Computer Communications 159 (2020) 161–174

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Geo-distributed efficient deployment of containers with Kubernetes

Fabiana Rossi *, Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli

Department of Civil Engineering and Computer Science Engineering, University of Rome Tor Vergata, Italy

ARTICLE INFO

Keywords:
Kubernetes
Containers
Elasticity
Placement
Self-management
Geographically distributed resources

ABSTRACT

Software containers are changing the way applications are designed and executed. Moreover, in the last few years, we see the increasing adoption of container orchestration tools, such as Kubernetes, to simplify the management of multi-container applications. Kubernetes includes simple deployment policies that spread containers on computing resources located in the cluster and automatically scale them out or in based on some cluster-level metrics. As such, Kubernetes is not well-suited for deploying containers in a geo-distributed computing environment and dealing with the dynamism of application workload and computing resources.

To tackle the problem, in this paper we present *ge-kube* (Geo-distributed and Elastic deployment of containers in Kubernetes), an orchestration tool that relies on Kubernetes and extends it with self-adaptation and network-aware placement capabilities. Ge-kube introduces flexible and decentralized control loops that can be easily equipped with different deployment policies. Specifically, we propose a two-step control loop, in which a model-based reinforcement learning approach dynamically controls the number of replicas of individual containers on the basis of the application response time, and a network-aware placement policy allocates containers on geo-distributed computing resources. To address the placement issue, we propose an optimization problem formulation and a network-aware heuristic, which explicitly take into account the non-negligible network delays among computing resources so to satisfy Quality of Service requirements of latency-sensitive applications. Using a surrogate CPU-intensive application and a real application (i.e., Redis), we conducted an extensive set of experiments, which show the benefits arising from the combination of elasticity and placement policies, as well as the advantages of using network-aware placement solutions.

1. Introduction

To increase the scalability of applications and reduce their response time, the recent trend is to use traditional cloud resources in combination with ever increasing edge/fog computing resources located at the network edges. Their exploitation allows us to decentralize the execution of applications, by moving the computation closer to data sources and consumers (e.g., users, IoT sensors/actuators). Nevertheless, the use of a geographically distributed infrastructure poses new challenges in determining the computing nodes that should host and execute each application instance. Network and system heterogeneity as well as non-negligible network delays among computing nodes are important factors that can negatively impact on latency-sensitive applications. Moreover, being applications subject to varying workloads and exposing stringent Quality of Service (QoS) requirements, their deployment should be also conveniently adapted at run-time.

Today's trend for application development and run-time management is to use software containers. Exploiting operating system level virtualization, containers benefit from a reduced overhead with respect to virtual machines (VMs). Moreover, containers enable to bundle together an application with all its dependencies (i.e., libraries, code), thus allowing for fast start-up, migration, and shutdown times. Considering the dynamism, heterogeneity, and geo-distribution of the emerging edge/fog computing environment, a crucial task is to conveniently solve at run-time the elasticity and placement problem of container-based applications.

An elastic application can dynamically allocate and de-allocate computing resources so to accordingly handle its workload. Elasticity can improve the application availability while limiting execution cost. The *elasticity problem* determines how to scale the number of containers (i.e., horizontal scaling) and/or the amount of computing resources assigned to each of them (i.e., vertical scaling) in face of workload variations [1]. While a fine-grained vertical scaling allows to more quickly react to small workload changes, a horizontal scaling operation makes easier to react to sudden workload peaks. The heterogeneity and geo-distribution of computing resources emphasize the importance of carefully determining which computing nodes will host and execute the application containers. Therefore, alongside the elasticity problem, also the *placement problem* (or scheduling problem) should be efficiently solved at run-time. An improper selection of the computing nodes used

* Corresponding author.
E-mail addresses: f.rossi@ing.uniroma2.it (F. Rossi), cardellini@ing.uniroma2.it (V. Cardellini), lopresti@info.uniroma2.it (F. Lo Presti), nardelli@ing.uniroma2.it (M. Nardelli).

https://doi.org/10.1016/j.comcom.2020.04.061
Received 25 November 2019; Received in revised form 20 March 2020; Accepted 30 April 2020
Available online 7 May 2020

to execute the application can negatively impact its performance and, in a pay-per-use scenario, can lead to higher execution costs. So far, only a limited number of works have studied how to jointly solve the elasticity and placement problems of containers in a geo-distributed environment (e.g., [2,3]). Most of the existing solutions consider either the elasticity of containers or their placement (e.g., [4–6]), and do not often consider the geographic distribution of cloud/fog computing resources (e.g., [7,8]).

When a complex container-based application or multiple applications should be deployed and executed, we resort to *orchestration tools* that automatize container provisioning, management, communication, and fault-tolerance. Although several orchestration frameworks exist in literature, as surveyed in [9,10], Docker Swarm and Kubernetes are the most popular solutions in academic and industrial scenarios. *Docker Swarm* [11] is an open-source platform to create, deploy, and manage containerized applications on distributed computing resources.[1] *Kubernetes* is an open-source platform introduced by Google in 2014 for managing microservice-based applications [12]. Experimental results demonstrate that Kubernetes performs better than other existing orchestration engines, such as Docker Swarm, Apache Mesos, and Cattle [13]. However, Kubernetes (like the other orchestration engines) is not suitable for managing applications in a geographically distributed environment. Indeed, it is equipped with a static threshold-based deployment policy that does not consider network delays and relies on cluster-oriented metrics to horizontally scale applications in face of workload variations [14]. Threshold-based elasticity policies that rely on cluster-oriented metrics are not well suited to scale latency-sensitive applications. Determining a good scaling threshold is cumbersome, because it requires to identify the relation between a system metric (i.e., utilization) and an application metric (i.e., response time). As regards container placement, the default Kubernetes scheduler spreads containers on cluster's worker nodes, not considering the heterogeneity and geographic distribution of the available computing resources. To overcome these limitations, in this paper we propose *ge-kube* (Geo-distributed and Elastic deployment of containers in Kubernetes), that relies on and extends the open-source Kubernetes platform. The main contributions of this paper are as follows.

- We design and implement *ge-kube*, an extension of Kubernetes that introduces self-adaptation and network-aware deployment capabilities through the newly designed Elasticity and Placement Managers components. They exploit decentralized adaptation control loops that can be easily equipped with different QoS-aware deployment policies.
- We integrate flexible scaling and placement policies in ge-kube. To address the elasticity problem, we propose a model-based reinforcement learning (RL) solution, which learns a suitable scaling policy from the experience. It relies on a penalty function that captures the cost of performing scaling operations. To efficiently place pods[2] on geo-distributed worker nodes, we propose an optimization problem formulation and a network-aware heuristic, which explicitly take into account the non-negligible network delay among worker nodes. The proposed solutions overcome some of the limitations of prior works in literature, which do not combine the horizontal and vertical dimensions of elasticity and do not consider the geo-distribution of computing resources.
- We run a thorough set of experiments aimed to evaluate the proposed deployment policies, using a surrogate CPU-intensive application and a real application (i.e., Redis). The experimental results show that our model-based RL solution can efficiently scale the application, so to meet requirements in terms of maximum application response time. The experimental results also show the benefit of allocating pods in a network-aware manner. When a network-aware placement policy is used, the geo-distributed Redis cluster can process 3 times more operations per second with respect to the default policy in Kubernetes.

[1] Docker supports the *swarm mode* starting from version 1.12 for natively managing a cluster of Docker Engines.

[2] A Kubernetes pod encapsulates an application's container or, in some cases, multiple containers.

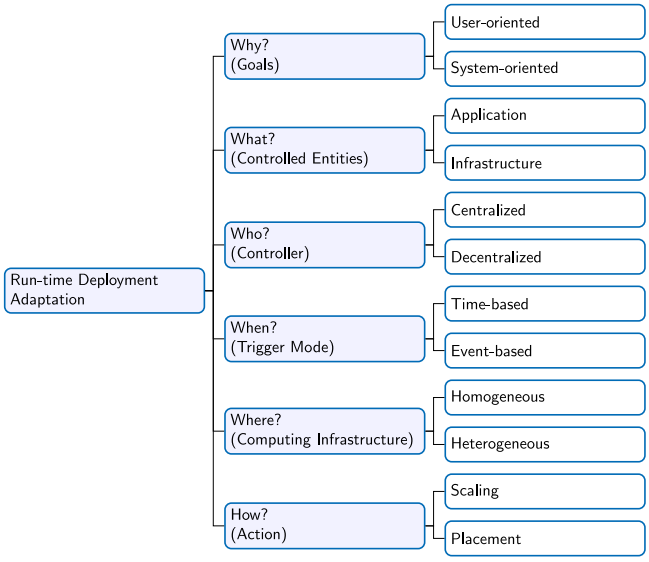

Fig. 1. Taxonomy of existing deployment solutions.

The remainder of the paper is organized as follows. In Section 2, we classify and discuss related work. In Section 3, we present the ge-kube architecture. Then, we propose RL-based elasticity policies and network-aware placement solutions for controlling the deployment of containerized applications (Sections 4 and 5). In Section 6, we evaluate the proposed solutions in a real geo-distributed environment using ge-kube and two different applications. We show the flexibility and efficacy of using network-aware solutions compared to the default scheduling policy of Kubernetes when the orchestration engine is used in a geo-distributed computing environment.

2. Related work

Cloud and edge-native applications [15,16] can operate at large scale, be long-running and subject to a variable incoming workload and transient failures. To preserve performance, their deployment should be accordingly adapted at run-time minimizing human intervention, and software containers can be used to simplify their deployment and management. To summarize and organize the most relevant approaches that deal with the deployment of containerized applications, we present a general taxonomy, built on the following six questions: *why*, *what*, *who*, *when*, *where*, and *how*. These questions help us to identify the key features of the existing deployment solutions, enabling to quickly pinpoint their commonalities and differences. Fig. 1 presents a high-level overview of the proposed taxonomy.

- *Why*: The question pinpoints the utility function and the quantitative metrics that the deployment solution aims to optimize or satisfy. To quantify the deployment objective, several metrics have been adopted in literature; we can broadly distinguish them in *user-oriented* and *system-oriented* metrics. A user-oriented metric models a specific aspect of the application performance, following the user's viewpoint: e.g., throughput, response time, cost, energy consumption. A system-oriented metric aims to quantify a specific aspect of the system, as can be perceived by the provider who wants, for example, to efficiently use the available computing resources. Most works consider user-oriented metrics (e.g., [2,17–21]), whereas few works consider a combination of user- and system-oriented deployment metrics (e.g., [3,6,22]).

