A Service-Defined Approach for Orchestration of Heterogeneous Applications in Cloud/Edge Platforms

Gabriele Castellano, Flavio Esposito and Fulvio Risso

***Abstract*—Edge Computing is moving resources toward the network borders, thus enabling the deployment of a pool of new applications that benefit from the new distributed infrastructure. However, due to the heterogeneity of such applications, specific orchestration strategies need to be adopted for each deployment request. Each application can potentially require different optimization criteria and may prefer particular reactions upon the occurrence of the same event. This paper presents a Service-Defined approach for orchestrating cloud/edge services in a distributed fashion, where each application can define its own orchestration strategy by means of declarative statements, which are parsed into a Service-Defined Orchestrator (SDO). Moreover, to coordinate the coexistence of a variety of SDOs on the same infrastructure while preserving the resource assignment optimality, we present DRAGON, a Distributed Resource AssiGnment and OrchestrationN algorithm that seeks optimal partitioning of shared resources between different actors. We evaluate the advantages of our novel Service-Defined orchestration approach over some representative edge use cases, as well as measure convergence and performance of DRAGON on a prototype implementation, assessing the benefits compared to conventional orchestration approaches.**

***Index Terms*—Orchestration, Mathematical optimization, Distributed algorithms, Distributed management.**

I. Introduction

With the expansion of Cloud Computing toward the edge of the network, the diversity of the involved applications and of their management requirements has been drastically exacerbated. Indeed, the largely heterogeneous set of (often distributed) applications running over Cloud/Edge platforms may have different and unpredictable deployment objectives; furthermore, reacting differently to network events, such as a traffic load increase, became a necessity. In the above circumstance, some applications may need to scale up or out, while others may migrate to a more convenient location. Others may even modify the service behavior without asking for additional resources. For instance, the optimization of a Content Distribution Network (CDN) service may require to monitor the average miss-rate on deployed caches, to identify occasional hot spots such as flash crowd during live events; this in turn requires optimizing the service by relocating *and possibly duplicating* some caches. Vice versa, a video streaming application may need to monitor the provided quality of service in terms of Frames Per Second (FPS), opting for the deployment of a more aggressive video transcoder whenever a particular high load deteriorates the current frame rate, instead of asking for more processing resources.

While large service providers may run their applications on ad-hoc platforms and therefore can define their best optimization strategies, most of the other providers have to rely on third-party Edge/Cloud platforms, which host heterogeneous applications and hence in need of adopting service-agnostic one-size fits-all orchestration strategies to handle the entire applications pool. Embedding of virtual services is thus accomplished by optimizing generic metrics such as energy saving, latency, load balancing [1]. Similarly, load increases are managed by taking into account conventional infrastructure metrics such as CPU and memory consumption through the traditional auto-scaling techniques (i.e., scaling up/down is performed whenever consumption goes above/below a certain threshold). In summary, existing orchestration approaches cannot *(i)* take their decisions based on service-specific parameters (i.e., cache miss-rate) and *(ii)* perform service-specific actions such as modifying the internal behavior of the service (e.g., switching the transcoder), as only infrastructure-related actions are possible. As a consequence, they often fail to optimize application-specific goals.

This paper fills this knowledge gap by proposing a novel Platform-as-a-Service (*PaaS*) approach where the orchestration component is fully distributed and provides the possibility to instantiate, prior to service deployment, small-scoped *Service-Defined Orchestrators* (SDOs), each dedicated to handling the life-cycle of a particular application. Such orchestrator may operate by modifying the current overall resource assignment (e.g., if the application needs more resources or may release some) as well as merely act on the application itself to adapt its operational state to the current infrastructure situation (e.g., switch the used video codec on a streaming service component). To avoid exacerbating applications development, which would also include such an orchestration module, we present a distributed architecture where SDOs are automatically synthesized starting from an *Orchestration Behavioral Model*, used by the service provider to specify metrics and objectives needed to build the proper orchestration strategy for the given application by means of high-level declarative statements.

The coexistence of such a variety of small orchestrators (SDOs) operating on a shared infrastructure introduces the problem of coordinating resource allocation. To address this problem, we extend our prior work [2] by designing a fully Distributed Resource AssiGnment and OrchestratioN (DRAGON) algorithm, which enables a pool of SDOs to reach an agreement on how infrastructure resources should be (temporarily) partitioned among them, by means of a fully distributed decision process. Our contributions are as follows:
Design contribution. We present a novel distributed orchestration architecture (Section III) and detail the design of its core component, namely the Service-Defined Orchestrator

G. Castellano and F. Risso are with the Department of Control and Computer Engineering, Politecnico di Torino, Italy, gabriele.castellano@polito.it, fulvio.risso@polito.it.
F. Esposito is with the Computer Science Department at Saint Louis University, USA, flavio.esposito@slu.edu.

(Section IV). We also formally define a Declarative Behavioral Model used to synthesize an SDO from an high-level description of its orchestration strategies.

Algorithmic contribution. We detail our DRAGON asynchronous algorithm (Section V) formalizing its complete multi-node version — the single-node version was previously presented in [2] — and we show how it provides non-improvable guarantees on resource assignment performance to independent SDOs and a convergence bound (Section VI).

Evaluation contribution. We assess the benefits of our approach analyzing three reference use cases: *(i)* QoS degradation for a video streaming application, *(ii)* cache placement for a CDN provider and *(iii)* edge migration for mobile gaming. Moreover, we evaluate both convergence and performance properties of DRAGON, comparing them with traditional approaches (Section VII). Our findings confirm the applicability of this approach in edge infrastructures and the performance advantages over conventional one-size fits-all paradigms.

II. Related Work

The orchestration of infrastructure resources is primarily investigated in several recent works. Most of them focus on the VNF deployment problem proposing algorithms that rely on a centralized solver [3]–[5]. Among them, some propose a joint computation of different phases of the problem to seek better optimization. For instance, [4] proposes an algorithm in which scaling, placement, and mapping are optimized jointly, while in [5] authors solve the embedding problem combined with the service composition one. Other works as [6], [7] investigate instead the problem of joint orchestration among multiple infrastructure providers, proposing distributed optimization approaches. In [6], the authors propose a game-theoretic approach, while [7] illustrates a decentralized algorithm on top of an existing multi-domain architecture developed in the 5Gex project [8]. This last one addresses relationships across multi-administrative domain orchestrators, distinguishing between *Resource Orchestration*, service-agnostic and performed at the infrastructure level, and *Service Orchestration*, i.e., service-specific management of a single slice [9]. Within the project, a distributed architecture enabling multi-domain resource orchestration is proposed, as well as analysis of their coordination [8], [10]. However, no focus is given on the interaction between service orchestrators, which is only theorized in [9] and is mostly outside the scope of the project.

Infrastructure level orchestration alone does not provide service specific optimization.

Indeed, recent work on edge computing [11]–[13] proposes ad-hoc optimization focusing single edge application separately. For instance, [11] optimizes the placement of roadside units on new generation vehicular networks; [12] focuses on the service placement problem in mobile applications, where the dynamism of user's location plays a key role; [13] proposes an optimal allocation for high-performance video streaming in 5G networks. While the above solutions enable optimization of isolated applications, to the best of out knowledge, it is still unclear how such a variety of service embedding algorithms can coexist on a shared infrastructure without undermining the overall performance optimality. Mesos [14] enables dynamic decisions on resource partitioning and allows the coexistence of diverse cluster computing frameworks, each one featuring different scheduling needs, on top of the same cloud infrastructure. This solution exploits a centralized master that assigns resources dynamically by making offers to demanding frameworks. SONATA [15] introduced the concept of service-specific optimization in the ETSI NFV reference architecture, extending it with Service-Specific Managers (SSMs) micro-services, so that service awareness can be dynamically introduced to the generic orchestrator. In this work, we even go further by enabling a fully distributed approach to remove such a centralized component. Indeed, mandating the existence of a centralized component (featured both in [14], [15]) may not be suitable in a scenario where services are executed on scattered compute nodes, e.g., at the edge of the network, which feature arbitrary and unpredictable topologies that evolve over time. In this context, we should rely on solutions that provide decentralized consensus (e.g., Paxos [16] and RAFT [17]) to reach agreement on resource assignment. In particular, RAFT is implemented in widespread SDN controllers to enable data-store replication and resiliency among multiple controller instances. Some of its limitations have already been highlighted in [18], where authors also propose an enhancement of RAFT to improve recovery times on the specific use case of SDN Controllers. More generally, none of [16], [17] simultaneously provides *(i)* guarantees on convergence time and performance, and *(ii)* a fully distributed approach.

III. Overall Architecture

This section introduces our distributed orchestration architecture (Figure 1). We identified three separated operational planes that have a correspondence with the well-established cloud layered model (XaaS) [19]. An Infrastructure Plane (i.e., Infrastructure-as-a-Service) provides elementary resources (such as computing, networking and storage) by virtualizing a set of edge or cloud servers (compute nodes) scattered across the network. A set of Service-Defined Orchestrators, each dedicated to the management of a given application, constitutes a distributed Platform-as-a-Service that we name Orchestration Plane. Finally, the whole set of edge/cloud end applications running on top of the distributed infrastructure constitutes the Service Plane (i.e., Software-as-a-Service).

A. Service Plane

To preserve the generality of our approach, we assume that applications may follow the micro-service paradigm [20], that is, services are composed by small components, each specialized on a given task. For instance, a video streaming application may feature a video source, a transcoder and a web server, each one potentially deployed on a separate location according to the placement decisions enforced by the orchestrator.

Figure 1 shows a Service Plane where each application requires the deployment of multiple components. The proposed architecture assumes that each application component may feature multiple valid implementations when physically

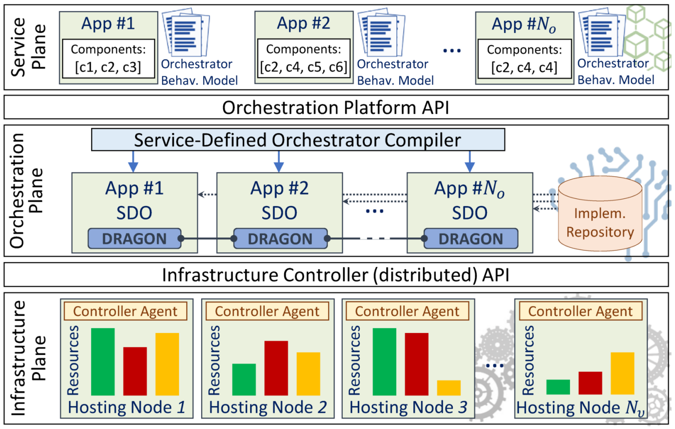


Fig. 1: Overall distributed orchestration architecture.

deployed on the infrastructure. Each implementation may feature different characteristics, for example, the execution environment could be over virtual machines, containers, or dedicated hardware. Each implementation may require different resources and provide different QoS levels. Based on the scenario, an application may benefit more from a particular implementation policy set. As shown in Figure 1, the Orchestration Plane features an *Implementation Repository* that stores valid implementations for well-known components, along with details about their configuration and resource requirements. Additionally, at deployment request time, an application may customize further each one of its components.

