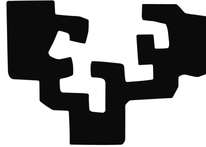
eman ta zabal zazu



Universidad del País Vasco
Euskal Herriko Unibertsitatea

Department of Automatic Control and Systems Engineering

ACOA: application quality of service aware orchestration architecture for the Edge to Cloud continuum

Adrián Orive Oneca

Supervisors:

Mª Isabel Sarachaga González

Aitor Agirre Andueza

February, 2023

(c)2023 ADRIAN ORIVE ONECA

Acknowledgements

These lines are dedicated to all those people who have made possible the completion of this thesis.

First of all, I would like to thank my thesis supervisors: Marga Marcos, who started this journey with me and provided valuable experience, Aitor Agirre, for his practical and down-to-earth approach, and Isabel Sarachaga, who joined halfway through but was essential for this outcome.

I would also like to thank all my colleagues from Ikerlan, my experience there has left memories that I will remember with joy. Especially to all the other PhD students, without their support it would have been much more difficult to overcome the gray days.

Finally, to my family and friends, who have been a fundamental pillar in overcoming all the obstacles that have appeared along the way.

From the bottom of my heart, thank you very much.

Abstract

Edge to Cloud continuum provides an infrastructure of heterogeneous nodes to execute distributed applications. The quality of service of these applications is tied to multiple non-functional requirements, such as response time or energy efficiency, which highly depends on the actual deployment (mapping) of the application components into the infrastructure nodes. In this context, new orchestration architectures are needed to manage both complex infrastructures and application non-functional requirements.

A novel Application-Centric Orchestration Architecture oriented to the Edge to Cloud continuum is proposed to address these challenges. This architecture takes advantage of multiple schedulers in order to improve the scheduling throughput and to provide a finer-tuned scheduling algorithm for each application. It proposes an infrastructure model used for node characterization, a workload model for application definition, and a set of interrelated system components needed to handle the orchestration tasks.

One of these system components implements a new protocol to monitor the network state. Latency, jitter, and packet-loss ratio measurements are obtained, focusing on having a minimal impact in both nodes and the network itself. These measurements are con-

sumed by the scheduling algorithms to manage application quality of service requirements.

Several tests have been performed to validate the architecture. On the one hand, parts of the architecture have been tested in isolation, such as the network monitoring protocol and the increased throughput of the multiple schedulers approach. On the other hand, a use case with two different applications from the railway domain has been used to assess the suitability of this architecture for applications with non-functional requirements. Furthermore, a theoretical analysis of the used scheduling algorithm has also performed and compared with the experimental results from the use case.

Index

1 Introduction **1**

1.1 Motivation 3

1.2 Objectives 8

1.3 Layout 9

2 Network state monitoring **13**

2.1 Introduction 15

2.2 Network state characterization 15

2.3 Requirement identification 17

2.4 Related work 18

2.5 Network state monitoring protocol 23

2.5.1 SWIM protocol viability assessment 24

2.5.2 Network State Monitoring mechanism 28

2.5.3 Implementation of SWIM-NSM 32

2.5.4 Assessment of SWIM-NSM 35

2.6 Conclusions 38

3 Application-Centric Orchestration Architecture **41**

3.1 Introduction 43

3.2 Requirement identification 44

3.3 Scheduling schemes 47

3.3.1 Monolithic scheme 48
3.3.2 Partitioned scheme 49
3.3.3 Two-level scheme 51
3.3.4 Shared state scheme 53
3.3.5 Distributed scheme 54
3.3.6 Comparison of scheduling schemes 55
3.4 Related work 56
3.5 Application-Centric Orchestration Architecture . . 60
3.5.1 Infrastructure model 62
3.5.2 Workload model 64
3.5.3 Architecture Components 66
3.6 Conclusions 75

4 Implementation of ACOA 79
4.1 Introduction 81
4.2 State of the technology 81
4.2.1 Kubernetes 83
4.2.2 Docker Swarm 90
4.2.3 Kubernetes vs Docker Swarm 94
4.3 ACOA implementation over Kubernetes 97
4.3.1 Extended Kubernetes infrastructure model . 97
4.3.2 Extended Kubernetes workload model . . . 98
4.3.3 Extended Kubernetes architecture 103
4.4 Conclusion 107

