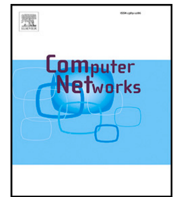
Computer Networks 247 (2024) 110444

Contents lists available at ScienceDirect

Computer Networks

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



Taming latency at the edge: A user-aware service placement approach

Carlo Centofanti *, Walter Tiberti, Andrea Marotta, Fabio Graziosi, Dajana Cassioli

Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, via Vetoio, L'Aquila, 67100, Italy

ARTICLE INFO

Keywords:
Edge computing
Orchestration
Service placement
Kubernetes
Latency optimization

ABSTRACT

Modern network and computing infrastructures are tasked with addressing the stringent demands of today's applications. A pivotal concern is the minimization of latency experienced by end-users accessing services. While emerging network architectures provide a conducive setting for adept orchestration of microservices in terms of reliability, self-healing and resiliency, assimilating the awareness of the latency perceived by the user into placement decisions remains an unresolved problem. Current research addresses the problem of minimizing inter-service latency without any guarantee to the level of latency from the end-user to the cluster. In this research, we introduce an architectural approach for scheduling service workloads within a given cluster, prioritizing placement on the node that offers the lowest perceived latency to the end-user. To validate the proposed approach, we propose an implementation on Kubernetes (K8s), currently one of the most used workload orchestration platforms. Experimental results show that our approach effectively reduces the latency experienced by the end-user in a finite time without degrading the quality of service. We study the performance of the proposed approach analyzing different parameters with a particular focus on the size of the cluster and the number of replica pods involved to measure the latency. We provide insights on possible trade-offs between computational costs and *convergence time*.

1. Introduction

The increase of Internet of Things (IoT) devices and the rising demand for real-time processing [1] have highlighted the importance of edge computing in the modern computing ecosystem. Unlike centralized cloud computing, edge computing shifts processing closer to the data sources, which include devices like smartphones, IoT sensors, Augmented Reality (AR), and Virtual Reality (VR) devices, and autonomous vehicles. This shift not only promises to offload computational resources from centralized data centers, thereby reducing their burden, but also signifies a paradigmatic move towards reduced latency and improved data processing efficiency. However, the decentralization process introduces new challenges, chief among them being the scheduling and allocation of services on edge nodes. This problem is known in the literature as the Service Placement problem [2]. The placement of services on appropriate edge nodes is crucial for the optimization of network-wide performance metrics and for enhancing end-user Quality of Service (QoS). The action of choosing a node to place a service on is also known as service scheduling. The heterogeneous nature of edge resources, coupled with dynamic and often unpredictable network traffic, amplifies the challenges of service scheduling in the distributed edge context. The challenge lies in reconciling multiple objectives: minimizing latency, maximizing resource utilization, optimizing geospatial service distribution, and catering to user-specific or application-specific requirements. An ineffectual scheduling risks not only a degradation of QoS but also potential resource imbalances, leading to network inefficiencies and potential bottlenecks.

A technological foundation of service placement is represent by Microservice (μS) architectural approach. Microservices represent a modern software design paradigm, to decompose applications into a collection of a set of loosely coupled, independently deployable, and secure services [3,4]. The evolution of μS architecture is known as cloud-native architecture. In contemporary research, cloud-native architecture is delineated as an architectural approach optimized for cloud computing environments [5,6]. This architecture predominantly employs microservices, containerization, continuous integration/continuous delivery, and orchestration tools, such as Kubernetes, ensuring resilience, flexibility, and efficient scaling in response to dynamic cloud-based workloads. Each microservice implements a specific responsibility, enabling fine-grained scalability, resiliency, and maintainability. The application is then composed as an aggregation of these microservices. However, when deploying microservices in distributed systems like edge computing environments, the placement of individual microservices on specific nodes becomes a critical concern [7]. The location of these services strongly influences the End-to-End (E2E) experienced latency of the entire application. For example, services that frequently

* Corresponding author.
E-mail address: carlo.centofanti1@univaq.it (C. Centofanti).

https://doi.org/10.1016/j.comnet.2024.110444
Received 31 October 2023; Received in revised form 22 March 2024; Accepted 18 April 2024
Available online 24 April 2024
1389-1286/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

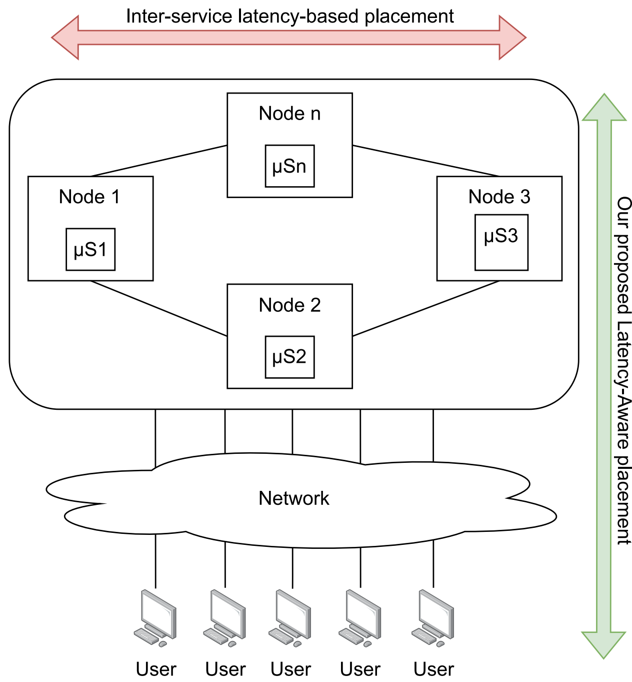


Fig. 1. High level concept of the inter-service latency-based placement vs. the proposed latency-aware placement approach.

communicate each other might be placed on nearby nodes to reduce the overall communication delay due to the required information exchange. In the literature this delay is referred to as *inter-service latency* and many studies emphasize the placement of microservices to optimize this latency. This approach is shown in Fig. 1 with the horizontal red arrow.

However, an even more critical aspect is represented by the latency between the users and the microservices they request, i.e., along the vertical green arrow in Fig. 1. In this complex scenario there is an exigent need for an architectural framework adept at making informed and fast service scheduling decisions at the edge.

To this purpose, we embark on an exploration of this pertinent challenge and extend our previous work [8] by proposing a novel Latency-Aware scheduler architecture tailored to service placement in edge computing environments, with a focus on addressing the low latency challenges intrinsic to this computational modality. Our Latency-Aware scheduler is designed to optimize the latency between the service and the user which can reduce further the experienced E2E latency in the context of a distributed cluster, by seamlessly reducing the Round Trip Time (RTT) while maintaining the QoS required by that user.

1.1. Contributions

This paper introduces several significant advancements in the field of service placement within edge computing environments, specifically focusing on the critical role of latency in shaping user experience and system performance. The novel contributions of this research are as follows:

- Introduction of a *Latency-Aware Scheduler architecture*, specifically designed for edge computing scenarios. This architecture enhances the end-to-end latency by optimizing the latency between the service and the user through service migration to the end user.
- Detailed exploration of the *impact of sentinel deployment on system scalability and performance*, facilitated by the development of an analytical framework and the introduction of a new quantitative metric. This analysis provides insights into the trade-offs between resource efficiency and latency reduction, advancing the discourse on distributed computing system design and operational management.
- *Empirical validation of the Latency-Aware scheduling approach* within a Kubernetes cluster, demonstrating its effectiveness in achieving optimal service placement efficiently. The established relationship between the number of sentinels and the system's convergence time highlights the intricate balance between resource allocation and system performance.

These contributions underscore the innovative approach of this research in addressing the complex challenges of latency-sensitive service placement within edge computing. By prioritizing latency as a key factor in scheduling decisions, this work enables more advanced user-centric and efficient distributed computing environments.

1.2. Paper organization

The remainder of this paper is organized as follows: Section 2 delves into the related works, providing an overview of previous studies and methodologies that bear relevance to our approach. In Section 3, we detail our system model, elucidating the design principles and architectural considerations. Section 4 presents the Implementation on Kubernetes, outlining the steps and processes employed to bring our model to fruition within the Kubernetes ecosystem. Section 5 focuses on the validation and performance evaluation, and provides a comprehensive discussion of our results, highlighting the key outcomes and implications of our findings. Finally, in Section 6, we draw conclusions from our study, summarizing the key takeaways and suggesting avenues for future research.

2. Related work

Due to its intrinsic distributed nature, the edge computing, and even more the fog paradigm, pose new challenges to orchestrators, because the management complexity is much higher in a distributed computing infrastructure than in a traditional centralized cloud-computing environment. To increase the elasticity and resilience of a fog computing system, cluster federation was applied in [9] by means of the Kubernetes Cluster Federation (KubeFed), augmented with a two-phase workload placement mechanism to smartly distribute applications' microservices among the federated infrastructure. The implementation resulted in a *decentralized control plane* able to take into account networking issues in a highly distributed architecture.

Current research contributions on service placement and computation offloading in fog computing-based IoT are surveyed in [10], whereas [11] focuses on nature-inspired approaches applied to the service placement and computation offloading in emerging edge technologies, formulated as NP-hard problems.

The baseline problem underlying the placement optimization of micro-services in a highly distributed environment is the container scheduling. There exist many surveys presenting container scheduling, and some others related to containers' orchestration in fog/edge architectures. In [12], a survey on the state-of-the-art of the Kubernetes orchestrator including relevant contributions from the 2016 to 2022 period is presented and open challenges and research opportunities on the topic are identified. It provides a study of empirical research on Kubernetes scheduling techniques and a new taxonomy for Kubernetes scheduling.

In order to realize the more reasonable allocation of Pod scheduling on Kubernetes, a new cost function combining the resource and load balancing costs is defined in [13] to optimize the task scheduling in the cloud computing environment by means of a meta heuristic algorithm derived from the combination of the adaptive particle swarm algorithm and an improved version of the ant colony algorithm.

However, a centralized scheduler model, as Kubernetes, has been proved not to perform well in all scenarios like, e.g., for resiliency and fault-tolerance scheduling. Moreover, it cannot solve problems like service high-availability, collocation interference, priority preemption or inherent rescheduling. Hence, a novel hybrid-state scheduling framework for the Kubernetes system has been proposed in [14]; it provides scheduling corrections for the processing of unscheduled and unprioritized jobs and assigns distributed agents to optimize locally the main tasks, whereas the application-level scheduler synchronizes the cluster state across all agents.

A policy-based scheduling is proposed in [15] for multi-cloud deployments, where realistic experiments have shown that the proposed method balances the resource allocation across multiple geo-distributed clusters and reduces the fraction of pending pods from 65% in the case of Kubernetes Federation to 6% for the same workload.

Unfortunately, the aforementioned papers do not address the specific low-latency requirement imposed to modern communication systems. Orchestration driven by application requirements still represents an open challenge in Kubernetes environment and some research efforts have been made in this direction. To achieve this goal, architectural solutions enabling network-awareness in scheduling decisions need to be investigated. A network-aware resource scheduling approach for delay-sensitive and data-intensive services in fog computing environments is proposed in [16] for container-based applications in smart city deployments, and validated on the Kubernetes platform. Implemented as an extension to the default scheduling mechanism available in Kubernetes, it enables resource provisioning decisions based on the current status of the network infrastructure, thus achieving reductions up to 80% in terms of network latency when compared to the default scheduling mechanism. The node selection is based on the minimization of the nominal RTT estimated on the basis of the target location for the service, as specified on the pod configuration file. However, the nominal RTT is an optimistic metric, whereas an experimental measure taken on top of the micro-service stack (as proposed in this paper) is a more effective metric for a latency-aware scheduler.

