a distinct job. This will make such a system easier to understand, modify and extend.

Based on the previously listed contributions, our model should be seen as an automated tool for failure where no micro cloud is irreplaceable. Such model, to the best of the authors' knowledge, has not been studied yet. The proposed model can exist as a stand-alone solution, the aiding component of existing orchestration engines, or part of the infrastructure for every cloud provider.

The rest of the paper is organized as follows: Section II discusses related work. The design and architecture of the system are formally presented in Section III. Section IV outlines an example of a high-level application for the

proposed model. For that purpose, we used the combination of area traffic control and healthcare applications during the COVID-19 outbreak in the city of Milan, Italy. Section V collects some concluding remarks and future directions of our research.

II. RELATED WORK

This section presents the relevant studies of the literature to our solution. The related work is summarized in two parts: (*A*) nodes organization and (*B*) platform model.

A. NODES ORGANIZATION

A zone-based organization for EC nodes presented by Guo *et al.* in [26], gives an interesting perspective on EC in the application of a smart vehicle. The authors showed how zone-based models enable continuity of dynamic services, and reduce the connection handovers. Also, they show how to enlarge the coverage of ESs to a bigger zone, thus expanding the computing power and storage capacity of ESs. Since one of the premises of EC is geo-distributed workloads, organizing ESs into zones and regions could potentially benefit EC.

Baktir *et al.* [27] explored the programming capabilities of software-defined networks (SDN). Findings show SDN can simplify the management of the network in a cloud-like environment. SDN is a good candidate for networking because it hides the complexity of the heterogeneous environment from the end-users. Kurniawan *et al.* [28] argue about very bad scalability in centralized delivery models like cloud content delivery networks (CDN). They proposed a decentralized solution using nano DCs as a network of gateways for internet services at home [28]. These DCs are equipped with some storage as well. Authors show a possible usage of nano DCs for some large-scale applications with much less energy consumption.

Ciobanu *et al.* [29] introduce an interesting idea called drop computing. The authors show that we can compose EC platforms ad-hoc, thus enabling collaborative computing dynamically, using a decentralized model over multilayered social crowd networks. Instead of sending requests to the cloud, drop computing employs the mobile crowd formed of nearby devices, hence enabling quick and efficient access to resources. The authors show an interesting idea of how to create a computing group ad-hoc. Creating ad-hoc platforms from crowd resources might raise a few possible concerns: (1) crowd nodes availabilities, and (2) offered resources. Crowd nodes might be a captivating idea as a backup option in cases we need more computing power or storage, and there are no more available resources to use.

Micro data-centers (MDCs) are an interesting model and area of rapid innovation and development. Greenberg *et al.* [30] introduce MDCs as DCs that operate in proximity to a big population (on contrary to nano DCs that serve a lot smaller population), thus minimizing the latency and costs for end-users [24], [30], and reducing the fixed costs of traditional DCs. The minimum size of a MDCs is defined by the needs of the local population [30], [31], with agility as a key feature. Agility here means the ability to dynamically grow and shrink resources and satisfy the demands and usage of resources from the most optimal location [30].

Different from the previously mentioned works, this study focuses on the descriptive dynamic organization of geo-distributed nodes over an arbitrary vast area that other solutions lack. To achieve such a task, the model here proposed is under the influence of the cloud computing organization but adapted for a different environment such as EC. This allows us to push the whole solution more towards edge computing as a service model like any other utility in the cloud.

B. PLATFORM MODEL

Kubernetes [32] is an open-source variant of Google orchestrator Borg [33]. All workloads end in the domain of one cluster [32]–[34]. Kubernetes is a promising solution for geo-distributed and EC environments due to its extensibility and existing tooling, but by design, Kubernetes operate in a completely different environment. On the other hand, some solutions show the Kubernetes can run in geo-distributed and EC environments. For example, Rossi *et al.* [34] focuses on adapting Kubernetes for geo-distributed workloads using a reinforcement learning (RL) solution, to learn a suitable scaling policy from experience. Like every other machine learning implementation, this could be potentially slow due to the required model training. KubeEdge [35] is a lightweight extension of Kubernetes framework allowing native containerized application orchestration capabilities to hosts at edge. Built upon Kubernetes it provides fundamental infrastructure support for network, applications deployment and metadata synchronization between cloud and edge. The comparison of the Kubernetes and our model is described in detail in Section III-E.

Ryden *et al.* [23] present a platform for distributed computing with attention to user-based applications. Unlike other systems, the goal is not to implement a resource management policy but to give users more flexibility for application development. Users implement applications using Javascript (JS) programming language, with some embedded native code for efficiency. Similarly [29], the authors use volunteer nodes to run all the workloads, with the difference that some nodes are storage, while others are calculation exclusively. Volunteer nodes refer to the nodes donated by regular people for some project in which their resources are used for distributed computing and/or storage [36]. Sandboxing technique protects nodes running applications from malicious code. Users develop their applications using JS only. This restriction comes from using Google Chrome Web browser-based Native Client (NaCl) sandbox [37]. JS is a popular language at the moment, but the restriction of a single language might be a deal-breaker for some usages. If standard virtual machines become too resource-demanding, a solution using containers could provide sandboxing and bring better resource utilization. It is an interesting idea to show how

users can develop their applications and run them in an EC environment.

Lèbre *et al.* [38] describe a promising solution of extending OpenStack, an open-source infrastructure as a service (IaaS) platform for fog/edge use cases. The authors tried to manage both cloud and edge resources using a NoSQL database. Implementation of a massively distributed multi-site IaaS, using OpenStack is a challenging task [38]. Communication between nodes of different sites can be subject to important network latencies [38]. The major advantage is that users of the IaaS solution can continue using the same familiar infrastructure. In [22] Shao *et al.* present a possible MDCs structure serving only the local population, in the smart city use-case. There are few industry operating frameworks for EC, like Amazon Greengrass [39], deeply connected to the entire Amazon cloud ecosystem. These frameworks are mainly used for user-based applications, while, for instance, GE. Predix [40] is a scalable platform used for industrial IoT applications.

Our work focuses on fully bringing cloud solutions closer to the ground and data sources, with adaptation for a different environment. The users will have a cloud-like application model, able to fully utilize the new design — the edge computing applications model. This model should allow users to develop applications regardless of technology, language or framework as long as they can be virtualized (e.g., microservices in containers or virtual machines).

III. DESIGN AND ARCHITECTURE

This section explains the model of EC as a service compared to the traditional CC model. The formation of such a system is presented with a formal model and a proof of concept implementation based on the proposed model.

A. TOWARDS EDGE CENTRIC COMPUTING AS A SERVICE

MDCs with a zone-based server organization are a good starting point for building EC as a service because we can extend the computing power and storage capacity to serve the local population. But, we need a more available and resilient system with less latency.

Looking at the CC design it can be seen that every part there contributes to a more resilient and scalable system. CC architecture consists of three main building blocks: cluster, region, and availability zone [41].

A cluster represents a set of computers working together, and they are viewed as a single working entity. A Region represents a geographic location where cloud resources (or clusters) are located. Regions are isolated and independent from each other, composed of few availability zones [41]. Every availability zone represents a logical DC in a region available for cloud customers to use. Every availability zone represents a logical DC inside a region to use, and has redundant and separate power, networking, and connectivity. If the zone fails, there are still more of them to serve user requests. CC customers can run their applications in multiple zones at the same time. With some adaptations, edge-centric computing (ECC) could use a similar architectural strategy to dynamically form disposable micro clouds, closer to the users.

Multiple nodes can form a cluster, providing resources for the system. Multiple node clusters can be combined into a bigger logical concept of region, increasing the availability and reliability of the system and applications. We are talking about geo-distributed systems, and the scenario is slightly different than in standard CC model.

The cloud region is a physical infrastructure element housing numerous racks of computers available for cloud users [41]. In the ECC, a region could be used to describe a set of clusters over an arbitrary geographic region. Regions can accept or release clusters and clusters can accept or release nodes. ECC regions are composed of at least one cluster but can be composed of many more. If the entire cluster goes down, the region can failover workloads to one of the other clusters in the same region achieving a more resilient and available system. To ensure less latency, the vast distance between clusters should be avoided. In normal circumstances we want the cluster to be as close as possible to the population making the requests. In CC, region extension requires connecting modules physically to the rest of the infrastructure [42].

Multiple regions form a second logical layer - topology. Topology is composed a minimum of one region and could span over more regions. When designing a topology, especially if regions need to share information, the vast distance between regions should be avoided, if possible. Topology handles regions in the same way region handles clusters. If the entire region goes down, the topology can failover workloads to one of the other regions achieving a more resilient and available system. If regions are not close to the population sending requests, that situation may affect latency. In ECC, the cloud has multiple roles. If the EC node fails, and there are no available regions or clusters to accept requests locally, we can still failover to the cloud as our final resort. The cloud should also store and process data on a much higher scale than EC nodes. Also, the cloud should be able to accept pre-processed data from EC nodes for further processing, if required.

With these simple abstractions, any geographic region can be covered with the ability to shrink or expand clusters, regions — micro clouds infrastructure. Compared to other similar models and solutions, this model allows users dynamic formation of micro clouds infrastructure and treat it fully disposable. This allow infrastructure abstraction to the level of software, creating infrastructure as software model. Available tools, principles, and techniques (e.g., reuse, testing, modeling, and evaluation) that can equally be used for our model.

Table 1 shows similar concepts between CC and ECC. The accent is on the difference between the physical part and the logical concepts in CC and ECC.

Separation on clusters, regions, and topologies is a matter of agreement and usages, similar to modeling in Big Data

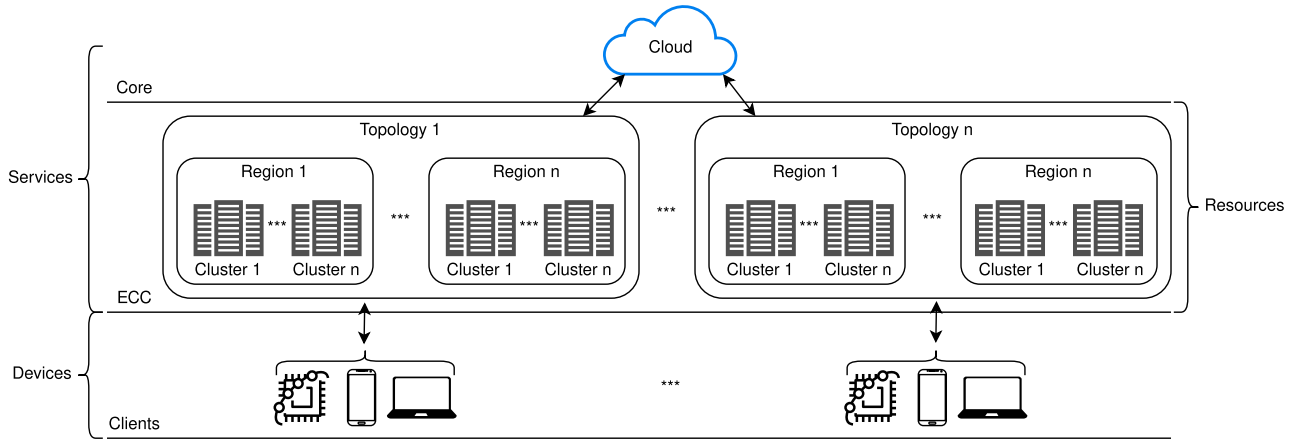


FIGURE 1. ECC as a service architecture with separation of concerns.