- *What*: The question identifies the architectural entities involved in the deployment adaptation actions (i.e., *application-level* and *infrastructure-level* entities). A policy determines infrastructure-level adaptations when it dynamically changes (e.g., resizes) the set of computing resources (e.g., [8,23,24]). A policy that computes application-level adaptations manipulates only the application (and its instances), but does not change the set of available computing resources (e.g., [2,17,25]).
- *Who*: To manage and adapt the application deployment autonomously, we need a deployment controller that provides self-adaptation mechanisms and can be equipped with deployment policies. Few solutions integrate the deployment controller within the application code (e.g., [26]). Having no separation of concerns, the application itself should also carry out the adaptation actions. Although this enables optimized scaling policies, it complicates the application design. Conversely, to improve software modularity and flexibility, most research efforts use an *external controller* (i.e., *orchestration tool*) to manage at run-time the application deployment (e.g., [4,27–30]). When the managed system is geo-distributed, a centralized controller introduces a single point of failure and a bottleneck for scalability. Moreover, it may be able to adapt only a limited number of entities and its efficacy may be negatively affected by the presence of network latencies among the managed system components. As described in [31], different patterns have been used in practice by decentralizing the self-adaptation functions. However, the most used one is the master-worker decentralization pattern (e.g., [29,30]). The master component collects the monitoring data, analyzes them, and dispatches the deployment adaptation actions to the decentralized executors. This relieves part of the burden from the master control node.
- *When*: This question investigates the temporal aspects of the deployment solutions. Since applications are long-running and subject to changing working conditions, many solutions are specifically tailored for addressing challenges and needs of run-time adaptation. In this scenario, we use the *when* question to identify the triggering mode, which determines whether an action is needed and, if so, plans it. In general, the action trigger can be either *time-based* (e.g., [17,25]) or *event-based* (e.g., [27]). A time-based trigger periodically executes the deployment policy. This strategy is simpler to be implemented than the event-based one, so it is the most used approach among the orchestration tools for adapting the application deployment at run-time.
- *Where*: This question aims to characterize the computing infrastructure modeled by the deployment solution. A large number of solutions investigate the deployment problem for applications running in a locally-distributed cloud environment. Therefore, they model a *homogeneous* computing infrastructure (e.g., [17,19,32,33]). Conversely, other deployment solutions explicitly consider *resource heterogeneity*, i.e., they take into account specific features of computing and networking resources, such as processing or storage capacity of resources, available resources, or network delays (e.g., [3,18,21,34]). These solutions can better encompass the specific features of geo-distributed computing environments.
- *How*: The *how* question identifies the set of actions to solve the application deployment problem at run-time. The possible actions include the scaling and placement of the application components. As regard the elasticity, horizontal and vertical scaling directions enable applications to dynamically adapt in face of workload variations. Most of the existing solutions consider only horizontal scaling operations (e.g., [4,8,18,27,32]), whereas fewer solutions exploit only vertical scaling (e.g., [19,25]). So far, only a limited number of research efforts investigate the benefits of combining both horizontal and vertical scaling operations (e.g., [3,17,35]). As regards the placement, it can be determined not only when the application is initially deployed but also at run-time; in the latter case, being triggered by some change in the deployment, such as that occurring after a scale-out decision. Nevertheless, so far, only a limited number of works has studied how to jointly solve the elasticity and placement problems of application instances (e.g., [2,3,18]).

We now further investigate the existing methodologies (*how* question) and orchestration tools (*who* question) that drive and execute the deployment of application instances using containers.

Deployment Policies. To determine which specific (scaling or placement) action should be performed in order to achieve the deployment objectives, a deployment policy is needed. The existing approaches for adapting the deployment of containers can be classified according to the methodologies they exploit: mathematical programming (e.g., [2,3,6,18,36–40]), heuristics (e.g., [7,27,28,33,41,42]) and machine learning solutions (e.g., [3,4,17,34]).

Mathematical programming approaches exploit methods from operational research. Most research efforts (e.g., [2,3,6,36–40]) focus on solving the placement problem of container-based applications. Huang et al. [39] model the mapping of IoT services to edge/fog devices as a Quadratic Programming problem, that, although simplified into an Integer Linear Programming (ILP) formulation, may suffer from scalability issues. Zhao et al. [6] deal with the scheduling of containerized applications solving a data locality-aware multiple knapsack problem. To address the limited scalability of the proposed (NP-hard) approach, they devise heuristics tackling the problem in a bottom-up fashion. This approach is well rooted in the fog environment, characterized by a hierarchical architecture. In [3], we propose a two-step approach that manages the run-time adaptation of container-based applications deployed over geo-distributed VMs. An ILP problem is formulated to place containers on VMs, minimizing the adaptation time and VM cost. In this paper, we integrate our previous ILP formulation as well as a network-aware heuristic in Kubernetes, so to define pods scheduling when Kubernetes manages geo-distributed computing resources. Only a limited number of research works consider the formulation of an optimization problem to drive scaling actions at run-time. Nardelli et al. [18] propose a multi-level optimization problem: at the first level, it deals with the elastic adaptation of the number and type of application instances (i.e., containers); at the second level, it oversees the container placement on a set of VMs that can be elastically acquired and released on demand.

Heuristics are the most popular approach for solving at run-time the deployment problem of containerized applications. The most popular solutions range from greedy approaches (e.g., [33,41]), to threshold-based heuristics (e.g., [4,27,28,43]), to specifically designed solutions (e.g., [7,42,44]). For example, Souza et al. [33] propose a greedy best-fit heuristic that allocates the application instances on fog nodes. If there is not enough processing capacity available on such nodes, cloud resources are used. Threshold-based policies are the most popular approach to scale containers at run-time (e.g., [4,27,28,43]). Barna et al. [27] propose a static threshold-based algorithm to scale containers in a locally-distributed cluster. Also in [28,43] static thresholds are used for planning the adaptation of container deployment taking into account CPU, memory, and network utilization. All these approaches require a manual tuning of the thresholds, which can also be application-dependent—so, in general, performing threshold tuning is not a trivial task. To overcome this limitation, Horovitz et al. [4] propose a novel approach that uses reinforcement learning techniques (i.e., Q-learning) to dynamically adapt the thresholds used to horizontally scale the containers.

In the field of *machine learning*, reinforcement learning (RL) is a special technique by which an agent can learn how to make good decisions through a sequence of interactions with the environment [45]. RL has been mostly applied to devise policies for VM allocation and provisioning (e.g., [23,32]) but it has explored to manage containers in

a limited way [4,17]). Most of the related works considered the classic model-free RL algorithms (e.g., [23,32,34]), which however suffer from slow convergence rate. To tackle this issue, Tesauro et al. [23] propose a solution that combines RL and queuing models within a hybrid approach. The queuing model is used to determine the initial allocation as well as to estimate the system performance. In [17], we propose a novel model-based RL solution that exploits what is known (or can be estimated) about the system dynamics to control the horizontal and vertical elasticity of containers. Simulation and prototype-based experiments have shown the flexibility of our approach: the model-based RL agent learns a better adaptation policy than Q-learning, that allows to meet QoS requirements expressed in terms of average response time. In this paper, we integrate the model-based RL policy (as well as Q-learning) in Kubernetes.

Orchestration Tools. Among the noteworthy orchestration tools to manage containerized applications, we can find Kubernetes, Docker Swarm, Amazon ECS, and Apache Hadoop YARN. To scale containers, they usually rely on best-effort threshold-based policies based on some cluster-level metrics (mostly, they resort on CPU utilization). Moreover, they deploy containers among computing nodes assuming that such resources are locally distributed and interconnected by means of fast communication links (i.e., geographic distribution is often neglected).

Some works aim to improve performance of the existing orchestration tools (e.g., [17,24,40,42,46,47]). Kubernetes includes the Horizontal Pod Autoscaler,[3] which periodically adjusts the number of pods based on the observed CPU utilization. Wu et al. [46] modify the Horizontal Pod Autoscaler to adapt at run-time the deployment of containerized data stream processing applications according to the predicted load arrival rate. To scale Docker containers in Kubernetes, Netto et al. [42] propose a specific solution that uses a state machine approach and provides the application with high availability, integrity, and strong consistency. The solution proposed by Santos et al. [47] uses node labels to decide where to deploy a specific service. First, all the nodes are marked with static labels indicating the round trip time (RTT) between the node and all possible service target locations. Then, considering the configured service target location, the node selection policy chooses a subset of nodes so to minimize the RTT indicated on the node labels. The service target location is statically defined in the pod configuration file. For this strategy, defining the node RTT labels is critical and it may suffer from scalability issues when employed in large-scale environments because it requires to compute the RTT between each pair of nodes, and from network delay spikes because the labels are statically defined. Christodoulopoulos et al. [24] propose Commodore to add infrastructure scalability features to Kubernetes, by allocating (or de-allocating) VMs depending on the cluster state. Garefalakis et al. [40] propose Medea, a cluster scheduler based on Apache Hadoop YARN; however, no network-aware scheduling policy is provided. In [17], we present Elastic Docker Swarm (EDS), an orchestration framework which extends Docker Swarm with elasticity capabilities, resorting on a decentralized control loop. However, EDS does not tackle the container placement in a network-aware manner.

Differently from the existing solutions, in this paper we propose ge-kube, an extension of the well-known open-source Kubernetes platform. As described in the next section, using a modular architecture, ge-kube includes flexible mechanisms that can be easily equipped with custom deployment policies.

Considering the taxonomy represented in Fig. 1, our ge-kube solution populates the six dimensions as follows:

- *Why*: optimization of a multi-objective utility function, which takes into account the application response time and resource utilization (i.e., both user and system-oriented metrics).
- *What*: ge-kube controls only containers, i.e., application-level entities, that can be allocated on the set of computing resources controlled by Kubernetes. In particular, we do not consider infrastructure-level entities within the control loop (i.e., the set of Kubernetes worker nodes is fixed and they are not acquired and released at run-time).
- *Who*: ge-kube exploits multiple control loops, organized according to the master-worker decentralization pattern. In particular, the planning components are logically centralized, so to benefit from a global view of the system and to compute the optimal deployment solution.
- *When*: ge-kube computes the initial application deployment and periodically evaluates it, so to react to changes of the working conditions by planning proper scaling actions. When new containers should be instantiated, as a result of either a scale-out or vertical scaling action, ge-kube triggers the execution of the placement policy.
- *Where*: our contribution considers a system infrastructure composed by geo-distributed and heterogeneous resources, which are interconnected with non-negligible network delays.
- *How*: our solution addresses the application replication and placement problem. To solve these problems, we consider approaches that exploit a variety of methodologies, ranging from mathematical programming, to heuristics and machine learning approaches. Encouraged by the promising simulation results presented in [3], we integrate in ge-kube the proposed deployment policies and evaluate them in a real geo-distributed computing environment.

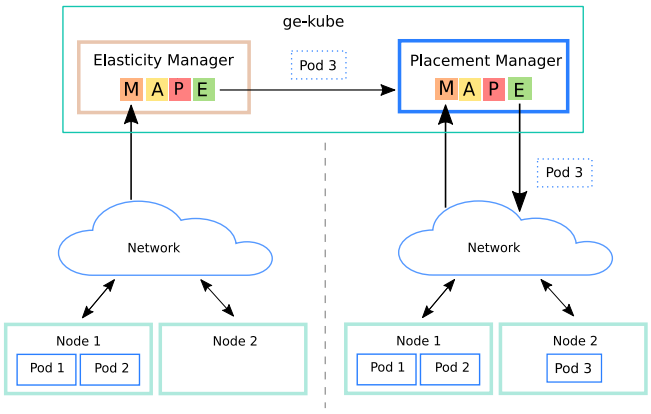

Fig. 2. High-level system overview of ge-kube. The Elasticity Manager chooses the scaling action to be performed according to the workload variations and the monitored metrics (in the example, adding Pod 3). The scaling action can trigger the Placement Manager, that determines the nodes that will host each new application instance.

3. Elastic geo-distributed deployment in Kubernetes

Ge-kube extends the Kubernetes orchestration engine with the goal to enrich it with self-adaptive and network-aware deployment capabilities. Fig. 2 presents a high-level system overview of ge-kube. In ge-kube, two logical components collaborate to jointly explore elasticity and placement dimensions: *Elasticity Manager* and *Placement Manager*. The Elasticity Manager changes the amount of containers used by the application by scaling them out or in, according to the workload variations and the monitored user-oriented and system-oriented metrics (i.e., application response time and resource utilization). When a scale-out operation is performed, the Placement Manager determines the placement of new application instances (i.e., pods), considering resource availability as well as network delays among the geo-distributed worker nodes. We perform elasticity and placement of pods as two distinct operations so to simplify the design of the self-adaptation policies which drive the decisions. This choice fits well within the

[3] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Kubernetes designing approach, which follows the single responsibility principle (i.e., each component oversees a specific task). The elasticity and network-aware capabilities are obtained by introducing two MAPE control loops [31] within Kubernetes. In a MAPE control loop, four main components are responsible for the self-adaptation functionalities: Monitor, Analyze, Plan, and Execute. The Monitor component collects data about the application and the execution environment. By analyzing monitoring data, the Analyze component determines whether the application deployment should be changed; in such a case, it triggers the Plan component, which uses a specific control policy to identify the adaptation action to perform. Ultimately, the Execution component implements the deployment changes. The modularity of Kubernetes and the vast amount of APIs it offers allow us to easily integrate into it the MAPE components. The proposed solution is general enough in that it follows the design principle of separation of mechanisms and policies: therefore, it can be easily equipped with different deployment policies from those we propose in this paper, e.g., that optimize different QoS metrics (such as availability, network usage, energy consumption) or adopt different methodologies.