In [16] a scheduling strategy is proposed based on static characterization of network latency between cluster nodes. [17] proposes a heuristic strategy for the deployment of placeholder pods for enhanced reliability and takes into account inter-node latency. [18,19] propose Kubernetes scheduler extensions to introduce network-awareness for efficient placement of pods. The above mentioned approaches focus on network awareness and rely on latency metrics collected at the network layer via independent applications responsible to monitor network performance parameters to optimize inter-container latency.

However, latency experienced by end-users includes network latency to reach the cluster and a non-negligible contribution of the application-layer which for example may be impacted by computational resource saturation, hardware heterogeneity through cluster nodes and other external conditions. The problem of optimizing E2E latency is not directly addressed in literature and can be solved reporting to the scheduler the measured E2E latency, as proposed in this work.

It is worth mentioning that the proposed approach enables fine-tuned mobility management which still represents an open issue in the context of service orchestration in decentralized architectures. A performance cost analysis of IPv6-mobility management in Kubernetes environment has been conducted in [20].

3. System model

To ensure operational efficiency, the concept of *service placement* is pivotal since making timely decisions without disrupting the user experience is imperative. Expanding upon this, we introduce the proposed *latency aware service placement*, an approach to ensure optimal system responsiveness and E2E performances.

Distributed clusters are a crucial concept in the domain of cloud computing and modern application deployment. This concept refers to multiple interconnected nodes or clusters spanning various geographical locations or different cloud providers [21]. By leveraging these geographically distributed systems, organizations can ensure high availability, fault tolerance, and optimal performance by serving end-users from the nearest data center. However, in clusters with geographically distributed nodes, varying latencies can be observed. We consider the system model shown in Fig. 2 representing a distributed cluster where nodes exhibit different latencies with respect to the end-user.

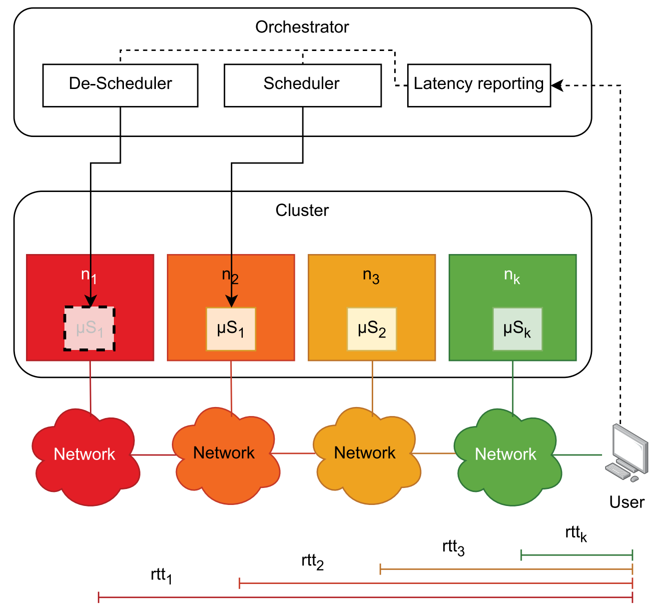


Fig. 2. System model.

The architecture of the considered distributed cluster comprises several elements. The *Nodes* are primary units, responsible for hosting services. Each node can be interpreted as a physical server, a virtual machine, or a container, depending on the deployment granularity. *Service Instances* are the deployable units of software in this system. These can be applications, microservices, or other computational entities. They are dispersed across nodes and are ready for execution. The *Network Fabric* encompasses physical links, switches, routers, and other networking equipment, determining communication paths, latencies, and communication speeds. *end-users*, either humans or systems, engage with the services furnished by the architecture. Their geographic positions and access patterns play into service placement and migration decisions.

All the mentioned elements are overseen by an *Orchestrator* which has the significant role of managing, deploying, and ensuring fault tolerance of services across nodes. Furthermore, it facilitates service migration between nodes, considering various factors, including latency optimization.

As illustrated in Fig. 2, being $\mathcal{N} = \{n_1, n_2, \dots, n_k\}$ the set of nodes possessing the capability to host multiple service instances. The overarching goal is to fine-tune the placement of these services taking into account latency, i.e. RTT, experienced by the users.

The RTT is illustrated at the top of the figure, with a wider RTT indicating greater network delay. Within this reference architecture, it is evident that the green node offers the lowest RTT. Thus, positioning the service S_k on this node would optimize the latency experienced by the end-user. As shown in Fig. 2 for the service S1, moving a service from a node with higher RTT (i.e. n_1) to one with a lower one (i.e. n_2), implies a reduction in the latency experienced by the user. The operation to move the service from one node to another is termed as migration. Transitioning the service to a specific node

must be executed transparently to ensure the end-user experiences no service disruptions or interruptions. In the considered architecture the operation of instantiating a service on a node is accomplished by the Scheduler component while the one of removing a service instance from a node is performed by the De-Scheduler.

Amid this context, the placement of a service within a distributed cluster based on latency becomes central. It is the aim of our system to position a service in such a manner that it is as proximate to the end-user in terms of response time as possible. This kind of optimization becomes especially significant for applications where swift responses are imperative, such as real-time multimedia sharing, online interactive experiences, and rapid financial transactions.

Achieving this optimal placement is not straightforward with conventional orchestration scheduling mechanisms. Hence, our system introduces a latency-aware service management approach. Central to our system is an iterative discovery strategy ensuring the best service allocation based on the latency experienced by application users. This strategy encompasses custom latency-aware scheduling and de-scheduling components integrated within the orchestration platform. These components work in unison, evaluating and, when necessary, relocating the service to the location providing the shortest latency to the end-user.

The task of collecting latency measures is performed by a Latency reporting element within the orchestrator. This tool operates on a client–server paradigm, with the client positioned at the user's side and the server within the orchestration's service instance. Communication between this tool and the de-scheduling component is managed through a messaging protocol, e.g. Message Queuing Telemetry Transport (MQTT).

To obtain achievable performance of the service over different nodes, the orchestrator creates a unique "sentinel" variant of the service to measure iteratively the E2E latency for specific locations in the cluster. After each iteration, only the service version with the most advantageous latency remains active, with decisions guided by the de-scheduling component. A new version then initiates at another location, and the cycle persists until the most favorable placement is discerned.

Our system differentiates between standard service instances and "sentinels". It initiates with a service's commencement at a location, followed by a sentinel's onset at an alternate location for latency measurement. This sequence is reiterated with subsequent sentinels, each contributing latency data. The service's position either remains or shifts depending on which location yields the shortest latency.

Moreover, the system remains vigilant about latency. Should latency exceed a predetermined threshold, the iterative discovery process recommences, prompting sentinels at diverse locations. This adaptability ensures the system's responsiveness to evolving network scenarios or changes in user locations.

Without the intervention of Sentinel replicas, latency from the user to the service endpoints remains anchored to the initial node allocation decisions made by Kubernetes. In scenarios where Kubernetes' scheduling decisions are primarily influenced by factors such as resource availability rather than latency, the resultant user-perceived latency can be suboptimal. This static approach to service deployment neglects the dynamic nature of network conditions and user locations, potentially locking the user experience into the latency characteristics of the initially selected nodes without the possibility for continuous optimization.

To clarify this aspect, consider the example of a cloud application designed to serve a global user base. Suppose the application is scaled to 100 instances (i.e., replicas), distributed across a network that encompasses 10 nodes in Europe, 10 in Asia, and 10 in North America. Utilizing a static allocation mechanism in conjunction with a load-balancer, service orchestrators might default to an even distribution, allocating approximately 3 replicas per node. However, from the perspective of user experience and perceived latency, such a configuration may not represent the optimal approach. Latency is inherently dynamic, subject to fluctuations influenced by many factors, including time of day and user traffic volume. At any given moment, a predominant segment of the user base might experience enhanced service quality by accessing replicas hosted on, for instance, European nodes. Conversely, a few hours later, the demographic shift might favor users predominantly located in North America, who would then find the latency from Asian or European nodes unsatisfactory.

Our methodology offers a solution to this issue by dynamically adjusting the allocation of replica-node pairs in response to real-time latency perceptions. While it is challenging to predict latency reductions in advance due to the variable nature of user and node configurations, our approach is designed to minimize latency, ensuring that optimal allocation is achieved through iterative adjustments.

Additionally, this principle can be effectively scaled and adapted to edge computing scenarios, where the complexity and scale of the system are reduced. The underlying strategy of relocating services closer to users to minimize latency remains consistent across different computing paradigms.

3.1. Real-world network latencies scenarios

We explore the practical implications of network latencies across a spectrum of global server locations provided by Amazon Web Services (AWS), with a particular focus on how these latencies impact user experiences and network reliability. The considered server destinations are Milan, google.com, Frankfurt, London, Stockholm, Ireland, Northern Virginia, Tokyo. The measurement procedure was designed to capture an accurate representation of network latencies. This process was facilitated through the deployment of a Bash script, leveraging the Internet Control Message Protocol (ICMP) protocol via the ping command. For each location pair, the script conducted 1000 measurements over a duration of 100 s, systematically repeating this measurement cycle every hour for a total period of 24 h. Subsequent to the data collection phase, statistical analysis was performed to calculate the average, minimum, and maximum latency values, as well as the standard deviation for each set of measurements. The Time To Live (TTL) values were directly taken from the ping command responses, while the number of network hops was derived through a calculation involving the subtraction from a default TTL value of 128, as it is the standard value for the ping command on Ubuntu systems. This methodological approach ensures a comprehensive dataset, providing a foundational basis for understanding the real-world dynamics of network latencies. Table 1 provides a comprehensive overview of the latency metrics observed from an Italian Gigabit Passive Optic Network (GPON) fixed access line connection originating in L'Aquila – a city in the center of Italy – to the above-mentioned server destinations. It should be noted that the domain google.com is resolved by Google's Domain Name System (DNS) and is directed to Google's Content Delivery Network (CDN) located in Milan. The study was further enriched by a measurement conducted within Milan itself, targeting a server situated in the same city to represent one of the intended destinations. The numerical values are reported in Table 2. This particular measurement was aimed at a local server in Milan to derive a real-world estimate of the potential end-to-end edge latency achievable within the current technological landscape. These metrics are expressed in milliseconds and include minimum (Min), average (Avg), maximum (Max) latencies, the standard deviation (Stddev) of latencies, TTL values, and the number of network hops required to reach each destination.

This analysis reveals significant variability in network performance, with latencies ranging from as low as 4 ms for local connections to over 300 ms for transcontinental communication to Tokyo. Notably, the standard deviation and the number of network hops provide insights into the stability and complexity of the paths traversed by packets across the network. For instance, the high standard deviation observed for the Tokyo connection suggests greater variability in latency, likely due to the longer and more complex route. Conversely, the relatively

Table 1
Ping and route statistics for an Italian fixed line from L'Aquila to various server locations.