TABLE 1. Similar concepts between CC and ECC.

Edge centric computing	Cloud computing
Topology (logical)	Cloud provider (logical)
Region (logical)	Region (physical)
Cluster (physical)	Zone (physical)

systems [43], [44]. For example, clusters could be as wide as the whole city or as small as all devices in a single household and everything in between. The city could be one region with parts of the city organized into clusters. Either a city topology or a country topology can be formed by splitting a city into multiple regions containing multiple clusters, or splitting a country into regions, with cities being regions.

Nodes inside the cluster should run some membership protocol. Gossip style protocols, like SWIM [45], could be used in conjunction with replication mechanisms [46]–[48] making the whole system more resilient.

In everything as a service model [49], EC as a service fits in between Containers as a Service (CaaS) and Platform as a Service (PaaS), depending on users' needs. Figure 1. shows the proposed model for ECC as a service, with SoC for every layer of the system.

B. SEPARATION OF CONCERNS MODEL

SoC is a vital part of any system because it establishes boundaries, and logical or physical constraints, which delineates a given set of responsibilities. This is especially important if creating a platform to be offered as a service.

Our SoC model for ECC as a service is based on the three core concepts: **devices**, **resources**, and **services**. Describing physical services with these three concepts, and specifying their relationships, was first proposed by Jin in [50]. Based on these concepts, our model is separated into three layers, as depicted in Figure 1.

The bottom layer consists of various client devices, or data creators, and service consumers. The second layer represents resources. Resources have a spatial feature and indicate the range of their hosting devices [50]. Developers must know the resource utilization and spread at any time, as well as the application's state and health. Resources represent EC nodes dynamically organized into disposable micro clouds, operating near users serving their requests locally. To be part of the system a node must satisfy four simple rules: (*i*) run an operating system with a file system, (*ii*) be able to run some application isolation tool, for example, a container or unikernel engine, (*iii*) have available resources for utilization, and (*iv*) have internet connection. These simple yet powerful rules could be helpful in certain situations. For example, if there is an increased demand for resources that the currently available infrastructure cannot support. In such a scenario, the inclusion of volunteer nodes (cf. page 114470) [23], [29] into the system can be allowed to depreciate load for an indefinite period. Services expose resources through an interface and make them available on the Internet [50]. These services operate on two levels. Front service responds to clients immediately if possible, or cache information [51], [52] inside micro clouds for future requests. Back services operate in the cloud and have multiple roles: (*i*) accept pre-processed data from the front services, (*ii*) accept queries from front service if the data user requested is not cached locally, (*iii*) serve user requests if there are no micro clouds able to serve user request locally, and (*iv*) they are also responsible for computation and storage, which is beyond the capabilities of micro clouds.

C. PROPOSED MODEL

Infrastructure deployment will not happen until the process is trivial [51], hence the key is to simplify ECC management. The main problem is that going to every node is tedious and time-consuming, especially in a geo-distributed environment.

The system we propose tackles this issue using remote configuration and it relies on three protocols: (*i*) **health-check** protocol informs the system about the state of every node, (*ii*) **cluster formation** protocol forms new clusters, and (*iii*) **list detail** protocol shows the current state of the system

to the user. For each of the three protocols we also present formal descriptions using multiparty asynchronous session types, that are explained in what follows.

1) MULTIPARTY ASYNCHRONOUS SESSION TYPES

Our protocols can be modeled using [53], an extension of *multiparty asynchronous session types* (MPST) [54] — a class of behavioral types tailored for describing distributed protocols relying on asynchronous communications. The type specifications are not only useful as formal descriptions of the protocols, but we can also rely on a modeling-based approach developed in [53] to validate our protocols satisfy multiparty session types safety (there is no reachable error state) and progress (an action is eventually executed, assuming fairness).

The first step in modeling the communications of a system using MPST theory is to provide a *global type*, that is a high-level description of the overall protocol from the neutral point of view. Following [53], the syntax of global types is constructed by:

$$\mathsf{G} ::= \{\mathsf{p} \dagger \mathsf{q}_i{:}\ell_i(\mathsf{T}_i).\mathsf{G}_i\}_{i\in I} \mid \mu t.\mathsf{G} \mid t \mid \mathtt{end} \quad (1)$$

where $\dagger \in \{\rightarrow, \twoheadrightarrow\}$ and $I \neq \emptyset$. In the above, $\{\mathsf{p} \dagger \mathsf{q}_i{:}\ell_i(\mathsf{T}_i).\mathsf{G}_i\}_{i\in I}$ denotes that *participant* p can send (resp. connects) to one of the participants q_i, for $\dagger = \rightarrow$ (resp. $\dagger = \twoheadrightarrow$), a *message* ℓ_i with the *payload* of *sort* T_i, and then the protocol continues as prescribed with G_i. $\mu t.\mathsf{G}_1$ is a recursive type, and t is a recursive variable, while $\mathtt{end}$ denotes a terminated protocol. We assume all participants are (implicitly) disconnected at the end of each session (cf. [53]).

The advance of using approach of [53], when compared to standard MPST (e.g., [54]), is in a relaxed form of choice (a participant can choose between sending to different participants), and, $\twoheadrightarrow$, that explicitly connects two participants, hence (possibly) dynamically introducing participants in the session. Both of these features will be significant for modeling our protocols (it will be discussed again).

The second step in modeling protocols by MPST is providing a syntactic projection of the protocol onto each participant as a local type, that is then used to type check the endpoint implementations. The definition of projector operator given in [53] is used. In essence, the projection of global type G onto participant p can result in $\mathsf{S}_\mathsf{p} = \mathsf{q}!\ell(\mathsf{T})\ldots$ (resp. $\mathsf{S}_\mathsf{p} = \mathsf{q}!!\ell(\mathsf{T})\ldots$) when $\mathsf{G} = \mathsf{p} \rightarrow \mathsf{q}{:}\ell(\mathsf{T})\ldots$ (resp. $\mathsf{G} = \mathsf{p} \twoheadrightarrow \mathsf{q}{:}\ell(\mathsf{T})\ldots$), and, dually, $\mathsf{S}_\mathsf{p} = \mathsf{q}?\ell(\mathsf{T})\ldots$ (resp. $\mathsf{S}_\mathsf{p} = \mathsf{q}??\ell(\mathsf{T})\ldots$) when $\mathsf{G} = \mathsf{q} \rightarrow \mathsf{p}{:}\ell(\mathsf{T})\ldots$ (resp. $\mathsf{G} = \mathsf{q} \twoheadrightarrow \mathsf{p}{:}\ell(\mathsf{T})\ldots$), while the projection operator "skips" the prefix of a global type if participant p is not mentioned neither as sender nor as receiver. Furthermore, a local type must be represented by the following syntax:

$$\mathsf{S} ::= +\{\mathsf{q}_i\alpha\ell_i(\mathsf{T}_i).\mathsf{S}_i\}_{i\in I} \mid \mu t.\mathsf{S} \mid t \mid \mathtt{end} \quad (2)$$

where $\alpha \in \{!, !!\}$ or $\alpha \in \{?, ??\}$ (in which case $\mathsf{q}_i = \mathsf{q}_j$ must hold for all $i, j \in I$, to ensure consistent external choice subjects, cf. [53, Page 6.]), and $I \neq \emptyset$. Trailing $\mathtt{end}$'s when specifying types for our protocols will be omitted. Interested reader can find details in [53].

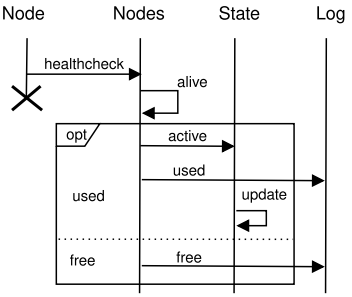


FIGURE 2. Low level health-check protocol diagram.

2) HEALTH-CHECK PROTOCOL

In a clustered environment, every node has a channel where it sends metrics as a health-check mechanism. This channel can be used, or a new one created, to spread actions to the nodes, for example, a cluster formation message. Figure 2. shows a low-level health-check protocol between a single node and the rest of the system, involving the following participants: Node, Nodes, State, and Log.

The participants which are included in Figure 2 follow the next protocol: (*i*) **Node** sends a health-check signal to the nodes service; (*ii*) **Nodes** accept health-check signals for every node, update node metrics and if node is used in some cluster, inform that cluster about the node state; (*iii*) **State** contains information about nodes in the clusters, regions and topologies; (*iv*) **Log** contains records of operations. Users can query this service.

Periodically, every node will inform the system about its existence via a health-check ping. This ping will also carry the information about available **label**s attached to that particular node. These labels play a very important role for the rest of the system, and they can be viewed/used as node attributes. Furthermore, they will be used when users want to query available nodes, and/or to form new clusters. For example, a predefined label **geolocation** can be used for cluster formation from nodes that are geographically close to each other.

Upon the node's ping, the rest of the system is informed about the ping if and only if (henceforth iff) the node is used in some cluster. Algorithm 1 describes how the system will store the node data and determine if the node is free or used. It is a *polynomial* or $\mathcal{P}$ algorithm class, meaning that the execution time is either given by a polynomial on the size of the input or can by such a polynomial be bounded. All subsequent algorithms fall into the $\mathcal{P}$ class category of algorithms.