An application deployment request consists of *(i)* the list of components to be deployed, along with their virtual topology, *(ii)* any custom implementation needed for the deployment and *(iii)* a declarative description of the orchestration strategies to be used to properly deploy and manage the application.

B. Orchestration Plane

Our approach defines a highly modular and dynamic Orchestration Platform, whose building blocks are our Service-Defined Orchestrators (SDOs), each *(i)* dedicated to a single application and *(ii)* generated and executed on demand. Note that such an approach makes the overall PaaS behavior defined by the application itself.

The orchestration platform accepts application deployment requests. These requests come with additional information that is used to drive the orchestration process (i.e., resource allocation, placement and run-time management) in a way that is optimal for that specific application. Metadata that comes with each application deployment request are in the form of an *Orchestration Behavioral Model*, which features declarative statements used to actually generate the Service-Defined Orchestrator by means of an *SDO Compiler*. Details regarding the Orchestration Behavioral Model and how its declarative statements are composed to generate an SDO are discussed in Section IV.

Whenever a new SDO is generated, it is instantiated as an extension of the existing platform and employed to manage the orchestration of the corresponding application. Orchestration is performed with respect to both *deployment* and *run time*. At deployment time, the SDO interacts with the infrastructure to decide where each component should be physically deployed (placement) and the amount of resources to reserve (resource allocation). At run time, the SDO monitors the state of both application components and infrastructure, reacting to suboptimal placement and resource allocation. SDO actions include rescheduling components or resizing applications on demand.

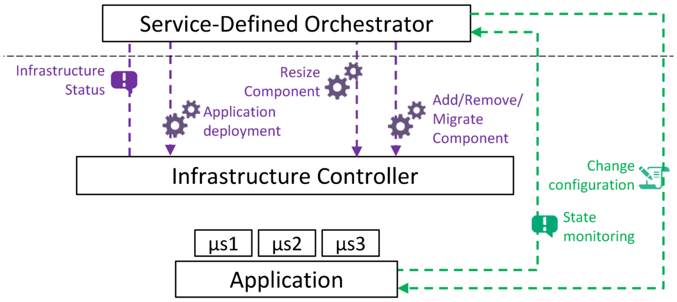


Fig. 2: Interactions between an SDO and *(i)* the infrastructure controller and *(ii)* components of the managed application.

C. Infrastructure Plane

After orchestration decisions have been taken, application components are physically deployed on a shared hosting infrastructure. The infrastructure is partitioned in multiple hosting nodes (Figure 1), each featuring different physical capacity in terms of resources of different types (e.g., CPUs, storage, network bandwidth, etc.). The current state of the hosting infrastructure and the state of each deployed application components are dynamically reported to the relevant SDOs by each hosting node, by means of a distributed message broker. Additionally, infrastructure nodes expose resource controller APIs through which SDOs can deploy and manage application components. Figure 2 highlights SDO interactions with both the infrastructure and application components.

IV. Service-Defined Orchestrator

This section provides details on the Service-Defined Orchestrator (SDO), the on-demand generated piece of the platform that manages deployment and run time of a single application. We first provide a formal definition of the Orchestration Behavioral Model, used to describe a specific SDO behavior through declarative rules. Then, the architecture and synthesis of an SDO are detailed, and a practical example is discussed.

A. Orchestrator Behavioral Model (OBM)

By definition, an SDO cannot be a generic module, as it should necessarily be specialized for each particular application. Application needs in an edge/cloud environment are usually unknown a priori, so we need a mechanism that allows, on demand, generation of any desired orchestration strategy. We propose an approach that derives a specialized SDO starting from a high level declarative description. In this section we formalize such a description through an *Orchestration Behavioral Model* (OBM). Our design generalizes the application specific approach of [21] by adapting some of the concepts from [22] on the formalization of declarative workflows, with the aim of subsuming any deployment orchestration strategy and encompassing multiple run-time situations.

An OBM instance is provided with the application as deployment metadata, specified by the service provider. It should feature at least the following: *(i)* parameters which the SDO should be aware of, such as infrastructure and/or application state; *(ii)* the objective that should be optimized; *(iii)* events that may occur and actions to be performed in response. We formally define the OBM by providing the following abstractions.

Definition 1. *(state $\mathcal{S}$). We define as state $\mathcal{S} = \mathcal{S}_A \cup \mathcal{S}_I \cup \mathcal{S}_O$ the set of variables, parameters and, in general, configurations, that the SDO can have access to. We distinguish three separate sets composing it: Application State ($\mathcal{S}_A$), Infrastructure State ($\mathcal{S}_I$) and SDO State ($\mathcal{S}_O$).*

Each element $s \in \mathcal{S}$ represents a generic readable and, possibly, configurable parameter within a given area, and is associated with few information (e.g., name, type, scope).

The *Application State* ($\mathcal{S}_A$) concerns the current deployment of each application component. It includes *(i)* the list of implementations currently chosen for each component (and where they have been physically deployed), *(ii)* their configuration and *(iii)* any *operational variable*, i.e., read-only data that the SDO may obtain by directly querying one or more components (e.g., the current miss-rate measured on a given deployed content cache).

The *Infrastructure state* ($\mathcal{S}_I$) is mainly the set of information the SDO obtains from the hosting nodes below, namely, their resource capacity and topology data. Since more than one SDO concurrently allocates resources over the same infrastructure, we add another piece of information to the Infrastructure State, i.e., how much resources the SDO is allowed to allocate at the moment. This is obtained by each SDO through our distributed agreement algorithm that will be described in Section V.

Additionally, the OBM features an *SDO State* ($\mathcal{S}_O$), which is maintained internally to the SDO itself and can be used to store some run-time information, thus enabling the definition of stateful behaviors.

Definition 2. *(constraints $\mathcal{C}$). We define a set of constraints $\mathcal{C}$, where each element $\gamma \in \mathcal{C}$ is a mathematical statement (equation or inequation) between two functions $f_L, f_R \colon \mathcal{S}^{|\mathcal{S}|} \to \mathbb{R}$ defined on state variables $s \in \mathcal{S}$.*

Constraints represent additional requirements associated to a given application. They can state the maximum latency between two components, specific characteristics of the physical nodes where a given component has to be deployed, and more. If constraints are specified in the OBM, they are interpreted by the generated SDO as hard requirements that should always be satisfied, thus discarding any deployment solution that would violate them. Moreover, since variations that may occur at run-time in the *State* may possibly lead to a constraint violation, *Actions* to be performed (see below) upon such violation must be specified for each declared constraint.

Definition 3. *(events E). Given a state $\mathcal{S}$, a particular set of variations that may occur at run-time on its variables may be declared to be an event $\boldsymbol{e} \in E$, where E is the set of all the events declared in the OBM.*

A service parameter that exceeds a given threshold, the amount of a given resource that drops below the configuration requirement, the expiration of a timer defined at run-time on the SDO State, and more, can identify situations where the application is suffering and reconfiguration actions should be performed. An *Event* may be defined on a single state variable variation, or even when more than one of the variables change in a predefined way. Each *variation* is defined through: *(i)* the reference to the state variable in question; *(ii)* the kind of variation that must be observed, i.e. *equal* to, *higher* or *lower* a threshold, or it simply changes in *any* way; *(iii)* the value, if any, to which the changing variable should be compared, which can be a static value (e.g., a string or a number) or even a reference other variables on the state. Each event is labeled with a name, which is used to associate *Action*(s) that should be performed upon its occurrence.

Definition 4. *(actions $\mathcal{A}$). Given an event $\boldsymbol{e} \in E$ that may occur on a state $\mathcal{S}$, we define as action $\boldsymbol{a} \in \mathcal{A}$, a vector of functions $a_i \colon \mathcal{S}^{|\mathcal{S}|} \to \mathcal{S}$, each giving the new value to "write" on a particular state variable $s_i \in \mathcal{S}$. The size of the action vector $\boldsymbol{a}$ represents the number of write operations to be performed on the state.*

Executing an action may consist in one or more of the following: *(i)* modify the configuration of a given application component; *(ii)* reschedule the deployment of the application, or scale/migrate just a particular component; *(iii)* update some local variables of the SDO State, e.g. to modify the SDO future behavior. Whenever an action is invoked, it takes as implicit parameters any variation registered by the triggering event.

Definition 5. *(objective o). Given a state $\mathcal{S}$ declared in an OBM, we define the objective of the associated SDO, and we denote it with $o \colon \mathcal{S}^{|\mathcal{S}|} \to \mathbb{R}$, the numerical function that the SDO should optimize during the application deployment.*

The objective function of an application should model one or more service QoS metrics (e.g., the frame rate in a video streaming application) through variables that correspond to placement and resource assignment decision during the deployment. The objective optimization process is modeled through a default, implicit *Action* $\boldsymbol{a}_o$, which is automatically invoked at deployment time. Additionally, one can declare to invoke the same action in response to some particular events, in order to reschedule application components on resources from scratch when necessary.

In declaring their own orchestration strategies through the OBM, service providers are able to take into account different situations the service may face while operating. The remote need of re-defining the strategies while the service is operating may constitute a limitation. However, in a real framework product, an SDO specified through declarative statements should provide an increased level of flexibility compared to developing and deploying ad-hoc software.

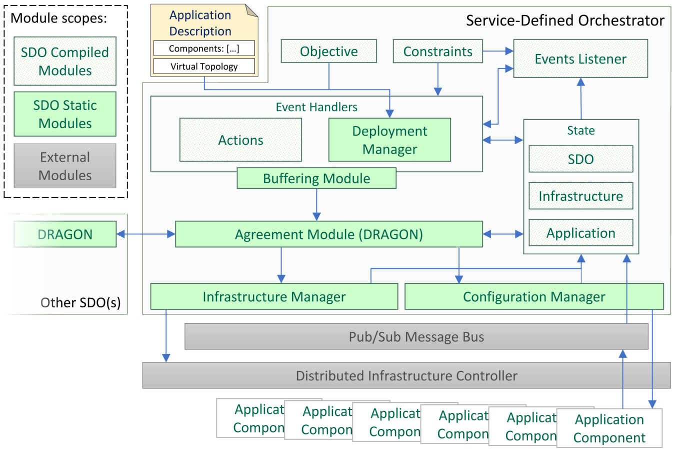


Fig. 3: Overall architecture of the Service-Defined Orchestrator.