Location	Min (ms)	Avg (ms)	Max (ms)	Stddev (ms)	TTL	#Hops
Milan	20.079	23.098	26.838	1.600	49	79
google.com	28.427	30.929	34.613	1.592	115	13
Frankfurt	32.873	34.707	49.142	1.896	35	93
London	43.837	45.281	47.242	0.824	44	84
Stockholm	54.934	56.649	61.209	0.888	44	84
Ireland	56.102	58.140	73.651	2.474	43	85
N. Virginia	111.657	113.236	118.181	0.987	36	92
Tokyo	248.467	256.050	300.447	12.533	29	99

Table 2
Ping and route statistics for an Italian fixed line from Milan to Milan server.

Location	Min (ms)	Avg (ms)	Max (ms)	Stddev (ms)	TTL	#Hops
Milan-to-Milan	4.013	6.172	20.624	2.492	51	13

lower standard deviation values for closer destinations indicate more stable latency performance.

These findings underscore the critical importance of geographic proximity, network infrastructure, and routing efficiency in determining real-world network latencies. Moreover, the dataset underscores a crucial consideration in network management: while the strategic placement of services in proximity to end-users is vital for reducing latency, network conditions are inherently time-variant. These conditions are significantly influenced by network congestion events, during which latency times and standard deviation values may see substantial increases. Remarkably, such dynamics can lead to scenarios where geographically closer locations experience higher latencies compared to more remote ones. This phenomenon highlights the complex interplay between geographic distance, network infrastructure, and temporal variations in network traffic, challenging the assumption that proximity invariably equates to lower latency. Understanding these nuances is critical for designing robust network systems capable of maintaining optimal performance and user experiences despite fluctuating network conditions.

4. Implementation on Kubernetes

Kubernetes[1] (abbr. *K8s*) is a powerful open-source container orchestration platform. Its primary role is to automate the deployment, scaling, and operation of containers across a cluster of host machines. While containers can help manage individual tasks, managing multiple containers and ensuring they interact seamlessly is challenging. K8s provides the technical tool to solve this kind of problems in an automated and seamless way. The K8s architecture comprises a master node (or *control plane*) and a number of worker nodes. The master node coordinates the cluster, while worker nodes host the containers inside logical groups of containers, known as *pods*. In K8s, a *pod* represents the smallest schedulable unit of workload.

Applications are usually deployed on K8s by means of higher-level abstractions, e.g., *Services*, *Deployements*, *Daemon Sets*, etc. Those abstractions avoid developers to manage complex and redundant set of pods manually and enable applications to benefit from additional features such as self-healing, replica management, assisted rollback, controlled rollout, etc.

In this section we provide an overview of the default Kubernetes scheduling strategy (which we adopt as benchmark) and show the details of our proposed latency-aware approach. Description of the fundamental elements of the proposed architecture is provided. Lastly, we analyze a step-by-step example to illustrate the functioning and cooperation of the involved elements.

[1] Kubernetes homepage: https://kubernetes.io.

4.1. Default Kubernetes scheduling

As previously mentioned, the scheduler is in charge to deploy pods in the K8s cluster. The K8s environment is normally equipped with a default scheduler, known as *kube-scheduler*, which takes scheduling decisions take into account the following factors:

- **Resource Requirements**: If a pod needs specific resources (CPU, memory), the scheduler ensures that the node where the pod places has enough of these resources
- **Node Conditions**: The scheduler ensures pods do not get scheduled on nodes that are in *NotReady* or *Unschedulable* state
- **Affinity and Anti-Affinity Rules**: These rules allow users to specify conditions regarding where pods should or should not be placed relative to others. For example, two pods that need frequent communication might have an affinity rule to be scheduled on the same node
- **Taints and Tolerations**: Nodes can *taint* themselves to repel certain pods unless the pod has a matching *toleration*
- **Node Selector**: It is a simple way to force a pod to run on a node using the labels mechanism provided by K8s

Kubernetes' scheduler, by considering the above factors, decides the optimal node for a pod to run on. Once a decision is made, the pod is bound to the chosen node, and the *kubelet* on that node is informed of the decision. The *kubelet* is an agent that runs on each node in a Kubernetes cluster and will oversee the lifecycle of the scheduled pods. In a healthy environment, there is no need to kill a running pod migrating it to another node of the cluster unless some resource limit is reached by the node and the *kubelet* is forced to kill that pod.

A representation of the K8s default scheduling process is shown in Fig. 3. Kubernetes relies heavily on its scheduler to ensure that pods are efficiently and correctly placed onto nodes within a cluster.

The scheduler's operation leverages on framework designed to assure extensibility and customization. This framework is structured around several stages, primarily: PreFilter, Filter, PostFilter, Score, NormalizeScore, Reserve, Permit, Bind, and PostBind. All those phases aim to let the developer guide the Scheduler's behavior. The main focus of the default scheduler is to take placement decision for pods in a fast and structured way. In a healthy cluster where each node has enough available resources, the default behavior is to randomly pick one node from the resultant set of nodes that fulfill all the requirements at each step. Fig. 3 shows an empty cluster at **Step 0**. The default scheduler is called to schedule a pod *P1*. The scheduler may pick each of the available nodes of the cluster, so resulting in one of the steps **Step 1a – Step 1d**. In the illustrated example, which shows a distributed cluster with varying latencies for each node, the default scheduler ignores the latencies when making its placement decision. **Step 1d** occurs with a probability of $1/k$, where k is the number of available nodes in the cluster.

A wider view is given by Fig. 4, where the whole scheduling process is analyzed from a software point of view. The process initiates when the Kubernetes scheduler requests the next unscheduled pod from the API server (Step 1). This pod is awaiting to be assigned to a node for execution. The API server returns an unscheduled pod object to the scheduler (Step 2), which triggers the scheduling algorithm (Step 3) to determine the best node for this pod based on various criteria such as resource requirements, node conditions, affinity and anti-affinity rules, taints and tolerations, and other factors that were previously described.

Once an appropriate node is selected, the scheduler communicates the chosen node updating the pod object's status to "scheduled" (Step 4), which is acknowledged by the API server (Step 5). The *kubelet* then polls the API server to get the scheduled pod (Step 6 and Step 7), proceeding to create the pod on the node (Step 8).

The pod's status is updated to "running" once it is successfully created and operational (Step 9) and the status is reported back to the API server (Step 10), which acknowledges back (Step 11). The API

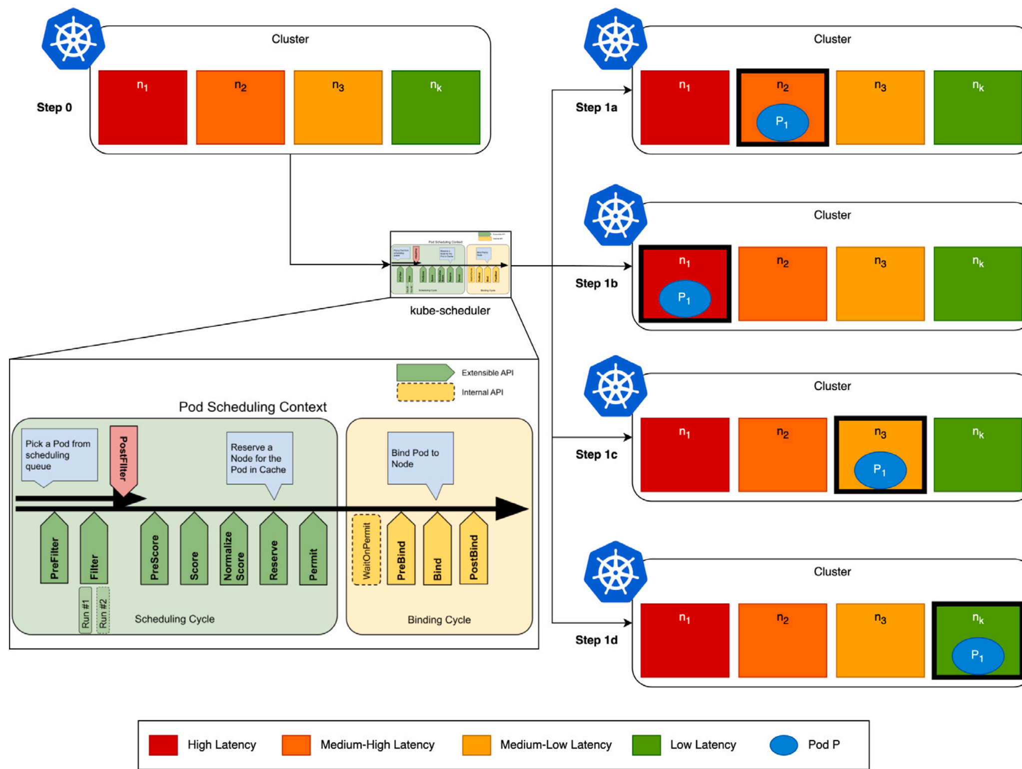


Fig. 3. The default Kubernetes Scheduler.
Source: Adapted from K8s scheduling framework [22].

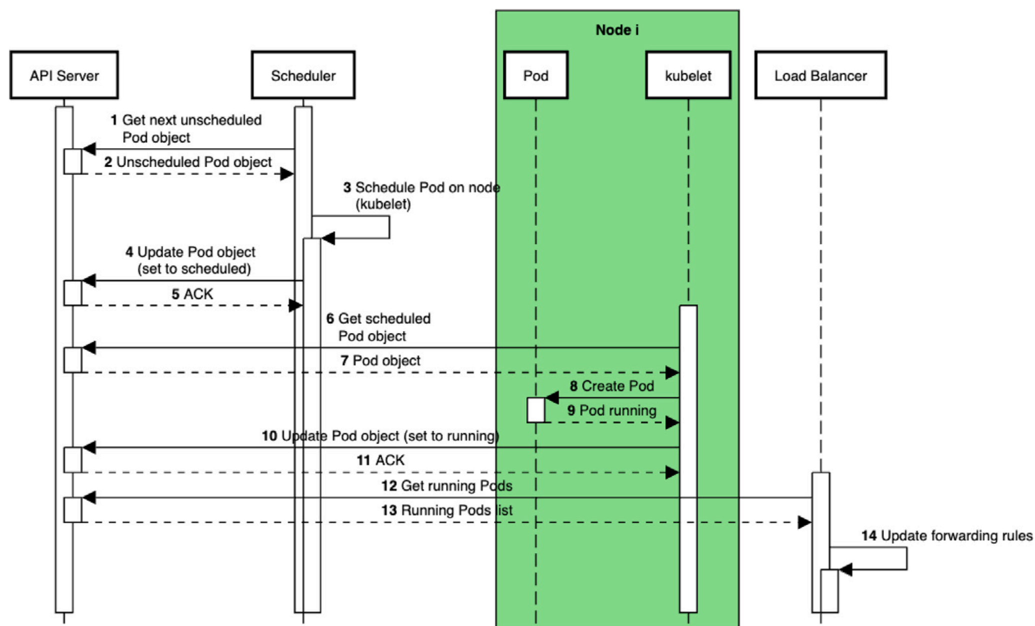


Fig. 4. Pod creation workflow with Load Balancer updates in a Kubernetes Cluster.

server maintains a list of all running pods thanks to the *etcd* component, which is continuously updated. The service object, created in advance in the deployment, constantly polls the API server to check for any state of the pods change within the cluster (Step 12 and Step 13). The final step in the scheduling process entails the updating of the Load Balancer's forwarding rules (Step 14). This critical action ensures that network traffic is correctly routed to the newly running pods, thereby sustaining the availability and performance of the application.