Set theory can be used to formally describe servers or nodes (terms are used interchangeably). In the beginning, the server set S is empty, denoted with $S = \emptyset$. Nodes are free iff they do not belong to any cluster. To determine the node state, a node-id structure can be used, for example. If the received health-check message from the particular node contains only node-id, it is free, otherwise, it is not. If there are n free nodes in the wild, denoted with s_i, where $i \in \{1, \ldots, n\}$, and they

Algorithm 1 Health-Check Data Received

```
input: event, config
if isNodeFree(event.id) then
    if exists(event.id) then
        renewLease(event.id, config.leaseTime);
        updateData(event.id, event.data);
    else
        leaseNewNode(event.id, config.leaseTime,
        event.data);
        saveMetrics(event.id, event.metrics);
else if isNodeReserved(event.id) then
    updateData(event.id, event.data);
else
    renewLease(event.id, config.leaseTime);
    updateData(event.id, event.data);
    saveMetrics(event.id, event.metrics);
    sendNodeACK(event.id);
```

notify the system with a health-check ping that they are free, they should then be added to the server set, and thus we have:

$$S_{new} = S_{old} \cup \bigcup_{i=1}^{n} \{s_i\}. \tag{3}$$

The order in which messages arrive is not significant. Nodes in the same cluster are equal to free nodes, and there are no special nodes. The only thing cared about is that nodes are alive and ready to accept some jobs. Algorithm 1 describes how the system stores the node data and determines if the node is free or used. We can describe the s_i server in the server set S as a tuple $s_i = (L, R, A, I)$, where:

- L is a set of ordered key-value pairs, i.e., $L = \{(k_1, v_1), \ldots, (k_m, v_m)\}$ where $k_i \neq k_j$, for each $i, j \in \{1, \ldots, m\}$ such that $i \neq j$. L represents node labels or server-specific features. We based labels on Kubernetes [34] labels concept, which is used as an elegant binding mechanism for its components.
- R is a set of tuples $R = \{(f_1, u_1, t_1), \ldots, (f_m, u_m, t_m)\}$ representing node resources, where f_i, u_i, t_i, for $i \in \{1, \ldots, m\}$ are as follows: f_i is the free resource value; u_i is the used resource value; t_i is the total resource value.
- $A = \{(l_1, r_1, c_1, i_1), \ldots, (l_m, r_m, c_m, i_m)\}$, representing running applications, where l_j, r_j, c_j, i_j, for $j \in \{1, \ldots, m\}$, are as follows: l_j represents labels, the same way we used for node labels; r_j is the resource set application requires; c_j is the configuration set application requires; i_j is the general information like name, port, developer.
- I represent a set of general node information like: name, location, IP address, id, cluster id, region id, topology id, etc.

If we want to assign m (fresh) labels to the i_{th} server, we start with empty labels set $s_i[L] = \emptyset$, then we add labels to server. Thus, we have

$$s_i[L]_{new} = s_i[L]_{old} \cup \bigcup_{j=1}^{m} \{(k_j, v_j)\}. \tag{9}$$

Every server from set S must have *a non-empty* set of labels, but the number of labels for every server may vary. For the label definition, arbitrary alphanumeric text can be used for both key and value, separated with colon sign (e.g., os:linux, arch:arm, model:rpi, cpu:2, memory:16GB, disk:300GB, etc.). Labels should be chosen carefully and agreed on upfront but should be able to be changed if needed.

Following the MPST (cf. Section III-C1), a formal description of the low-level health-check communication protocol (cf. Figure 2) is now presented. The global protocol G_1, presented in (4), as shown at the bottom of the page, conforms the informal description given on page 114473: `node` connects `nodes` with *health_check* message and a payload of type T_1 that is required by the system to properly register node. Then, depending on the received information, `nodes` **either** connects `state` with *active* message informing the node status with a payload typed with T_2 (that contains information required to register active health-check sender), and then also connects `log` with the same message, **or** directly connects `log` informing the node is *free*.

Notice that in G_1 (cf. (4)) we indeed have a choice of `nodes` sending either to `state` or to `log`. Such communication pattern could not be directly modeled using standard MPST approaches, such as, e.g., [54]. Also, notice that `state` will be introduced into the session only when receiving from `nodes`. Hence, if the session after the first ping from `node` to `nodes` proceeds with the second branch (i.e., connecting `nodes` with `log`), then `state` is not considered as stuck, as it would be in standard MPST such as, e.g., [54], but rather idle.

Projecting global type G_1 onto participants `node`, `nodes`, `state`, and `log` we obtain local types $\mathsf{S}_{\texttt{node}}$, $\mathsf{S}_{\texttt{nodes}}$, $\mathsf{S}_{\texttt{state}}$, and $\mathsf{S}_{\texttt{log}}$, respectively, as presented in (5) – (8), as shown at the bottom of the page. For instance, type $\mathsf{S}_{\texttt{nodes}}$, given in (6), specifies `nodes` can receive the ping message from `node`, after which it will dynamically introduce either

$$\mathsf{G}_1 = \texttt{node}\twoheadrightarrow\texttt{nodes}{:}\mathit{health\_check}(\mathsf{T}_1).\begin{cases}\texttt{nodes}\twoheadrightarrow\texttt{state}{:}\mathit{active}(\mathsf{T}_2).\texttt{nodes}\twoheadrightarrow\texttt{log}{:}\mathit{used}(\mathsf{T}_2)\\ \texttt{nodes}\twoheadrightarrow\texttt{log}{:}\mathit{free}(\mathsf{T}_2)\end{cases} \tag{4}$$

$$\mathsf{S}_{\texttt{node}} = \texttt{nodes}!!\mathit{health\_check}(\mathsf{T}_1) \tag{5}$$

$$\mathsf{S}_{\texttt{nodes}} = \texttt{node}??\mathit{health\_check}(\mathsf{T}_1).+\begin{cases}\texttt{state}!!\mathit{active}(\mathsf{T}_2)].\texttt{log}!!\mathit{used}(\mathsf{T}_2)\\ \texttt{log}!!\mathit{free}(\mathsf{T}_2)\end{cases} \tag{6}$$

$$\mathsf{S}_{\texttt{state}} = \texttt{nodes}??\mathit{active}(\mathsf{T}_2) \tag{7}$$

$$\mathsf{S}_{\texttt{log}} = +\begin{cases}\texttt{nodes}??\mathit{used}(\mathsf{T}_2)\\ \texttt{nodes}??\mathit{free}(\mathsf{T}_2)\end{cases} \tag{8}$$

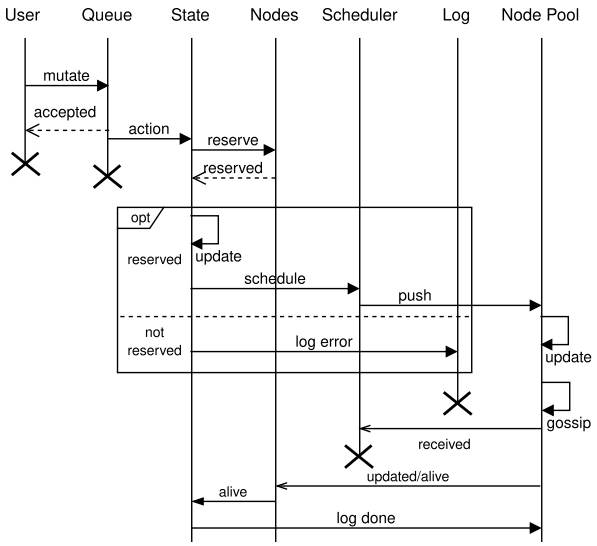


FIGURE 3. Low level cluster formation communication protocol diagram.

`state` or `log` into the session, where in the former case it also connects `log` (but now with message *free*).

3) CLUSTER FORMATION PROTOCOL

Another communication protocol in our system appears in the cluster formation process. Users can dynamically form new clusters. Two different actions are distinguished here. The first action is a user-system communication, where the user sends a query to the system to obtain a list of available nodes based on the query parameters. These query parameters will be tested against the available node labels. The query result should be sorted and grouped by geolocation if such a label is present in node labels, otherwise node IP address could be used to determine node approximate location. The second action starts when the user sends a message to the system with a new specification. Users may choose to form new clusters based on multiple attributes (e.g., geolocation, node architecture, specific resources, etc.). Nodes geolocation should be attributed when forming clusters, but it should not be mandatory and it should be left on user decision. In this setting, the system involves participants: User, Queue, Scheduler, State, Nodes, Log, and NodesPool, that cooperate to dynamically form new clusters, regions, or topologies, adhering to the scenario shown in Figure 3.

The participants follow the protocol that is now formally described: (*i*) **User** query Nodes service, based on some predefined criteria. The user sends created message to Queue, and gets either a response *ok*, or *error* if the message cannot be accepted due to missing rights or other issues. This operation is called *mutation*; (*ii*) **Queue** accepts a user message and passes it to State. Messages are handled in FIFO (First In, First Out) order. The queue prevents system congestion, with received messages; (*iii*) **State** accepts mutation messages from Queue and tries to store new information about the cluster, region, or topology. If Nodes service can reserve all desired nodes, the system will store new user desired information and send a message to Scheduler to physically create clusters of desired nodes; (*iv*) **Nodes** accept messages from State. If possible, it will reserve desired nodes, otherwise, it will send an error message to Log service. On a health-check message, if a node is used in some cluster, it will inform that the node is alive; (*v*) **Scheduler** waits for a message sent from State, and pushes cluster formation messages to desired nodes; (*vi*) **Log** contains records of operations. Users can query this service to see if their tasks are finished or have any problems; (*vii*) **Nodes Pool** represents the set of n free nodes that will accept mutation messages. On message receive, every node will: (*i*) start gossip protocol to inform other nodes from the mutation message about cluster formation, and (*ii*) send an event to Scheduler and Nodes service that it is alive and can receive messages.

If a user wants to get a list of free nodes, he must create a query using *the selector*, which is the set of key-value pairs desired by the user. Algorithm 2 describes steps required to perform a proper node lookup based on a received selector value.