B. SDO Architecture

Figure 3 shows the architecture of the Service-Defined Orchestrator. The figure distinguishes between modules that are dynamically generated from the Orchestration Behavioral Model, and those that are fixed (hence application independent). Each of the former modules has a direct correspondence to a precise piece of the OBM.

The *State Module* maintains the run-time information about all the state variables described in the dedicated section of the OBM (Definition 1), distinguishing between SDO internal variables, information related to the infrastructure and to each deployed application component. An *Event Listener* implements the detection of events declared in the OBM through Definition 3. Whenever there is a variation on one of the relevant state variables, the Event Listener checks for events that may have occurred. Additionally, this module also checks if a state variation causes a violation of one of the defined constraints. In any of these cases, the Event Listener invokes the corresponding *Handler*, that is one (or more) of the Actions defined in the OBM. If the particular event that occurred requires to reschedule the entire deployment of the application, action $\boldsymbol{a}_o$, i.e., the application deployment, is invoked instead. This action implements the optimization of the application objective declared on the OBM and is performed by the *Deployment Manager*, which schedules a solution based on *(i)* the current state, *(ii)* defined constraints and *(iii)* the *Application Description*. Upon SDO startup, this deployment action is automatically triggered by the Event Listener.

Whenever an action is triggered, all writing operations are buffered by a helper module, which checks if any of them requires the acquisition of additional resources from the physical infrastructure. If this is the case, the execution is mediated by an *Agreement Module*, which, through a distributed consensus algorithm, negotiates resource assignment with other SDOs operating over the same infrastructure (details are given in Section V). After the agreement, adjustments deriving from executing the action are propagated to the relevant modules.[1] In particular, an *Infrastructure Manager* acts as an interface towards the infrastructure controller and is in charge of allocating resources on needed hosting nodes and scaling up/down instantiated components. On the other hand, a *Configuration Manager* pushes any new configuration on the appropriate deployed component. Both changes on infrastructure and on components are also propagated into the corresponding portion of state maintained within the SDO. Additionally, the state is also updated any time a change notification is received from the infrastructure or any application component. Such communication occurs over a pub/sub based message bus, while the state and configuration of each component are described using the YANG language.

C. A practical use case: the video streaming application

We now provide a practical example of an orchestration strategy declared through our model. As a reference use case, we use a video streaming application. For the sake of brevity and clarity, we only focus the scope of a single component within the service run-time.

Let us consider a video streaming application whose components are: *(i)* a video transmitter (the media source), *(ii)* a transcoder, *(iii)* a web server and *(iv)* a series of clients consuming the output video streams. The transcoder takes as input the original video stream and generates multiple output streams at different bit rates and resolutions, so that each client may select the most appropriate stream based on the available bandwidth. Let us assume that the available implementation is configurable with the number of output streams to be generated. Each of these configurations has associated a minimum requirement in terms of CPU resources. The transcoder may suffer in situations where resources assigned are not enough to guarantee the proper generation of output streams.

In such a case, an example of a service-specific orchestration strategy is to fix the number of output streams (transcoder configuration) according to the available resources. A constraint is defined on a variable of the Infrastructure State, i.e., the available CPU resources. The constraint definition references a parameter in the transcoder component configuration: when the available CPU value drops below the requirement specified for the current configuration, the constraint is violated and an event is triggered. The action associated with this violation features a single write operation defined as follows: the variable to modify is on the Application State, i.e., the set of output streams; the new value to assign is computed by selecting, among the available setups, the one that *(i)* provides the higher number of output streams and *(ii)* fits the newly available CPU resource. This run-time orchestration strategy, alone, does not require any write operation on the Infrastructure State.

V. Distributed Resource AssiGnment and OrchestratioN (DRAGON) Algorithm

Using more than one orchestrator to allocate resources over the same physical infrastructure is a natural approach to enable service-centric optimization. However, this introduces the problem of coordinating resource allocation while preserving the resource assignment optimality. In fact, each orchestrator can potentially seek different optimization criteria due to

[1] Note that such adjustments may derive both from the execution of a local Action, and from any change on the equilibrium with external SDOs.

heterogeneous requirements of applications. In this section, we first define the (NP-hard) orchestrator-resources assignment problem by leveraging linear programming and then present DRAGON, a distributed approximation algorithm.

A. Resource Assignment Problem

Let us model an *application* as a multiset — a set in which element repetition is allowed — whose elements are selected among N_μ (abstract) application components to be embedded on a shared (physical) edge infrastructure. A *component* is an abstract instance of a physical function, e.g., a load balancer, a video transcoder or a content cache, which can be implemented by selecting the best possible physical *implementation* among the N_f available ones. In fact, each implementation may feature different characteristics such as execution environment (virtual machine, container, dedicated hardware), required resources, or the capability to provide a specific level of QoS.

The infrastructure is partitioned in N_v hosting nodes, each one with potentially different physical capacities. We assume that each implementation consumes a given amount of resources such as CPU, storage, memory, network bandwidth, etc., which are modeled with N_ρ different types.

Finally, let us assume that the deployment of each application is managed by a Service-Defined Orchestrator. We consider a total of N_o SDOs, all simultaneously demanding resources from a shared edge infrastructure, each one following a potentially different optimization strategy. We assume that the SDO will select the best (feasible) implementations that are required to realize application components, then allocate them in the most appropriate location.

Our goal is to maximize a global utility U while finding an infrastructure-bounded orchestrator-resources assignment that allows the deployment of each application. We define an orchestrator-resources assignment to be *infrastructure-bounded* if the consumption of all assigned components allocated on each hosting node does not exceed the $\boldsymbol{\rho}_n$ available resources on that node.

We model the orchestrator-resources assignment problem with an integer program; its binary decision variable x_{ijn} is equal to one if an instance of the implementation j has been assigned to the SDO i on hosting node n and to zero otherwise.

$$\text{maximize} \qquad \sum_{i=1}^{N_o}\sum_{j=1}^{N_f}\sum_{n=1}^{N_v} U_{ijn}(\boldsymbol{x}_i)x_{ijn} \tag{1.1}$$

subject to

$$\sum_{i=1}^{N_o}\sum_{j=1}^{N_f} x_{ijn}c_{jk} \leq \rho_{nk} \qquad \forall k \in \mathcal{K},\ \forall n \in \mathcal{N} \tag{1.2}$$

$$\sum_{j=1}^{N_f}\sum_{n=1}^{N_v} x_{ijn} = \sum_{m=1}^{N_\mu} (\sigma_{im})y_i \qquad \forall i \in \mathcal{I} \tag{1.3}$$

$$\sum_{j=1}^{N_f}\left(\sum_{n=1}^{N_v} x_{ijn}\right)\lambda_{mj} \geq y_i \qquad \forall m \in \mathcal{M},\ \forall i \in \mathcal{I} \tag{1.4}$$

$$\sum_{n=1}^{N_v} x_{ijn} \leq 1 \qquad \forall j \in \mathcal{J},\ \forall i \in \mathcal{I} \tag{1.5}$$

$$\sum_{j=1}^{N_f} x_{ij} \geq 1 - N_f y_i \qquad \forall i \in \mathcal{I} \tag{1.6a}$$

$$\sum_{j=1}^{N_f} x_{ij} \leq 1 N_f y_i \qquad \forall i \in \mathcal{I} \tag{1.6b}$$

$$x_{ijn} \in \{0,1\} \qquad \forall (i,j,n) \in \mathcal{I} \times \mathcal{J} \times \mathcal{N} \tag{1.7a}$$

$$y_i \in \{0,1\} \qquad \forall i \in \mathcal{I} \tag{1.7b}$$

$$c_{jk} \in \mathbb{N} \qquad \forall (j,k) \in \mathcal{J} \times \mathcal{K} \tag{1.7c}$$

$$\rho_{nk} \in \mathbb{N} \qquad \forall (n,k) \in \mathcal{N} \times \mathcal{K} \tag{1.7d}$$

$$\lambda_{mj} \in \{0,1\} \qquad \forall (m,j) \in \mathcal{M} \times \mathcal{J} \tag{1.7e}$$

$$\sigma_{im} \in \{0,1\} \qquad \forall (i,m) \in \mathcal{I} \times \mathcal{M} \tag{1.7f}$$

where $\boldsymbol{x}_i \in \{0,1\}^{N_f \times N_v}$ is the *assignment vector* for SDO i, whose $j^{th} \times n^{th}$ element is x_{ijn}. The auxiliary variables y_i are equal to 1 if at least an instance of any component implementation has been assigned to SDO i, and 0 otherwise (constraints 1.6a, 1.6b, 1.7b). The index sets are defined as $\mathcal{I} \triangleq \{1,\ldots,N_o\}$, $\mathcal{M} \triangleq \{1,\ldots,N_\mu\}$, $\mathcal{J} \triangleq \{1,\ldots,N_f\}$, $\mathcal{K} \triangleq \{1,\ldots,N_\rho\}$ and $\mathcal{N} \triangleq \{1,\ldots,N_v\}$. The variable ρ_{nk} represents the amount of resource k available on node n; furthermore, we denote $\boldsymbol{\rho}_n \in \mathbb{N}_n^N$ the capacity of node $n \in \mathcal{N}$. With $c_{jk} \in \mathbb{N}$ we capture the cost of implementation j in terms of resource k; thus, we name $\boldsymbol{c}_j \in \mathbb{N}_\rho^N$ the cost vector of implementation $j \in \mathcal{J}$. We set $\lambda_{mj} = 1$ if the abstract component m can be implemented (i.e., deployed) through j, while $\sigma_{im} = 1$ if m is needed by the application orchestrated by SDO i.

The *utility function* models the overall gain $U_{ijn}(\boldsymbol{x}_i)$, i.e., the utility that the system gains by assigning $\boldsymbol{c}_j$ resources to SDO i, allowing it to add the implementation j to its assignment vector $\boldsymbol{x}_i$. Note that the gain does not depend merely from the given application component; in fact, it depends *(i)* on the chosen implementation and *(ii)* on the node selected for the deployment. Note how constraint (1.2) ensures that the solution is infrastructure-bounded, while constraints (1.3 and 1.4) avoid partial allocations.

In the reminder of this section we introduce DRAGON (Distributed Resource AssiGnment and OrchestratioN), a novel approximation algorithm that we designed to solve the NP-hard Problem 1 through a distributed approach.