An analogous process occurs during the de-scheduling phase, as depicted in Fig. 5, where a "Delete Pod" request initiates the sequence. This request leads to an announcement to the Load Balancer, instructing it to cease forwarding requests to the pod slated for deletion. Subsequently, the pod is effectively removed from the node. This de-scheduling operation is crucial for maintaining the cluster's service reliability, ensuring that resources are efficiently allocated and that traffic is directed only to active pods capable of handling requests.

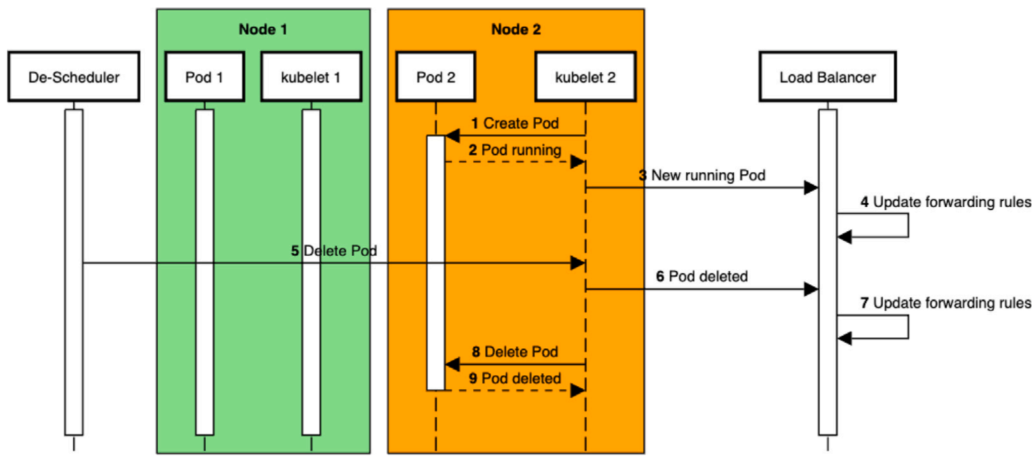


Fig. 5. Pod deletion workflow with Load Balancer updates in a Kubernetes Cluster.

4.2. Latency-aware Kubernetes scheduling strategy

The pod allocation performed by the Kubernetes scheduler is not always optimal from the point-of-view of the end-user latency. Once the application pods are allocated on the available cluster nodes, an user can experience high or low latency depending on which pod is providing response to the user requests.

In order to minimize (and stabilize) the latency perceived by the application users, we propose a mechanism that iteratively forces Kubernetes scheduler to allocate the pods on the nodes which offer the minimum measured (at runtime) E2E latency as experienced by the end-user. The proposed approach is summarized by the following steps:

1. Decide a number of *sentinel instances* S of the application. This is the number of instances (i.e., pods) that will be used to measure the application latency on the cluster nodes.
2. Allocate $R + S$ instances, where we refer to R as the number of replicas that are required from the developer. This guarantees that, at any moment, at least R replicas will be available to serve user requests. R can be 1 in case redundancy in not required by the developer. It is worth mentioning that when multiple instances of the same pod are deployed (whether they are sentinels or replicas), the load balancer within the K8s service of each deployment evenly forwards traffic to all healthy and running pods available.
3. Wait for users to provide feedback on their currently experienced latency and rank nodes according to the reported E2E latency.
4. Remove the pods from the S nodes with worst latency, and replace them with S new pods. Keep track of the removed nodes and force the Kubernetes scheduler to ignore them for new pod allocations. This step is repeated until all the nodes are either marked or running an application pod, i.e., the optimal (in terms of latency) pod allocation is achieved.

Our implementation of the system architecture is based on a setup that encompasses a distributed Kubernetes cluster composed by a collection of k worker nodes, denoted as $N = \{n_1, n_2, \dots, n_k\}$, under the supervision of a Control Plane, as illustrated in the diagram in Fig. 6. In the figure, for the sake of presentation, we color each node according to the latency between the user and the node.

The architectural components required for the implementation are listed below and explained in the following subsections:

1. the *Latency-Aware* Scheduler;
2. the *Latency-Aware* De-Scheduler;
3. the MQTT latency reporting channel and the MQTT broker.

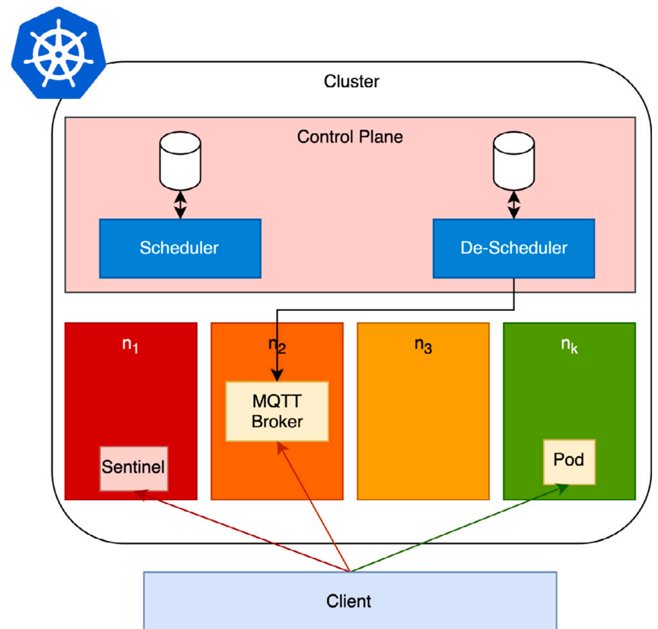


Fig. 6. System Architecture: a Kubernetes cluster with $k = 4$ nodes, a MQTT broker and a control plane featuring the Latency-Aware Scheduler and the Latency-Aware De-Scheduler. Nodes are colored according the supposed latency: red for high latency, orange for medium-high latency, yellow for medium-low latency and green for low latency.

4.3. Latency-aware scheduler

In the Kubernetes Scheduling Framework, as illustrated in Fig. 3, the orchestration of pods, is managed through a series of well-defined extension points across two main phases: the scheduling cycle and the binding cycle. The scheduling cycle is responsible for selecting a suitable node for the pod, while the binding cycle applies this decision to the cluster. These cycles constitute the scheduling context that ensures system resilience and efficient resource management, even under scenarios of substantial load.

The robustness of the system in managing multiple sentinels, even in high resource occupancy situations, stems from the framework's ability to prioritize workloads and optimize resource allocation dynamically. Sentinel pods, serving as replicas of the service pods, are scheduled based on the same criteria as regular pods, including resource requirements, node conditions, and various scheduling policies defined by the administrator.

During the scheduling cycle, various plugins work in concert to evaluate the feasibility of placing a pod on a node. For instance, Filter plugins screen nodes to ensure sufficient resources are available for the pod's demands. If a sentinel pod cannot be allocated due to resource constraints, it is placed in the internal unschedulable Pods list, preventing any negative impact on the system's performance. Furthermore, Reserve and Permit plugins work to prevent race conditions by temporarily reserving resources on nodes and ensuring that pods are only scheduled when the system can accommodate them without compromising the stability of other services.

When resource occupancy is high, the system's QueueingHint function plays a vital role in deciding whether sentinel pods can be requeued for scheduling. This ensures that only when there is a genuine opportunity for the sentinel pods to be scheduled without disrupting existing workloads, they are considered for placement. This mechanism is crucial in maintaining a balance between high availability and resource efficiency.

The *Latency-Aware Scheduler* is our custom Kubernetes scheduler that adds an additional scheduling policy when a pod needs to be scheduled for execution on a node of the cluster. In particular, following the proposed approach, the Latency-Aware Scheduler is set up to check available worker nodes and keep track of which node has a pod of a target application running, following the Algorithm 1. Upon a request to schedule a new pod of the target application, the Latency-Aware Scheduler forces the node selection to be restricted to nodes which have never host a pod for that application before, thus avoiding nodes that are (a) currently running an application pod or (b) ran an application pod recently.

This behavior can cause the Latency-Aware Scheduler to run out of unmarked nodes: to avoid this, the Latency-Aware Scheduler includes a *clean-up* mechanism that periodically unmarks all the nodes, so that they become again available to host an application pod. This ensures also that, in case the latency changes over time, the Latency-Aware scheduler can adapt automatically to find the new optimal pod allocation.

Finally, the Latency-Aware Scheduler make sure that, when a target application is set to require R working replicas, it instead schedules additional S replicas required by the proposed mechanism to measure the E2E latency (i.e., the *sentinels*). While this process is transparent to the application, the number S is configurable. This allows deployers to balance the trade-off between having additional workload due to the S extra replicas and the time required by the proposed approach to find the optimal allocation. We refer to this time as the *Convergence Time* T_C.

A representation of the Latency-Aware Scheduler is shown in Fig. 7.

Algorithm 1: Scheduling logic

Input: Application App
Input: Set of nodes in the cluster N
Input: Set of visited nodes V
Input: Set of untested nodes U
Output: The selected node $n_i \in N$

$U \leftarrow N - V$
while $U \neq \{\}$ **do**
    $n_i \leftarrow \text{random}(U)$
    $V \leftarrow V \cup \{n_i\}$
    scheduleProbe(n_i)
    $U \leftarrow N - V$
return n_i

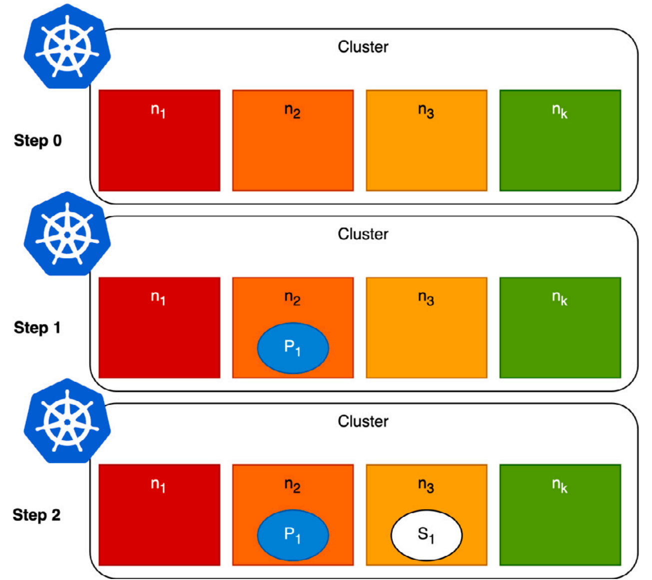


Fig. 7. Latency-Aware: scheduling process of pod P_1 with a *sentinel* replica S_1.

4.4. Latency-aware de-scheduler

In Kubernetes, pods are meant to be *unique* and *replaceable*, i.e., moving a pod from a node to another is not a direct operation, but rather a combination of (a) the deletion of the pod and (b) the creation of a *new* pod. The Latency-Aware Scheduler ensures that the latter operation (creation of the new pod) involves only a restricted set of nodes. On the other hand, the proposed approach requires a controlled (i.e., latency-based) pod deletion.

This task is performed by our *Latency-Aware De-Scheduler*. This component is responsible for listening to incoming latency measurements, deduce the subject (i.e., pod and node names) of the measurement and keep an updated ranking of nodes according to their latency. Finally, when the Latency-Aware De-Scheduler has received measurements for all the cluster nodes currently running application pods, it triggers the deletion of the pods running on the S nodes with worst latency.