3.1. Docker and Kubernetes

A software container exploits operating system-level virtualization to bundle an application and its dependencies (e.g., libraries and configurations) so to smoothly run it on (physical or virtual) machines and to isolate and limit resource usage (e.g., using *cgroup*). Although several container management systems exist, Docker is by far the most popular solution to create, distribute, and run applications inside containers. According to Docker, a container is an instance of a Docker image, which is made of a number of layers stacked on top of each other and contains the system libraries, tools, and other files and dependencies for the executable code. Several (public and private) container registries have been developed to improve container image sharing and software reuse (e.g., Docker Hub, OpenShift Container Catalog, Google Container Registry). When a container should be instantiated on a machine, the latter downloads the necessary layers of the container image from an external repository, if the image is not already locally available.

Although deploying a simple container can be easily accomplished using only the container management system, managing a complex application (or multiple applications) during its whole lifetime requires an *orchestration engine*. Kubernetes[4] is an open-source container orchestration engine that simplifies the deployment, management, and execution of containers on distributed computing resources. Besides dealing with configuration, coordination, and communication among containers, Kubernetes can replicate containers for improving resource usage, load distribution, and fault-tolerance.

Kubernetes has a master-worker architecture. The master is in charge of preserving at run-time the desired state of the cluster by managing resources (using workers) and by orchestrating applications (using pods). A worker node is a physical or virtual machine that offers its computational capability for executing pods in a distributed manner. A pod is the smallest deployment unit in Kubernetes. It consists of a single container or a reduced number of tightly coupled containers.[5] Kubernetes allows to configure each container of a pod with specific resource requests and limits. A *resource request* is the minimum amount of (CPU and/or memory) resources needed by the container. A *resource limit* is the maximum amount of resources that can be assigned to the container. Limits and requests for CPU resources are measured in CPU units (e.g., vCPU, hyper-thread).[6] Limits and requests for memory are measured in bytes. Although resource requests and limits can only be specified on individual containers, Kubernetes controls only the overall resource requests and limits used by each pod.

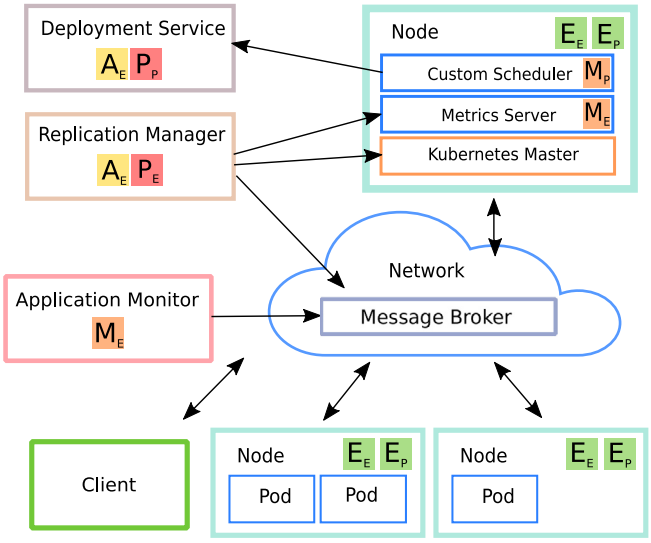

Fig. 3. Architecture of ge-kube. The entities that implement the Elasticity and Placement Manager functions are represented with subscripts E and P, respectively.

Kubernetes can run multiple instances (or replicas) of a pod using a *ReplicaSet*. A ReplicaSet ensures that a given number of pods are up and running, to provide a specific service. When multiple containers run within a pod, they are co-located and scaled as an atomic entity. To simplify the deployment of applications, Kubernetes includes *Deployment* controllers, which build upon the ReplicaSet concept to expose a higher level abstraction. As such, it simplifies the ReplicaSets update. When a Deployment controller runs the application, it can create and destroy pods dynamically. To expose pods as network services, i.e., to make them accessible from external nodes, Kubernetes uses a *Service*, an abstraction that defines a logical set of pods and a policy by which to access them.

When a new application should be executed in Kubernetes, the master node uses the *scheduler* to identify the worker nodes that will host and execute all the application pods. *Kube-scheduler* is the default scheduler for Kubernetes; for each pod to run, it searches for a suitable hosting deployment node using a two-step process described in Section 5.2, since we use the Kubernetes default scheduling policy in the experimental evaluation. Kube-scheduler can be extended to consider other factors while taking scheduling decisions, so to include resource requirements, policy constraints, and affinity and anti-affinity specifications. There are three ways to add new scheduling rules in Kubernetes: (1) by adding rules to the default scheduler; (2) by implementing a new custom scheduler that runs instead of (or alongside with) the standard scheduler; or (3) by implementing a "scheduler extender" that the standard scheduler calls out as a final step when making scheduling decisions.

When we deploy Kubernetes in a geographically distributed environment, we can more easily control pods placement by designing a new custom scheduler. As such, it can conveniently take into account the non-negligible network delays among worker nodes running latency-sensitive applications. The modular architecture of Kubernetes simplifies the development of new custom schedulers. Exploiting this feature, we design a new network-aware deployment solution for Kubernetes.

3.2. Ge-kube architecture

Fig. 3 represents in detail the architecture of ge-kube. It includes decentralized components that implement the Elasticity Manager (components with subscript *E*) and the Placement Manager (components with

[4] https://kubernetes.io.

[5] All containers within a pod are always co-located and share the same execution environment.

[6] https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container.

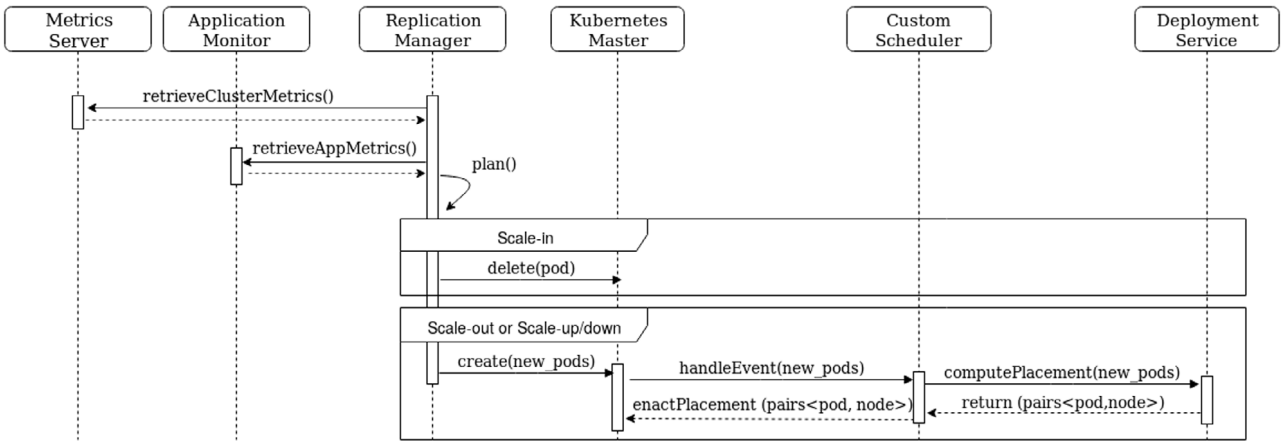

Fig. 4. Sequence diagram representing the execution of the scaling operations in ge-kube. For newly created pods, the Custom Scheduler is notified to define the pods placement on worker nodes. To this end, the Deployment Service implements the placement policy.

subscript *P*). The Elasticity and Placement Managers realize the MAPE control loops within ge-kube, changing the application deployment at run-time.

The *Metrics Server* and the *Application Monitor* represent the MAPE Monitor components of the Elasticity Manager. The Metrics Server[7] is a cluster-level component which periodically collects metrics (i.e., CPU utilization) from all the Kubernetes pods and nodes. Since a pod is a collection of containers, the pod CPU utilization is the sum of the CPU utilization of all its inner containers. Similarly, the node CPU utilization is the total CPU utilization of all pods running on that node. The monitored metrics are aggregated and exposed through a RESTful API. The Application Monitor collects and publishes application-level metrics (i.e., response time) on a message broker (i.e., Apache Kafka). The *Replication Manager* is the centralized control entity, in charge of running the Analyze and Plan phases of the MAPE control loop. First, it receives the monitoring information through the message broker and the Metrics Server. Then, it analyzes and plans scaling actions using an adaptation policy. In this paper, we rely on RL-based elasticity policies, however other policies can be easily integrated. To execute the scaling actions, the Replication Manager uses the standard Kubernetes APIs. Following a master-worker decentralization pattern, a single master component (i.e., Replication Manager) runs the Analyze and Plan phases, while multiple independent worker components execute the Monitor and Execute phases. We observe that the centralization of the Analyze and Plan phases simplifies the design of scaling policies for the Replication Manager.

When new pods should be executed, Kubernetes needs to allocate them on worker nodes. We observe that, besides introducing pods with horizontal scale-out operations, also vertical scaling requires to allocate new pods. Indeed, when a vertical scaling changes pod configuration (e.g., to update CPU limits), Kubernetes spawns new pods as a replacement of those with the old configuration. We develop our placement policies within a library that can be also used by different orchestration engines. To integrate them within Kubernetes, we develop a *Custom Scheduler* and a *Deployment Service*. The Custom Scheduler is deployed as a pod in Kubernetes: besides retrieving the status of (other) pods and worker nodes, it can define pods placement as the default Kube-scheduler does. When a new placement should be computed, the Custom Scheduler identifies the hosting nodes by interacting with the Deployment Service. The latter provides a standard interface to use our placement policies; in particular, it exposes our library through RESTful APIs. The binding between the pod and the chosen node is enacted by the Custom Scheduler through the Kubernetes master.

[7] https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/.

In Fig. 4, we present the sequence diagram of the scaling operations, showing the interaction between the main components of ge-kube. The Replication Manager periodically retrieves system and application metrics from the Metrics Server and the Application Monitor. By analyzing the collected information, it can plan adaptation actions. When a scale-in operation should be performed, it interacts with the Kubernetes master to decrease the number of active pods. By default, Kubernetes deletes first the most recently created pods. Performing a scale-out or a vertical scaling operation requires to add new pods in Kubernetes. In such a case, the Replication Manager simply interacts with Kubernetes for creating new pods. At this stage, Kubernetes needs to identify the worker nodes where the pods will be executed, so it triggers the execution of the Custom Scheduler. Exploiting the available monitoring information, the Custom Scheduler creates a snapshot of the Kubernetes cluster and sends a scheduling request to the Deployment Service. The latter computes the pod-worker node binding by using the configured placement policy and returns it to the Custom Scheduler, that can ultimately enact it in Kubernetes.

Devising an efficient application deployment solution strongly depends on the assumptions about the domain it will be applied to. In Section 4, we describe the system model and the policies used by the Replication Manager to scale the application deployment at run-time. Then, in Section 5 we describe the placement policies used by the Deployment Service.