B. DRAGON Overview

Each SDO i runs a DRAGON agent, which iterates between a local and a distributed phase. Locally, the SDO builds its assignment vector $\boldsymbol{x}_i$, i.e., the set of component implementations to be deployed on each node. This is used to participate to a resource election process by voting resources needed on each hosting node. Each vote models the benefit that an SDO would gain from the resources demanded on a given node, and is directly related with the SDO private utility. Voting and elections are performed at the node level. At first (*Orchestration Phase*), each agent performs the election locally, based on its state awareness. During an *Agreement Phase*, agents communicate and update their votes to ensure the convergence of the election process by mean of max-consensus. SDOs that are "elected", i.e., that they win the distributed election, gain the right to allocate the demanded amount of (virtual) resources on a certain number of (physical) nodes.

Note that the assignment vector $\boldsymbol{x}_i$ of each SDO i does not need to be exchanged. Agents are aware of the resource demand from their peers, but are unaware of the details regarding which application components they wish to allocate.

To detail the algorithm, we give the following definitions:

Definition 6. *(private utility function $\boldsymbol{u}_i$). Given a set $\mathcal{I}$ of SDOs allocating a set $\mathcal{J}$ of component implementations over a set $\mathcal{N}$ of hosting nodes, we define private utility function of SDO $i \in \mathcal{I}$, and we denote it with $\boldsymbol{u}_i \colon \mathcal{J} \times \mathcal{N} \to \mathbb{R}$, the utility $u_{ijn} \in \mathbb{R}$ that SDO i gains by adding implementation $j \in \mathcal{J}$ to its assignment vector $\boldsymbol{x}_i$ and deploying it on node $n \in \mathcal{N}$, i.e., implementing an application component on n through j.*

Each SDO may have a different (conflicting) objective and may have no incentive to disclose its utility; however, our model, and so our algorithm, maximizes a global objective (Equation 1.1), that in DRAGON is a policy. Since we assume that a Pareto optimality is sought, the global utility is a function of the applications private utilities, i.e.,

$$\boldsymbol{U}_i(\boldsymbol{x}_i) = f(\boldsymbol{u}_i(\boldsymbol{x}_i)),\ \forall i \in \mathcal{I}.$$

DRAGON needs a vote vector that we define as follows.

Definition 7. *(vote vector $\boldsymbol{v}^i$). Given a distributed voting process among a set $\mathcal{I}$ of N_o SDOs, allocating resources on a set $\mathcal{N}$ of N_υ hosting nodes, we define $\boldsymbol{v}^i \in \mathbb{R}_+^{N_o N_\upsilon}$ to be the vector of current winning votes known by SDO $i \in \mathcal{I}$ on hosting node $n \in \mathcal{N}$. Each element $v^i_{\iota n}$ is a positive real number representing last vote of SDO ι on node n as known by i, if i thinks that ι is a winner of the election phase for node n. Otherwise, $v^i_{\iota n}$ is 0.*

Since SDOs compute resource assignments in a distributed fashion, they could possibly have different views until an agreement on the election winner(s) is reached; we use the apex i to refer to the vote vector as seen by SDO i at each point in the agreement process. During the algorithm description, for clarity, we omit the apex i when we refer to the local vector (the same applies also for the following vectors).

Definition 8. *(demanded resource vector $\boldsymbol{r}^i$). Given a voting process among a set $\mathcal{I}$ of N_o SDOs on N_ρ different types of shared resources distributed among a set $\mathcal{N}$ of N_υ hosting nodes, we define as demanded resource vector $\boldsymbol{r}^i \in \mathbb{N}_+^{N_o \times N_\upsilon \times N_\rho}$, the vector of total resources currently requested by each SDO on every node; each element $\boldsymbol{r}^i_{\iota n} \in \mathbb{N}^{N_\rho}$ is the amount of resources requested by SDO $\iota \in \mathcal{I}$ on node $n \in \mathcal{N}$ with its most recent vote $v^i_{\iota n}$ known by $i \in \mathcal{I}$.*

Definition 9. *(voting time vector $\boldsymbol{t}^i$). Given a set $\mathcal{I}$ of N_o SDOs participating to a distributed voting process over a set $\mathcal{N}$ of N_υ hosting nodes, we define as voting time vector $\boldsymbol{t}^i \in \mathbb{R}_+^{N_o \times N_\upsilon}$, the vector whose element $t^i_{\iota n}$ represents the timestamp of the last vote $v^i_{\iota n}$ known by $i \in \mathcal{I}$ for SDO $\iota \in \mathcal{I}$ on node $i \in \mathcal{I}$.*

Definition 10. *(neighborhood $\bar{\mathcal{I}}_i$). Given a set $\mathcal{I}$ of SDOs, we define neighborhood $\bar{\mathcal{I}}_i \subseteq \mathcal{I} \setminus \{i\}$ of SDO $i \in \mathcal{I}$, the subset of SDOs directly connected to i.*

Algorithm 1 `DRAGON` for SDO i at iteration t

```
orchestration($\boldsymbol{v}(t-1),\ \boldsymbol{r}(t-1),\ \boldsymbol{\rho}$)
if $\exists \iota \in \mathcal{I} :\ v_\iota(t) \neq v_\iota(t-1)$ then
    send($i',\ t$), $\forall i' \in \bar{\mathcal{I}}_i$
receive($i',\ t$), $\forall i' \in \bar{\mathcal{I}}_i$
agreement($i',\ t$), $\forall i' \in \bar{\mathcal{I}}_i$
```

Algorithm 2 `orchestration` for SDO i at iteration t

```
Input: $\boldsymbol{v}(t-1),\ \boldsymbol{r}(t-1),\ \boldsymbol{t}(t-1),\ \boldsymbol{\rho},\ \boldsymbol{c}$
Output: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t)$
1: if $t \neq 0$ then
2:     $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t) = \boldsymbol{v}(t-1),\ \boldsymbol{r}(t-1),\ \boldsymbol{t}(t-1)$
3: do
4:     $\bar{v_i} = v_i(t)$
5:     if $v_i(t-1) \neq 0\ \wedge\ v_i(t) = 0$ then                ▷ outvoted
6:         embedding($t$)                      ▷ find next $\boldsymbol{x}_i$ maximizing $\boldsymbol{u}_i$
7:         voting($\boldsymbol{x}_i,\ \boldsymbol{c}$)                           ▷ vote $\boldsymbol{x}_i$ using $\boldsymbol{U}$
8:     election($\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{\rho}$)
9: while $\bar{v_i} \neq v_i(t)$                          ▷ repeat until not outvoted
```

The notion of neighborhood is generalizable with the set of agents reachable within a given latency upper bound.

We are now ready to describe DRAGON (Algorithm 1), by detailing its two main phases.

C. Orchestration Phase

After the initialization of local vectors $\boldsymbol{v}(t)$, $\boldsymbol{r}(t)$ and $\boldsymbol{t}(t)$ for the current iteration t (Algorithm 2, line 2), each DRAGON agent uses Algorithm 2, line 8 to elect the current winners according to the known votes updated at the last iteration. If agent i has been outvoted (Algorithm 2, line 5), the algorithm starts to iterate among *(i)* an *embedding routine* (Algorithm 2, line 6), which computes the next suitable assignment vector $\boldsymbol{x}_i$ maximizing i's private utility, *(ii)* a *voting routine* (Algorithm 2, line 7) where agent i votes for the resources that follow the last computed assignment vector and *(iii)* the *election routine* (Algorithm 2, line 8), which uses votes to compute winning agents.

The iteration continues until agent i does not get outvoted anymore (Algorithm 2, line 9). This may happen if either *(i)* the selected assignment vector allows i to win the election or *(ii)* there are no more suitable assignments $\boldsymbol{x}_i$ (then no new votes have been generated).

Remark. *To guarantee convergence, DRAGON forbids outvoted SDOs to re-vote with an higher utility value on resources that they have lost in past rounds. Re-voting is, however, allowed only on residual resources.*

Note that an asynchronous agreement may never terminate unless we forcefully timeout the consensus process. However, we use the theory of max-consensus to show that the agreement stops as long as we have reliable communication and each vote traverses the network at least once (Section VI).

1) Embedding Routine. Either during the first iteration ($t = 0$), or any time SDO i is outvoted, DRAGON invokes an embedding routine (Algorithm 2, line 6) that, based on the private policies of i, computes the next best suitable assignment vector $\boldsymbol{x}_i$. Therefore, this routine is in turn private for each SDO, and strictly dependent from the specific nature

Algorithm 3 `voting` for SDO i at iteration t

```
Input: $\boldsymbol{x}_i,\ \boldsymbol{c}$
Output: $\boldsymbol{v}_i(t), \boldsymbol{r}_i(t), t_i(t)$
1: $t_i(t) = t$                                             ▷ vote time
2: if $\boldsymbol{x}_i \neq \mathbf{0}$ then                               ▷ valid assignment
3:     for all $n \in \mathcal{N}$ do
4:         $r_{ink}(t) = \Sigma_j x_{ijn} c_{jk},\ \forall k \in \mathcal{K}$     ▷ resources required on node
5:         $v_{in}(t) = \texttt{score}(\boldsymbol{x}_i, n)$               ▷ vote new assignment
```

Algorithm 4 `election` for SDO i at iteration t

```
Input: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{\rho}$
Output: $\boldsymbol{v}(t)$
1: do
2:     for all $n \in \mathcal{N}$ do
3:         $\mathcal{W}_n = \texttt{node\_election}(\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ n,\ \boldsymbol{\rho}_n)$
4:     $\mathcal{W}^F = \texttt{election\_recount}(\mathcal{W}_n\ \forall n)$        ▷ detect false-winners
5:     $\boldsymbol{v}_\iota = \mathbf{0},\ \forall \iota \in \mathcal{W}^F$                   ▷ reset false-winners votes
6: while $\mathcal{W}^F \neq \varnothing$                  ▷ repeat until no false-winners are detected
7: $\boldsymbol{v}_\iota = \mathbf{0},\ \forall \iota \in \mathcal{I} \setminus \bigcup_{n \in \mathcal{N}} \mathcal{W}_n$                        ▷ reset loser votes
```

Algorithm 5 `node_election` on node n at iteration t

```
Input: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ n,\ \boldsymbol{\rho}_n$
Output: $\mathcal{W}_n$
1: $\bar{\boldsymbol{\rho}}_n = \boldsymbol{\rho}_n$                                      ▷ residual resources
2: $\mathcal{W}_n = \varnothing$                                            ▷ winner set
3: do
4:     $\mathcal{I}_b = \{i \in \mathcal{I} |\ r_{ink}(t) \leq \bar{\rho}_{nk}, \forall k \in \mathcal{K}\}$            ▷ valid candidates
5:     $\omega = \arg\max_{i \in \mathcal{I}_b \setminus \mathcal{W}} \left\{ \frac{v_{in}(t)}{\|\boldsymbol{r}_{in}(t)\|} \right\}$      ▷ candidate with higher vote
6:     $\mathcal{W}_n = \mathcal{W}_n \cup \{\omega\}$                            ▷ add to winners
7:     $\bar{\rho}_{nk} = \bar{\rho}_{nk} - r_{\omega nk},\ \forall k \in \mathcal{K}$         ▷ decrease residual resources
8: while $\mathcal{I}_b \setminus \mathcal{W} \neq \varnothing$              ▷ repeat until no candidate remains
```

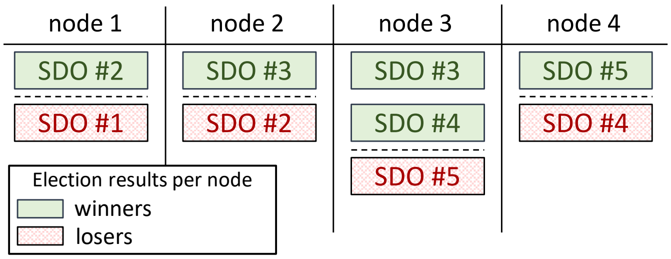


Fig. 4: Example of false winners after an election routine: SDO #2 prevents #1 to allocate needed resources on node 1, although #2 cannot be deployed, since it lost elections on node 2.

of the orchestrated application. More in detail, when a set of operations are pending upon the execution of an action (Section IV-B), a corresponding routine is built and passed from the framework to the DRAGON agent in order implement the desired result under distributed agreement. For what concerns DRAGON, this routine can be viewed as a private decision process that selects, for each component needed by the application, both the implementation $j \in \mathcal{J}$ to be used and the node $n \in \mathcal{N}$ where j should be deployed.