Algorithm 2: De-scheduling logic

Input: Application App
Input: De-Scheduling Threshold T
Input: Latency Measurements LM

while *True* **do**
    $M \leftarrow$ number of measured pods for App
    if $M \geq T$ **then**
        `// Retrieve the pod with maximum RTT`
        $p_k \leftarrow \max_{RTT}(LM, App)$
        `// Deschedule` p_k `and clear its entry in` LM
        Deschedule(p_k)
        $LM[App][p_k] \leftarrow$ None

An example of the Latency-Aware De-Scheduler is depicted in Fig. 8, and the underlying algorithm is presented in Algorithm 2. In the figure, nodes are colored according to their latencies. At the initial phase (**Step 0**), we have a configuration with $R = 1, S = 1$ (one sentinel only), node n_2 running pod P_1, node n_3 running a recently launched sentinel pod S_1 and nodes n_1, n_4 have no pods running and are available for pod allocation. After Step 0, the Latency-Aware De-Scheduler receives the measurement about S_1, that provides a latency measurement for node n_3.

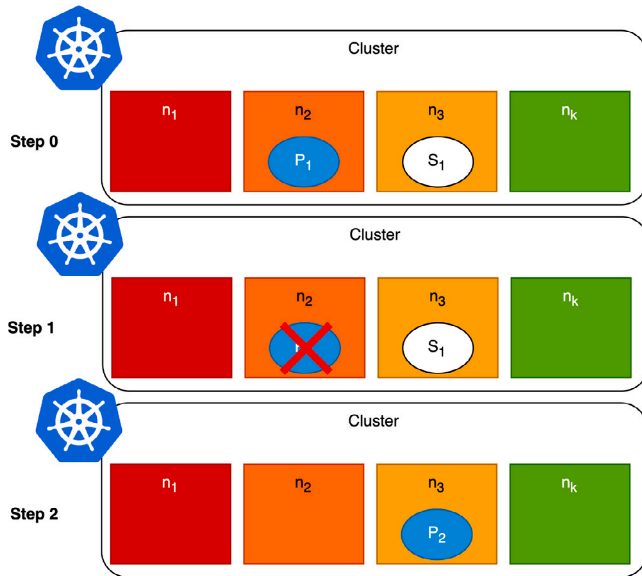


Fig. 8. Latency-Aware De-Scheduler: the pod running on the higher-latency node is removed and the sentinel become a proper application pod - $S_1 \rightarrow P_2$.

At **Step 1**, the Latency-Aware De-Scheduler has ranked all the nodes running pods according to their latency and triggers the removal of $S = 1$ pods, i.e., the pod P_1 running on node n_2 (the one with the worst latency). The removal of P_1 causes the sentinel pod S_1 to cease being a sentinel and to become a proper application pod. To enforce this, we changed the name of S_1 into P_2.

Finally, at **Step 2**, P_1 has been successfully deleted and the cluster now runs the application pod in a node offering better latency with respect to the starting allocation.

4.5. Latency reporting channel

Since the final objective of the proposed approach is to reduce the perceived E2E latency as perceived *by the users*, it is crucial to establish a communication channel that they can provide feedback on the latency they are currently experiencing. This means that we require users to measure and communicate the latency measurements to the control plane, i.e., the Latency-Aware De-Scheduler. The decision on which kind of communication channel, the protocol and the format to be used to send latency measurements is left to the deployers.

We realized the communication among the clients and the Latency-Aware Descheduler by using the MQTT publisher/subscriber protocol and a MQTT broker. During application execution, the latency measurements are *published* by the application clients using an application-specific MQTT topic (e.g., *latency-reporting-app1*). At the same time, the Latency-Aware De-Scheduler, already *subscribed* to the application topics, listen to incoming messages, receive the latency measurements published and perform node ranking and deletion according the description in Section 4.4.

The format adopted to send a latency measurement is a JSON data structure (shown in Fig. 9) with the application name, a timestamp and the measured round trip time latency. The Latency-Aware De-Scheduler, upon the reception of a message, enrich the received latency measurement with the responding pod ID and the hosting node ID. This structure allows us to store information for multiple applications and to handle them separately. This way the Latency-Aware De-Scheduler can navigate through the stored latency measurements of running pods along the information about the nodes hosting them. Additionally, a timestamp is for checking the *freshness* of the measurements and, if required, add an aging mechanism to ensure that the measurements are

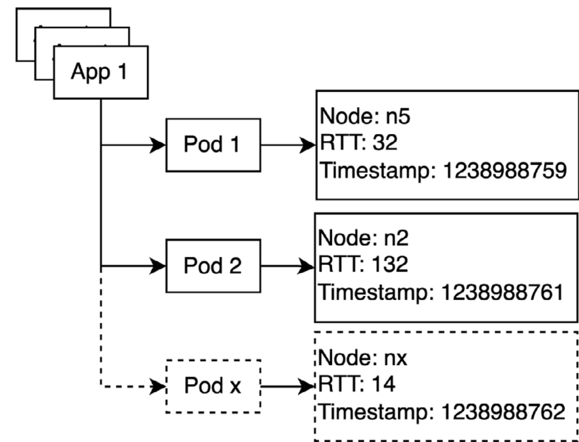


Fig. 9. Latency Measurements data structure used to collect and store latency information.

always reflecting the most recent node latencies. Finally, this structure ensures a fast removal of the latency measurements when a pod is removed by the Latency-Aware De-Scheduler.

4.6. Combined operation

A combined example of both the Latency-Aware Scheduler and the Latency-Aware De-Scheduler is shown in Fig. 10. In the initial status (**Step 0**) we have a 4-nodes Kubernetes cluster (with nodes colored according the supposed latency) in which an application is being deployed and configured to have $R = 1$ (one replica required), $S = 1$ (one sentinel pod). Then, the following steps take place:

Step 1 The first application sentinel pod is allocated by the Latency-Aware Scheduler. Since there is no other running pod, the sentinel becomes directly a proper application pod, P_1. The current allocation is given by this initial condition, with node n_2 being the allocation for the $R = 1$ application pod.

Step 2 Sentinel pod S_1 is allocated onto node n_3. After receiving a latency measurement from both pod P_1 and S_1 (i.e., nodes n_2 and n_3), the Latency-Aware De-Scheduler triggers the deletion of the pod running on the (only, since $S = 1$) node with worst latency, i.e., P_1 on node n_2.

Step 3 The sentinel pod S_1 becomes the proper application pod P_2.

Step 4 A new sentinel pod S_2 is allocated by the Latency-Aware Scheduler onto node n_1. The Latency-Aware De-Scheduler receives the latency measurement from the user related to pod S_2 and node n_1. Since n_1 has the worst latency, S_2 is deleted.

Step 5 Pod S_2 has been deleted and node n_3 remains the best current allocation.

Step 6 A new sentinel pod S_3 is allocated by the Latency-Aware Scheduler onto node n_4. The Latency-Aware De-Scheduler receives the latency measurement from the user related to pod S_3 and node n_4. Since n_3 has the worst latency, P_2 is deleted.

Step 7 The sentinel pod S_3 becomes a proper application pod P_3 and, since all the nodes in the cluster have been marked, the allocation of the $R = 1$ pods on the cluster is now the optimal allocation.

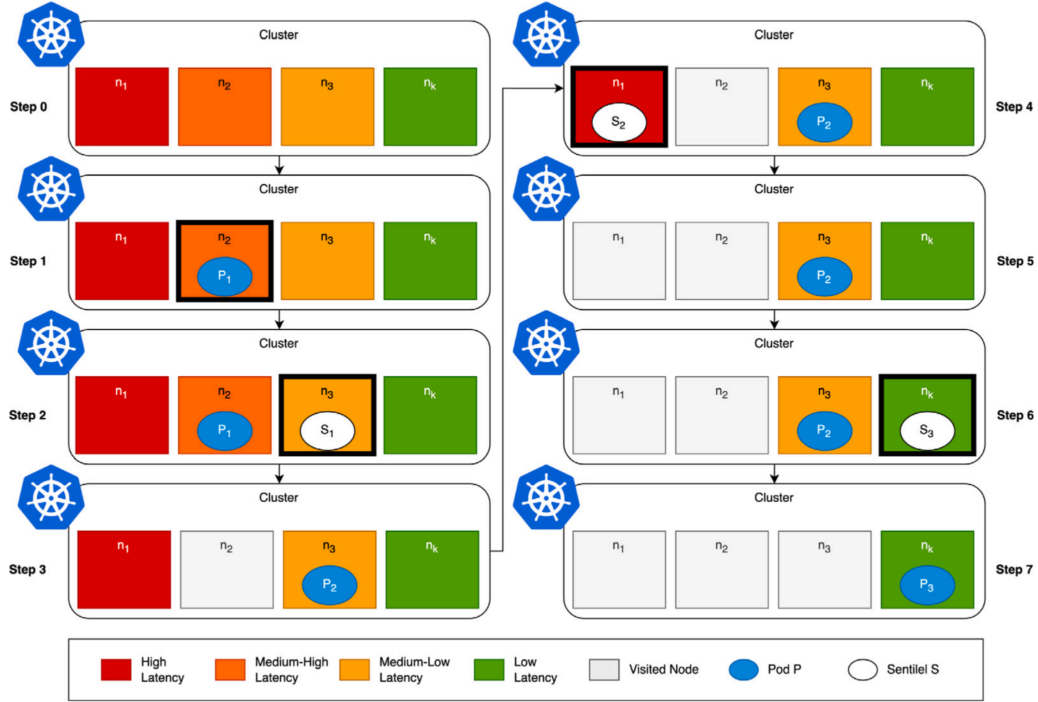


Fig. 10. Operation of the combined Latency-Aware Scheduler and De-Scheduler operations: The application pod P_1 is iteratively "moved" from node n_2 (medium-high perceived latency) to node n_4 (low latency).

From the perspective of the Latency-Aware scheduler, the sentinel is a role that is assigned to a replica in order to measure the latency experienced by users on a node. In case of descheduling, this role is passed to another replica automatically, with no downtime or service interrupts since, during this operation, we ensure that at least R replicas are always simultaneously available, triggering the replica removal only when the number of active replicas is greater than R. Nonetheless, Kubernetes ensures that pod/replica removal is handled gracefully with no or minimal downtime (i.e., high-availability property). This allows our Latency-Aware scheduling approach to work flawlessly even during scheduling and descheduling operations.

5. Results

In order to validate the implementation of the proposed approach, we set up a Kubernetes cluster on a commercial cloud platform with a set $\mathcal{N}$ of $k = 10$ nodes. We suppose that such nodes are geographically distributed, so that application instances running on them suffer from a different E2E latency (as perceived by the end-users) depending on which node physically hosts the responding application instance. Since it is not generally possible to select and position the cluster nodes to have clear latency differences when using a cloud provider, we decided to enforce this assumption by hard-coding a known set of simulated delays to applications depending on the node on which they are running. This enabled us to quickly validate the approach in different latency scenarios without changing the setup or the cloud provider configuration.