Algorithm 2 Nodes Lookup

```
input: query
Initialize: nodes ← []
foreach node ∈ freeNodes() do
    if len(node.labels) == len(query) ∧
    node.haveAll(query) then
        nodes.append(node)
return nodes
```

We start with the empty selector $Q = \emptyset$, in which we append key-value pairs. Hence, when a user submits a set of p key-value pairs we have that:

$$Q_{new} = Q_{old} \cup \bigcup_{i=1}^{p} \{(k_i, v_i)\}. \tag{10}$$

Once the query is submitted, for every server in the set S, we need to check:

(i) the cardinality of the i_{th} server's set of labels and the query selector are identical in size

$$|s_i[L]| = |Q|\,, \text{ and} \tag{11}$$

(ii) every key-value pair from query set Q is present in the i_{th} server's labels set $s_i[L]$, hence the following predicate must yield true:

$$P(Q, s_i) = \left(\forall (k, v) \in Q \; \exists (k_j, v_j) \in s_i[L] : k = k_j \wedge v \leq v_j\right) \tag{12}$$

The i_{th} server from the server set S will be present in the result set R, iff both rules are satisfied:

$$R = \{s_i \mid |s_i[L]| = |Q| \wedge P(Q, s_i), i \in \{1, \ldots, n\}\} \tag{13}$$

If the result set R is not empty, nodes are reserved for configurable time so that other users cannot see (and try to

use) them, and, finally, reserved nodes with message data *md* are added to the task queue set

$$TQ_{new} = TQ_{old} \cup \{(R, md)\}. \tag{14}$$

When the task comes to execution, the task queue sends messages to every node. Algorithm 3 describes the process required for cluster formation.

Algorithm 3 Clustering Formation Message

```
input: request, config
nodes ← searchFreeNodes(data.query)
reserveNodes(nodes, config.time)
pushMsgToQueue(nodes, data)
key ← saveTopologyLogicState(data)
watchForNodesACK(key)
```

Algorithm 4 describes steps after nodes receive a cluster formation message, that are explained next.

Algorithm 4 Node Reaction to Clustering Message

```
input: event
switch event.type do
    case formationMessage
        updateId(event.topology, event.region,
        event.cluster)
        newState ← updateState(event.labels,
        event.name)
        sendReceived(newState)
        nodes ← pickGossipNodes(event.nodes)
        startGossip(nodes)
```

Users can choose to override labels with their own or keep existing ones when including nodes in the cluster. If the node is free, or if the user did not change the node labels on cluster formation, the system will use default labels. On message received, the node will pick and contact a configurable subset of nodes $R_g \subset R$, and start the gossip protocol, propagating information about nodes in the cluster (e.g, new, alive, suspected, dead, etc.). When every node inside the newly formed cluster has a complete set of nodes R obtained through gossiping, the cluster formation process is over. Topology, region, or cluster formation should be done descriptively using YAML, or similar formats.

In the following, a low-level cluster formation communication protocol (cf. Figure 3) is formally described using the same extension of multiparty session types [53] as for the health-check protocol. Global protocol G_2, presented in (15), as shown at the bottom of the page, conforms the informal description of the cluster formation protocol given on page 114475. The protocol starts with `user` connecting `state` by message *query* and a payload typed with T_1 that contains user query data, and then `state` forwards the message by connecting `nodes`. Then, the protocol possibly enters into a loop, specified with μt, depending on the later choices. Further, `nodes` replies a response *resp* to `state`, that, in turn, forwards the message to `user`. The payload of the message is typed with T_2 that has response data, based on a given query. At this point, `user` sends to `state` one of three possible messages: (*i*) *mutate*, and the mutation process, described with global protocol G', presented in (16), as shown at the bottom of the page, starts; (*ii*) *quit*, in which case the protocol terminates; or, (*iii*) *query* — this means the process of querying starts again, the query message is forwarded to `nodes` and the protocol loops, returning to the point marked with μt. The third branch is the only one in which protocol loops. Also, notice that `user` − `state` and `state` − `nodes` are connected before specifying recursion. Hence, even after many recursion calls, these connections will be unique (thus, there is no need to disconnect them before looping).

The mutate protocol G' (cf. (16)), activated in the first branch in G_2, starts with `user` sending `create` message to `state`, specifying also information about new user desired state typed with T_3, and `state` replies back with *ok*. Then, `state` sends *ids* of the nodes to be reserved (specified in the payload typed with T_4) to `nodes`, that, in turn sends one of the two possible messages to `state`: (*a*) *rsrvd*, denoting all nodes are reserved and the protocol proceeds as prescribed with G'', given in (17), as shown at the bottom of the page, or (*b*) *error*, with error message in the payload, informing there has been unsuccessful reservation of nodes, in which case `state` connects `log` reporting the error and the protocol terminates.

$$\mathsf{G}_2 = \texttt{user}\twoheadrightarrow\texttt{state}{:}\mathit{query}(\mathsf{T}_1).\texttt{state}\twoheadrightarrow\texttt{nodes}{:}\mathit{query}(\mathsf{T}_1).\\ \mu t.\texttt{nodes}\rightarrow\texttt{state}{:}\mathit{resp}(\mathsf{T}_2).\texttt{state}\rightarrow\texttt{user}{:}\mathit{resp}(\mathsf{T}_2).\begin{cases}\texttt{user}\rightarrow\texttt{state}{:}\mathit{mutate}().\mathsf{G}'\\ \texttt{user}\rightarrow\texttt{state}{:}\mathit{quit}()\\ \texttt{user}\rightarrow\texttt{state}{:}\mathit{query}(\mathsf{T}_1).\texttt{state}\\ \quad\rightarrow\texttt{nodes}{:}\mathit{query}(\mathsf{T}_1).t\end{cases} \tag{15}$$

$$\mathsf{G}' = \texttt{user}\rightarrow\texttt{state}{:}\mathit{create}(\mathsf{T}_3).\texttt{state}\rightarrow\texttt{user}{:}\mathit{ok}().\texttt{state}\rightarrow\texttt{nodes}{:}\mathit{ids}(\mathsf{T}_4).\\ \begin{cases}\texttt{nodes}\rightarrow\texttt{state}{:}\mathit{rsrvd}().\mathsf{G}''\\ \texttt{nodes}\rightarrow\texttt{state}{:}\mathit{err}(\mathsf{String}).\texttt{state}\twoheadrightarrow\texttt{log}{:}\mathit{err}(\mathsf{String})\end{cases} \tag{16}$$

$$\mathsf{G}'' = \texttt{state}\twoheadrightarrow\texttt{sched}{:}\mathit{ids}(\mathsf{T}_5).\texttt{sched}\twoheadrightarrow\texttt{pool}{:}\mathit{update}(\mathsf{T}_6).\texttt{pool}\rightarrow\texttt{sched}{:}\mathit{ok}().\texttt{pool}\twoheadrightarrow\texttt{nodes}{:}\mathit{nids}(\mathsf{T}_4).\texttt{nodes}\\ \rightarrow\texttt{state}{:}\mathit{succ}().\texttt{state}\twoheadrightarrow\texttt{log}{:}\mathit{succ}() \tag{17}$$

Finally, in G'' `state` connects `sched` (Scheduler) with message *ids* and the payload that contains other data imported for mutation to be completed (typed with T_5). Then, `sched` connects `pool` (Nodes Pool) with *update* specified with T_6, after which `pool` replies back with *ok*, and connects to `nodes` sending new id's *nids* typed with T_4 (that contains successfully reserved user desired nodes). Now `nodes` notifies `state` the action was successful, that in turn connects `log` with the same message, and the protocol terminates.

We may now obtain the projections of global type G_2 onto the participants `user`, `state`, `nodes`, `log`, `pool`, and `sched` as presented in (18) – (24), as shown at the bottom of the page.

For instance, type $S_{\texttt{sched}}$, given in (24), specifies that participant `sched` gets included in the session only after receiving from `state` message *ids*, then `sched` connects `pool` with *update* message, after which expects to receive *ok* message and finally terminates.

Global type G_2 could also be modeled directly using standard MPST models (such as [54]). However, in such models, the projection of G_2 onto, for instance, participant `sched` would be undefined (cf. [53]). Since we follow the approach of [53] with explicit connections, projection of G_2 onto `sched` is indeed defined as $S_{\texttt{sched}}$.

4) LIST DETAIL PROTOCOL

The last communication protocol in our system appears in the information retrieval process. Namely, on formed topologies, using labels, the user can specify what part of the system he wants to retrieve, for example, to visualize on some dashboard. Two options are available: (*i*) **global view** of the system — all topologies the user manages, or (*ii*) **specific clusters** details — complete details about specified clusters like resource utilization over time (using stored metrics information), node information, and running or stopped services. Similar to the query operation, both rules (11) and (12) must be satisfied in order for information to be present in the response. One additional piece of information that may be specified is whether the user wants a detailed view or not. If detailed view information is presented in a request, the user will get a detailed view. Figure 4. shows a low-level view of the list operation protocol, where users can get details about the formed system. In this setting, the system involves participants: User, State, Nodes, and Log.

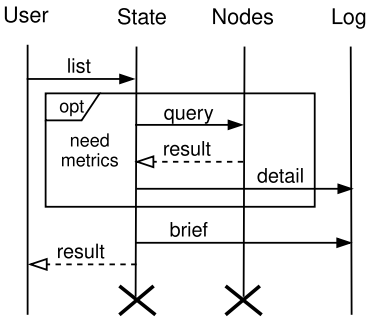


FIGURE 4. Low level view of list operation communication.

The participants roles in the protocol are now informally described: (*i*) **User** sends a list request to State service; (*ii*) **State** accepts the list request and the query local state based on the user selector. If a detail view is required, the state gets metrics data from Nodes service; (*iii*) **Nodes** contain node metrics data, and if required, it may send this data to State; (*iv*) **Log** contains records of all operations. Users can query this service.