5 ACOA assessment 111
5.1 Introduction 113
5.2 Shared-state scheme validation 113
5.3 Railway use case 115
5.3.1 Infrastructure description 116

5.3.2 Workload characterization 119
5.3.3 Deployment evaluation 132
5.4 Theoretical bounds for e2e response time scores . . 137
5.5 Conclusions 141

6 Conclusions and future lines 145
6.1 Conclusions 147
6.2 Future lines 149

A SWIM-NSM wire protocol 153
A.1 Introduction 154
A.2 Header 154
A.2.1 Version block 154
A.2.2 Additional blocks 155
A.3 Detection message 155
A.3.1 Common detection message blocks 156
A.3.2 Ping detection message 157
A.3.3 Ping request detection message 158
A.3.4 Ack detection message 160
A.3.5 Forward ack detection message 161
A.4 Dissemination messages 162
A.4.1 Common dissemination message blocks . . . 162
A.4.2 Alive dissemination message 164
A.4.3 Suspect dissemination message 166
A.4.4 Confirm dissemination message 168

References 173

Glossary 183

List of Figures

1.1 Fog/Edge layers different representations. 4
1.2 IoT domains. 6

2.1 SWIM-NSM ping interaction. 31
2.2 SWIM-NSM ping request interaction. 32
2.3 Number of messages required by SWIM-NSM. 37

3.1 Monolithic scheduling scheme. 48
3.2 Partitioned scheduling scheme. 50
3.3 Two-level scheduling scheme. 52
3.4 Shared-state scheduling scheme. 53
3.5 Distributed scheduling scheme. 55
3.6 Infrastructure model. 63
3.7 Workload model. 64
3.8 ACOA architecture. 67
3.9 ACOA data flow. 68
3.10 Management of application schedulers. 71
3.11 Management of application components. 74

4.1 Kubernetes Replica Set, Deployment and Daemon Set. 84
4.2 Kubernetes components. 86
4.3 Kubernetes HA setup components. 87
4.4 Docker Swarm components. 91

4.5 Docker Swarm HA setup components. 93
4.6 ACOA abstractions. 98
4.7 UML component diagram of ACOA architecture over K8s. 106

5.1 Deployment time depending on the number of schedulers. 115
5.2 Smoke monitoring application components. 121
5.3 Speed profiling application components. 127
5.4 Smoke monitoring application best-case deployment. . 136
5.5 Smoke monitoring application worst-case deployment. . 136
5.6 Speed profiling application best case deployment. . . . 137
5.7 Speed profiling application worst case deployment. . . 137
5.8 Theoretical worst-case score. 140

List of Tables

2.1 Network state monitoring methods metrics. 22
2.2 Network state monitoring methods comparison. 22
2.3 Raspberry Pi 3 Model B characteristics. 35

3.1 Scheduler schemes requirement fulfillment. 56
3.2 Identified requirements fulfillment by related work. . . 61
3.3 ACOA requirement fulfillment. 76

4.1 Kubernetes and Docker Swarm requirement fulfillment. 96
4.2 Kubernetes extended components for ACOA. 104

5.1 Collisions during component deployment. 115
5.2 Emulated railway infrastructure. 118
5.3 Train node labels. 118
5.4 Network metric bounds. 120
5.5 Smoke monitoring deployment nodes selection. 134
5.6 Speed profiling deployment nodes selection. 135

A.1 Maximum major and minor version values per block size.155
A.2 Version block for v1.0. 155
A.3 Detection message header block. 156
A.4 Ping detection message header block. 157
A.5 Ping request detection message header block. 158

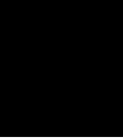
A.6 Ack detection message header block. 160
A.7 Forward ack detection message header block. 161
A.8 Dissemination message header block. 163
A.9 Variable length encoding of 1. 163
A.10 Variable length encoding of 127. 164
A.11 Variable length encoding of 128. 164
A.12 Variable length encoding of 72 057 594 037 927 936. . . . 165
A.13 Alive dissemination message header block. 165
A.14 Suspect dissemination message header block. 166
A.15 Confirm dissemination message header block. 168