2) Voting Routine. After a new assignment vector has been built, each DRAGON agent executes a voting routine, updating the time of its most recent vote; if the assignment vector is valid, all demanded resources are updated and voted, through a *score function* derived from the global utility (Algorithm 3). Since voting is performed at node level, this routine generates a vote for each hosting node involved in the current assignment $\boldsymbol{x}_i$. Although the raw global utility itself may be used as score function to compute votes, in Section V-E we give recommendation on which function should be used to guarantee convergence and optimal approximation bound (Section VI). Since the value of t_i is updated in any case (Algorithm 3, line 1), if SDO i does not find any suitable assignment vector, the recent timestamp associated with an empty vote will let its peers know that i agrees with an election definitively lost.

3) Election Routine. The last step of the *Orchestration Phase* (Algorithm 2, line 8) is a resource election that decides which SDOs are capable of allocating the demanded resources on the chosen hosting nodes (Algorithm 4). Based on the most recent known votes $\boldsymbol{v}(t)$, the related resource demands $\boldsymbol{r}(t)$ and the capacity $\boldsymbol{\rho}_n$ of each node, this procedure selects SDOs by means of a greedy approach. For every node $n \in \mathcal{N}$ (Algorithm 4, line 3-4), the `node_election` subroutine (Algorithm 5) *(i)* discards every SDO whose demanded resources $\boldsymbol{r}_i$ exceed the residual node capacity and *(ii)* selects the one with the highest ratio vote to demanded resources (Algorithm 5, lines 4-5). The one elected is then added to the winner set of that particular node and the amount of resources assigned to the new winner is removed from the residual ones (Algorithm 5, lines 6-7). The greedy election on each node ends when either all candidates result winners, or residual node resources are not enough for any of those remaining.

In Section VI we show that the greedy heuristic gives guarantees on the optimal approximation.

Remark. *In DRAGON an assignment $\boldsymbol{x}_i$ is considered valid only if SDO i wins all elections on each node n involved in the assignment $\boldsymbol{x}_i$. If any node election is lost, DRAGON resets the vote vector and a new assignment is built from scratch to avoid suboptimal assignments.*

Since elections are performed separately for each node of the infrastructure, the election routine includes a conflict resolution subroutine named *election-recount* (Algorithm 4, line 4), which handles potential suboptimality deriving as a result of elections. Consider the assignment scenario in Figure 4; most resources of node 1 have been assigned to SDO #2, thus preventing the deployment of SDO #1; however, having #2 lost the election on node 2, releases its previous vote on node 1 at the next iteration. Therefore, app. #1 could be considered a winner.

The election-recount subroutine copes with this problem by identifying which SDO should be removed from the election so that the solution is optimized. We call these SDOs *false-winners*, i.e., SDOs that only won a subset of the needed nodes, preventing peers that would maximize the global utility to win. False-winners are identified recursively. Given a potential false-winner ω, i.e., it lost elections on a subset of needed nodes, the idea is to temporarily extend residual resources on these nodes, whether that amount of resources have been assigned to other false-winners during the previous election round. If the extended residual resources are still in deficit with respect to the resources that ω needs, then the candidate is a false-winner as well.

For instance, let's consider the election results in Figure 4. To determine if SDO #2, that is a winner for node 1, is valid or not, we check if it is possible to free some resources on node

2, where it lost. On that node, the only winner is SDO #3; however, it is a valid one, since it has not lost any other needed node. Therefore, SDO #2 definitely lost node 2, and can be removed from the winners of node 1 (it is a false-winner), thus enabling SDO #1 to win the elections. Situations in which some SDOs *cross-lose* nodes are resolved in favor of the one whose vote on any node is the highest; see e.g., Figure 4: app. #4 and #5 *cross-lost* nodes 3 and 4.

The election process is repeated until the recount subroutine does not detect any false-winners (Algorithm 4, line 6). When the election result is confirmed, votes of SDOs that did not win the election are reset (Algorithm 4, line 7).

D. Agreement Phase

Once vectors $\boldsymbol{v}^{i'}$, $\boldsymbol{r}^{i'}$ and $\boldsymbol{t}^{i'}$ are received from every neighbor i', each agent runs an Agreement Phase. During this phase, SDOs make use of a consensus mechanism to reach an agreement on their vote vector $\boldsymbol{v}^i$, hence on the overall resources assignment (Algorithm 6). By adapting the definition of *consensus* [23] to the orchestrator-resources assignment problem, we define our own notion of consensus on the election results as follows:

Definition 11. *(election-consensus). Let us consider a set $\mathcal{I}$ of N_o SDOs sharing a computing edge infrastructure through an election routine driven by, for each SDO $i \in \mathcal{I}$, the vote vector $\boldsymbol{v}^i(t) \in \mathbb{R}_+^{N_o \times N_v}$, the demanded resource vector $\boldsymbol{r}^i(t) \in \mathbb{R}_+^{N_o \times N_v \times N_\rho}$ and the voting time vector $\boldsymbol{t}^i(t) \in \mathbb{N}^{N_o \times N_v}$. Let $e : \mathbb{R}_+^{N_o \times N_v}, \mathbb{N}^{N_o \times N_v \times N_\rho} \to 2^{\mathcal{I}}$ be the election function, that given a vote vector $\mathbf{v}$ and the demanded resources $\boldsymbol{r}$ gives a set of winners. Given the consensus algorithm for SDO i at iteration $t+1$, $\forall \iota \in \mathcal{I}$,*

$$v_\iota^i(t+1) = v_\iota^{i'}(t), \; \boldsymbol{r}_\iota^i(t+1) = \boldsymbol{r}_\iota^{i'}(t), \\ \textit{with } i' = \operatorname*{arg\,max}_{i' \in \mathcal{I}_i \cup \{i\}} \{t_\iota^{i'}(t)\}, \tag{2}$$

election-consensus *among the SDOs is said to be achieved if $\exists \bar{t} \in \mathbb{N}$ such that, $\forall t \geq \bar{t}$ and $\forall i, i' \in \mathcal{I}$,*

$$\begin{cases} e(\boldsymbol{v}^i(t), \boldsymbol{r}^i(t)) \equiv e(\boldsymbol{v}^{i'}(t), \boldsymbol{r}^{i'}(t)) \\ \boldsymbol{v}_\iota^i(t) \neq \mathbf{0} \iff \iota \in e(\boldsymbol{v}^i(t)), \; \forall \iota \in \mathcal{I}, \end{cases} \tag{3}$$

i.e., on all SDOs the election function computes the same winner set and only winner votes are non zero.

The agreement on votes of an SDO ι is performed by every SDO i once received vectors $\boldsymbol{v}^{i'}$, $\boldsymbol{r}^{i'}$ and $\boldsymbol{t}^{i'}$ from each i' in its neighborhood, comparing them and selecting the most recent information received, if any (Equation 2). Since DRAGON is asynchronous by design, at each iteration t the agreement phase can start even if agents have received a vote message from only a subset of their neighbors.

E. Recommendations on the score function

DRAGON's score function is a policy. Many policies may work well in practice, but in some cases they may lead to arbitrarily bad performance. As we will see in the next section, DRAGON guarantees both convergence and a given performance lower bound as long as the function maximized during the election routine is submodular (Definition 12). In this section we give recommendation on the score function $\mathcal{V}$ that each SDO should use during the voting routine described in Algorithm 3 to satisfy this property. Analytic results are shown in the next section.

Algorithm 6 `agreement` with SDO i' at iteration t

```
Input: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t),\ \boldsymbol{v}^{i'}(t),\ \boldsymbol{r}^{i'}(t),\ \boldsymbol{t}^{i'}(t)$
Output: $\boldsymbol{v}(t),\ \boldsymbol{r}(t),\ \boldsymbol{t}(t)$
for all $\iota \in \mathcal{I}$ do
    for all $n \in \mathcal{N}$ do                          ▷ for every hosting node
        if $t_{\iota n}(t) < t_{\iota n}^{i'}(t)$ then          ▷ received newer vote
            $v_{\iota n}(t) = v_{\iota n}^{i'}(t)$
            $r_{\iota n k}(t) = r_{\iota n k}^{i'}(t),\ \forall k \in \mathcal{K}$
            $t_{\iota n}(t) = t_{\iota n}^{i'}(t)$
```

Let $\mathcal{U}_{in}(\boldsymbol{x}_i) = \Sigma_j U_{ijn}(\boldsymbol{x}_i) x_{ijn}$ be the overall node utility of SDO i on node n. To guarantee convergence of the election process, we let each peer i communicate its vote on node n obtained from the score function:

$$\mathcal{V}_i(\boldsymbol{x}_i, \mathcal{W}_n, n) = \min_{\omega \in \mathcal{W}_n} \{\mathcal{U}_{in}(\boldsymbol{x}_i), \mathcal{S}_{in}(\omega)\}, \tag{4}$$

where $\mathcal{W}_n \subseteq \mathcal{I}$ is the current winner set for node n, i.e., $v_{\omega n}(t) \neq 0 \; \forall \omega \in \mathcal{W}_n$, and $\mathcal{S}_{in}$ is defined as

$$\mathcal{S}_{in}(\omega) = \begin{cases} +\infty & \text{if } i \text{ never voted on } n, \\ \|\boldsymbol{r}_{in}(t)\| \, \frac{v_{\omega n}(t)}{\|\boldsymbol{r}_{\omega n}(t)\|} & \text{otherwise.} \end{cases}$$

Since $\mathcal{U}_{in}(\boldsymbol{x}_i) \geq 0$ by definition, if i computes each vote with the function $\mathcal{V}$, it follows that, $\forall (i, n) \in \mathcal{I} \times \mathcal{N}$, $\mathcal{V}_i(\boldsymbol{x}_i, n) \geq 0$. Note how, if it is not the first time that i votes on n, the vote $v_{in}(t)$ generated at iteration t never results as an outvote of any SDO that has been previously elected on node n, during the election process described in Algorithm 5.