4. Replication Manager

To dynamically adapt the application deployment exploiting horizontal and vertical elasticity, we have equipped the Replication Manager with reinforcement learning based policies. Differently from the popular threshold-based approaches, we aim to design a flexible solution that can customize the adaptation policy without the need of manually tuning various configuration knobs. RL is a special method belonging to the branch of machine learning. It refers to a collection of trial-and-error methods by which an agent can learn how to make good decision through a sequence of interactions with the environment. One of the main challenges in RL is to find a good trade-off between the exploration and exploitation phases. To minimize the obtained cost, a RL agent must prefer actions that it found to be effective in the past (*exploitation*). However, to discover such actions, it has to explore new actions (*exploration*). At any decision step t, the Replication Manager uses the RL agent to perform the Analyze and Plan phases of the control loop. In the Analyze phase, the RL agent determines the application state (i.e., Deployment controller state) and updates the expected long-term cost (i.e., Q-function). In the Plan phase, according to an action selection policy, the Replication Manager uses the RL

agent to identify the scaling action to be performed. One of the most popular action selection policies is the simple ϵ-greedy, which also addresses the exploitation vs. exploration dilemma. At any decision step, the RL agent chooses, with probability ϵ, a random action to improve its knowledge of the application dynamics (i.e., exploration), whereas, with probability $1-\epsilon$, it chooses the best known action (i.e., exploitation). Most of the time the ϵ-greedy policy exploits the system knowledge, while exploring sub-optimal adaptation actions with low probability.

We define the application state as $s_t = (k_t, u_t, c_t)$, where k_t is the number of application instances (i.e., pods), u_t is the pods CPU utilization, and c_t is the CPU resource limit of each pod. We denote by $\mathcal{S}$ the set of all the application states. We assume that $k_t \in \{1, 2, \dots, K_{\max}\}$ and, being the pods CPU utilization (u_t) and CPU resource limit (c_t) real numbers, we discretize them by defining that $u_t \in \{0, \bar{u}, \dots, L\bar{u}\}$ and $c_t \in \{\bar{c}, 2\bar{c}, \dots, M\bar{c}\}$, where $\bar{u}$ and $\bar{c}$ are suitable quanta. Observe that the minimum amount of CPU resources (i.e., CPU resource request) needed by containers in a pod cannot change at run-time, so we can conveniently set it as $\bar{c}$. Formally, for each state $s \in \mathcal{S}$, we define the set of possible adaptation actions as $\mathcal{A}(s) \subseteq \{-\bar{c}, -1, 0, 1, \bar{c}\}$, where $\pm\bar{c}$ defines a vertical scaling (i.e., $+\bar{c}$ to add CPU share and $-\bar{c}$ to remove CPU share), ± 1 defines a horizontal scaling (i.e., $+1$ to scale-out and -1 to scale-in), and with 0 the *do nothing* decision. Obviously, not all the actions are available in any application state, due to the upper and lower bounds on the CPU resource limits and maximum number of pods per application (i.e., $K_{\max}$). The execution of action a_t in s_t leads to the transition in a new application state (i.e., s') and to the payment of an immediate cost.

We define the immediate cost $c(s_t, a_t, s')$ as the weighted sum of different terms, normalized in the interval $[0, 1]$, where 0 represents the best value (no cost), 1 the worst value (highest cost):

$$c(s_t, a_t, s') = w_{\text{perf}} \cdot c_{\text{perf}} + w_{\text{res}} \cdot c_{\text{res}} + w_{\text{adp}} \cdot c_{\text{adp}} \tag{1}$$

where w_{adp}, w_{perf} and w_{res}, with $w_{\text{adp}} + w_{\text{perf}} + w_{\text{res}} = 1$, are non negative weights that allow us to express the relative importance of each cost term, and c_{perf}, c_{res} and c_{adp} are the *performance penalty*, the *resource cost* and the *adaptation cost*, respectively. The *performance penalty*, c_{perf}, is paid whenever the average application response time exceeds the target value $R_{\max}$. The *resource cost*, c_{res}, is proportional to the number of application instances (i.e., pods) and assigned CPU resource limit. The *adaptation cost*, c_{adp}, captures the number of adaptations. Although we consider stateless applications, performing a scaling operation in Kubernetes introduces an adaptation cost. This depends on the traffic routing strategy used in Kubernetes, which forwards the application requests to the newly added pod, even if not all containers in the pod are already running. A scale-out operation introduces an adaptation cost inversely proportional to the number of application instances (i.e., k). Conversely, when the RL agent chooses a scale-up or a scale-down operation (vertical scaling), we estimate the adaptation cost as unitary. To perform vertical scaling, Kubernetes performs a rolling update that gradually creates a subset of pods with new configuration and deletes a subset of pods with old configuration. By default, Kubernetes ensures that at least 75% of the desired number of pods are up and running during the update; in this stage, the application availability decreases and only a subset of the incoming requests are processed. The formulation of the immediate cost function in (1) is general enough and can be easily customized with other QoS requirements.

The received immediate cost contributes to update the Q-function. The Q-function consists in $Q(s, a)$ terms, which represent the expected long-term cost that follows the execution of action a in state s. The existing RL policies differ in how they update the Q-function and select the adaptation action to be performed (i.e., action selection policy) [45].

Model-based RL. To estimate the Q-function, we consider a model-based solution which we have extensively evaluated in [3,17] using simulation and prototype-based experiments. At any decision step t, the proposed model-based RL solution does not use an action selection policy (e.g., ϵ-greedy action selection policy) but it always selects the best action in term of Q-values, i.e., $a_t = \arg\min_{a' \in A(s_t)} Q(s_t, a')$.

To update the Q-function, the traditional RL solutions (e.g., Q-learning and Dyna-Q) use a simple weighted average:

$$Q(s_t, a_t) \leftarrow (1-\alpha) Q(s_t, a_t) + \alpha \cdot [c_t + \gamma \min_{a' \in \mathcal{A}(s')} Q(s', a')] \tag{2}$$

where $\alpha \in [0, 1]$ is the *learning rate* parameter and $\gamma \in [0, 1)$ the *discount factor*. Conversely, our model-based solution directly updates it using the Bellman equation:

$$Q(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a) \left[c(s, a, s') + \gamma \min_{a' \in \mathcal{A}} Q(s', a') \right] \quad \begin{matrix} \forall s \in \mathcal{S}, \\ \forall a \in \mathcal{A}(s) \end{matrix} \tag{3}$$

where $p(s'|s, a)$ and $c(s, a, s')$ are, respectively, the transition probabilities and the cost function $\forall s, s' \in \mathcal{S}$ and $a \in A(s)$. Thanks to the experience, the proposed model-based solution is able to maintain an empirical model of the unknown external system dynamics (i.e., $p(s'|s, a)$ and $c(s, a, s')$), speeding-up the learning phase. Further details on how to estimate $p(s'|s, a)$ and $c(s, a, s')$ can be found in [3,17], where we have also shown the flexibility and benefits of the novel RL solution: while model-free RL algorithms (i.e., Q-learning and Dyna-Q) suffer from slow convergence rate, the model-based approach can successfully learn the best adaptation policy, according to the user-defined deployment goals.

5. Deployment Service

When the Replication Manager chooses a vertical scaling, all the running pods are terminated and are replaced with new ones having the updated CPU resource. Similarly, when a scale-out is performed, a new pod is created. At this point, the Custom Scheduler determines a suitable mapping between the set of new pods and the set of available nodes through the Deployment Service. The Deployment Service solves the placement problem relying on a scheduling policy. In Sections 5.1 and 5.2, we describe the network-aware and benchmark placement policies. The network-aware policies solve the scheduling problem taking into account the available computing resources as well as the non-negligible network delays between nodes. In Section 5.2, we present simple yet common placement policies that consider resource availability on nodes as the main selection criterion (i.e., *Greedy First-fit Heuristic* and *Round-robin Heuristic*). For sake of comparison, we also describe Kube-scheduler, the default scheduling policy in Kubernetes.

5.1. Network-aware placement policies

In the following, we focus on identifying deployment solutions that minimize the time needed to deploy every application instance (i.e., *deployment time*) and the amount of allocated computing resources, taking into account the network delays between nodes.

5.1.1. Optimal placement problem formulation

Domain Entities. We define the set of pods to be scheduled as P, the CPU resource limit required by each pod p on the hosting node as c, and the set of image layers of containers in pod p as I_p. We refer to the set of worker nodes as N. Each node $n \in N$ is characterized by: C_n, the amount of available CPU cores; I_n, the set of container image layers available on node n; and l_n, the set of pods already running on node n. We define as $d_{n,m}$ the logical network delays between each pair of nodes $n, m \in N$. At this level, we consider the *logical connectivity* between nodes, which results by the underlying physical network paths and routing strategies.

We model the pod placement with binary variables $x_{p,n}$, $p \in P$, $n \in N$: where $x_{p,n} = 1$ if pod p is deployed on node n and $x_{p,n} = 0$ otherwise. We denote by $\boldsymbol{x}$ the placement vector for pods, where $\boldsymbol{x} = \langle x_{p,n} \rangle$, $\forall p \in P$, $\forall n \in N$.

Deployment Time. We define the deployment time of pods $D(\cdot)$ as the time needed to retrieve images of all containers in the pods. Formally, given the placement vector $\boldsymbol{x}$, we have:

$$D(\boldsymbol{x}) = \sum_{p \in P} \sum_{n \in N} D_{p,n}(\boldsymbol{x}) \tag{4}$$

where $D_{p,n}(\boldsymbol{x})$ denotes the time needed to deploy the pod p on n. We define $D_{p,n}(\boldsymbol{x})$ as

$$D_{p,n}(\boldsymbol{x}) = \beta_{p,n} x_{p,n} \tag{5}$$

where $\beta_{p,n}$ models a penalty related to the deployment time of container images I_p not yet in I_n:

$$\beta_{p,n} = \begin{cases} 0 & I_p \subseteq I_n \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

Node Cost. The node cost $Z(\boldsymbol{x})$ counts the number of nodes used for running the application:

$$Z(\boldsymbol{x}) = \sum_{n \in N} z_n \tag{7}$$

$n \in N$ is active (i.e., $z_n = 1$) if it hosts at least one pod. We find convenient to introduce ξ_n constants, with $n \in N$, modeling the presence of pods already running on n; specifically, we have that $\xi_n = 1$, if $|I_n| > 0$, and $\xi_n = 0$, otherwise. Therefore, we can defined z_n as the logical OR between all placement variables $x_{p,n}$, $\forall p \in P$, and ξ_n; formally, we have: $z_n = (\vee_{p \in P} x_{p,n}) \vee \xi_n$.

Application Constraints. The application runs on geo-distributed nodes and exposes strict QoS requirements. Specifically, we consider applications requiring an average response time smaller than $R_{\max}$. Although scaling the application may improve performance, we clearly see that determining an appropriate application placement is of key importance. If replicas are placed on worker nodes interconnected with high delay, the application latency increases and, in the worst case, it can be unfeasible to satisfy $R_{\max}$. Therefore, we should reduce the network delay between the active nodes. We consider that the placement policy can allocate the application only on worker nodes whose network delay is below the critical value ND_{max}. Formally, we have that $d_{n,m} \cdot \bar{z}_{n,m} \leq ND_{max}, \forall n, m \in N$, where $\bar{z}_{n,m}$ is a binary variable representing whether any application pod runs in m and any other runs in n (i.e., $\bar{z}_{n,m} = z_n \cdot z_m$).