List of Code Snippets

4.1 Example YAML definition of an application in ACOA. 99
4.2 Example YAML definition of an application scheduler configuration in ACOA. 100
4.3 Example YAML definition of an application components in ACOA. 100
4.4 Example YAML definition of an application channels in ACOA. 101
4.5 Example YAML definition of an application paths in ACOA. 101
4.6 Example YAML definition of an application constraints and criteria in ACOA. 103
5.1 YAML file structure for the smoke monitoring application. 121
5.2 YAML file fragment for the components definition of the smoke monitoring application. 122
5.3 YAML file fragment for the channels definition of the smoke monitoring application. 123
5.4 YAML file fragment for the paths definition of the smoke monitoring application. 124
5.5 YAML file fragment for the constraints definition of the smoke monitoring application. 124

5.6 YAML file fragment for the criteria definition of the smoke monitoring application. 125
5.7 YAML file structure for the speed profiling application. 127
5.8 YAML file fragment for the components definition of the speed profiling application. 128
5.9 YAML file fragment for the channels definition of the speed profiling application. 129
5.10 YAML file fragment for the paths definition of the speed profiling application. 130
5.11 YAML file fragment for the constraints definition of the speed profiling application. 130
5.12 YAML file fragment for the criteria definition of the speed profiling application. 131
5.13 Pod declaration for the smoke monitoring application measurement component. 132

CHAPTER 1



Introduction

"The first step in solving a problem is to recognize that it does exist."

- Zig Ziglar

1.1 Motivation

Computing paradigms have been swinging back and forth between centralized and distributed computing [1]. Initially, data storage and processing capabilities were centralized in mainframes and accessed by terminals. Personal computers did not replace mainframes until late 1980s or early 1990s, representing the first pendulum swing from centralized to distributed computing. Distribution is enabled by the computing power and storage capacity increment due to Moore's law [2][3]. The underlying argument was moving data and computing closer to the user, thus allowing the usage of the data in a timely manner.

The opposite swing happened in the mid 2000s, centralizing data and applications in cloud data centers. These data centers are made of clusters of powerful devices with high aggregated computing and storage capacities. Tasks from multiple users are executed in the same devices, allowing a higher resource utilization. The cloud computing paradigm reduces the Total Cost of Ownership (TCO), benefits from Economies of Scale (EoS), improves energy efficiency and reduces maintenance costs.

Currently the next swing towards distributed computing is taking place as highlighted by technologies such as Cyber-Physical Systems (CPSs) [4] or Wireless Sensor Networks (WSN) [5]. This allows time-constrained or energy-aware applications benefit from the processing, storage, and networking capabilities of distributed computing, giving birth to new paradigms: fog and edge computing. As a relevant difference with the precedent swing towards distribution, where personal computers made mainframes obsolete,

these new paradigms do not completely replace the cloud computing paradigm, they extend it (figure 1.1A).

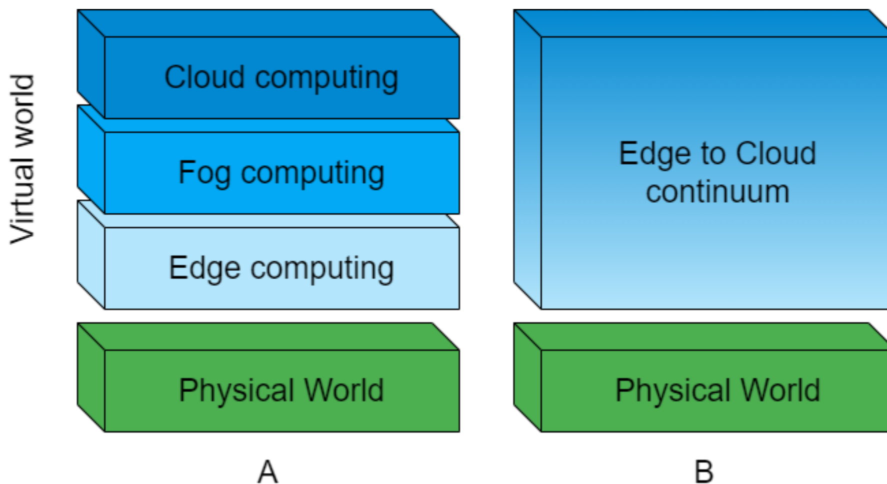


Figure 1.1: Fog/Edge layers different representations.