Algorithm 5 describes steps after the state receives a list message.

$$S_{\texttt{user}} = \texttt{state}!!\mathit{query}(T_1).\mu t.\texttt{state}?\mathit{resp}(T_2).+\begin{cases}\texttt{state}!\mathit{mutate}().\texttt{state}!\mathit{create}(T_3).\texttt{state}?\mathit{ok}()\\ \texttt{state}!\mathit{quit}()\\ \texttt{state}!\mathit{query}(T_1).t\end{cases} \tag{18}$$

$$S_{\texttt{state}} = \texttt{user}??\mathit{query}(T_1).\texttt{nodes}!!\mathit{query}(T_1).\mu t.\texttt{nodes}?\mathit{resp}(T_2).\texttt{user}!\mathit{resp}(T_2). + \begin{cases}\texttt{user}?\mathit{mutate}().\texttt{user}?\mathit{create}(T_3).S'\\ \texttt{user}?\mathit{quit}()\\ \texttt{user}?\mathit{query}(T_1).\texttt{nodes}!\mathit{query}(T_1).t\end{cases} \tag{19}$$

$$S' = \texttt{user}!\mathit{ok}().\texttt{nodes}!\mathit{ids}(T_4).+\begin{cases}\texttt{nodes}?\mathit{rsrvd}().\texttt{sched}!!\mathit{ids}(T_5).\texttt{nodes}?\mathit{succ}().\texttt{log}!!\mathit{succ}()\\ \texttt{nodes}?\mathit{err}(\mathsf{String}).\texttt{log}!!\mathit{err}(\mathsf{String})\end{cases} \tag{20}$$

$$S_{\texttt{nodes}} = \texttt{state}??\mathit{query}(T_1).\mu t.\texttt{state}!\mathit{resp}(T_2). + \begin{cases}\texttt{state}?\mathit{ids}(T_4).+\begin{cases}\texttt{state}!\mathit{rsrvd}()\\ \texttt{state}!\mathit{err}(\mathsf{String}).\texttt{poll}??\mathit{nids}(T_4).\texttt{state}!\mathit{succ}()\end{cases}\\ \texttt{state}?\mathit{query}(T_1).t\end{cases} \tag{21}$$

$$S_{\texttt{log}} = +\begin{cases}\texttt{state}??\mathit{succ}()\\ \texttt{state}??\mathit{err}(\mathsf{String})\end{cases} \tag{22}$$

$$S_{\texttt{pool}} = \texttt{sched}??\mathit{update}(T_6).\texttt{sched}!\mathit{ok}().\texttt{nodes}!!\mathit{nids}(T_4) \tag{23}$$

$$S_{\texttt{sched}} = \texttt{state}??\mathit{ids}(T_5).\texttt{pool}!!\mathit{update}(T_6).\texttt{pool}?\mathit{ok}() \tag{24}$$

Algorithm 5 List of Current State of the System

```
input: request
Initialize: data ← []
foreach (topology, isDetail) ∈ userData(request.query)
do
    if isDetail then
        data.append(topology.collectData())
    else
        data.append(topology.data())
return data
```

Next, we present a formal description of the list communication protocol (cf. Figure 4) by using [53] (cf. Section III-C1). Global type G_3, presented in (25), as shown at the bottom of the page, starts with `user` connecting `state` with one of the two possible messages: (*i*) *list*, specifying a request for a detailed view, where sort T_1 identifies which parts of the system user wants to view in details, after which `state` connects `nodes` with *query* message, with a payload of sort T_2, containing specification of which nodes need to show their metrics data, and then protocol proceeds with `nodes` replying to `state` *result* message and a payload identifying parts of the system user wants to see in greater detail typed with T_3. Then, `state` connects `log` with `details` and also sends `result` to `user`, and finally terminates; (*ii*) *list**, specifies no need for a detailed view is specified, where a payload of sort T_4 denotes user-specified parts of the system the user wants to view, but without greater details. Then, `state` also connects `log` with *brief* and a payload typed with T_5 identifying parts of the system user wants to see without greater detail. Then, `state` replies to `user` with `result` message, and the protocol terminates.

The same as for health-check and the cluster formation protocols, the projections of global type G_3 is here also presented, modeling the list protocol, onto participants `user`, `state`, `nodes`, and `log` (see (26) – (29)), as shown at the bottom of the page.

For instance, type $\mathsf{S}_{\mathtt{log}}$, present in (29), specifies `log` gets included in the session only after receiving from `state`, either message *detail*, or message *brief* , and then terminates.

Similarly as for G_2, we remark G_3 could also be modeled using standard MPST (e.g., [54]), but again the projection types would be undefined, while following the approach of [53] with explicit connections, all valid projections have been obtained.

D. APPLICATIONS MODEL

Traditional DCs propose specially designed cloud-native applications [55], that are easier to scale, more available, and less error-prone when compared to traditional web applications [55].

Edge-native applications [20] should use the full potential of EC infrastructure and keep the good features of their cloud counterparts. Applications may be split into front and back processing services. The front processing service is an edge-native application running inside MDCs to minimize latency, while the back service runs in traditional DCs as a cloud-native application to leverage greater resources. These edge-native applications will handle user requests coming to nearby MDCs, and communicate to cloud-native applications when needed generically (e.g., service mesh) [56].

Separation like that gives developers better flexibility and large design space. The frontend services model should be event-driven, with a subscription policy to message streams using topics [57].

The processing strategy is in the developer's hands, depending on the nature of the use case. Some examples include: (*i*) events that notify users if some value is above or below some defined threshold, (*ii*) stream or processing data as it comes to the system, (*iii*) batch processing does processing in predefined times over some collection of data, or (*iv*) other, something that falls outside these models, or it is the composition of multiple operations at once.

E. ENHANCEMENT OF THE EXISTING SOLUTIONS

The protocols defined in this paper could serve as a base layer for future ECC as a service implementation if the system is being developed from scratch. It is a base layer because, on top of the solution based on these protocols, other services and features like scheduling, storage, applications, management, monitoring, etc. can be implemented. These protocols will ensure proper node registration into the system,

$$\mathsf{G}_3 = \begin{cases} \mathtt{user}\twoheadrightarrow\mathtt{state}{:}\mathit{list}(\mathsf{T}_1).\mathtt{state}\twoheadrightarrow\mathtt{nodes}{:}\mathit{query}(\mathsf{T}_2).\mathtt{nodes}\rightarrow\mathtt{state}{:}\mathit{result}(\mathsf{T}_3).\mathtt{state} \\ \twoheadrightarrow\mathtt{log}{:}\mathit{detail}(\mathsf{T}_3).\mathtt{state}\rightarrow\mathtt{user}{:}\mathit{result}(\mathsf{T}_3) \\ \mathtt{user}\twoheadrightarrow\mathtt{state}{:}\mathit{list}^*(\mathsf{T}_4).\mathtt{state}\twoheadrightarrow\mathtt{log}{:}\mathit{brief}(\mathsf{T}_5).\mathtt{state}\rightarrow\mathtt{user}{:}\mathit{result}(\mathsf{T}_5) \end{cases} \tag{25}$$

$$\mathsf{S}_{\mathtt{user}} = + \begin{cases} \mathtt{state}!!\mathit{list}(\mathsf{T}_1).\mathtt{state}?\mathit{result}(\mathsf{T}_3) \\ \mathtt{state}!!\mathit{list}^*(\mathsf{T}_4).\mathtt{state}?\mathit{result}(\mathsf{T}_5) \end{cases} \tag{26}$$

$$\mathsf{S}_{\mathtt{state}} = + \begin{cases} \mathtt{user}??\mathit{list}(\mathsf{T}_1).\mathtt{nodes}!!\mathit{query}(\mathsf{T}_2).\mathtt{nodes}?\mathit{result}(\mathsf{T}_3).\mathtt{log}!!\mathit{detail}(\mathsf{T}_3).\mathtt{user}!\mathit{result}(\mathsf{T}_3) \\ \mathtt{user}??\mathit{list}^*(\mathsf{T}_4).\mathtt{log}!!\mathit{brief}(\mathsf{T}_5).\mathtt{user}!\mathit{result}(\mathsf{T}_5) \end{cases} \tag{27}$$

$$\mathsf{S}_{\mathtt{nodes}} = \mathtt{state}??\mathit{query}(\mathsf{T}_2).\mathtt{state}!\mathit{result}(\mathsf{T}_3) \tag{28}$$

$$\mathsf{S}_{\mathtt{log}} = + \begin{cases} \mathtt{state}??\mathit{detail}(\mathsf{T}_3) \\ \mathtt{state}??\mathit{brief}(\mathsf{T}_5) \end{cases} \tag{29}$$

organization, and reorganization into clusters, regions, and topologies, bringing disposable micro clouds to the users at the network edge.

In contrast to the proposed model, the existing orchestrator engines like Kubernetes, Apache Mesos, Docker Swarm, etc. operate one cluster level [32]–[35], [58], and a single cluster could span over multiple availability zones, minimizing the chance that a failure in one zone impairs services in other zones [58]. Kubernetes even allow multi-cluster deployments [58], treating these clusters as disposable — ''treating **clusters** as cattle, not pets'' (i.e. numerous servers/clusters built using automated tools designed for failure, where no servers/clusters is irreplaceable [59]).

Our model goes one step further, proposing the creation of disposable micro clouds, thus, gives users more dimensions to operate and optimize their infrastructure. We can build numerous micro clouds designed for failure using automated tools where no micro cloud is irreplaceable — ''treating **micro clouds** as cattle, not pets''.

The proposed model can be integrated into existing solutions to serve as a node register. Users can register new nodes into the system, allowing them to be used by some existing orchestration engine. For example, existing orchestrator engines run a small agent software on every machine that can accept new commands, new nodes into a cluster, or release existing ones. The users can provide a specification, which available EC nodes need to part of the micro clouds. The system will communicate with the existing orchestrator agent to register/unregister them with the existing cluster. Also, we can rely on the orchestrator health-check mechanism. On every health-check message received, the orchestrator can inform our system that a node used in some cluster is alive. Unused nodes can rely on our health-check protocol to inform the system that they are still available for utilization.

Here, our model will preserve the topology of the nodes allowing cloud providers and orchestrator engines to make micro clouds disposable, abstracting infrastructure to the level of software — infrastructure as software [25]. The benefit of this approach lies in the already available tools, principles, and techniques (e.g., reuse, testing, modeling, and evaluation) that can equally be used for the disposable micro cloud infrastructure definitions.

The presented model is not competing with the existing orchestrator tools. Its sole purpose is to be free nodes register and micro clouds infrastructure descriptor offered as a stand-alone service bringing disposable micro clouds model to the users. It can be integrated and connected to any existing orchestration and scheduling tool, leveraging existing mechanisms and best practices.

Cloud providers might even create their connector, based on the proposed model, for various orchestration engines. They can offer dynamically created, disposable micro clouds as a service to their users, using infrastructure as software principles.

F. PROOF OF CONCEPT IMPLEMENTATION

As we already discussed in the introduction, the focus of this paper is not on the implementation details but on a formal description of the model and its protocols used to organize geo-distributed nodes into infrastructure that resembles a cloud but operates closer to the users and data sources. However, to confirm that an implementation of the presented model is feasible, we created a proof of concept solution and tested it in laboratory conditions. We did not analyze metrics (e.g., performance and network overheads or scalability aspects) since laboratory conditions are significantly different than real-world scenarios.

Based on the developed protocols, the proof of concept solution can dynamically form disposable micro clouds, abstracting infrastructure to the level of software in laboratory conditions.

We have implemented a proof of concept system of the proposed model using the microservice architecture shown in Figure 5.

All services are implemented using the Go programming language. As a storage layer, etcd was used, a popular open-source key-value database, and for metrics storage, we used the open-source time-series database InfluxDB. Communication between microservices is implemented in an RPC manner using gRPC, and Protobuf as a message definition. gRPC and Protobuf are open-source tools designed by Google to be scalable, interoperable, and available for general purposes. Communication between nodes and the system is carried out using NATS, an open-source messaging system. Health checking and action push to nodes are implemented over NATS in a publish-subscribe manner. To communicate with the platform, we have developed a simple command-line interface (CLI) application also using the Go programming language that sends JSON encoded messages over HTTP to the system.