Optimal Problem Formulation. We formulate the optimal mapping of the pods on the geo-distributed nodes as an ILP model. Our problem formulation considers an objective function that minimizes the deployment time and the node cost. We define the objective function $F(\boldsymbol{x})$ as follows:

$$F(\boldsymbol{x}) = \frac{D(\boldsymbol{x})}{D_{\max}} + \frac{Z(\boldsymbol{x}) - Z_{\min}}{Z_{\max} - Z_{\min}} \tag{8}$$

where $D_{\max}$ and $Z_{\max}$ ($Z_{\min}$) denote the maximum (minimum) value for the overall expected adaptation time and cost, respectively. Formally, $D_{\max} = |P|$, $Z_{\min} = 1$, and $Z_{\max} = |N|$. After normalization, each metric ranges in the interval $[0, 1]$, with 0 the best possible case, and 1 to the worst case. The Optimal Placement Problem is formulated as follows:

$$\min_{\boldsymbol{x}} F(\boldsymbol{x}) \quad \textbf{subject to}:$$

$$\sum_{n \in N} x_{p,n} = 1 \qquad \forall p \in P \tag{9}$$

$$\sum_{p \in P} c_p x_{p,n} \leq C_n \qquad \forall n \in N \tag{10}$$

$$d_{n,m} \cdot \bar{z}_{n,m} \leq ND_{\max} \qquad \forall n, m \in N \tag{11}$$

$$\frac{\xi_n}{\Gamma} + \frac{1}{\Gamma} \sum_{p \in P} x_{p,n} \leq z_n \leq \sum_{p \in P} x_{p,n} + \xi_n \qquad \forall n \in N \tag{12}$$

$$z_n + z_m - 1 \leq \bar{z}_{n,m} \leq \frac{z_n + z_m}{2} \qquad \forall n, m \in N \tag{13}$$

$$x_{p,n} \in \{0, 1\} \qquad \forall p \in P, \forall n \in N \tag{14}$$

$$z_n \in \{0, 1\}, \bar{z}_{n,m} \in \{0, 1\} \qquad \forall n, m \in N \tag{15}$$

Eq. (9) requires that a pod p is allocated on one and only one node. Eq. (10) allows to find a correct placement ensuring that a hosting node exposes only its available resources. Eq. (11) expresses the application constraint on network delay between the hosting nodes. These nodes are identified using z_n and $\bar{z}_{n,m}$, whose definition is in Eqs. (12) and (13). Γ is a large constant, such that $\Gamma \geq |P| + 1$.

5.1.2. Network-aware greedy heuristic

The ILP formulation models an NP-hard problem [3]; so, it does not scale well as the problem instance increases. To overcome this limitation, we propose a *network-aware greedy heuristic* (see Algorithm 1). It solves a variant of the bin-packing problem, while taking into account the deployment time of pods, the available computing resources and the network delays between active worker nodes.

The Network-aware Greedy heuristic schedules one pod at a time. For each pod p and worker node n, the heuristic computes the node contribution to the objective function $F(\boldsymbol{x})$ (8), whose value results by computing $F(\boldsymbol{x})$ considering only the terms related to the pod p and node n (line 3). Then, it sorts the available nodes by their contribution to the objective function in ascending order (i.e., the first nodes of the list offer a reduced deployment time and node cost). Afterwards, the heuristic identifies the first worker node in the sorted list that can host the pod and that has a network delay with all other worker nodes used for the placement below $ND_{\max}$ (line 11). When the node is found, it is used for the pod placement (line 19). Since the heuristic greedily evaluates nodes from the sorted list, the first candidate node minimizes also the pod deployment time. It is worth to observe that the proposed heuristic avoids spreading pods among the computing nodes, while preferring to minimize the number of active nodes.

Algorithm 1 Network-aware Greedy Heuristic

```
    Input P: pods to schedule
    Input N: available nodes
 1: for all p ∈ P do
 2:     for all n ∈ N do
 3:         n.computeLocalObjFunction(p)
 4:     end for
 5:     N.sortByLocalObjFunctionAsc()
 6:     i = 0
 7:     n = null
 8:     S = new Scheduling()
 9:     while i < |N| do
10:         n = N.get(i)
11:         if n.canHost(p) ∧ maxNetDelay(S ∪ n) ≤ ND_max then
12:             break
13:         end if
14:         i++
15:     end while
16:     if n == null then
17:         throw Error("unable to find a placement")
18:     end if
19:     S.schedule(p, n)
20: end for
```

5.2. Benchmark placement policies

In this section, we present existing placement policies against which we will evaluate our network-aware solutions. First, we describe two well-known placement policies, that are often implemented in computing frameworks, namely Greedy First-fit and Round-robin. Then, we describe the placement solution implemented by the default scheduler of Kubernetes.

Greedy First-fit Heuristic. The Greedy First-fit heuristic determines the application placement using a greedy approach. It is one of the most

Table 1
Average network round-trip time (RTT) between data centers hosting Kubernetes worker nodes. Delays are expressed in milliseconds.

	us-central1	*eu-west1*	*eu-north1*	*northamerica-northeast1*
us-central1	1	100	135	32
eu-west1	100	1	33	82
eu-north1	135	33	1	115
northamerica-northeast1	32	82	115	1

popular solutions used to solve the bin packing problem. Our implementation considers the application instances (i.e., pods) as elements to be (greedily) allocated in bins, representing computing resources. Specifically, the heuristic adds the available cluster nodes to a list and sorts it in ascending order of available resources. For each pod, Greedy First-fit policy defines the placement on the first node selected from the sorted list.

Round-robin Heuristic. The Round-robin heuristic aims to evenly distribute pods on worker nodes, without considering network delays among nodes. Basically, it organizes the cluster nodes in a circular list and records the latest node used for scheduling. When a new pod needs to be allocated, the heuristic assigns it to the first worker node with enough computing resources, starting from the current position on the circular list.

Kube-scheduler. Kubernetes includes a default scheduler, named *kube-scheduler*, which allocates pods on worker nodes according to an extensible policy. The default scheduling strategy is *spread*, which distributes pods so to optimize for the node with the least number of pods. For each pod, Kube-scheduler chooses a destination node through a two-step procedure.

In the first step, Kube-scheduler identifies those worker nodes that can run the pod by applying a set of filters (i.e., predicates). For example, if the pod declared a key–value selector (i.e., *label*), the scheduler can select worker nodes matching the key–value pair. Thus, the first step discards the nodes that cannot satisfy the required resources and the label matching defined in the pod configuration file.

In the second step, the Kube-scheduler ranks the nodes remaining after the first step through a set of priority functions. Each priority function assigns a score to each node and the final score of each node is calculated by adding up all the given weighted scores. After having assigned the score to the remaining nodes, the one having the highest score is chosen to run the pod. If multiple nodes achieve the same score, one of them is randomly selected. The idea of the second step is to spread pods by minimizing the number of pods belonging to the same service on nodes with the same label.

6. Experimental results

In this section, we use ge-kube, our extension of Kubernetes, to evaluate the proposed deployment policies in a real geo-distributed environment.

We define two sets of experiments aimed to show the benefits of our model-based RL solution and network-aware placement policies when the application managed by ge-kube is deployed in a geo-distributed environment.

First, in Section 6.2, we use a surrogate CPU-intensive application that computes the Fibonacci number. The application is subject to a varying workload, so we can show the benefits of combining elasticity and placement policies. Then, in Section 6.3, we deploy a Redis cluster[8] on ge-kube, so to further investigate the results achieved by the placement policies and generalize their outcomes. We compare the network-aware policies, presented in Section 5.1, against the benchmark placement policies, presented in Section 5.2.

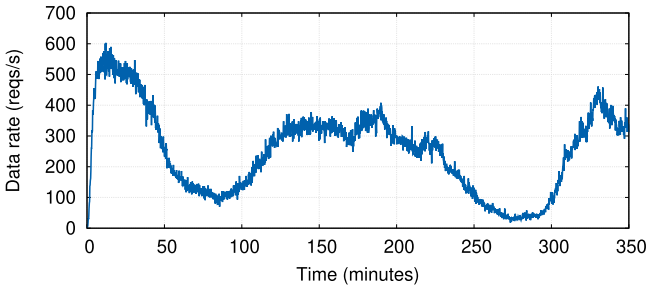

Fig. 5. Workload used for the reference application.

6.1. Experimental setting

We deploy ge-kube on a cluster of 12 virtual machines of the Google Cloud Platform[9]; each virtual machine has 2 vCPUs and 7.5 GB of RAM (type: *n1-standard-2*). To evaluate our solution in a geo-distributed environment, we acquire virtual machines in 4 different regions (or zones), where data centers are interconnected with non-negligible delays (i.e., *us-central1*, *eu-west1*, *eu-north1*, *northamerica-northeast1*). The zones and the average network delays between them are reported in Table 1. As placement policies, we consider the optimal ILP formulation (referred as *OPT*), the network-aware greedy heuristic (*NetAware*), the *Greedy First-fit* heuristic, the *Round-robin* heuristic, and the default policy included in Kubernetes, named *kube-scheduler*. To solve the optimal ILP formulation, we use CPLEX 12.8.

6.2. Combining scaling and placement policies

We consider a CPU-intensive reference application that computes, upon request, the sum of the first n elements of the Fibonacci sequence (using an algorithm with complexity $O(n^2)$). As shown in Fig. 5, the application receives a varying number of requests. It follows the workload of a real distributed application [48], accordingly amplified and accelerated so to further stress the application resource requirements.

The application requires $R_{\max} = 100$ ms as target response time and the maximum network delays between nodes to be $ND_{\max} = 85$ ms. To meet the user-oriented QoS requirement, it is important not only to adapt the application deployment according to the workload variations by taking proper scaling decisions, but also to wisely choose the hosting nodes for the application. To dynamically scale the application at run-time, we equip the Replication Manager with our model-based RL solution. In [3,17], we have extensively shown that the model-based RL agent can learn a suitable adaptation policy, under different weight configurations of the immediate cost function (1). Exploiting information about the system dynamics, it learns a better adaptation policy than model-free RL algorithms (e.g., Q-learning). Considering the large set of scaling and placement policies configurations, in this paper we mainly consider the model-based RL solution (*MB*, for short) for scaling the application at run-time. However, for sake of comparison, we also present the results achieved by the model-free Q-learning policy (*QL*) when used in combination with the optimal placement policy.

The Replication Manager executes the Analyze phase of the MAPE control loop every 3 minutes. To learn an adaptation policy, the RL algorithms have been tuned with the following parameters: discount factor $\gamma = 0.99$ and, for Q-learning, learning rate $\alpha = 0.1$ and $\epsilon = 10\%$. To discretize the application state (as explained in Section 4), we use $K_{\max} = 10$, $\bar{u} = 0.1$, and $\bar{c} = 10\%$. We consider the set of weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$ in (1). With this weight configuration, optimizing the application response time is more important than saving resources; moreover, adaptation costs are negligible. The Placement

[8] https://redis.io/.

[9] https://cloud.google.com/.

Table 2
Fibonacci application performance under different configurations of elasticity and placement policies.

Elasticity Policy	*Placement Policy*	R_{max} *violations* (%)	*Average pods CPU utilization* (%)	*Average CPU limit* (%)	*Average number of pods*	*Adaptations* (%)	*Average number of used nodes*
Model-based	OPT	22.76	43.24	66.83	5.75	61.79	3.79
	NetAware	11.54	44.00	64.23	5.71	31.89	3.07
	Greedy First-fit	22.14	55.86	87.10	3.65	70.23	4.46
	Round-robin	16.26	42.04	62.20	5.89	42.28	5.24
	kube-scheduler	20.16	44.02	91.29	4.04	55.65	2.85
Q-learning	OPT	94.35	86.00	80.32	1.62	70.16	1.34

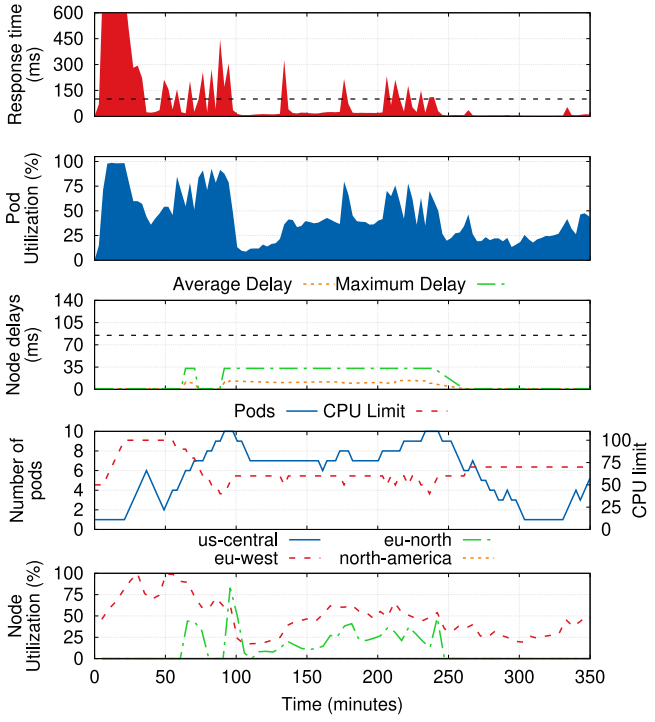

Fig. 6. Fibonacci application performance using the *Model-based RL scaling* policy and the *Optimal placement* policy.