The term fog computing comes from the analogy with cloud computing, representing a thinner cloud closer to the physical world. The fog layer lies between the edge and cloud layers. It provides interconnectivity and processing capabilities. Compared to the devices found in the cloud layer, fog layer devices are smaller, but they still offer considerable computing and storage capabilities.

The term edge computing is due to the fact that this new paradigm operates in the frontier between the physical and the virtual worlds. It is formed by sensors and actuators that act as CPSs, giving virtual capacities to otherwise physical objects. Devices in this layer are usually resource-constrained but they can be found in high quantities.

However, the bounds between these three layers are fuzzy. A device that belongs to a layer from the point of view of one of the

applications in the system may belong to a different layer from the point of view of another application. This is the reason why, recently, the three layers are considered as a single Edge to Cloud continuum, without categorizing devices explicitly into any of the layers (figure 1.1B).

In this context, applications are realized by different components that exchange messages among them. These components are placed in the different devices of the system, also known as nodes. Properly selecting where to deploy each component in order to meet the application requirements is refer to as orchestrating these applications. This process can be performed manually, but as the number of nodes and applications in the system increases, it becomes increasingly difficult. Orchestration tools that determine the component deployments automatically are used instead.

Cloud environments are a perfect fit for some of the common applications of Internet of Things (IoT) [6] such as big data storage, historic data analysis or online monitoring. Deploying these applications require to take into consideration their CPU time, memory usage or disk space requirements. However, the broad set of IoT domains (figure 1.2) also imposes additional needs for some of their applications that cannot be fully fulfilled by the cloud. Orchestrating these new applications require to factor other non-functional requirements such as low response time, energy-awareness, or multiple working modes.

Some IoT applications need low response time. Orchestration tools were designed for environments where all the nodes in the cluster have similar computing power and negligible latency among them in comparison with the user-cloud latency. Thus, network parameters are not being considered at the scheduling algorithm

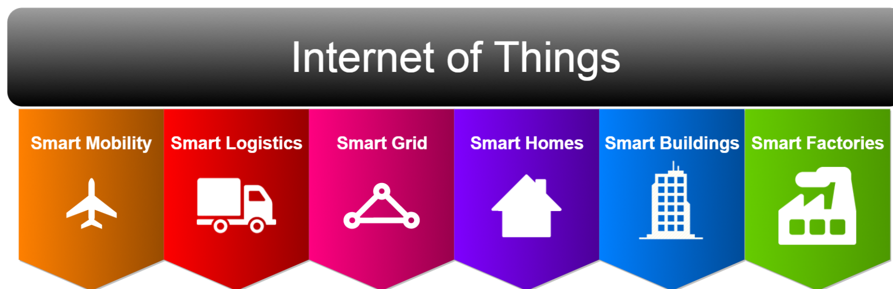


Figure 1.2: IoT domains.

that selects the node where each component of the application is going to be executed.

An example of an application where time reactivity (i.e., low response time) is of high importance could be the speed profiling algorithms that are used by trains. In order to optimize the fuel consumption, these profilers determine the speed that each train needs to have in each moment according to the railway conditions (slopes, turns), the desired goals (reach the next station in a certain time window) and the state of the rest of the trains in the surroundings. While the first two factors are mainly static, the last one makes this application highly dynamic. Additionally, failing to compute the updated profile in time can result in user experience degradation, e.g., delays due to train congestion. Therefore, the need of in time reaction within a dynamic environment arises in some of the IoT applications.

Some other IoT applications have the need of reducing the computing-related energy costs. In a cloud data center, the cost associated to the different nodes is considered to be the same as the nodes have similar hardware characteristics and the electricity cost is the same. Systems with nodes in different geographical

places can take advantage of night electric prices that are usually lower. Energy efficiency is also relevant as the source is not always the electric network; some devices may be battery-powered. Additionally other factors such as energy provenance (e.g., fossil fuels, green or nuclear energy) may be of interest. These parameters are not being considered in current scheduling algorithms.