5.1. Considerations on the convergence time

A critical aspect of the proposed approach is the time needed to *migrate* the application pods to the nodes that offer the least E2E latency for users. This duration, denoted as T_C is a pivotal performance metric in our evaluation. Specifically, T_C represents the time interval required to reach the optimal scheduling decision that minimizes the perceived E2E latency at the application layer for the end-user. T_C is influenced by several factors. In particular, accounting for the deployment of multiple sentinels, we extend our previous model in [8]. Thus, T_C, which we refer to as the convergence time, can be modeled as:

$$T_C = \frac{N}{S} \times (T_O + T_S + T_D + T_{COM}), \quad (1)$$

where:

- N denotes the number of nodes present in the Kubernetes cluster;
- S is the number of sentinels
- T_O is the required Observation Time to consistently measure the latency from a responding service;
- T_S refers to the Scheduling Time, which our Latency Aware Scheduler requires to allocate the pod via the Kubernetes REST API;
- T_D stands for the De-Scheduling Time, necessary for our De-Scheduler to terminate and remove the chosen pod using the Kubernetes REST API;
- T_{COM} is the Communication Time needed to relay messages to the de-scheduler through the MQTT Broker.

We empirically assessed the average time taken by Kubernetes APIs for scheduling and de-scheduling actions. Based on our evaluations, we set $T_S = 0.7$ s and $T_D = 0.7$ s. Given that the communication time T_{COM} is in the order of milliseconds, we deem the impact of T_{COM} to be negligible.

Given that the values of T_S and T_D depend on the Kubernetes cluster configuration, we assess the convergence time by altering the two remaining variables, namely T_O and N, as illustrated in Fig. 11. The convergence time T_C increases linearly with T_O and the slope becomes steeper as N increases. As shown in Figs. 11(b) through 11(d), the introduction of more sentinels results in a linear decrease of T_C. The determination of the optimal value for T_O is a trade-off between the system's responsiveness and the precision of the measurements. It is worth mentioning that here we assumed only one sentinel alongside one replica of the pod ($R = 1, S = 1$) in the iterative discovery process. We also introduce T_T, denoted as the *Trigger Timer*, which represents a predefined time interval established by the developer to orchestrate the subsequent execution of our algorithm. It is imperative that this timer

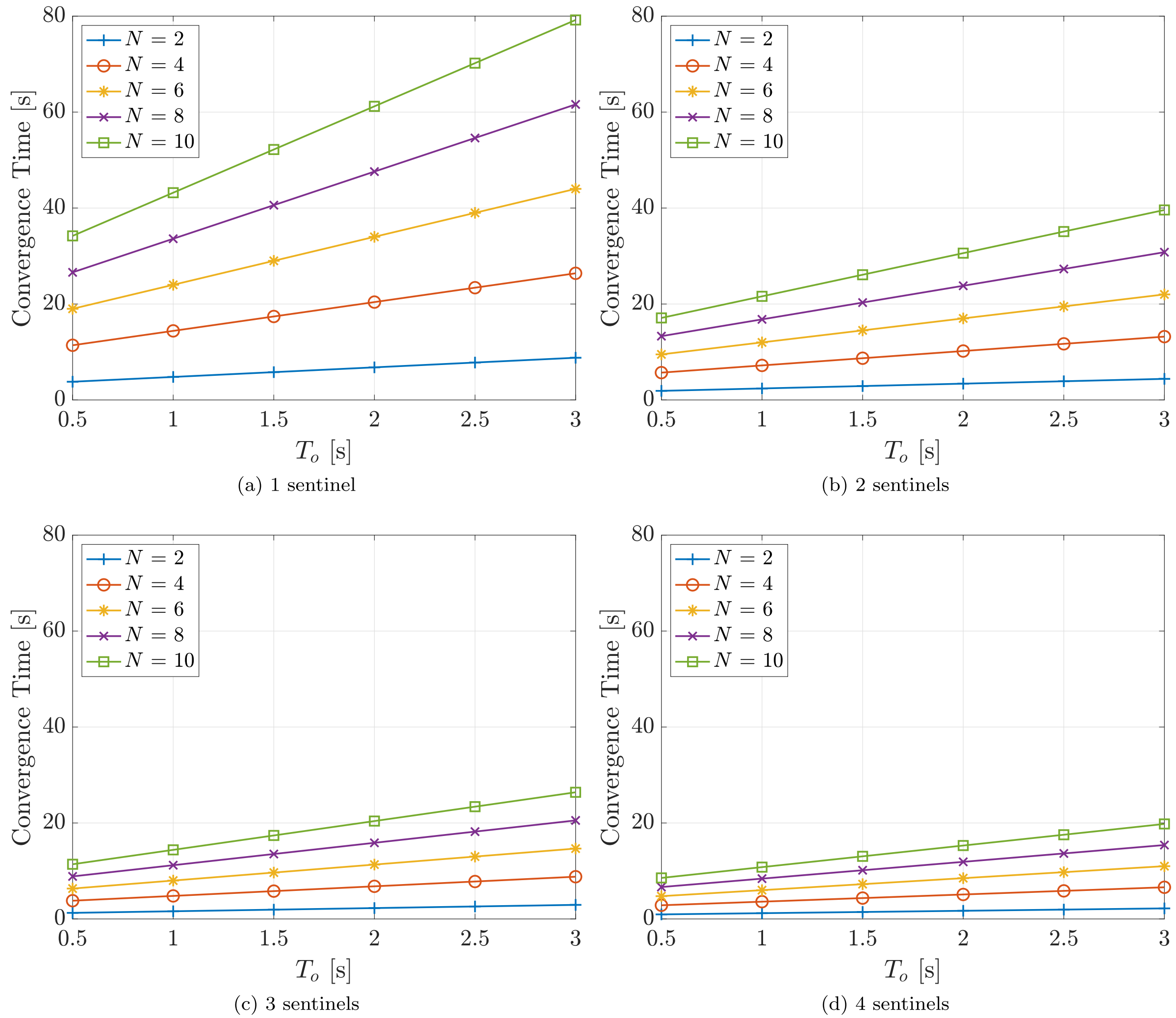


Fig. 11. Theoretical convergence time for different number of sentinels.

is set to a duration exceeding T_C, as configuring it to a lesser interval means to restart the execution before the end. Upon the expiration of T_T, our mechanism embarks on a new cycle of execution. It is worth noting that at this point the probability to find a node with lower latency is reduced as the node selected in the previous run was the best available option. In the following section, we will analyze the impact of the number of sentinels on the convergence time.

5.2. Experimental setup

To validate our Service Placement approach, it was essential to implement it within a real Kubernetes cluster. We used 10 machines from Amazon Web Services (AWS) to establish the cluster, with an additional machine allocated for deploying our client application. All the Virtual Machine (VM)s were deployed within the same Availability Zone (AZ), ensuring that the nodes were co-located within a singular AWS data center. This choice was instrumental in minimizing potential network interferences. Given that the latency between nodes remained significantly below 1 ms, we could effectively neglect this latency in our analysis. For our experiments, we provisioned AWS *t3.medium* instances, each equipped with 2Virtual Central Processing Unit (vCPU) and 4GiB of Random Access Memory (RAM). These VMs run Ubuntu Server 22.04.03 LTS and are equipped with 8GiB of AWS Elastic Block Store (EBS) gp2 Solid State Drive (SSD) for storage purposes, and controlled network impairments on latency are applied accordingly to Table 3.

Table 3
Simulated latencies for network experimentation in a distributed cluster.

Node	Latency [ms]	Equivalent location
n_1	0	Edge Computing
n_2	10	Metro
n_3	25	National
n_4	50	Continental
n_5	75	Continental
n_6	100	Intercontinental
n_7	150	Intercontinental
n_8	200	Intercontinental
n_9	300	Global
n_{10}	500	Extreme Distance
client	0	–

On the top of the Ubuntu Server, we installed Microk8s v1.27.5 and created a cluster composed by the 10 nodes. One more VM has been deployed in AWS to run the client-side part of a sample distributed web REST-based application. The client VM has the same hardware resources allocated to any other node of the cluster.

In this section, we evaluate the performance of the introduced Service Latency-Aware Placement approach. Being the pioneers in attempting to minimize End-to-End latency by dynamically relocating the service to the node closest in terms of latency to the end-user, we need to demonstrate first that our algorithms can chose the best node in a real environment. For this purpose, we varied the number

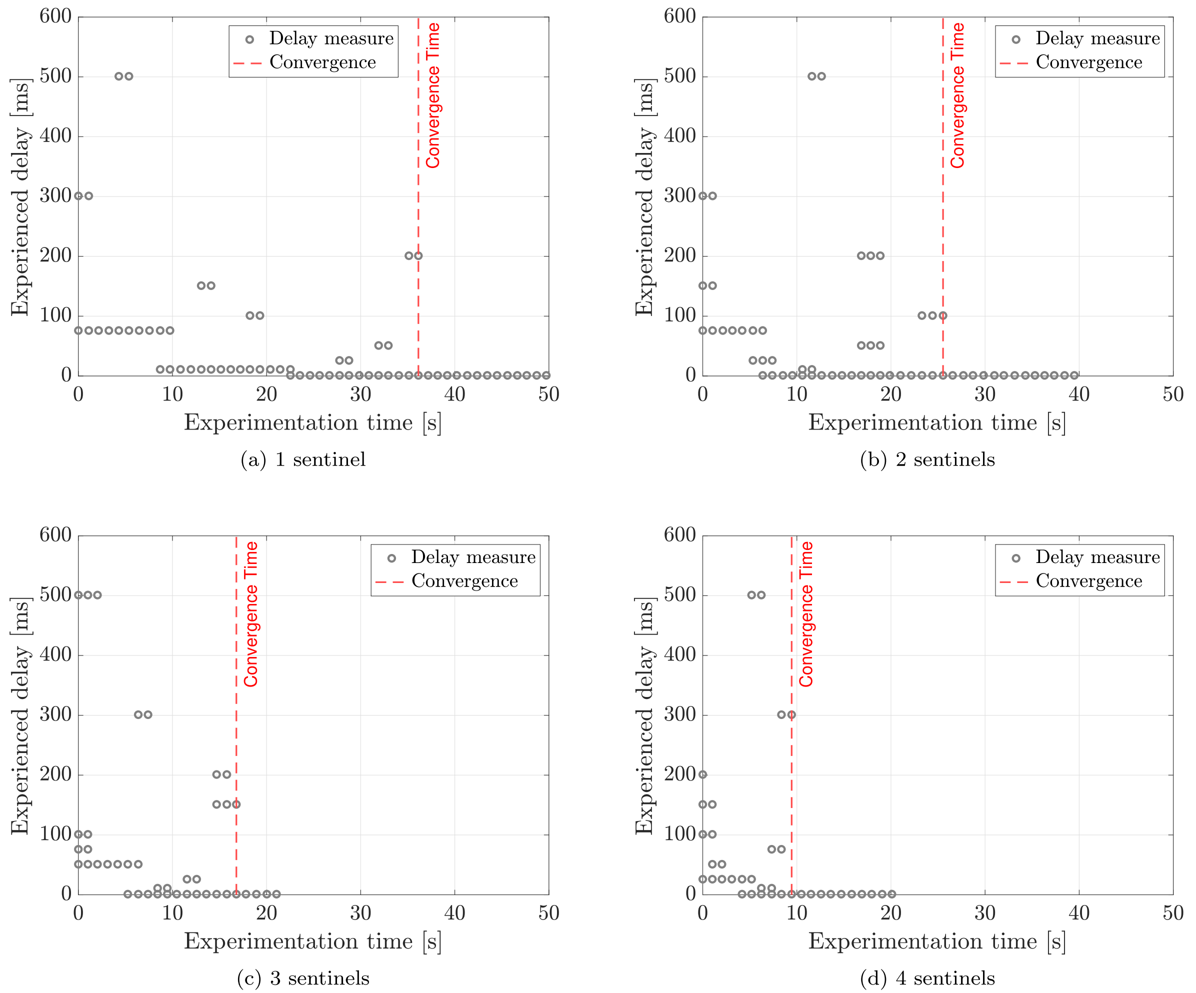


Fig. 12. Convergence examples for different number of sentinels.

of sentinels to demonstrate the procedure's accuracy, and subsequently set up a series of experiments to study the *convergence time* by altering the *sentinel* number.