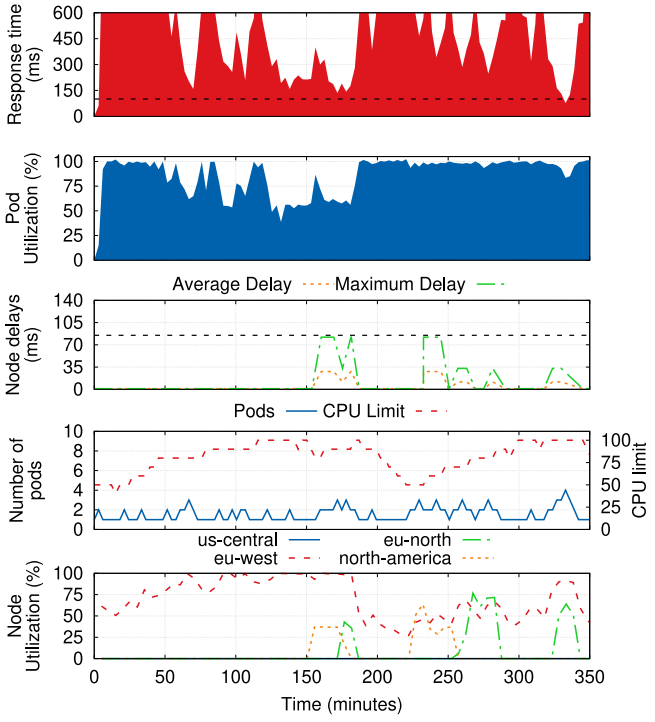

Fig. 7. Fibonacci application performance using the *Q-learning RL scaling* policy and the *Optimal placement* policy.

Manager executes the Analyze phase of its MAPE control loop as soon as a new pod should be deployed. The binding between the pod and node depends on the chosen placement policy.

We summarize the experimental results in Table 2, and Figs. 6–11. We can readily see that the application placement differs largely when the different policies are used. Figs. 6–11 also show that the placement and elasticity problems are closely related one another. A good placement policy helps to meet stringent QoS requirements and, on the other hand, a good elasticity policy can mitigate the adoption of a sub-optimal placement. Moreover, the figures show the placement policy effect on the average node utilization. For sake of clarity, we group nodes in regions (i.e., *us- central*, *eu-west*, *eu-north*, *north-america*) and report the average CPU utilization of (only) active worker nodes.

Fig. 6 shows the application performance when the model-based RL solution and the optimal placement policy drive the scaling and placement of the application pods, respectively. We observe that, in the first minutes of the experiment, the scaling policy continuously changes the deployment. Indeed, the RL agent starts with no knowledge on the adaptation policy, so it begins to explore the cost of each adaptation action. As soon as a suitable adaptation policy is learned, the RL solution can successfully scale the application and meet the application response time requirement, even when the Round-robin policy computes the placement (see Fig. 10). The learned adaptation policy deploys a number of pods that follows the application workload (see Figs. 5 and 6). On average, the application runs with 5.75 pods with a CPU limit of 67%. Performing vertical scaling introduces response time peaks due to the Kubernetes routing policy. The RL agent learns that vertical scaling operations are more expensive than horizontal ones: indeed, 63% of adaptation actions are for horizontal scaling. Thanks to the optimal placement policy, pods are scheduled on worker nodes belonging to data centers that are close one another (i.e., eu-west1 and eu-north1). The network delay between pods is on average 7 ms, whereas the maximum delay is 33 ms, which is well below the ND_{max} requirement. Fig. 6 shows also that the optimal placement policy deploys pods on two neighbor data centers, thus allowing to satisfy the application ND_{max} requirement.

When the optimal placement policy is used in combination with the model-free Q-learning scaling policy, the application performance is very different (see Fig. 7). In particular, due to the complexity of the computing environment, the model-free scaling policy cannot learn an adaptation policy within the time interval of the experiment (i.e., 350 minutes). Fig. 7 clearly shows that the application deployment is continuously updated, with the Replication Manager that performs both horizontal and vertical scaling operations. As a consequence, the application response time exceeds R_{max} most of the time, even though the optimal placement policy is in charge of defining the application scheduling. Conversely, taking advantage of the system knowledge, the model-based solution drastically reduces the number of R_{max} violations. Besides that, the optimal placement policy schedules pods on worker nodes in data centers located in Europe (i.e., *eu-west1* and *eu-north1*), respecting the ND_{max} requirement. Since Q-learning has lower

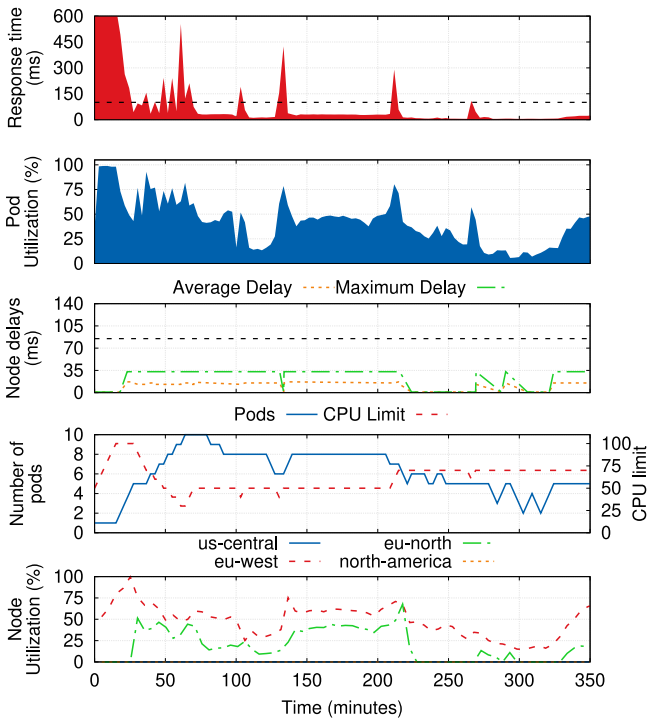

Fig. 8. Fibonacci application performance using the *Model-based RL scaling* policy and the *Network-aware placement heuristic*.

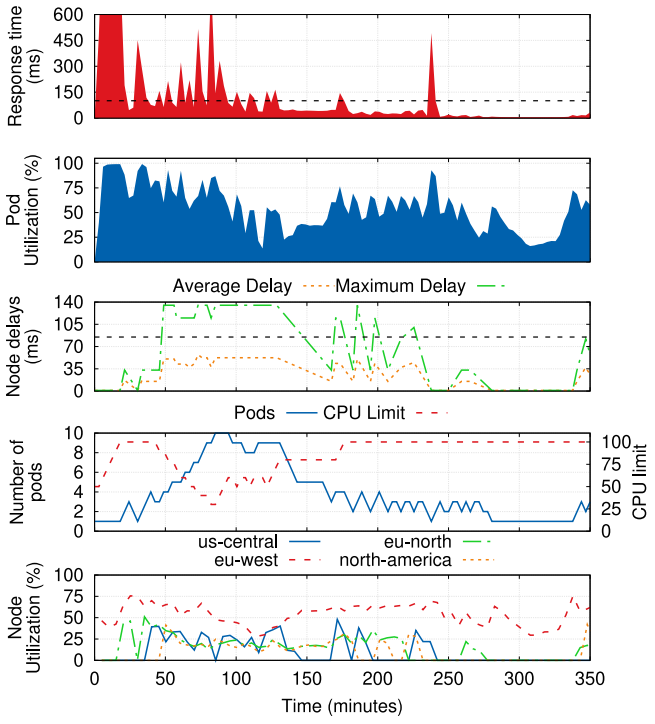

Fig. 9. Fibonacci application performance using the *Model-based RL scaling* policy and the *Greedy First-fit placement* policy.

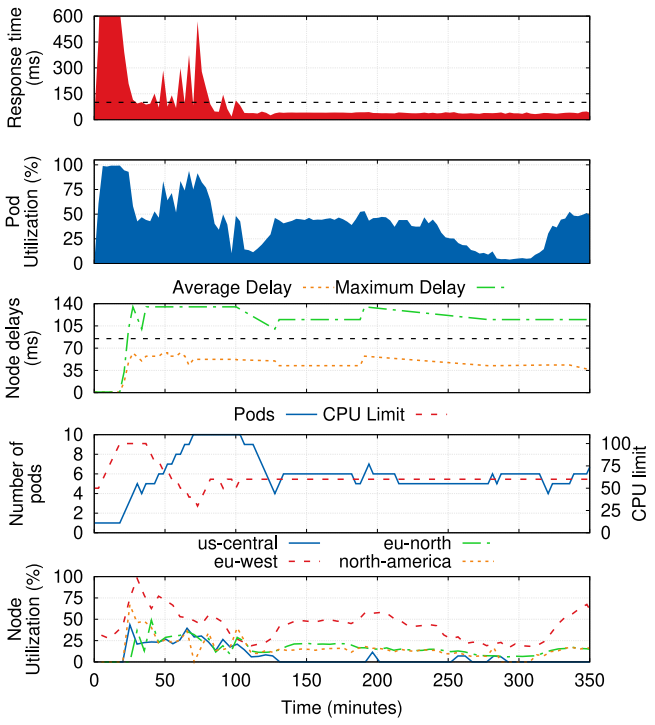

Fig. 10. Fibonacci application performance using the *Model-based RL scaling* policy and the *Round-robin placement* policy.

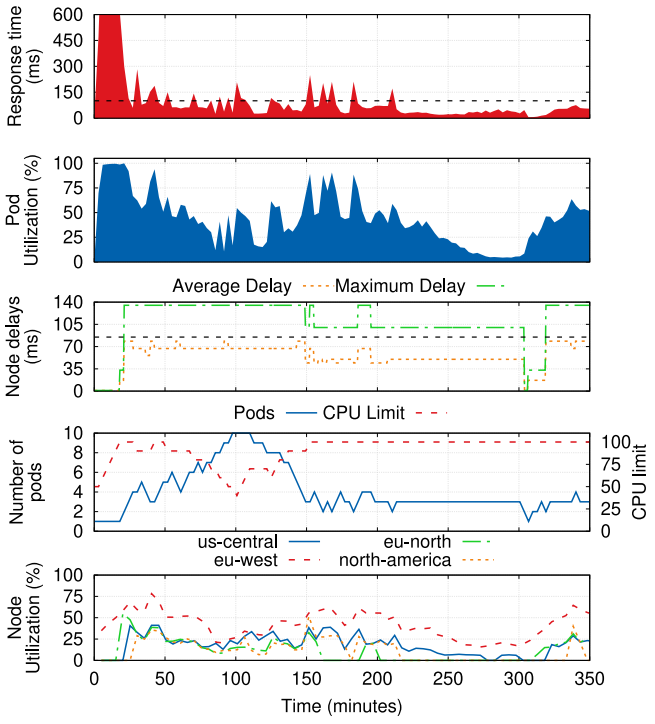

Fig. 11. Fibonacci application performance using the *Model-based RL scaling* policy and the *Kube-scheduler*.

performance than model-based, from now on, we only consider the model-based RL solution for scaling containers.