Application examples that would benefit from these energy-aware algorithms could be those that involve unmanned ground vehicles (UGVs) powered by electric batteries. Computationally expensive loads executed at UGVs, such as path finding or mapping algorithms, could be offloaded to nodes connected to the electric network. Thereby, the duration of the batteries of the UGVs would be increased, reducing the time they need to expend charging and increasing their uptime. However, offloading these computationally expensive loads to cloud servers may take too much time, increasing the time the UGV would be waiting for a response and reducing the uptime. An increase of the uptime of each UGV directly relates with a reduction of the amount of UGVs needed. Therefore, there is also a need of algorithms that are not only network-aware, but also energy-aware, in order to maximize the UGV uptime.

There are also IoT applications that have multiple working modes. Each of these modes may impose different requirements. Current scheduling algorithms are not able to modify their deployment strategies based on the application requirements.

An example of applications with unique needs for different working modes can be found in vertical transportation. Modern lift systems in big buildings use traffic algorithms that assign lifts to each of the users, optimizing the energy consumed by the lifts and the overall transit time to destination. These algorithms are consid-

ered a premium feature and do not offer any noticeable advantage in house buildings. However, in big buildings, these algorithms offer a real advantage during high traffic peaks, but the constrained resources of embedded lift controllers are not always enough. They also require a timely response, thus not being addressable in the cloud either. These scenarios raise different deployment requirements in each case, and therefore, modifying the scheduling and deployment strategies in a per application basis is also needed.

The Edge to Cloud continuum takes advantage of the benefits of cloud, fog, and edge computing, being able to fulfill the Quality of Service (QoS) requirements of IoT applications. These applications will be deployed throughout the continuum, by selecting the nodes that better fit their QoS requirements. The infrastructure management in this continuum is more complex as it involves a variety of node types with different capabilities and available resources.

This new paradigm requires new orchestration architectures that manage these complex infrastructures. They should take care of distributing these applications among the cluster nodes according to each application QoS requirements. End users need to be abstracted of all this complexity, and provided with a straightforward way of defining their applications.

1.2 Objectives

In this context, the main objective of this research work is to design and develop an orchestration architecture that enables the runtime management of distributed containerized applications running on the Edge to Cloud continuum. This architecture should consider

the QoS characterization of the applications in order to dynamically reconfigure the deployment of their components over the clusterized infrastructure nodes. To achieve this main objective, some partial objectives have been identified:

1. Propose mechanisms to customize scheduling and deployment strategies based on non-functional requirements of applications.

2. Explore the viability of distributed or hybrid scheduling algorithms as an alternative to centralized ones in order to get rid of bottlenecks that hinder scalability.

3. Propose mechanisms to monitor network status. The network state will serve as an input for the dynamic scheduling and deployment algorithms in order to guarantee non-functional requirements.

1.3 Layout

The document has been structured in a total of six chapters including the current one. The layout used in the rest of the chapters is as follows.

Since the application QoS is affected by the state of the network, the second chapter focuses on network state monitoring. The properties that need to be measured are defined and the relevant requirements in the Edge to Cloud continuum context are identified. Current monitoring approaches are evaluated before presenting an alternative protocol that better addresses the requirements

imposed by the Edge to Cloud continuum, as well as its implementation and assessment.

This network state monitoring protocol is included in ACOA (Application-Centric Orchestration Architecture), that is presented in chapter three. First, the requirements for an orchestration architecture oriented towards the Edge to Cloud continuum are identified. These requirements are used to evaluate different scheduling schemes and related works to assess their suitability for this context. Finally, a novel orchestration architecture is presented to address all these requirements. ACOA's design includes an infrastructure model to characterize the nodes, a workload model to define the applications that will be executed, and a set of system components required to implement such an architecture.

The implementation of ACOA is detailed in the fourth chapter. The state of the technology is analyzed in order to take advantage of already existing tools. One of these tools has been selected to implement the new Edge to Cloud orchestration architecture.

The assessment of ACOA is presented in chapter five. First, the suitability of the shared-state schema is evaluated regarding the scalability improvement. Second, a use case in the railway domain with two different applications allows assessing ACOA's suitability to satisfy their QoS requirements. Third, a theoretical analysis of the response time is performed to measure the influence of different application and infrastructure properties, and it is compared with the previous results.