VI. Convergence and Performance Guarantees

In this section we present results on the convergence properties of our DRAGON distributed approximation algorithm. As defined in Definition 11, by convergence we mean that a valid solution to the orchestrators-resources assignment problem is found in a finite number of steps. Infeasibility is also a valid solution. Moreover, starting from well-known results on submodular functions, in this section we show that DRAGON guarantees an $(1 - e^{-1})$-approximation bound, and that this bound is also optimal, i.e. there is no better guarantee, unless $NP \subseteq DTIME(n^{O(\log \log n)})$.

Note that, if (4) is used as a score function, the election routine of DRAGON is equivalent to a greedy algorithm attempting to find, for each node n, the set of winner SDOs $\mathcal{W}_n \subseteq \mathcal{I}$ such that the set function $z_n : 2^{\mathcal{I}} \to \mathbb{R}$, defined as

$$z_n(\mathcal{W}_n) = \sum_{\omega \in \mathcal{W}_n} \mathcal{V}_\omega(\boldsymbol{x}_\omega, \mathcal{W}_n, n), \tag{5}$$

is maximized. By construction of $\mathcal{V}$, we have that z_n is monotonically non-decreasing and $z(\varnothing) = 0$.

Definition 12. *(submodular function). A set function $z : 2^{\mathcal{I}} \to \mathbb{R}$ is submodular if and only if, $\forall \iota \notin \mathcal{W}' \subset \mathcal{W}'' \subseteq \mathcal{I}$,*

$$z(\mathcal{W}'' \cup \{\iota\}) - z(\mathcal{W}'') \leq z(\mathcal{W}' \cup \{\iota\}) - z(\mathcal{W}'). \tag{6}$$

This means that the marginal utility of adding ι to the input set, cannot increase due to the presence of additional elements. Next we show that the total score z_n (5) is submodular. Our intuition behind its submodularity is that the score function $\mathcal{V}_n$ can, at most, decrease due to the presence of additional elements in $\mathcal{W}_n$. Formally, we have:

Lemma VI.1. *z_n (5) is submodular.*

Proof. Since $\mathcal{W}'_n \subset \mathcal{W}''_n$, we have

$$\min_{\omega \in \mathcal{W}''_n} \left\{ \|\boldsymbol{r}_{\iota n}(t)\| \frac{v_{\omega n}(t)}{\|\boldsymbol{r}_{\omega n}(t)\|} \right\} \leq \min_{\omega \in \mathcal{W}'_n} \left\{ \|\boldsymbol{r}_{\iota n}(t)\| \frac{v_{\omega n}(t)}{\|\boldsymbol{r}_{\omega n}(t)\|} \right\},$$

and so, for (4),

$$\mathcal{V}_\iota(\boldsymbol{x}_i, \mathcal{W}''_n, n) \leq \mathcal{V}_\iota(\boldsymbol{x}_i, \mathcal{W}'_n, n). \quad (7)$$

By definition of z_n, the marginal gain of adding ι to $\mathcal{W}_n$ is

$$z_n(\mathcal{W}_n \cup \{\iota\}) - z_n(\mathcal{W}_n) = \mathcal{V}_\iota(\boldsymbol{x}_i, \mathcal{W}_n, n), \forall \iota \notin \mathcal{W}_n \subseteq \mathcal{I},$$

therefore, substituting in (7), we have the claim. □

Convergence Guarantees. A necessary condition for convergence in DRAGON is that all SDOs are aware of which are the winning votes for an hosting node. This information needs to traverse all SDOs in the communication network (at least) once. Theorem VI.2 shows that a single information traversal is also sufficient for convergence.

The communication network of a set of SDOs $\mathcal{I}$ is modeled as an undirected graph, with unitary length edges between each couple $i', i'' \in \mathcal{I}$ such that $i'' \in \bar{\mathcal{I}}_{i'}$ and $i' \in \bar{\mathcal{I}}_{i''}$, being $\bar{\mathcal{I}}_{i'} \subseteq \mathcal{I} \backslash \{i'\}$ and $\bar{\mathcal{I}}_{i''} \subseteq \mathcal{I} \backslash \{i''\}$ respectively the neighborhood of i' and i''.

Theorem VI.2. *(Convergence of synchronous DRAGON). Consider an infrastructure of N_v hosting nodes, whose resources are shared among N_o SDOs through an election process with synchronized conflict resolution over a communication network with diameter D. If the communication channels are reliable and the function (5) maximized during the election routine is submodular, then DRAGON needs at most $N_o^2 N_v D$ iterations to converge.*

Proof. We first show by induction that agents agree on the first k assignments in at most kN_oD iterations. Given the submodularity of z_n, the assignment $(i_1^\star, n_1^\star)$ with the highest vote computed at iteration 1 can be outvoted at most $N_o - 1$ times, i.e., until every agents voted on node $n_1^\star$ at least once. Since each time D iterations are needed to propagate the vote, every agent will have agreed on the highest vote $v_{i_1^\star n_1^\star}$ at most after N_oD iterations. Let us suppose that at iteration hN_oD all agents agree on the first k-best assignments. Since the next-best vote propagated at iteration $k+1$ can be outvoted at most $N_o - 1$ times, it follows that every agent will have agreed on $(i_{h+1}^\star, n_{h+1}^\star)$ by iteration $hN_oD + N_oD$. Then, together with $(i_1^\star, n_1^\star)$ being agreed to at N_oD, every agent will have agreed on $(i_k^\star, n_k^\star)$ within kN_oD iterations. In DRAGON each compute node may be assigned to each SDO, then, in the worst case there is a combination of N_oN_v assignments. Therefore, agents reach agreement in at most $N_o^2 N_v D$ iterations. □

As a direct corollary of Theorem VI.2, we compute a bound on the number of messages that SDOs have to exchange in order to reach an agreement on resource assignments. Because we only need to traverse the communication network at most once for each combination SDOs per hosting nodes $(i, n) \in \mathcal{I} \times \mathcal{N}$, the following result holds:

Corollary VI.2.1. *(DRAGON Communication Overhead). The number of messages exchanged to reach an agreement on the resource assignment of N_v nodes among N_o non-failing SDOs with reliable communication channels using the DRAGON algorithm is at most $N_o^2 N_v D N_{msp}$, where D and N_{msp} are respectively the diameter of the communication network and its the minimum spanning tree.*

Performance Guarantees. The election routine in DRAGON is trivially extended with partial enumeration [24], leading to the following two results (for brevity, the extension has been omitted in Algorithm 4).

Theorem VI.3. *(DRAGON Approximation Bound). DRAGON extended with partial enumeration yields an $(1 - e^{-1})$-approximation bound with respect to the optimal assignment.*

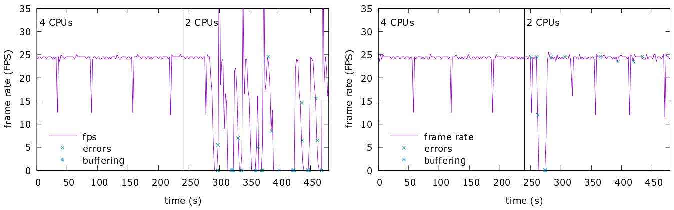
Proof. (sketch) During the election routine, DRAGON uses a greedy heuristic to assign node resources to a set of winners $\mathcal{W}_n$. The objective of the heuristic is to maximize the value of the set function $z_n(\mathcal{W}_n)$ without exceeding the node capacity (knapsack constraint). From a recent result on submodular functions [25], we know that a greedy approximation algorithm used to maximize a non decreasing submodular set function subject to a knapsack constraint is bounded by $(1 - e^{-1})$ if the algorithm is combined with the enumeration technique due to [24]. Being the set function $z_n(\mathcal{W}_n)$ positive, monotonic and non-decreasing, it remains to show that the utility used by DRAGON is submodular, which comes from Lemma VI.1; hence the claim holds. □

Theorem VI.4. *(DRAGON Approximation Optimality). The DRAGON approximation bound of $(1 - e^{-1})$ is optimal, unless $NP \subseteq DTIME(n^{O(\log \log n)})$.*

Proof. (sketch) To show that the approximation bound given by DRAGON is optimal, we first show that the orchestrators-resources assignment problem addressed by DRAGON can be reduced from the (NP-hard) *budgeted maximum coverage problem* [24]. Given a collection S of sets with associated costs defined over a domain of weighted elements, and a budget L, find a subset $S' \subseteq S$ such that the total cost of sets in S' does not exceeds L, and the total weight of elements covered by S' is maximized. We reduce the orchestrators-resources assignment problem from the budgeted maximum coverage problem by considering *(i)* S to be the collection of all the possible set of orchestrators, i.e., $S = 2^{\mathcal{I}}$, *(ii)* L to be the total amount of resources available on the hosting node (in this particular case $N_\rho = 1$), and *(iii)* weight and costs to be votes and demanded resources of each SDO. Since [24] shows that $(1 - e^{-1})$ is the best approximation bound for the budgeted maximum coverage problem unless $NP \subseteq DTIME(n^{O(\log \log n)})$, the claim holds. □

VII. Experimental Results

To validate our approach, we implemented a prototype of both the SDO compiler and DRAGON. Our code is available at [26], [27]. Our evaluation focuses on two major sets of results; we first provide evidence of the advantages of the Service-Defined Orchestration approach, analyzing three use cases: stream management on a video streaming application, cache placement for a CDN provider and process migration for mobile gaming; then, we assess both DRAGON's asynchronous convergence properties and performance.



(a) One-size fits-all approach

(b) Service-Defined approach.

Fig. 5: Frame rate over time for a video streaming application in a resource constrained situation.

A. Service-defined Orchestration

We show the advantages for service providers when deploying their applications on an infrastructure that adopts our service-defined orchestration approach and does not restrict the available orchestration strategies. Our aim is to show that, for example, a CDN provider that relies on a third party platform/infrastructure to serve a certain area may benefit from using its own cache placement algorithm (e.g., [28]), running over DRAGON, rather than depending on a one-size fits-all embedding orchestrator.