On average, the optimal placement policy takes 59 ms to compute the placement, although, in the worst case, the resolution time grows up to 1.4 s, thus suggesting the adoption of efficient network-aware placement heuristics. The network-aware greedy heuristic allows to obtain application performance close to that achieved by the optimal placement policy (see Fig. 8). It schedules the application pods on worker nodes interconnected with an average network delay of 11 ms and maximum delay of 33 ms (thus satisfying the $ND_{\max}$ requirement). Also in this case, the placement policy deploys the application in European data centers, maintaining a reduced number of active nodes

(i.e., 3.07 on average). Thanks to the combination of the model-based RL solution and network-aware placement heuristic, the application exceeds $R_{\max}$ only 12% of the time.

The Greedy First-fit and Round-robin heuristics do not take into account network delays while computing the application placement. As a consequence, the RL agent learns a scaling policy that slightly differs from the previous ones (see Figs. 9 and 10). Specifically, the RL agent allocates more resources to the application to overcome the downside of a not convenient placement. When our model-based RL solution is used in combination with the Greedy First-fit heuristic, the RL agent changes the application deployment for the 70% of the time. The Greedy First-fit heuristic policy tries to minimize at run-time the number of active nodes, placing (if possible) the newly added pods on nodes that already host at least one pod. The application uses on average 4.5 worker nodes, spread on all the data centers. Fig. 9 shows that, in the last part of the experiment, the RL agent uses a reduced number of pods that can access to the whole CPU unit. With this configuration, the placement heuristic schedules few pods that, by chance, are allocated on nodes with limited network delays (below 85 ms).

The Round-robin heuristic spreads the application pods on every data center, using on average 5.2 worker nodes, which is the highest number of nodes with respect to the other configurations. Although the average network delay between nodes ranges between 40 ms and 60 ms, its maximum values ranges between 115 ms and 135 ms, thus violating $ND_{\max}$. Fig. 10 shows that, for the first 100 min, the RL agent reconfigures continuously the application deployment, performing both horizontal and vertical scaling operations. Consequently, all the computing nodes are used at least once to schedule the (new added or resized) pods. In the second part of the experiment, the RL agent applies only few changes to the application deployment, probably because it over-provisions computing resources. From Fig. 10, we can observe an unbalanced pod distribution among the data centers. This follows from the combination of the Round-robin placement policy and the default Kubernetes scale-in policy, which removes first the most recently added pods. Note that, in this work, we do not change the scale-in policy and we do not adapt at run-time the application placement. As a consequence, most of the running pods are allocated in the *us-central1* data center.

To conclude, we combine the model-based RL solution with the default scheduler of Kubernetes, kube-scheduler, which does not take into account network delays while selecting nodes for pods. The scheduling policy spreads pods on 2.85 nodes, on average, and the maximum network delay exceeds $ND_{\max}$ most of the time during the experiment. Nonetheless, the RL agent learns an adaptation policy that results in a 20% of response time violations (see Fig. 11). In the first 150 minutes, it switches between horizontal scaling and vertical scaling operations, using nodes from all the data centers. Then, the RL agent reacts to workload changes using only horizontal scaling operations: usually, it uses between 2 and 3 pods, each accessing all the CPU unit. kube-scheduler spreads the application instances between Europe and America (i.e., *eu-west1*, *eu-north1*, and *us-central1*), so the application response time is generally higher than that resulting from the other deployment configurations.

To summarize the behavior of the evaluated elasticity and placement policies, in Fig. 12, we report the overall distribution of the application response time using boxplots. Each boxplot reports the 5th, 25th, 50th, 75th, and 95th percentile of the application response time. Furthermore, we represent the maximum response time $R_{\max}$ with a horizontal dotted line. We can readily observe that the network-aware policies obtain a better response time distribution, with median value of 10–15 ms. Then, we find all the other benchmark placement policies that obtain higher value of application response time. In particular, the Greedy First-fit, Round-robin, and kube-scheduler obtain the median application response time of 26 ms, 37 ms, and 37 ms, respectively. When Q-learning drives the application elasticity, the application has a very high response time, and even the 25th percentile is above the critical value $R_{\max}$.

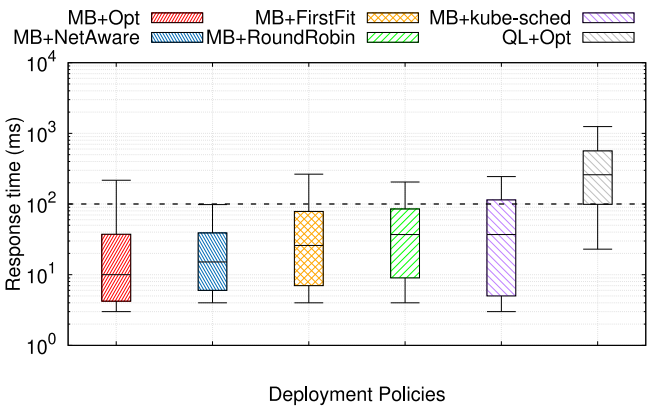

Fig. 12. Response time distribution of the Fibonacci application when deployed with the different policies. Boxplots report 5th, 25th, 50th, 75th, and 95th percentile of the response time distribution. For the sake of brevity, we indicate the model-based RL scaling policy as MB, and the Q-learning scaling policy as QL.

This set of experiments clearly shows the benefits of using network-aware placement policies when Kubernetes is deployed on geo-distributed computing resources. Furthermore, the experiments show how the Elasticity Manager and Placement Manager can cooperate to drive the application deployment so to meet QoS requirements in face of changing working conditions. The optimal problem formulation cannot be used for large scale problem instances, because it models an NP-hard problem. Therefore, we designed a network-aware greedy heuristic that, also in this small-size problem instance, allowed to reduce the problem resolution time from 59 ms (of the optimal placement formulation) to few milliseconds.

6.3. Deploying Redis cluster

In this experiment, we deploy a Redis cluster on ge-kube, using the different network-aware and benchmark placement policies. Redis is a popular key–value data store often included in web applications to implement in-memory distributed caching. We consider the same computing infrastructure of the previous experiment, which consists of 12 virtual machines acquired across 4 regions.

The Redis cluster includes 3 master nodes; every node is responsible for a subset of the hash slots. Hash slots result by applying a hash function on the key of the key–value pair to store. For the experiment, we require that Redis is deployed on nodes interconnected with a network delay below $ND_{\max} = 50$ ms.

Redis supports different kinds of abstract data structures (e.g., strings, lists, maps, sets, sorted sets) and exposes different primitives to operate on them. In this experiment, we consider only a subset of the available operations, namely *get*, *set*, *lpop*, *lpush*, and *hset*: *get* returns the value of a key; *set* assigns a string value to the key; *lpop* removes and returns the first element of the list stored in a key; *lpush* inserts all the specified values at the head of the list stored in a key; and *hset* adds a value to a dictionary stored in a key. To benchmark the cluster, we use the *redis-benchmark*[10] utility, that simulates 50 concurrent clients performing 10^5 operation requests.

Fig. 13 shows the application throughput, when the Redis cluster is deployed using the different placement policies. We repeat each experiment 5 times; each histogram reports the average value as well as the minimum and maximum values represented using a vertical line. Overall, the behavior of the different placement policies is consistent with the previous experiment. The network-aware solutions define a placement taking network delays into account, so they deploy the Redis

[10] https://redis.io/topics/benchmarks.

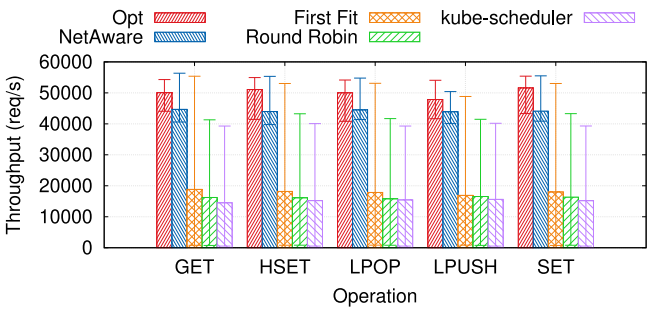

Fig. 13. Performance of a Redis Cluster deployed using the different placement policies. Throughput of different operations on the Redis data store.

cluster on nodes that are close to one another. The maximum network delay between the Redis masters is below 35 ms.

The optimal placement solution computes the best placement, which obtains on average the highest throughput for each operation: on average, it performs $50{\times}10^3$ operations per seconds. This happens when Redis is deployed on a single data center. The network-aware greedy heuristic often finds the best application placement, although it sometimes spreads the Redis cluster across two data centers (i.e., *eu-west1* and *eu-north1*). Anyway, it always meets the network delay requirements $ND_{\max}$. When deployed with the network-aware heuristic, Redis performs on average 44×10^3 operations per second.

The benchmark heuristics compute a different placement for the Redis cluster, which can also include worker nodes interconnected with non-negligible network delays. When such nodes are considered, the application performance of Redis is significantly reduced. In the worst case, the cluster can satisfy only 529 requests per second (the worst case is obtained with kube-scheduler). The Greedy First-fit heuristic shows, on average, better performance than the other benchmark policies, because it tends to co-locate the Redis nodes within few data centers. Using this policy, Redis can process on average 18×10^3 requests/s. Round-robin and kube-scheduler result in similar application performance, although in this experiment Round-robin computes a slightly better application placement. When Round-robin is used, Redis performs on average 16×10^3 operations per second, whereas when kube-scheduler is used, it accommodates 15×10^3 requests/s.

Overall, the network-aware placement policies allow to improve the Redis cluster performance, when deployed in a geo-distributed computing environment. In particular, when the network-aware greedy heuristic is used, Redis can process 2.91 times more operations per second than the deployment by the classic Kubernetes scheduling policy.

7. Conclusions

Kubernetes is one of the most popular container orchestration engines used in the academic and industrial world. Kubernetes has been originally designed to operate in a locally-distributed cluster, so it includes a placement policy that deploys an application by simply spreading its containers among the worker nodes. This approach is not well-suited to tackle the dynamism and heterogeneity of the new emerging geo-distributed environment. Therefore, in this paper we proposed ge-kube, an extension of Kubernetes that introduces self-adaptation and network-aware scheduling capabilities. As such, it allows to efficiently deploy and dynamically scale applications that run in geo-distributed environments. To manage the application elasticity, we integrate in ge-kube a model-based RL solution that scales the application containers so to satisfy requirements on application-oriented metrics. To manage the application placement, we integrate in ge-kube network-aware placement policies. However, the modular architecture of ge-kube and the self-adaptation mechanisms we introduced are general enough and can be easily equipped with other deployment policies. We conducted an extensive evaluation of the resulting system using both a surrogate application and a real application (i.e., Redis). The experimental evaluation showed the benefits of combining elasticity and placement policies, as well as the importance of using network-aware placement policies when Kubernetes is deployed in a geo-distributed environment.

As future work, we plan to extend the proposed heuristics so to efficiently control the scaling and placement of multi-component applications (e.g., microservices). Considering also stateful applications, we want to investigate the impact of performing online state migrations in terms of performance penalty, so to include such information in the adaptation loop that controls the application deployment. Similarly, we are interested in modeling other performance metrics (e.g., network usage, fault-tolerance, energy consumption), that can be of interested for applications running on geo-distributed resources.

CRediT authorship contribution statement

Fabiana Rossi: Conceptualization, Methodology, Software, Visualization, Writing - original draft. **Valeria Cardellini:** Conceptualization, Methodology, Supervision, Writing - review and editing. **Francesco Lo Presti:** Conceptualization, Methodology, Supervision, Writing - review and editing. **Matteo Nardelli:** Conceptualization, Methodology, Validation, Writing - review and editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We gratefully acknowledge the support we have received from Google with the GCP research credits program.

References

[1] A. Barnawi, S. Sakr, W. Xiao, A. Al-Barakati, The views, measurements and challenges of elasticity in the cloud: A review, Comput. Commun. 154 (2020) 111–117, http://dx.doi.org/10.1016/j.comcom.2020.02.010.