The sixth chapter highlights the main contributions of this work and outlines some potential future research lines.

CHAPTER 2

Network state monitoring

"Make it simple, but significant."

- Don Draper

2.1 Introduction

The network state among all the nodes that constitute an Edge to Cloud infrastructure directly impacts the QoS of the applications running on them. Taking this state into account when orchestrating where each component should be executed allows selecting better suited nodes. Therefore, the applications performance is improved based on their QoS needs.

The state of communication networks varies over time. Measurements taken an hour ago do not properly characterize the current state of the network. Therefore, this state needs to be continuously monitored.

The current chapter defines the different metrics that characterize the state of the network and identifies the requirements of a network monitoring system targeted at Edge to Cloud architectures. Next, the fulfillment of current monitoring approaches is checked against these metrics and requirements. Finally, the proposed network state monitoring approach is designed, implemented, and assessed.

2.2 Network state characterization

A communication network connects all the nodes in the cluster to enable them to exchange messages. The connection between any two nodes in the network is known as “link”. In order to monitor the state of a network, several metrics need to be considered for each link.

The organization responsible for the standardization of the Internet is called IETF (Internet Engineering Task Force) [7]. "The

mission of the IETF is to produce high quality, relevant technical and engineering documents that influence the way people design, use, and manage the Internet in such a way as to make the Internet work better" [8]. Some of these documents formally define the main properties of a link: latency, jitter, packet-loss rate, and bandwidth.

- The **latency** is defined as the time needed for a packet to travel from a source node to a destination node. It measures the delay incurred in sending a message from one node to another. It was first defined in RFC 2679 [9], which was later obsoleted by RFC 7679 [10].

- The **jitter** represents the variability of the latency between different packet transmissions. It provides a measure of the stability of the link. It is defined in RFC 3393 [11].

- The **packet-loss rate** measures the amount of unsuccessful packet transmissions between two nodes. It reflects the reliability of the link. It was first defined in RFC 2680 [12], which was later obsoleted by RFC 7680 [13].

- The **bandwidth** represents the amount of data that can be transmitted per time unit. It provides a measure of the capacity of a link. It is defined in RFC 5136 [14].

These four properties characterize the state between any pair of nodes. The state of the whole set of links constitutes the network state, and it needs to be monitored.

2.3 Requirement identification

Monitoring network properties in the context of an Edge to Cloud orchestration architecture raises some important requirements that need to be fulfilled.

The main goal of an Edge to Cloud architecture is to run the user applications in a distributed fashion. Monitoring the network state is a system process, and accordingly, it cannot hoard too many resources. This applies to both the CPU and memory resources of the nodes as well as the network resources.

This problem is even more relevant for clusters with a vast number of nodes, as the ones that can be found in the Edge to Cloud continuum context. The number of links (L) has a quadratic relation (2.1) with the number of nodes in the cluster (N).

$$L = f(N) = 0 + 1 + 2 + ... + (N - 1) = \sum_{i=0}^{N-1} i = \frac{N^2 - N}{2} \quad (2.1)$$

Additionally, these links are not limited to a local network, they can span multiple networks with several routers. As some of these routers are not owned by the same company, the needed solution cannot require modifying their behavior.

In this context, the following requirements for Edge to Cloud continuum network state monitoring can be extracted:

- The CPU time required should be low, in order to leave as much CPU time available to the execution of the user applications.

- The number and size of the exchanged messages should be small so that they do not interfere with the foreground processes.

- Scalability is also a major concern, due to the quadratic relation between the number of links and nodes. The measuring tools should comply with the above two requirements even for a vast number of nodes.

- Only nodes owned by the architecture can be modified, as the routers may not be owned by the same company.

Existing monitoring approaches need to fulfill these requirements in order to be a good fit for the Edge to Cloud continuum.

2.4 Related work

The most basic method to obtain network metrics, which is used as a reference for the following ones, is to use a heartbeat-based protocol with acknowledgment [15]. Latency measurements can be obtained by measuring the time elapsed between the sent message and the received response. Jitter can be easily computed from the latency measurements. Packet-lose rate can also be calculated by tracking the amount of successful and failed messages. Bandwidth, however, cannot be measured with this method.