At first, we setup a virtualization infrastructure to orchestrate the deployment and run-time of a video streaming application. Additionally, we setup a simulated environment to evaluate two different edge use cases: *(i)* cache placement for a CDN provider [29], and *(ii)* edge migration for mobile gaming [30]. In our tests, we compared the provided QoS resulting from different deployment approaches, also varying the concurrency level by adding some concurrent applications, thus evaluating the behavior when resources become scarce.

Video Streaming. We deployed an use case analogous to the one described in Section IV-C. A VM generating an RTMP stream through FFmpeg implements the video transmitter, while both the web server and the transcoder have been implemented through Wowza [31], generating 9 streams at different bit rates. All these components have been deployed on a KVM based infrastructure (Hypervisor Debian Linux 4.14.0-3 on i7-6700 CPU 3.40 GHz, RAM 32 GB). On a second machine, we run the VLC software to consume the output streams and measure the QoS in terms of frame rate. We emulated a scenario in which, after 240 seconds, some of the CPUs originally allocated for the transcoder VM are no longer available. Figure 5 summarizes our findings.

(i) When resources are reduced for the transcoder VM, a one-size fits-all orchestrator can neither understand that the transcoder is suffering (it has no generic parameter to base itself upon) nor can it identify a solution in such a constrained situation, i.e. the VM cannot be scaled out since there are no more resources locally and no other edge nodes are available to migrate the VM. As a consequence, Figure 5a shows a significant degradation of the provisioned service.

(ii) If the application is managed by an SDO, a custom action can be defined to be executed whenever it is not possible to assign a given amount of CPUs to the transcoder component; in such a case, a possible service-defined solution may be to configure the transcoder component to disable some of the generated output streams (e.g., those at a higher resolution), thus reducing its workload and resource requirements. Figure 5b shows that this behavior effectively preserves the frame rate after switching configuration.

CDN Caches. A CDN provider provisions content caches over an edge network where user density dynamically changes across compute nodes. The objective of the provider is to minimize the average miss-rate occurring on deployed caches. The CDN application should be adapted on events where a set of users shifts from a node to another. In our tests we simulated a set of 100 users moving over a network of 10 edge computing nodes. To visualize the user distribution among nodes, we also report the Gini index (a high index indicates that most users are located near few host nodes). We summarize our findings in a few take home messages:

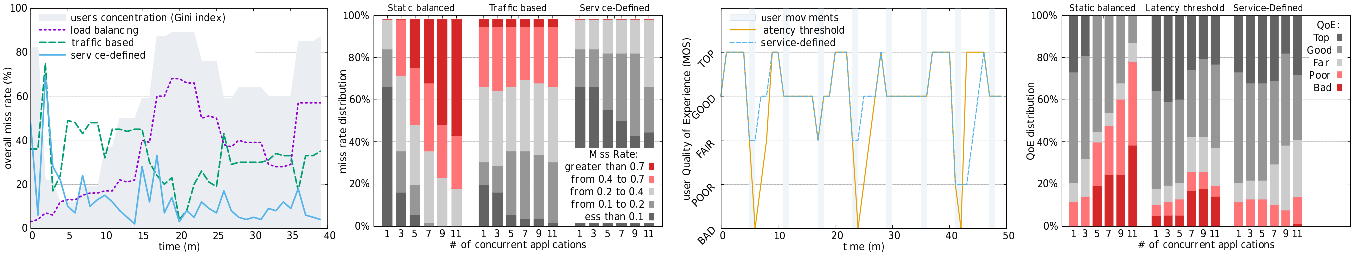
(i) A one-size fits-all approach that places caches by balancing the resource consumption per node achieves good performance when users are well distributed, but the number of miss-rate grows fast when the concentration increases (Figure 6a). A similar result is obtained by statically partitioning the resources among coexistent applications (Figure 6b) when their number is high with respect to the available resources.

(ii) A one-size fits-all approach that places caches according with the traffic load on each node achieves optimal miss-rates when users are concentrated on few nodes, while the performance is poor otherwise. This is because a low traffic amount on a certain node does not necessarily mean that users are consuming less variety of contents. Figure 6b shows a slight degradation when increasing the concurrency.

(iii) If application caches are placed by an SDO based on current miss-rate on each node, mandating resource partitioning to DRAGON, optimal miss-rate both for low and high users concentration is achieved (Figure 6a). Moreover, note how Figure 6b does not show a noticeable QoS degradation when increasing the number of concurrent SDOs, showing the scalability of our approach. This is because DRAGON seeks optimal resource partitioning with regard to the application objectives.

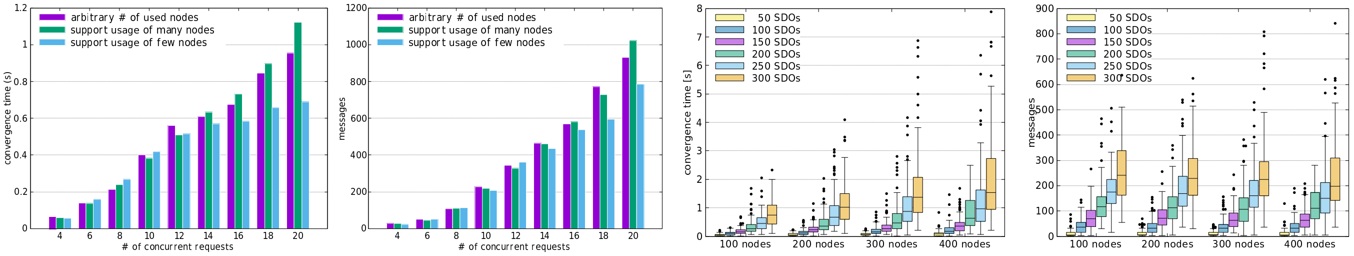
Mobile Gaming. A gamer moves into an area served by multiple edge nodes. Whereas it may be convenient to relocate (part of) the game application components to better fulfill the latency requirements, the relocation may happen in a crucial phase of the game, causing undesirable service degradation [30]. Therefore, if the deployment is managed by an SDO, it may be instructed to recognize the time frame in which a relocation is most appropriate (e.g., after the gamer reaches a checkpoint or during the loading of a new level).

In our tests we simulated a user moving every 6 minutes



(a) Miss rate over time. (b) Miss rate distributions. (c) QoE over time. (d) QoE distribution.

Fig. 6: (ab) Evaluation of a CDN cache provisioning application comparing different placement strategies: (a) miss rate over time varying the geographical users distribution; (b) distribution of measured miss rate varying the number of concurrent applications. (cd) Evaluation of a mobile gaming application for different deployment strategies: (c) QoE over time perceived by a user moving in different areas; (d) QoE distribution varying the number of concurrent applications.



(a) Convergence time. (b) Number of exchanged messages. (c) Convergence times. (d) Number of exchanged messages.

Fig. 7: DRAGON convergence evaluation. (ab) Results on prototype varying the number of simultaneous requests, also comparing different system policies. (cd) Larger scale simulation varying the number of concurrent SDOs and hosting nodes, where resource allocation requests are randomly performed over time.

across a network of 10 edge nodes. We measured the Quality of Experience perceived by the user based on latency and packet loss, using the same Mean Opinion Score (MOS) described in [32] for medium-paced games. Our findings are summarized as follows (Figure 6cd):

(i) Statically partitioning resources between applications does not scale (Figure 6d): the application may be unable to migrate components on needed nodes, since resources are assigned to other peers, despite not being currently used.

(ii) If the resources are managed by a one-size fits-all orchestrator that minimizes the end-to-end latency, the user often experiences a QoE level that we label as `bad` due to some process relocation occurring during the game session (Figure 6c). Figure 6d shows that the percentage of `bad` QoE measurements even may increase with the concurrency.

(iii) If the relocation decision is taken by an SDO, and resources are dynamically assigned with DRAGON, it is possible to define a behavior that does not migrate the service rapidly whenever the user moves away; even if this may temporarily increase the latency, it prevents undesirable service degradation during a game session and the overall perceived QoE results improved (Figure 6c). Figure 6d also shows that this approach scales well with the number of concurrent SDOs.

B. DRAGON Properties Evaluation

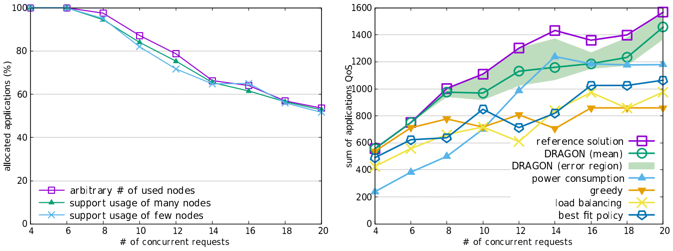
In [2], we conducted an evaluation in a simulated environment; in this paper we extend the evaluation of DRAGON in an environment with 4 physical nodes (deployed on the CloudLab distributed research infrastructure [33]), each with a different amount of computing resources (CPU, memory and storage). We run 6 diverse application components, whose implementation can be chosen among 9 different options; on average, each implementation uses about 13% of a node capacity. These numbers, combined with the rest of our parameter space, allowed us to test the behavior of the algorithm when the hosting resources are saturated, even running a moderate number of SDOs. All tests have been repeated varying the number of concurrent SDOs.

Convergence Evaluation. DRAGON convergence properties have been evaluated by measuring the time needed to reach consensus and the total number of messages exchanged. To stress the convergence of the algorithm, we evaluated it when up to 20 allocation requests arrive simultaneously.

Figure 7ab shows our results comparing three system policies: *(i)* components of an application are preferably allocated on the lowest number of nodes; *(ii)* components of an application are spread across as many nodes as possible; *(iii)* no preference on the number of nodes is given. For each configuration, we ran 25 instances, gradually varying the average number of components per application (with averages from 2.4 to 3.6 components). Plots show mean values; all confidence intervals (not shown) were statistically significant.

In particular, Figure 7a shows the mean convergence times. We found that, when a large number of SDOs interact, encouraging the system to use fewer nodes significantly lowers convergence time. Some consequences of this policy are *(i)* a reduced probability to lose a node election and *(ii)* re-voting on residual resources located on additional nodes is discouraged. Hence, the highest convergence times have been registered enforcing the usage of many nodes, while convergence is slightly faster when SDOs are free to arbitrarily decide the number of nodes to use. The total number of exchanged messages follows a similar behavior (Figure 7b). However, in this case the previous trend is not marked as for convergence times.

Other than offline deployment, we also evaluated online convergence on a large scale simulation, where an increasing



(a) Percentage of allocated applications.

(b) Sum of deployed application QoS.