[2] X. Guan, X. Wan, B.Y. Choi, S. Song, J. Zhu, Application oriented dynamic resource allocation for data centers using Docker containers, IEEE Commun. Lett. 21 (3) (2017) 504–507, http://dx.doi.org/10.1109/LCOMM.2016.2644658.

[3] F. Rossi, V. Cardellini, F. Lo Presti, Elastic deployment of software containers in geo-distributed computing environments, in: Proc. of IEEE ISCC '19, 2019, pp. 1–7, http://dx.doi.org/10.1109/ISCC47284.2019.8969607.

[4] S. Horovitz, Y. Arian, Efficient cloud auto-scaling with SLA objective using Q-learning, in: Proc. of IEEE FiCloud '18, 2018, pp. 85–92, http://dx.doi.org/10.1109/FiCloud.2018.00020.

[5] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, X. Koutsoukos, Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications, in: Proc. of IEEE CLOUD '18, 2018, pp. 82–89, http://dx.doi.org/10.1109/CLOUD.2018.00018.

[6] D. Zhao, M. Mohamed, H. Ludwig, Locality-aware scheduling for containers in cloud computing, IEEE Trans. Cloud Comput. (2018) http://dx.doi.org/10.1109/TCC.2018.2794344.

[7] C. Guerrero, I. Lera, C. Juiz, Genetic algorithm for multi-objective optimization of container allocation in cloud architecture, J. Grid Comput. 16 (1) (2018) 113–135, http://dx.doi.org/10.1007/s10723-017-9419-x.

[8] M. Nardelli, C. Hochreiner, S. Schulte, Elastic provisioning of virtual machines for container deployment, in: Proc. of ACM/SPEC ICPE '17 Comp., 2017, pp. 5–10, http://dx.doi.org/10.1145/3053600.3053602.

[9] E. Casalicchio, Container orchestration: A survey, in: A. Puliafito, K.S. Trivedi (Eds.), Systems Modeling: Methodologies and Tools, Springer International Publishing, Cham, 2019, pp. 221–235, http://dx.doi.org/10.1007/978-3-319-92378-9_14.

[10] M.A. Rodriguez, R. Buyya, Container-based cluster orchestration systems: A taxonomy and future directions, Softw. Pract. Exp. 49 (5) (2019) 698–719, http://dx.doi.org/10.1002/spe.2660.

[11] Swarm mode overview, 2019, https://docs.docker.com/engine/swarm/.

[12] K. Hightower, B. Burns, J. Beda, Kubernetes Up and Running: Dive into the Future of Infrastructure, O'Reilly Media, Inc., 2017.

[13] I.M.A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli, Container orchestration engines: A thorough functional and performance comparison, in: Proc. of IEEE ICC '19, 2019, pp. 1–6, http://dx.doi.org/10.1109/ICC.2019.8762053.

[14] S. Taherizadeh, M. Grobelnik, Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications, Adv. Eng. Softw. 140 (2020) http://dx.doi.org/10.1016/j.advengsoft.2019.102734.

[15] D. Gannon, R. Barga, N. Sundaresan, Cloud-native applications, IEEE Cloud Comput. 4 (5) (2017) 16–21, http://dx.doi.org/10.1109/MCC.2017.4250939.

[16] M. Satyanarayanan, G. Klas, M. Silva, S. Mangiante, The seminal role of edge-native applications, in: Proc. of IEEE EDGE '19, 2019, pp. 33–40, http://dx.doi.org/10.1109/EDGE.2019.00022.

[17] F. Rossi, M. Nardelli, V. Cardellini, Horizontal and vertical scaling of container-based applications using reinforcement learning, in: Proc. of IEEE CLOUD '19, 2019, pp. 329–338, http://dx.doi.org/10.1109/CLOUD.2019.00061.

[18] M. Nardelli, V. Cardellini, E. Casalicchio, Multi-level elastic deployment of containerized applications in geo-distributed environments, in: Proc. of IEEE FiCloud '18, 2018, pp. 1–8, http://dx.doi.org/10.1109/FiCloud.2018.00009.

[19] A. Asnaghi, M. Ferroni, M.D. Santambrogio, DockerCap: A software-level power capping orchestrator for Docker containers, in: Proc. of IEEE EUC '16, 2016, pp. 90–97, http://dx.doi.org/10.1109/CSE-EUC-DCABES.2016.166.

[20] K. Kaur, T. Dhand, N. Kumar, S. Zeadally, Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers, IEEE Wirel. Commun. 24 (3) (2017) 48–56, http://dx.doi.org/10.1109/MWC.2017.1600427.

[21] M. Abbasi, M. Yaghoobikia, M. Rafiee, A. Jolfaei, M.R. Khosravi, Efficient resource management and workload allocation in fog-cloud computing paradigm in IoT using learning classifier systems, Comput. Commun. 153 (2020) 217–228, http://dx.doi.org/10.1016/j.comcom.2020.02.017.

[22] A. Mseddi, W. Jaafar, H. Elbiaze, W. Ajib, Joint container placement and task provisioning in dynamic fog computing, IEEE Internet Things J. (2019) http://dx.doi.org/10.1109/JIOT.2019.2935056.

[23] G. Tesauro, N.K. Jong, R. Das, M.N. Bennani, A hybrid reinforcement learning approach to autonomic resource allocation, in: Proc. of IEEE ICAC '06, 2006, pp. 65–73, http://dx.doi.org/10.1109/ICAC.2006.1662383.

[24] C. Christodoulopoulos, E.G.M. Petrakis, Commodore: Fail safe container scheduling in Kubernetes, in: Proc. of AINA '19, Springer International Publishing, 2019, pp. 988–999, http://dx.doi.org/10.1007/978-3-030-15032-7_83.

[25] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Autonomic vertical elasticity of Docker containers with ElasticDocker, in: Proc. of IEEE CLOUD '17, 2017, pp. 472–479, http://dx.doi.org/10.1109/CLOUD.2017.67.

[26] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Elasticity in cloud computing: State of the art and research challenges, IEEE Trans. Serv. Comput. 11 (2018) 430–447, http://dx.doi.org/10.1109/TSC.2017.2711009.

[27] C. Barna, H. Khazaei, M. Fokaefs, M. Litoiu, Delivering elastic containerized cloud applications to enable DevOps, in: Proc. of SEAMS '17, 2017, pp. 65–75, http://dx.doi.org/10.1109/SEAMS.2017.12.

[28] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, A. Leon-Garcia, Elascale: Autoscaling and monitoring as a service, in: Proc. of CASCON '17, 2017, pp. 234–240, URL http://dl.acm.org/citation.cfm?id=3172795.3172823.

[29] M.S. de Brito, S. Hoque, T. Magedanz, R. Steinke, A. Willner, D. Nehls, O. Keils, F. Schreiner, A service orchestration architecture for fog-enabled infrastructures, in: Proc. of FMEC '17, 2017, pp. 127–132, http://dx.doi.org/10.1109/FMEC.2017.7946419.

[30] S. Hoque, M.S. d. Brito, A. Willner, O. Keil, T. Magedanz, Towards container orchestration in fog computing infrastructures, in: Proc. of IEEE COMPSAC '17, Vol. 2, 2017, pp. 294–299, http://dx.doi.org/10.1109/COMPSAC.2017.248.

[31] J.O. Kephart, D.M. Chess, The vision of autonomic computing, IEEE Comput. 36 (1) (2003) 41–50, http://dx.doi.org/10.1109/MC.2003.1160055.

[32] H. Arabnejad, C. Pahl, P. Jamshidi, G. Estrada, A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling, in: Proc. of IEEE/ACM CCGrid '17, 2017, pp. 64–73, http://dx.doi.org/10.1109/CCGRID.2017.15.

[33] V. Souza, X. Masip-Bruin, E. Marín-Tordera, S. Sànchez-López, J. Garcia, G. Ren, A. Jukan, A.J. Ferrer, Towards a proper service placement in combined fog-to-cloud (F2C) architectures, Future Gener. Comput. Syst. 87 (2018) 1–15, http://dx.doi.org/10.1016/j.future.2018.04.042.

[34] Z. Tang, X. Zhou, F. Zhang, W. Jia, W. Zhao, Migration modeling and learning algorithms for containers in fog computing, IEEE Trans. Serv. Comput. 12 (5) (2019) 712–725, http://dx.doi.org/10.1109/TSC.2018.2827070.

[35] L. Baresi, S. Guinea, A. Leva, G. Quattrocchi, A discrete-time feedback controller for containerized cloud applications, in: Proc. of ACM SIGSOFT FSE '16, 2016, pp. 217–228, http://dx.doi.org/10.1145/2950290.2950328.

[36] H.R. Arkian, A. Diyanat, A. Pourkhalili, MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications, J. Netw. Comput. Appl. 82 (2017) 152–165, http://dx.doi.org/10.1016/j.jnca.2017.01.012.

[37] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, P. Hu, DRAPS: dynamic and resource-aware placement scheme for Docker containers in a heterogeneous cluster, in: Proc. of IEEE IPCCC '17, 2017, pp. 1–8, http://dx.doi.org/10.1109/PCCC.2017.8280474.

[38] M.I. Naas, P.R. Parvedy, J. Boukhobza, L. Lemarchand, iFogStor: An IoT data placement strategy for fog infrastructure, in: Proc. of IEEE ICFEC '17, 2017, pp. 97–104, http://dx.doi.org/10.1109/ICFEC.2017.15.

[39] Z. Huang, K.-J. Lin, S.-Y. Yu, J.Y. jen Hsu, Co-locating services in IoT systems to minimize the communication energy cost, J. Innov. Digit. Ecosyst. (ISSN: 2352-6645) 1 (1) (2014) 47–57, http://dx.doi.org/10.1016/j.jides.2015.02.005.

[40] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, S. Rao, Medea: Scheduling of long running applications in shared production clusters, in: Proc. of EuroSys '18, ACM, 2018, http://dx.doi.org/10.1145/3190508.3190549, 4:1–4:13.

[41] E. Yigitoglu, M. Mohamed, L. Liu, H. Ludwig, Foggy: A framework for continuous automated IoT application deployment in fog computing, in: Proc. of IEEE AIMS '17, 2017, pp. 38–45, http://dx.doi.org/10.1109/AIMS.2017.14.

[42] H.V. Netto, A.F. Luiz, M. Correia, L. de Oliveira Rech, C.P. Oliveira, Koordinator: A service approach for replicating Docker containers in Kubernetes, in: Proc. of IEEE ISCC '18, 2018, pp. 58–63, http://dx.doi.org/10.1109/ISCC.2018.8538452.

[43] H. Khazaei, H. Bannazadeh, A. Leon-Garcia, SAVI-IoT: A self-managing containerized IoT platform, in: Proc. of IEEE FiCloud '17, 2017, pp. 227–234, http://dx.doi.org/10.1109/FiCloud.2017.27.

[44] A. Kwan, J. Wong, H. Jacobsen, V. Muthusamy, Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres, in: Proc. of IEEE ICDCS '19, 2019, pp. 80–90, http://dx.doi.org/10.1109/ICDCS.2019.00017.

[45] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, second ed., MIT Press, Cambridge, MA, 2018.

[46] Y. Wu, R. Rao, P. Hong, J. Ma, FAS: A flow aware scaling mechanism for stream processing platform service based on LMS, in: Proc. of ICMSS '17, ACM, 2017, pp. 280–284, http://dx.doi.org/10.1145/3034950.3034965.

[47] J. Santos, T. Wauters, B. Volckaert, F. De Turck, Towards network-aware resource provisioning in Kubernetes for fog computing applications, in: Proc. of IEEE NetSoft '19, 2019, pp. 351–359, http://dx.doi.org/10.1109/NETSOFT.2019.8806671.

[48] Z. Jerzak, H. Ziekow, The DEBS 2015 grand challenge, in: Proc. ACM DEBS 2015, 2015, pp. 266–268, http://dx.doi.org/10.1145/2675743.2772598.