Every node runs a simple daemon program implemented as an actor system using the Go programming language. When a message arrives, the proper actor will react based on the message type, or discard it if the type is not supported. Before daemon starts, the user needs to specify identifier, name, labels, health-check interval, and system address using the YAML configuration file. Based on the configuration file, the daemon will start a background health-check mechanism, and it will subscribe to the system, using an identifier as a subscription topic. The background thread will contact the system repeatedly using a contact interval time, specified in the configuration file. On every health check, the node will send labels, names, id, and metrics to the system (e.g., CPU utilization, total, used, free ram or disk, etc.). The specified labels will be used when the user is querying for available nodes, while the node id will be used for reservation when forming a cluster.

The system operates with four commands:

- **mutate** (orange arrows in Figure 5.) change the system state by creating, editing, or deleting clusters, regions, and topologies. When a user wants to perform a mutation

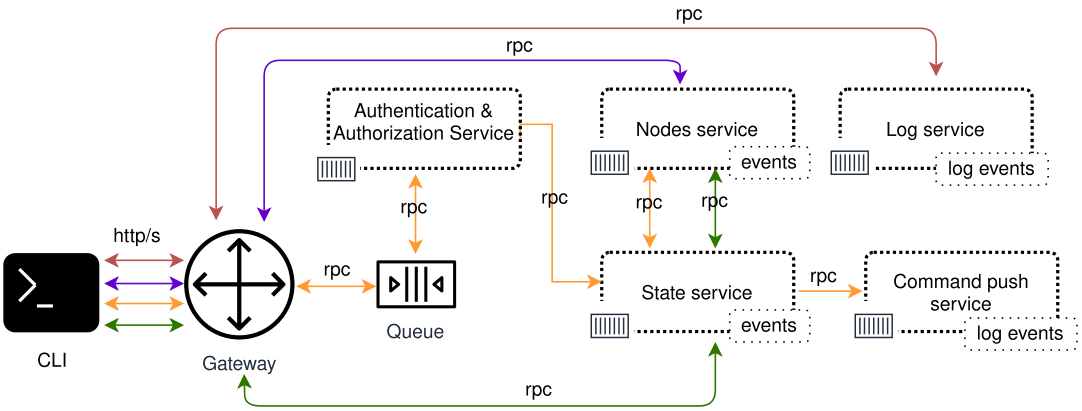


FIGURE 5. Proof of concept implemented system.

over the system, the desired state needs to be specified using a YAML file. The users specify which nodes are forming the cluster. Optionally, users can also specify labels and names on the node level, and retention period on the cluster level. The retention period is used to describe how long metrics are going to be kept. When the retention period expires, the metrics data will be deleted or moved to another location. Users can target a specific system queue, by adding a metadata part in the configuration file. With this ability, users can have specific queues just for the mutation to avoid long waiting times if other queues are full. When forming a topology, users can assign a name and set of labels to the entire topology. These parameters will be used when the user wants to query all topologies to get full information about regions, clusters, and nodes inside a topology;

- **list** show the current state of the system for the registered user (blue arrows in Figure 5.). Using labels, the user can specify what part of the system he\she wants to see. He\She can get a global view of the system - all topologies he\she manages, or details about a single topology (i.e., regions, clusters, and nodes in a single topology). Users can specify a more detailed view of a single cluster, meaning they will get information about what resources the cluster has, but also resource utilization over time (using stored metrics information);
- **query** operation show all or filtered free nodes registered to the system (yellow arrows in Figure 5.). When a user wants to get information about free nodes, he\she needs to submit a selector value composed of multiple key-value pairs. The selector will be used to compare the labels of every free node existing in the system. The nodes satisfying the rules defined in Section III-C3 will be in the result;
- **logs** operation show stored logs and traces to the user (purple arrows in Figure 5.). Here, the user can see the state of his\her operations and actions.

We plan to analyze different metrics as part of our future work (cf. Section V) in real-world scenarios. As a real-world scenario, we can use any of the examples described in Section IV.

IV. APPLICATION AREAS

This section explores the case study on a few examples. Section (*A*) shows how the presented model could be used. Section (*B*) presents what the benefits of such a model would be. In Section (*C*) we discuss how it compares to other, similar, models.

A. FINDINGS

Various computing and network-intensive applications that require real-time processing of geo-distributed environmental data may benefit from the proposed model. Applications like big data streaming, smart healthcare, or even power grids could be used as example applications serving only local population needs. We can go even further and think of a smart city platform, empowered by the proposed model, that unites applications into a cohesive unit, offering developers a new platform for their innovative applications.

Users can create different applications, knowing that the platform, or operating system, will take care of resources. For example, we can represent a city as a *topology*, where parts of the city are separated into one or more *regions*. Every *region* could accommodate multiple *clusters*, depending on the size of the population that needs to be serviced. If the population grows, or shrinks, resources (*regions* and *clusters*) can be dynamically reorganized in the best possible way. Users will have a notion that the "cloud" is closer to them, minimizing the potentially huge round-trip time of the cloud [60].

Government authorities can be responsible for the infrastructure deployment and maintenance, or they can create an arrangement with some provider that will be responsible for the infrastructure. At any given point in time, authorities, or their representatives can monitor infrastructure utilization using *list detail protocol*. This **does not** mean that they will take a look at data that is sent, but rather what applications are running, their state, health, and so on. Authorities can apply data sharing policies, and these policies could be applied on

topology, region, or even cluster level, to control what type of data is allowed to be shared.

This idea highlights the *separation of concerns* model presented earlier in the paper. The *devices* will collect some data at their origin, while *resources* will handle the data **locally** with *frontend services*, and if necessary, sending data to the cloud, where *backend service* may do more processing, train models, or store the data for some future usage.

Separating physical areas into groups, regions, and clusters allows us to naturally follow a certain phenomenon in detail and act proactively on that target area. For example, our model can be used for monitoring power grids, ensuring hospitals and other tenants have constant power during some catastrophic event. Existing solutions could benefit from integration with our model because of the additional monitoring of various parameters (e.g., weather conditions, vegetation growth, animal migration) in real-time and act accordingly. This will prevent outages and high damages by controlling power transmission to the consumers. If outages are predictable, it can be planned to send crew early to fix problems, minimizing overall downtime.

In healthcare applications, our model can be used for real-time monitoring, for instance, pulmonary or asthmatic patients when going for a walk in some area. Considering that in the area devices are measuring the air quality parameters, streaming the data to the frontend service (that analyzes the data and runs inside the cluster in the closest region), may result in informing the patient (and possibly also medical personal) about the danger before the damage is made.

B. COVID-19 MILAN AREA TRAFFIC CONTROL EXAMPLE

As an example of a use-case that can benefit from our model the city of Milan, Italy, can be considered, in the context of the recent COVID-19 outbreak. The city of Milan is divided into nine municipalities, numbered from 1 to 9. There is here Milan *topology* in which every municipality can have one or more *regions*. Depending on population density, implemented applications, and needs, every region can have multiple *clusters* serving only the population nearby.

Right now, the natural administrative subdivision of the city is being followed. If this changes in the future, reorganization of resources (*regions* and *clusters*) is easy and could be done dynamically, using *cluster formation protocol*. A prerequisite for this to be done: there are EC nodes deployed on the territory, and nodes are connected to the system using *health check protocol*.

During the COVID-19 outbreak in the city of Milan, an increased amount of ambulance vehicles and medical personnel had to be routed to hospitals fast. If the area of the city of Milan were separated and enhanced with the platform for smart area traffic control, utilizing the principles of our model, the ambulance vehicles could be targeted and given a higher priority, than regular vehicles.

In this scenario, EC clusters can run two kinds of *frontend services*, specifically tailored for this application: a service that follows the ambulance vehicles either using some machine learning techniques, specialized hardware, or using some other technique and a service that will regulate the traffic light control.

In a catastrophic scenario, a sudden increased number of ambulance vehicles causes an increased need for resources at the *frontend service* that regulates the traffic light control (also because of, e.g., decisions that require more processing power). Our *clusters* can be rearranged, or even a special *cluster* can be dedicated just for this purpose. If more resources are needed even *regions* can be changed, and finally, the whole *topology* can be changed to support increased requests and may be merged with a city nearby.

Patient health can be monitored in real-time [61]–[63] and data transferred to give health workers crucial information about patients on their arrival, while robotic systems can help in diagnosis and screening [64]. Combined with area traffic control application, such a platform would increase patient chances for survival, and at the same time reduce the hospital spending on unnecessary tests.

Another special *frontend service* could extract depersonalized data and transfer it to the *backend service* for the research purpose. For all mentioned above, we believe a service based on our model at the time of the COVID-19 outbreak in Milan could have been helpful for researchers, providing valuable insight in real-time. Other cities and countries could reuse the same applications strategy, if all goes well, or make the adjustments to best suit their needs. Such a service may exist only during the outbreak, when the situation is under control the service could be terminated.

When the pandemic is gone, or fewer resources are needed, the previous arrangement of the resources can be restored. If all of a sudden there is the next pandemic spike, the same strategy can be reused.

Collected data may be transferred to the *backend service* for deeper analysis, behavior modeling, or whatever other purpose. This should be regulated by the government if they want such sensitive pieces of information to be sent to the cloud or not.

C. DISCUSSION

The main advance of ECC compared to the traditional cloud-only approach is the acceleration of the communication speed. The system is more robust to network failures because local requests can still be accepted [56]. The cloud could bring huge latency for some real-time applications.

By separating some area, a city for example, into *regions* and giving them some storage and processing resources in form of *clusters* all connected into one coherent system, huge potential is being given to future developers and new human-centered applications. The most important thing is that the system would be able to operate these resources, efficiently organize them, and make them available for utilization. This is what the cloud has done a few decades ago.

The proposed model allows organization and reorganization of resources in a similar way the cloud does, allowing users to develop applications without some specialized

infrastructure for different applications. Compared to similar models, there are few benefits and few downsides.

Some of existing models [26], [61]–[63], [65] require specialized infrastructure for solving a single problem or a single application. The proposed model is more oriented towards developing a broader specter of applications without the need for a specialized hardware or software. Users should build their applications, similarly as they are building them for the cloud.