Fig. 8: Performance evaluation of DRAGON comparing (a) different system utilities and (b) DRAGON solutions against *(i)* three one-size fits-all common approaches and *(ii)* a reference solution obtained through a centralized solver.

number of SDOs demand resources over time (Figure 7cd). The convergence is evaluated on the variation of the number of SDOs and hosting nodes. Figure 7c shows that the number of concurrent SDOs affects convergence times more than the number of available nodes. Although this result is expected (see Theorem VI.2), the increase of processing time may be partially due to the limited number of physical CPUs (the simulation runs on an i7-4770 CPU @ 3.40GHz, where each SDOs is a separate process).

Figure 7d shows that increasing the number of available nodes does not introduce noticeable variations on the total number of exchanged messages. This result suggests that the number of steps DRAGON requires to converge (hence also the number of exchanged messages) does not significantly depend on the number of nodes in the problem. Since, instead, Figure 7c highlights that convergence times increase, we can conclude that what changes is the duration of every iteration, as more nodes need to be processed.

Performance Evaluation. Figure 8a compares DRAGON performance for the same three system policies previously introduced. The plot shows the percentage of applications successfully deployed after the distributed assignment process. We found that, when the number of concurrent SDOs stays below 8, all requests are allocated, since the overall resource demand is bounded by the total amount of available resources. Above that threshold, all analyzed policies achieve approximately the same average allocation ratio, with the exception of the "few-nodes-policy", whose allocation ratio is lower for less than 12 SDOs, although it shows the fastest convergence time (Figure 7ab). This is because, when resources on the already used nodes terminate, this policy discourages the usage of residual resources available on other nodes. However, this disadvantage disappears as the number of applications grows, since the system implicitly introduces more allocation options. This result suggests that DRAGON allocation ratio scales well with the application concurrency regardless the system policy.

Finally, to evaluate the performance in practice we compared DRAGON with traditional orchestration approaches. In particular, we compare against three one-size fits-all allocation policies, i.e., a centralized orchestrator that uses the same objective function to optimize the deployment of all applications: *(i)* minimization of total *power consumption*, *(ii) greedy* selection of the potentially best performing component implementations, *(iii) load balancing* among nodes. Figure 8b also shows the performance obtained by switching between these three policies based on which one *fits best* each application needs. Additionally, we plot the *reference solutions*, obtained by running a centralized solver to the Problem 1. Values obtained with this experiment set have been used as a reference to evaluate the other approaches.

Figure 8b compares solutions in terms of *overall Quality of Service*, i.e., the sum of the QoS obtained by each application successfully deployed[2]. Varying the number of concurrent SDOs, for each configuration we ran DRAGON multiple times. Results are shown with a 95% confidence interval. Centralized algorithms always give the same solution. Results show that deploying each application according with its own objective through DRAGON provides a considerably higher QoS compared to one-size fits-all approaches, despite DRAGON being a distributed algorithm. In particular, for less than 8 requests, i.e., before the resources start to run out, DRAGON is always equivalent to the reference solution, considered as optimal. For a higher number of requests (and thus, of distributed instances), as expected, the mean QoS departs from the optimal. However, the total QoS continues to grow, following the trend of the reference solution. This result suggests that DRAGON effectively prefers the deployment of applications that introduce higher utilities to the overall solution.

Other findings from Figure 8b are summarized as follows. *(i)* A common objective that minimizes the overall power consumption provides poor total QoS, except for a high number of applications, since this strategy accommodates the largest number of requests. *(ii)* Greedily selecting the best performing implementation provides high values of overall QoS only when there are few applications. Finally, *(iii)* switching among different common strategies based on the one that fits best each application does not necessarily provide a higher QoS. This is because some generic allocation strategies work well when they are applied to all applications in the same way (e.g., load balancing and power consumption minimization). Noticeably, none of the one-size fits-all approaches is able to increase the overall QoS after resources are saturated.

VIII. Conclusion

This paper proposes a distributed and Service-Defined approach for orchestrating cloud/edge applications. The proposal enables individual applications to define their own orchestration strategy by means of a behavioral declarative model, which is used to dynamically generate a service-specific orchestrator (Service-Defined Orchestrator - SDO). This component is in charge of both the deployment (e.g., resource allocation) and run-time orchestration (e.g., scaling, reconfiguration) of the given application. With respect to the service deployment, we define DRAGON, a time-bounded distributed approximation algorithm that solves the problem of optimally partitioning a shared pool of resources between multiple SDOs. DRAGON allows them to coexist over a shared infrastructure by means of a dynamic agreement on how resources have to be (temporary) assigned, providing

[2] The QoS of each application have been modeled through its private utility. Values have been normalized between 0 and 100 for each component.

guarantees on both convergence time and performance. Our evaluation assesses the scalability of convergence and performance properties. Moreover, we evaluate our Service-Defined approach over three representative edge use cases, showing how an infrastructure provider may enable its customers (i.e., service providers) to implement their preferred orchestration strategy for their services, without the restrictions deriving by relying on a conventional one-size fits-all orchestrator.

Acknowledgment

The authors would like to thank Tierra Telematics and NSF CNS-1647084 for their support, Antonio Manzalini (TIM) for the support and the many fruitful discussions, and Emanuele Fia for his work on the first prototype.

References

[1] F. Esposito *et al.*, "Slice embedding solutions for distributed service architectures," *ACM Computing Surveys (CSUR)*, vol. 46, p. 6, 2013.

[2] G. Castellano *et al.*, "A distributed orchestration algorithm for edge computing resources with guarantees," in *IEEE International Conference on Computer Communications (INFOCOM 2019)*, 2019.

[3] J. Du, L. Zhao, J. Feng, and X. Chu, "Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee," *IEEE Transactions on Communications*, vol. 66, no. 4, pp. 1594–1608, 2018.

[4] S. Dräxler, H. Karl, and Z. Á. Mann, "Jasper: Joint optimization of scaling, placement, and routing of virtual network services," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 946–960, 2018.

[5] B. Spinnewyn *et al.*, "Coordinated service composition and embedding of 5g location-constrained network functions," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1488–1502, 2018.

[6] A. Leivadeas *et al.*, "A graph partitioning game theoretical approach for the vnf service chaining problem," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 890–903, 2017.

[7] P. T. A. Quang, A. Bradai, K. D. Singh, G. Picard, and R. Riggio, "Single and multi-domain adaptive allocation algorithms for vnf forwarding graph embedding," *IEEE Transactions on Network and Service Management*, vol. 16, no. 1, pp. 98–112, 2018.

[8] C. J. Bernardos *et al.*, "5gex: realising a europe-wide multi-domain framework for software-defined infrastructures," *Transactions on Emerging Telecommunications Technologies*, vol. 27, pp. 1271–1280, 2016.

[9] R. Guerzoni *et al.*, "Analysis of end-to-end multi-domain management and orchestration frameworks for software defined infrastructures: an architectural survey," *Transactions on Emerging Telecommunications Technologies*, vol. 28, no. 4, p. e3103, 2017.

[10] G. Darzanos, M. Dramitinos, and G. D. Stamoulis, "Coordination models for 5g multi-provider service orchestration: Specification and assessment," in *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 2017, pp. 262–274.

[11] S. Mehar *et al.*, "An optimized roadside units (rsu) placement for delay-sensitive applications in vehicular networks," in *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*. IEEE, 2015, pp. 121–127.

[12] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, p. 5.

[13] N.-S. Vo *et al.*, "Optimal video streaming in dense 5g networks with d2d communications," *IEEE Access*, vol. 6, pp. 209–223, 2018.

[14] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[15] H. R. Kouchaksaraei *et al.*, "Programmable and flexible management and orchestration of virtualized network functions," in *2018 European Conference on Networks and Communications*. IEEE, 2018, pp. 1–9.

[16] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[17] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference*, 2014, pp. 305–319.

[18] E. Sakic and W. Kellerer, "Response time and availability study of raft consensus in distributed sdn control plane," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 304–318, 2017.

[19] B. P. Rimal *et al.*, "A taxonomy and survey of cloud computing systems," in *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*. Ieee, 2009, pp. 44–51.

[20] N. Dmitry and S.-S. Manfred, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014.

[21] M. Moser, "Declarative scheduling for optimally graceful qos degradation," in *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*. IEEE, 1996, pp. 86–94.

[22] T. T. Hildebrandt and R. R. Mukkamala, "Declarative event-based workflow as distributed dynamic condition response graphs," *arXiv preprint arXiv:1110.4161*, 2011.

[23] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[24] S. Khuller *et al.*, "The budgeted maximum coverage problem," *Information Processing Letters*, vol. 70, no. 1, pp. 39–45, 1999.

[25] M. Sviridenko, "A note on maximizing a submodular set function subject to a knapsack constraint," *Operations Research Letters*, vol. 32, no. 1, pp. 41–43, 2004.

[26] E. Fia, "SDO Compiler Prototype," https://github.com/netgroup-polito/sdo-compiler.

[27] G. Castellano, "DRAGON," https://github.com/netgroup-polito/dragon.

[28] D. Karger *et al.*, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.

[29] T. Taleb *et al.*, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, pp. 1657–1681, 2017.

[30] V. Sciancalepore *et al.*, "A double-tier MEC-NFV architecture: Design and optimisation," in *Standards for Communications and Networking (CSCN), 2016 IEEE Conference on*. IEEE, 2016, pp. 1–6.

[31] "Wowza streaming engine," https://www.wowza.com/.

[32] M. Jarschel *et al.*, "An evaluation of QoE in cloud gaming based on subjective tests," in *Fifth conference on Innovative mobile and internet services in ubiquitous computing (imis)*. IEEE, 2011, pp. 330–335.

[33] R. Ricci *et al.*, "Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

Gabriele Castellano is pursuing his Ph.D. degree at Politecnico di Torino, Italy, where he received his Master's degree in Computer Engineering in 2016. During is Ph.D. career, he spent five months as visiting student at Saint Louis University, Saint Louis, MO. His research interests include service virtualization, resource orchestration, distributed algorithms and software defined networking.

Flavio Esposito received the M.S. degree in telecommunication engineering from the University of Florence, Italy, and the Ph.D. degree in computer science from Boston University in 2013. He is an Assistant Professor with the Computer Science Department, Saint Louis University and a Visiting Research Assistant Professor with the EECS Department, University of Missouri at Columbia. His research interests include network management, network virtualization, and distributed systems.

Fulvio Risso received the M.Sc. (1995) and Ph.D. (2000) in computer engineering from Politecnico di Torino, Italy. He is currently Associate Professor at the same University. His research interests focus on high-speed and flexible network processing, edge/-fog computing, software-defined networks, network functions virtualization. He has co-authored more than 100 scientific papers.