The main aspect we want to examine is whether increasing the number of sentinels S leads to a reduction in the convergence time. To achieve this, we gathered metrics from various runs in our testbed, altering the number of sentinels involved. For clarity, we only plot single runs in Fig. 12 to illustrate the *convergence time* during a single iteration of the experiment.

Fig. 12(a) illustrates an execution of the experiment using one *sentinel*. The results indicate that the RTT measured fluctuates throughout the experiment, aligning closely with the anticipated values based on the network latencies characterizing the nodes of the cluster. The plot displays the time interval for the experiment on the x-axis, spanning up to 50 s. On the y-axis, the observed RTT is plotted. Multiple RTT values at the same time index indicate that at that specific moment, a sentinel was running concurrently with the pod providing the service.

The K8s load balancer distributes the traffic between the two replicas of the service (one pod and one sentinel). It is worth noting that the values on the y-axis represent average values calculated over $T_O = 1$ s and reported with a periodicity T_O. A key observation is that after a certain duration, the latency stabilizes at the lowest achievable value and remains consistent at that level. This observation can be attributed to the tested scenario where user mobility is not accounted for, and the network conditions are assumed to be relatively stable. In a real-world environment, these two assumptions may not always hold true. Network latency is time-variant and influenced by parameters and conditions beyond the scope of this study. However, this does not invalidate our findings because the *Latency-Aware Placement* process can always be restarted to determine if network conditions have changed and if another node now offers the lowest possible latency. The *convergence time* is crucial in this context and must be minimized to enable a more frequent execution of the *Latency-Aware Placement* process. Fig. 12(a) also displays a red vertical line that marks the *convergence time* for that particular experimental run.

In our system model and experimental setup, we have considered the potential impact of environmental noise, such as Additive White Gaussian Noise (AWGN), on the network's latency dynamics. Our observations indicate that the inclusion of AWGN does not significantly alter the process of selecting the optimal node for service placement. This resilience is largely due to the method by which client devices measure latency. Latency values are measured over a T_O interval and reported as average values. This averaging process effectively mitigates the impact of transient noise spikes, ensuring that node selection remains stable despite underlying variability in network conditions.

To gain a deeper understanding of the *convergence time*, we repeated the experiment multiple times to determine the mean, minimum, and maximum values for this particular scenario. The mean value for *convergence time* and one *sentinel* is 40 s, as shown in Fig. 13.

The repetition of the same experiment with two *sentinels* is shown in Fig. 12(b). In this instance, as many as three delay measurements ($R = 1, S = 2$) can occur at a specific time. It is evident that the *convergence time* is reduced and consequently the experimental duration for this run is slightly shorter than that observed in Fig. 12(a). The instances where fewer than three sentinels are displayed at a specific

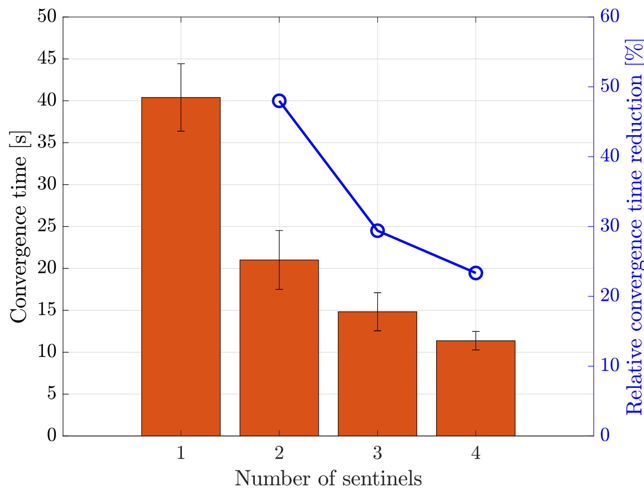


Fig. 13. Average convergence time for different number of replicas and relative reduction in a cluster composed by 10 nodes.

time can be attributed to the inherent nature of K8s architecture. In fact, K8s does not ensure that multiple replicas of a pod, scheduled simultaneously, will commence operations concurrently.

Subsequently, we conducted the same experiment using three *sentinels*, with the results illustrated in Fig. 12(c). We observed a further reduction in the *convergence time*, though the decrease was slightly less pronounced than in previous trials. Following the *convergence time*, the minimal latency persisted until the conclusion of the experiment.

We conducted our final experiment using four *sentinels*, as illustrated in Fig. 12(d). We observed a further reduction in the *convergence time*, though the decrease was modest.

These values are insufficient for a comprehensive statistical analysis, so we replicated each of the four experiments 100 times to gather more precise data. Fig. 13 presents the consolidated results of our experiments, showing the average values and variance for each experimental set. With a single sentinel, we observe the longest *convergence time* and the greatest variability in the measurement. The mean *convergence time* in this case is around 40 s, which is in line with the model in Eq. (1). By increasing the number of sentinels, we not only reduce the *convergence time* but also decrease the measurement variance. This arises because our scheduler is designed to schedule containers in groups, thereby requiring fewer steps to explore the entire cluster. Results in Fig. 13 are in line with the theoretical behaviors depicted in Fig. 11. This validates the proposed theoretical model.

Fig. 13 also illustrates the relative reduction in *convergence time* (expressed as a percentage) achieved by increasing the number of sentinels. In our experiments, transitioning from a single *sentinel* to two led to a 50% reduction in *convergence time* while an increase from three to four *sentinel* resulted in a reduction of less than 25%.

Apart of the *convergence time*, a final consideration concerns the amount and the impact of the de-scheduling and re-scheduling operations required to reach the optimal allocation. From the users' point of view, both the impact of de-scheduling and re-scheduling operations and their number is transparent, since our approach ensures a minimum number of available replicas (R) while Kubernetes itself ensures graceful de-scheduling/scheduling operations with minimal service down-time. At system level, while it is desirable to minimize the number of operations required to reach the optimal allocation, this is not feasible when no additional information about the network state are available. In other terms, as long as the end-user latency to every node at every point in time is not estimable a-priori, our iterative approach is the best solution to ensure the achievement of an optimal allocation. In this regards, we observed that the number of de-scheduling and re-scheduling operations decreases significantly after the optimal allocation has been achieved. During our experiments we noted that, after an optimal allocation has been achieved, non-extreme latency variations cause the future optimal allocation to be almost always within a very limited number of re-scheduling operations. Nevertheless, we are planning on adopting machine learning techniques to obtain a partial estimation of end-user latency and to further reduce the number of operations and the convergence time to reach the optimal allocation.

5.3. Scalability and performance analysis of sentinel deployment

In the domains of edge and distributed computing, the efficiency of resource allocation is crucial for assuring scalability and optimal performance under diverse workloads. The architecture of our proposed Kubernetes-based Low-latency Scheduler allows for an analytical assessment of the Overload O associated with the deployment of a specified number of sentinels, represented as S, together with R replicas, required by the service.

Eq. (2) defines the Offload O, calculated as the percentage increase in the total number of deployed entities within a Kubernetes environment, relative to the original number of pod replicas. Specifically, R represents the count of pod replicas. On the other hand, S denotes the number of sentinels. The equation aims to quantify the overhead introduced by the deployment of sentinels in addition to the original pod replicas, thereby offering a metric to evaluate the scalability and resource utilization within the orchestrated environment. By multiplying the ratio of the sum of replicas and sentinels to the number of replicas ($\frac{R+S}{R}$) by 100, the equation outputs O, the percentage that reflects the offload, thereby providing insights into the balance between enhanced user experience through reduced latency and the resource implications of such an optimization.

$$O = \left(\frac{R+S}{R}\right) \times 100, \tag{2}$$

This quantitative metric permits a rigorous exploration of the sentinel pods' proportional impact on the system. An augmented S to R ratio reflects a strategic focus aimed at reducing service latency and minimizing the system's convergence time (T_C), albeit at the expense of elevated resource allocation during the duration of T_C. Conversely, a diminished ratio illustrates a strategy of conservative resource deployment, potentially increasing T_C but enhancing resource efficiency overall. Notably, the cost escalates linearly with R, while T_C decreases in a non linear manner, as evidenced in Fig. 13. This trend is reported in Fig. 14 for clarity.

The resolution of this trade-off is at the discretion of the developer, who is empowered to set these parameters, thus enabling the selection of an optimal strategy that aligns with the specific demands of each application and conforms to overarching business objectives.

However, it is essential to consider that relative to the execution time of a service, even substantial increases, such as 1600%, constitute manageable spikes for the cluster and remain present only for a brief period of time, denoted as T_C. If minimizing T_T is not a necessity, then the cost associated with these resources becomes negligible in scenarios such as emergencies and latency-sensitive applications.

6. Conclusions and future works

The microservices architectural paradigm has significantly transformed contemporary software development practices by splitting complexity by promoting modularity. Building upon this foundation, cloud-native architectures have been devised to exploit enhanced scalability, availability, and resilience. Service placement becomes more relevant as those architecture goes close to the edge, in support of real time or near real time applications. Current literature focuses on reducing the inter-application latencies, minimizing the communication time within the cluster through co-location of services that need to interact often. Our proposed approach explores a different dimension of this problem, i.e., the latency measured from the service location to

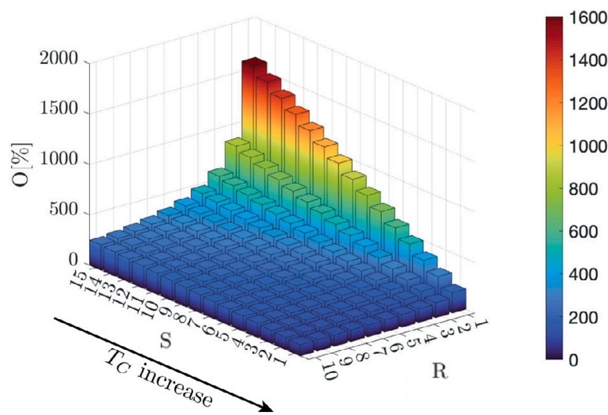


Fig. 14. Pod deletion workflow with Load Balancer updates in a Kubernetes Cluster.

the end-user in a distributed cluster. In this context, our work presents a robust approach to dynamically migrate a service among the nodes without disrupting the user-level service, aiming to place it to the node exhibiting the minimal latency within the cluster, by means of a novel "sentinel-based" mechanism.