Specialized models are developed and optimized for a specific use case to take the maximum out of hardware and software. They might outperform the proposed model in terms of speed. The proposed model offers much more development freedom for the users, in terms of agility and applicability. Users will be able to create new interesting human-centered applications in the future, utilizing both CC and ECC.

Existing application models [23], [26], [27] limit users in terms of technology for building applications. Limitations are not always a bad thing (e.g., better security, known access patterns). However, in many cases users may find such limitations too restraining, for example, when their applications can be optimized with some other technology.

The proposed model does not limit users on how to develop applications, what technology, language, or framework to use. As long as the application could be virtualized (e.g., using virtual machines, containers, or unikernes). In this way, developers may reuse already acquired knowledge and best practices to develop their applications.

If a catastrophic event like the COVID-19 outbreak is in the human population, a city can organize its storage and processing resources according to priority. If all cities in a state apply the same model, then the whole state can organize its resources and completely manage its digital infrastructure, and make applications that will help its citizens in those tough times.

V. CONCLUSION AND FUTURE WORK

This paper presents a possible solution for the organization of geo-distributed EC nodes, with the addition of few proven abstractions from cloud computing like zones and regions forming micro clouds model. These abstractions allow coverage of any geographic area and yield a more available and reliable EC system, allowing us to treat micro clouds disposable.

The organization and reorganization of these abstractions are done descriptively, and their size is determined by the population needs. We present a cloud to EC mapping and give a formal model of the system, with clear SoC and edge-native application model for future EC as service development. The paper also presents a proof of concept implementation with a real-world example application, and discusses integration into existing solutions, allowing existing scheduling tools to operate in micro clouds environment.

As part of our current and future work, we are planning to test the prototype implementation in a real-world geo-distributed environment (e.g., measurements of different parameters relevant to detect hazardous occurrences, real-time detection, and alerting of changes in air quality essential for lung patients, management in power grids, etc.) to analyze the performance, network overheads, and scalability aspects of the proposed model. We are also planning to extend existing open-source tools like Kubernetes with the proposed principles.

Beside that, we are planning to extend the proposed model with remote management with configurations, security credentials, and actions over nodes in one or multiple clusters. Also, we are planning to extend the system with namespaces for usage in environments with many users in multiple teams. Namespaces provide the virtual clusters, on the same physical cluster. Additionally, we are planning to implement: complete architecture and applications monitoring, role-based access, and quotas control for different users. These features will be helpful to any administrator of such a system.

Another direction for future work is the implementation of a scheduling system for user-developed applications. Users can develop their applications with different models: (*i*) **PaaS**, where the platform is doing all the management and offers a simple interface for developers to deploy their applications, or (*ii*) **CaaS** if users require more control over resources requirements, deployment, and orchestration decisions. Last, but not least, we are planning to add several security layers to protect the system in general from malicious users.

REFERENCES

[1] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 854–864, Dec. 2016, doi: 10.1109/JIOT.2016.2584538.

[2] W. Vogels, "A head in the clouds the power of infrastructure as a service," in *Proc. 1st Workshop Cloud Comput. Appl.*, 2008.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2009-28, Feb. 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[4] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges," *J. Internet Services Appl.*, vol. 1, no. 1, pp. 7–18, May 2010, doi: 10.1007/s13174-010-0007-6.

[5] M. D. da Assunção, A. da S. Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *J. Netw. Comput. Appl.*, vol. 103, pp. 1–17, Feb. 2018, doi: 10.1016/j.jnca.2017.12.001.

[6] S. K. A. Hossain, M. A. Rahman, and M. A. Hossain, "Edge computing framework for enabling situation awareness in IoT based smart city," *J. Parallel Distrib. Comput.*, vol. 122, pp. 226–237, Dec. 2018, doi: 10.1016/j.jpdc.2018.08.009.

[7] J. Cao, Q. Zhang, and W. Shi, *Edge Computing: A Primer* (Springer Briefs in Computer Science). Cham, Switzerland: Springer, 2018. [Online]. Available: https://dblp.org/rec/series/sbcs/CaoZS18.bib, doi: 10.1007/978-3-030-02083-5.

[8] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. and Eliazar, "Why does the cloud stop computing?: Lessons from hundreds of service outages," in *Proc. 7th ACM Symp. Cloud Comput.*, M. K. Aguilera, B. Cooper, and Y. Diao, Eds, Santa Clara, CA, USA, Oct. 2016, pp. 1–16, doi: 10.1145/2987550.2987583.

[9] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, and M. G. Rabbani, "Data center network virtualization: A survey," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 2, pp. 909–928, 2nd Quart., 2013, doi: 10.1109/SURV.2012.090512.00043.

[10] P. G. Lopez, A. Montresor, D. J. Epema, A. Datta, T. H. Higashino, A. L. Iamnitchi, and M. Barcellos, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, 2015, doi: 10.1145/2831347.2831354.

[11] M. B. A. Karim, B. I. Ismail, M. Wong, E. M. Goortani, S. Setapa, L. J. Yuan, and H. and Ong, "Extending cloud resources to the edge: Possible scenarios, challenges, and experiments," in *Proc. Int. Conf. Cloud Comput. Res. Innov.*, Singapore, May 2016, pp. 78–85, doi: 10.1109/ICCCRI.2016.20.

[12] S. Alonso-Monsalve, F. García-Carballeira, and A. Calderón, "A heterogeneous mobile cloud computing model for hybrid clouds," *Future Gener. Comput. Syst.*, vol. 87, pp. 651–666, Oct. 2018, doi: 10.1016/j.future.2018.04.005.

[13] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017, doi: 10.1109/MC.2017.9.

[14] H. Ning, Y. Li, F. Shi, and L. T. Yang, "Heterogeneous edge computing open platforms and tools for Internet of Things," *Future Gener. Comput. Syst.*, vol. 106, pp. 67–76, May 2020, doi: 10.1016/j.future.2019.12.036.

[15] F. Bonomi, R. A. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for Internet of Things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments* (Studies in Computational Intelligence), vol. 546, N. Bessis and C. Dobre, Eds. Cham, Switzerland: Springer, 2014, pp. 169–186. [Online]. Available: https://dblp.org/rec/series/sci/BonomiMNZ14.bib, doi: 10.1007/978-3-319-05029-4_7.

[16] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, vol. 5, pp. 6757–6779, 2017, doi: 10.1109/ACCESS.2017.2685434.

[17] A. Khune and S. Pasricha, "Mobile network-aware middleware framework for cloud offloading: Using reinforcement learning to make reward-based decisions in smartphone applications," *IEEE Consum. Electron. Mag.*, vol. 8, no. 1, pp. 42–48, Jan. 2019, doi: 10.1109/MCE.2018.2867972.

[18] M. Chen, Y. Hao, Y. Li, C.-F. Lai, and D. Wu, "On the computation offloading at ad hoc cloudlet: Architecture and service modes," *IEEE Commun. Mag.*, vol. 53, no. 6, pp. 18–24, Jun. 2015, doi: 10.1109/MCOM.2015.7120041.

[19] R. V. Aroca and L. M. G. Gonçalves, "Towards green data centers: A comparison of x86 and ARM architectures power efficiency," *J. Parallel Distrib. Comput.*, vol. 72, no. 12, pp. 1770–1780, Dec. 2012, doi: 10.1016/j.jpdc.2012.08.005.

[20] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante, "The seminal role of edge-native applications," in *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, Milan, Italy, Jul. 2019, pp. 33–40, doi: 10.1109/EDGE.2019.00022.

[21] W. L. Hürsch and C. V. Lopes. (1995). *Separation of concerns*. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5223

[22] Y. Shao, C. Li, Z. Fu, L. Jia, and Y. Luo, "Cost-effective replication management and scheduling in edge computing," *J. Netw. Comput. Appl.*, vol. 129, pp. 46–61, Mar. 2019, doi: 10.1016/j.jnca.2019.01.001.

[23] M. Ryden, K. Oh, A. Chandra, and J. B. Weissman, "Nebula: Distributed edge cloud for data intensive computing," in *Proc. IEEE Int. Conf. Cloud Eng.*, Boston, MA, USA, Mar. 2014, pp. 57–66, doi: 10.1109/IC2E.2014.34.

[24] C. Shi, K. Habak, P. Pandurangan, M. H. Ammar, M. Naik, and E. W. Zegura, "COSMOS: Computation offloading as a service for mobile devices," in *Proc. 15th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, J. Wu, X. Cheng, X. Li, and S. Sarkar, Eds., Philadelphia, PA, USA, Aug. 2014, pp. 287–296, doi: 10.1145/2632951.2632958.

[25] B. Fitzgerald, N. Forsgren, K.-J. Stol, J. Humble, and B. Doody, "Infrastructure is software too!" *SSRN Electron. J.*, Jan. 2015, doi: 10.2139/ssrn.2681904.

[26] H. Guo, L.-L. Rui, and Z.-P. Gao, "A zone-based content pre-caching strategy in vehicular edge networks," *Future Gener. Comput. Syst.*, vol. 106, pp. 22–33, May 2020, doi: 10.1016/j.future.2019.12.050.

[27] A. C. Baktir, A. Ozgovde, and C. Ersoy, "How can edge computing benefit from software-defined networking: A survey, use cases, and future directions," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2359–2391, 4th Quart., 2017, doi: 10.1109/COMST.2017.2717482.

[28] I. Kurniawan, H. Febiansyah, and J. Kwon, *Cost-Effective Content Delivery Networks Using Clouds and Nano Data Centers*, vol. 280. Berlin, Germany: Springer-Verlag, Jan. 2014, pp. 417–424, doi: 10.1007/978-3-642-41671-2_53.

[29] R.-I. Ciobanu, C. Negru, F. Pop, C. Dobre, C. X. Mavromoustakis, and G. Mastorakis, "Drop computing: Ad-hoc dynamic collaborative computing," *Future Gener. Comput. Syst.*, vol. 92, pp. 889–899, Mar. 2017, doi: 10.1016/j.future.2017.11.044.

[30] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 68–73, 2009, doi: 10.1145/1496091.1496103.

[31] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018, doi: 10.1109/JIOT.2017.2750180.

[32] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, 2016, doi: 10.1145/2890784.

[33] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst., EuroSys*, L. Réveillère, T. Harris, and M. Herlihy, Eds., Bordeaux, France, Apr. 2015, pp. 1–17, doi: 10.1145/2741948.2741964.

[34] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Comput. Commun.*, vol. 159, pp. 161–174, Jun. 2020, doi: 10.1016/j.comcom.2020.04.061.