Heartbeat-based protocol requires a low amount of CPU time since the involved math is simple. It is very lightweight as nodes do not require to exchange data, and thus, the size of the messages is basically the overhead of the used transport protocol. Regarding

scalability, the number of messages exchanged every period grows quadratically with the number of nodes, which is not desired.

In order to improve the scalability, several approaches have been followed in the related work. They can be categorized in active or passive methods. Active methods inject new messages to the network in order to obtain the measurements, while passive methods use the existing packets of the network.

In [16], the authors propose a centralized network measuring method. Each node in the cluster reports timestamps and message sizes aggregated per flow to a centralized node. A flow is defined as the messages of a specific protocol exchanged from a certain endpoint (address and port) to a different one. These timestamps and message sizes are used to compute latency, packet-lose rate, and consumed bandwidth in a central processing unit. It must be noted that the consumed bandwidth only allows establishing a lower bound for the link bandwidth. Capturing timestamps and message sizes is done in a passive way, not requiring additional packets to be sent. However, the transmission of this data to the centralized processing unit requires new packets. The size of these messages is small, but the amount is directly proportional to the load of the network. In order to reduce the impact, the authors provide a deterministic sampling mechanism that filters which packets should be reported to the centralized processing unit.

A different approach is taken in [17]. The authors propose a network monitoring design that extends NetQuest, a framework for inferring the network state based on measurements on a subset of the available links. They use it to obtain latency and packet-lose metrics. Individual metrics are obtained in an active manner, and sent to a central node to infer the metrics of the rest of the

links. The framework requires knowledge of the topology of the network, and uses a Bayesian algorithm to solve a NP-Complete problem in order to select the most adequate paths to measure. Additionally, their proposed modification performs a 20-minute-long process on each new node that is later used to restrict how many paths can start in that node. The node and network resources used by this approach are small. However, all the inferring is done in a centralized node impacting scalability, and a 20-minute-long warm up plus solving an NP-Complete problem is required for each node addition.

A Network Bandwidth Predictor (NBP) is developed in [18]. The authors propose a neural network-based predictor for the bandwidth that relies on continuous active measurements of the throughput of each link. The throughput is measured by sending a pair of messages in a periodic fashion. A first small message is used to measure the latency between the nodes and a second large message measures the throughput. The size of the network is not considered, and no method is provided to enhance the scalability.

A segmented approach is followed in [19]. The links between all nodes in the system are split into segments considering the network switches. The measurement mechanism is deployed in all the switches that are part of the network and introduce timestamps into existing traffic packets. These timestamps are used to compute aggregated latency, jitter, and packet-loss rates for each segment. Further investigation in [20] allowed to estimate these network properties in a per-flow basis instead of aggregated per segment. Unlike the previous approaches, in both cases the switches that conform the network need to be modified; therefore, they are not a good fit for the Edge to Cloud continuum where the vast ma-

jority of the switches are not property of the company interested in the network measurements.

A new measuring architecture called MAPLE (Measurement Architecture for Packet LatEncies) is proposed by the same authors in [21]. It provides a finer-grained measuring mechanism that stores latencies per packet instead of per-segment or per-flow basis. It also provides a querying mechanism so that nodes can obtain the network state metrics that are being stored in the switches. However, MAPLE still requires the switches of the network to be modified to implement the measuring mechanism, which will usually not be doable in the target scenarios,

Table 2.1 shows which metrics can be obtained with each of the discussed methods. As it can be observed, only the last presented approach is able to measure the bandwidth. Large messages need to be sent for every pair of nodes which makes this method very intrusive.

Table 2.2 shows the behavior of each of the discussed methods related to the requirements highlighted in section 2.3: low CPU time, lightweight messages, scalability and using only owned nodes. Two values are provided for [17], the left one corresponds to each of the nodes, and the right one to the centralized inferring process.

The presented approaches do not fulfill the identified requirements. Measuring the bandwidth in a lightweight fashion is still an unresolved challenge. But even if only considering the other three properties, the scalability and the lightweight-ness need to be tackled. Furthermore, the simplest approach, heartbeats, is one of the most complying of the listed approaches.

Table 2.1: Network state monitoring methods metrics.