In our research, we have demonstrated the feasibility of our approach by designing and implementing a custom scheduler in a K8s cluster. Results demonstrate that the convergence to the optimal placement is achieved within a finite time. We provide a seamless migration from the current node to a more performant one in terms of latency measured by the end-user. A relationship between the number of sentinels and the convergence time has been established for our system. Our analysis indicates that utilizing two sentinels reduces the convergence time by 50% compared to the case with only one sentinel. However, this reduction in time is accompanied by increased costs, directly proportional to the number of deployed sentinels. Thus, the decision to scale the number of sentinels while balancing costs is up to the developer, based on the specific application requirements and the target trade-off between costs and benefits of reaching sooner the optimal placement.

Future works will integrate Service Level Agreement (SLA) into the scheduler to further decrease convergence time and minimize the energy consumption to enhance the overall system energy sustainability.

CRediT authorship contribution statement

Carlo Centofanti: Conceptualization, Data curation, Investigation, Methodology, Project administration, Software, Validation, Writing – original draft, Writing – review & editing. **Walter Tiberti:** Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Andrea Marotta:** Conceptualization, Formal analysis, Funding acquisition, Methodology, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Fabio Graziosi:** Funding acquisition, Supervision, Writing – review & editing. **Dajana Cassioli:** Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Dajana Cassioli reports financial support was provided by European Commission. Fabio Graziosi reports financial support was provided by Ministero dell Universit e della Ricerca. Andrea Marota reports financial support was provided by Italian Government. Carlo Centofant reports financial support was provided by Italian Government. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work has been partially supported by the European Union through H2020-MSCA-RISE OPTIMIST project (GA: 872866), SNS Joint Undertaken SEASON project (GA: 101096120), NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem (ECS00000041 - VITALITY - CUP E13C22001060006), and partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART" - CUP E83C22004640001).

Disclosures

During the preparation of this work the authors used ChatGPT in order to assist with general language refinement. After using this tool, the authors reviewed and edited the wording as needed and take full responsibility for the content of the publication.

References

[1] Cisco, Cisco Annual Internet Report (2018–2023) White Paper, vol. 10, (no. 1) Cisco, San Jose, CA, USA, 2020, pp. 1–35.

[2] T.P. Sadatacharapandi, S. Padmavathi, Survey on service placement, provisioning, and composition for fog-based IoT systems, Int. J. Cloud Appl. Comput. (2022) http://dx.doi.org/10.4018/ijcac.305212.

[3] A. Hannousse, S. Yahiouche, Securing microservices and microservice architectures: A systematic mapping study, Comp. Sci. Rev. 41 (2021) 100415.

[4] I.K. Aksakalli, T. Çelik, A.B. Can, B. Tekinerdoğan, Deployment and communication patterns in microservice architectures: A systematic literature review, J. Syst. Softw. 180 (2021) 111014.

[5] N. Kratzke, P.-C. Quint, Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study, J. Syst. Softw. 126 (2017) 1–16.

[6] J. Alonso, L. Orue-Echevarria, V. Casola, A.I. Torre, M. Huarte, E. Osaba, J.L. Lobo, Understanding the challenges and novel architectural models of multi-cloud native applications–a systematic literature review, J. Cloud Comput. 12 (1) (2023) 1–34.

[7] X. Zhao, Y. Shi, S. Chen, MAESP: Mobility aware edge service placement in mobile edge networks, Comput. Netw. 182 (2020) 107435.

[8] C. Centofanti, W. Tiberti, A. Marotta, F. Graziosi, D. Cassioli, Latency-aware kubernetes scheduling for microservices orchestration at the edge, in: 2023 IEEE 9th International Conference on Network Softwarization, NetSoft, 2023, pp. 426–431, http://dx.doi.org/10.1109/NetSoft57336.2023.10175431.

[9] F. Faticanti, D. Santoro, S. Cretti, D. Siracusa, An application of kubernetes cluster federation in fog computing, in: Conference on Innovation in Clouds, Internet and Networks, 2021, http://dx.doi.org/10.1109/icin51074.2021.9385548.

[10] K. Gasmi, S. Dilek, S. Tosun, S. Ozdemir, A survey on computation offloading and service placement in fog computing-based IoT, J. Supercomput. (2021) http://dx.doi.org/10.1007/s11227-021-03941-y.

[11] D. Kumar, G. Baranwal, Y. Shankar, D.P. Vidyarthi, A survey on nature-inspired techniques for computation offloading and service placement in emerging edge technologies, World Wide Web (2022) http://dx.doi.org/10.1007/s11280-022-01053-y.

[12] M.C. Carrión, Kubernetes scheduling: Taxonomy, ongoing issues and challenges, ACM Comput. Surv. (2022) http://dx.doi.org/10.1145/3539606.

[13] Z. Weiguo, M. Xi-lin, Z. Jin-zhong, Research on kubernetes' resource scheduling scheme, in: International Conference on Communication and Network Security, 2018, http://dx.doi.org/10.1145/3290480.3290507.

[14] O.-M. Ungureanu, C. Vladeanu, C. Vlădeanu, R.E. Kooij, Kubernetes cluster optimization using hybrid shared-state scheduling framework, in: International Conference on Future Networks and Distributed Systems, 2019, http://dx.doi.org/10.1145/3341325.3341992.

[15] M.A. Tamiru, G. Pierre, J. Tordsson, E. Elmroth, Mck8s: An orchestration platform for geo-distributed multi-cluster environments, in: International Conference on Computer Communications and Networks, 2021, http://dx.doi.org/10.1109/icccn52240.2021.9522318.

[16] J. Santos, T. Wauters, B. Volckaert, F.D. Turck, Towards network-aware resource provisioning in kubernetes for fog computing applications, in: IEEE Conference on Network Softwarization, 2019, http://dx.doi.org/10.1109/netsoft.2019.8806671.

[17] L. Toka, Ultra-reliable and low-latency computing in the edge with kubernetes, J. Grid Comput. 19 (3) (2021) 31.

[18] Ł. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, M. Hong, Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh, in: IEEE INFOCOM 2021-IEEE Conference on Computer Communications, IEEE, 2021, pp. 1–9.

[19] A.C. Caminero, R. Muñoz-Mansilla, Quality of service provision in fog computing: Network-aware scheduling of containers, Sensors 21 (12) (2021) 3978.

[20] Á. Leiter, P.B. osy, M. Kis, L. Bokor, Performance costs for IPv6-based mobility management on the top of kubernetes, in: IEEE Conference on Network Softwarization, 2023, http://dx.doi.org/10.1109/netsoft57336.2023.10175456.

[21] A.N. Toosi, R.N. Calheiros, R. Buyya, Interconnected cloud computing environments: Challenges, taxonomy, and survey, ACM Comput. Surv. (2014) http://dx.doi.org/10.1145/2593512.

[22] Kubernetes, Scheduling framework, Kubernetes, 2023, https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/. (Accessed 9 October 2023).

Dr. Carlo Centofanti received his Ph.D. in Information and Communication Technology from the Department of Information Engineering, Computer Science, and Mathematics at the University of L'Aquila, Italy. He earned his M.Sc. degree in Computer Science Engineering and his Bachelor's degree in ICT Engineering from the same university. His research areas include Edge Computing, Software Defined Networking, and Network Slicing. He has actively contributed to ECSEL-RIA AfarCloud Project (GA: 783221), and is actively contributing to MSCA-RISE OPTIMIST Project (GA: 872866), and SNS-SEASON Project (GA: 101096120).

Walter Tiberti is an Assistant Professor at the Department of Information Engineering, Computer Science and Mathematics of the University of L'Aquila, where he received his B.Sc, M.Sc. degree in Computer Engineering and his Ph.D. in Information and Communications Technology working on embedded systems security, in particular, wireless sensor network security. He was a visiting researcher at the Research Centre in Real-Time and Embedded Computing Systems (CISTER) of Porto (Portugal). He is part of the technical committee member of IEEE conferences and he isinvolved in italian initiatives to promote nationwide cyber-security education and training.His research work is focused on software security, network security and cryptography.

Andrea Marotta is Assistant Professor at the Department of Information Engineering, Computer Science and Mathematics of the University of L'Aquila, Italy. He received his M.Sc. degree in Computer Engineering and his Ph.D. in Information and Communications Technology from the University of L'Aquila in 2015 and 2019, respectively. He was a visiting researcher at the Group for Research on Wireless (GROW) at the Instituto Superior Técnico/University of Lisbon, at the Research Institute of Communication, Information and Perception Technologies (TeCIP) of Scuola Superiore di studi universitari e di perfezionamento Sant'Anna in Pisa, and at the Computer Networks Lab (NetLab) of the University of California, Davis. He actively participates in the TPC of IEEE Communications Society conferences. He performs research on Network Reliability, SDM Optical Networks, Multi-Access Edge Computing, 5G Software Defined Access, CoMP Coordinated Scheduling.

Fabio Graziosi was born in L'Aquila, Italy, in 1968. He received the Laurea degree (cum laude) and Ph.D. in Electronic Engineering from the University of L'Aquila, Italy, in 1993 and 1997, respectively. Since February 1997, he has been with the Department of Electrical and Information Engineering at the University of L'Aquila, where he currently holds the position of Full Professor in Communications Systems.

The research activity led to more than 200 publications in international journals and conferences and was initially focused on modeling and performance evaluation of wireless systems in complex propagation scenarios. Subsequently, the research approach has been enriched thanks to the contamination with other scientific areas, i.e. electronics, computer science, and control systems. Research activities on wireless sensor networks and networked embedded systems actually fall within this exciting, multidisciplinary context which has been progressively extended to scientific areas historically far from Communications, such as Structural Engineering.

Dajana Cassioli is Associate Professor of Telecommunications Engineering at the University of L'Aquila, Italy, where she is the Head of the Study Program in Telecommunications Engineering: Advanced Technologies and Services. Her main research interests are in wireless communications, 5G/B5G networks and cybersecurity. She is the Chair of the IEEE ComSoc RCC SiG on Propagation Channels for 5G&B and the Diversity, Equity and Inclusion Activity Coordinator of the IEEE Italy Section. She is Past Chair of IEEE WIE AG Italy Section (2016–2022) and IEEE VT06/COM19 Italy Chapter (2011–2017). Since 2015 she is the coordinator of the University of L'Aquila Node of the CINI National Lab of Cybersecurity, where she led the CyberEquality WG (2020–2021). She has been awarded the ERC StG VISION (Video-oriented UWB-based Intelligent Ubiquitous Sensing) - 2010 and the ERC PoC Grant iCARE (MobIle health-Care system for monitoring toxicity and symptoms in cAncer patients Receiving diseasE-oriented therapy) - 2016. She was the CEO (2014–2018 and 2019) of the spin-off of the University of L'Aquila "Smartly: Natives of Smart Living srl." She served as the IEEE EUROCON 2023 WIE Chair, the IEEE ICC 2023 CISS Co-Chair, PIMRC2018 Industry Co-Chair, RTSI WIE Chair in 2018, 2019 and 2020, MELECON2020 and MetroInd4.0, and TPC member of several International Conferences (ICC, PIMRC, VTC, GLOBECOM, etc.). She serves/served as Associate Editor of IET Electron. Lett. and IEEE Communicat. Lett., and Executive Editor of Wiley Internet Technol. Lett. and Transact. on Emerging Telecommun. Technol. In 2000 she was Summer Manager at the Wireless Systems Research Dep. - AT&T Labs-Research, NJ, USA. She participated in the definition of the standard channel model for the IEEE 802.15.4 standard (2005). In 2022 she was a visiting short-term scholar at the University of Southern California, LA, USA.