[35] Linux Foundation. *KubeEdge*. Accessed: Nov. 7, 2020. [Online]. Available: https://kubeedge.io/

[36] *Volunteer Computing*, Boinc, Berkeley, CA, USA, Accessed: Jul. 5, 2020. [Online]. Available: https://boinc.berkeley.edu/trac/wiki/VolunteerComputing

[37] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Commun. ACM*, vol. 53, no. 1, pp. 91–99, Jan. 2010, doi: 10.1145/1629175.1629203.

[38] A. Lèbre, J. Pastor, A. Simonet, and F. Desprez, "Revising openstack to operate fog/edge computing infrastructures," in *Proc. Int. Conf. Cloud Eng.*, Vancouver, BC, Canada, Jan. 2017, pp. 138–148, doi: 10.1109/IC2E.2017.35.

[39] A. Kurniawan, *Learning AWS IoT: Effectively Manage Connected Devices on the AWS Cloud Using Services Such as AWS Greengrass, AWS Button, Predictive Analytics and Machine Learning*. Birmingham, U.K.: Packt, 2018. [Online]. Available: https://books.google.rs/books?id=7NRJDwAAQBAJ

[40] General Electric. *GE Predix*. Accessed: Nov. 7, 2020. [Online]. Available: https://www.ge.com/digital/iiot-platform/

[41] F. R. de Souza, C. C. Miers, A. Fiorese, M. D. de Assunção, and G. P. Koslovski, "QVIA-SDN: Towards QoS-aware virtual infrastructure allocation on SDN-based clouds," *J. Grid Comput.*, vol. 17, no. 3, pp. 447–472, Sep. 2019, doi: 10.1007/s10723-019-09479-x.

[42] J. R. Hamilton, "An architecture for modular data centers," in *Proc. CIDR*, Asilomar, CA, USA, Jan. 2007, pp. 306–313. [Online]. Available: http://cidrdb.org/cidr2007/papers/cidr07p35.pdf

[43] K. Sonbol, Ö. Özkasap, I. Al-Oqily, and M. Aloqaily, "EdgeKV: Decentralized, scalable, and consistent storage for the edge," *J. Parallel Distrib. Comput.*, vol. 144, pp. 28–40, Oct. 2020, doi: 10.1016/j.jpdc.2020.05.009.

[44] J. Wang, D. Crawl, I. Altintas, and W. Li, "Big data applications using workflows for data parallel computing," *Comput. Sci. Eng.*, vol. 16, no. 4, pp. 11–21, Jul. 2014, doi: 10.1109/MCSE.2014.50.

[45] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable weakly-consistent infection-style process group membership protocol," in *Proc. Int. Conf. Dependable Syst. Netw.*, Bethesda, MD, USA, 2002, pp. 303–312, doi: 10.1109/DSN.2002.1028914.

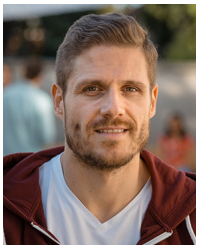
[46] C. Li, J. Bai, Y. Chen, and Y. Luo, "Resource and replica management strategy for optimizing financial cost and user experience in edge cloud computing system," *Inf. Sci.*, vol. 516, pp. 33–55, Apr. 2020, doi: 10.1016/j.ins.2019.12.049.

[47] E. Cau, M. Corici, P. Bellavista, L. Foschini, G. Carella, A. Edmonds, and T. M. Bohnert, "Efficient exploitation of mobile edge computing for virtualized 5G in EPC architectures," in *Proc. 4th IEEE Int. Conf. Mobile Cloud Comput., Services, Eng. (MobileCloud)*, Oxford, U.K., Mar. 2016, pp. 100–109, doi: 10.1109/MobileCloud.2016.24.

[48] W. Yu and C.-L. Ignat, "Conflict-free replicated relations for multi-synchronous database management at edge," in *Proc. IEEE Int. Conf. Smart Data Services (SMDS)*, Beijing, China, Oct. 2020, pp. 113–121. [Online]. Available: https://hal.inria.fr/hal-02983557

[49] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a service (XaaS) on the cloud: Origins, current and future trends," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, New York, NY, USA, Jun. 2015, pp. 621–628, doi: 10.1109/CLOUD.2015.88.

[50] X. Jin, S. Chun, J. Jung, and K. Lee, "Iot service selection based on physical service model and absolute dominance relationship," in *Proc. 7th IEEE Int. Conf. Service-Oriented Comput. Appl. (SOCA)*, Matsue, Japan, Nov. 2014, pp. 65–72, doi: 10.1109/SOCA.2014.24.
[51] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Comput.*, vol. 8, no. 4, pp. 14–23, Oct./Dec. 2009, doi: 10.1109/MPRV.2009.82.
[52] Y. Yao, B. Xiao, W. Wang, G. Yang, X. Zhou, and Z. Peng, "Real-time cache-aided route planning based on mobile edge computing," *IEEE Wireless Commun.*, vol. 27, no. 5, pp. 155–161, Oct. 2020, doi: 10.1109/MWC.001.1900559.
[53] R. Hu and N. Yoshida, "Explicit connection actions in multiparty session types," in *Fundamental Approaches to Software Engineering* (Lecture Notes in Computer Science), vol. 10202, M. Huisman and J. Rubin, Eds. Uppsala, Sweden: Springer, Apr. 2017, pp. 116–133, doi: 10.1007/978-3-662-54494-5_7.
[54] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," in *Proc. 35th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, San Francisco, CA, USA, Jan. 2008, pp. 273–284, doi: 10.1145/1328438.1328472.
[55] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 16–21, Sep. 2017, doi: 10.1109/MCC.2017.4250939.
[56] R.-A. Cherrueau, M. Delavergne, and A. Lebre, "Geo-distribute cloud applications at the edge," in *Proc. 27th Int. Eur. Conf. Parallel Distrib. Comput.*, Lisbon, Portugal, Aug. 2021, pp. 1–15. [Online]. Available: https://hal.inria.fr/hal-03212421
[57] M. Beck, M. Werner, S. Feld, and T. Schimper, "Mobile edge computing: A taxonomy," in *The 6th Int. Conf. Adv. Future Internet (AFIN)*, Jan. 2014, pp. 48–55.
[58] Kubernetes. *Running Multiple Zones*. Accessed: Nov. 7, 2020. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/multiple-zones/
[59] P. Andrade, T. Bell, J. Eldik, G. Mccance, B. Panzer-Steindel, M. Santos, and U. Schwickerath, "Review of CERN data centre infrastructure," *J. Phys. Conf. Ser.*, vol. 396, Dec. 2012, Art. no. 042002.
[60] X. Huang and N. Ansari, "Content caching and distribution at wireless mobile edge," *IEEE Trans. Cloud Comput.*, early access, May 18, 2020, doi: 10.1109/TCC.2020.2995403.
[61] M. Al-Khafajiy, T. Baker, C. Chalmers, M. Asim, H. Kolivand, M. Fahim, and A. Waraich, "Remote health monitoring of elderly through wearable sensors," *Multim. Tools Appl.*, vol. 78, no. 17, pp. 24681–24706, 2019, doi: 10.1007/s11042-018-7134-7.
[62] Y. J. Jeon and S. J. Kang, "Wearable sleepcare kit: Analysis and prevention of sleep apnea symptoms in real-time," *IEEE Access*, vol. 7, pp. 60634–60649, 2019, doi: 10.1109/ACCESS.2019.2913849.
[63] G. Chiarini, P. Ray, S. Akter, C. Masella, and A. Ganz, "MHealth technologies for chronic diseases and elders: A systematic review," *IEEE J. Sel. Areas Commun.*, vol. 31, no. 9, pp. 6–18, Sep. 2013, doi: 10.1109/JSAC.2013.SUP.0513001.
[64] Y. Shen, D. Guo, F. Long, L. A. Mateos, H. Ding, Z. Xiu, R. B. Hellman, A. King, S. Chen, C. Zhang, and H. Tan, "Robots under COVID-19 pandemic: A comprehensive survey," *IEEE Access*, vol. 9, pp. 1590–1615, 2021, doi: 10.1109/ACCESS.2020.3045792.
[65] E. Baccarelli, P. G. V. Naranjo, M. Scarpiniti, M. Shojafar, and J. H. Abawajy, "Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study," *IEEE Access*, vol. 5, pp. 9882–9910, 2017, doi: 10.1109/ACCESS.2017.2702013.



MILOŠ SIMIĆ received the B.Sc. and M.Sc. degrees in computer science from the Faculty of Technical Sciences, University of Novi Sad, in 2014 and 2015, respectively, where he is currently pursuing the Ph.D. degree. He has been a Teaching Assistant with the Department of Computing and Control Engineering, Faculty of Technical Sciences, University of Novi Sad, since 2015. His research interests include distributed systems, cloud computing, edge computing, big data, and service oriented architectures.

IVAN PROKIĆ received the B.Sc. and M.Sc. degrees in mathematics from the Faculty of Sciences, University of Novi Sad, in 2012 and 2014, respectively, and the Ph.D. degree in applied mathematics from the Faculty of Technical Sciences, University of Novi Sad. He has been a Teaching Assistant with the Department of Fundamentals Sciences, Faculty of Technical Sciences, University of Novi Sad, since 2020. His research interests include formal methods, more specifically concurrency theory, process calculi, and type theory.

JOVANA DEDEIĆ received the B.Sc. and M.Sc. degrees in applied mathematics from the Faculty of Sciences, University of Novi Sad, in 2010 and 2011, respectively, where she is currently pursuing the Ph.D. degree with the Faculty of Technical Sciences. She has been a Teaching Assistant with the Department of Fundamentals Sciences, Faculty of Technical Sciences, University of Novi Sad, since 2012. Her research interests include process calculi, concurrency theory, expressive power of languages with concurrency, and type theory.

GORAN SLADIĆ received the B.Sc. (Dipl.-Ing.), M.Sc. (Magister), and Ph.D. degrees in computer science from the Faculty of Technical Sciences, University of Novi Sad, in 2002, 2006, and 2011, respectively. He has been an Associate Professor with the Department of Computing and Control Engineering, Faculty of Technical Sciences, University of Novi Sad, since 2016. He has published over 70 articles and participated in more than 20 projects. His research interests include cyber security, blockchain, software architectures, and context-aware computing.

BRANKO MILOSAVLJEVIĆ received the Ph.D. degree from the University of Novi Sad, in 2003. He is currently a Professor of computer science with the Faculty of Technical Sciences, University of Novi Sad. He has published over 150 articles and participated or lead in more than 20 projects. His research interests include digital libraries, net-centric software architectures, and context-aware computing.