Monitoring Methods		**Metric**			
		Latency	**Jitter**	**Packet-lose rate**	**Bandwidth**
Heartbeat	[15]	✓	✓	✓	✗
Serral, Cabellos, and Domingo	[16]	✓	✗	✓	✗
Song and Yalagandula	[17]	✓	✗	✓	✗
Eswaradass, Sun, and Wu	[18]	✓	✗	✗	✓
Kompella et al.	[19]	✓	✓	✓	✗
Lee, Duffield, and Kompella	[20]	✓	✓	✓	✗
Lee, Duffield, and Kompella	[21]	✓	✓	✓	✗

Table 2.2: Network state monitoring methods comparison.

Monitoring Methods		**Low CPU time**	**Lightweight**	**Scalable**	**Owned nodes**
Heartbeat	[15]	✓	✓	✗	✓
Serral, Cabellos, and Domingo	[16]	✓	✓	✗	✓
Song and Yalagandula	[17]	✓/ ✗	✗	✓/ ✗	✓
Eswaradass, Sun, and Wu	[18]	✓	✗	✗	✓
Kompella et al.	[19]	✓	✓	✗	✗
Lee, Duffield, and Kompella	[20]	✓	✓	✗	✗
Lee, Duffield, and Kompella	[21]	✓	✓	✗	✗

2.5 Network state monitoring protocol

The presented related work approach towards scalability relies on reducing the amount of network state measurements by inferring some of these metrics from a smaller sample in a centralized unit. This approach presents two main drawbacks: precision and centralization.

Inferring metrics will not yield results as precise as measuring them. This issue has even more importance in Edge to Cloud scenarios, where the network connections are very heterogeneous, and therefore, it is harder to infer an accurate value for the network properties of each link.

The centralized inferring approach allows increasing the scalability by reducing the number of measurements needed. However, the load imposed to this centralized node will increase as the number of nodes, and consequently links, increases.

Since Edge to Cloud scenarios are expected to support architectures with a high number of nodes, a distributed approach may fit this paradigm better. In distributed processing systems, detecting the potential failures of the cluster nodes is of vital importance, as the tasks that were being performed by these failing nodes need to be handled by other working nodes.

This section presents a distributed network state monitoring approach targeted at the Edge to Cloud continuum that focuses on the aforementioned requirements: low CPU time, lightweight messages, scalability and using only owned nodes. It is based on the scalability approach followed in group membership protocols, whose goal is to maintain an up-to-date list of the properly working

nodes in the cluster, by detecting node failures and removing them from the list.

Group membership and network state monitoring protocols fulfill two different tasks. For both cases, the most basic protocol relies on heartbeats, which has been used in group membership field for over 35 years [22]. However, the approach followed to improve the scalability in the group membership field differs. It reduces the amount of network load by only targeting one node at each period instead of all of them. This approach can be used to provide a better scalability for Edge to Cloud network state monitoring.

This section describes a scalable group membership protocol, and compares it to heartbeat-based ones. This protocol is then extended in order to provide network state monitoring mechanisms. A library implementing this protocol is developed and its viability is assessed.

2.5.1 SWIM protocol viability assessment

SWIM stands for Scalable Weakly-consistent Infection-style process group Membership protocol [23]. It was designed to solve the scalability issues that heartbeat-based group membership protocols have when the number of nodes grows significantly.

The SWIM protocol is split into two separate mechanisms[1]: failure detection and dissemination. The first one oversees other nodes in order to detect when they are no longer available and modify its internal list accordingly. The second one propagates

[1]The authors originally call both of these parts components. In this document, the term mechanism will be used instead to avoid confusion with application components.

these internal list state changes in an infectious style so that all nodes in the group are informed in logarithmic time.

The failure detection mechanism checks the state of a node by sending a ping message. An acknowledgement message is expected before a configured deadline. If this deadline is not met, the node will try to perform an indirect check, by requesting several nodes to perform a check on its behalf with a ping request message. They will send a normal ping message to the specified node and will only acknowledge the ping request if they are able to successfully receive an acknowledgement message from the target node. These requested ping messages will not trigger additional ping request delegation nor will change the state of the internal list of the requested node.

When neither the original node nor the delegated nodes are able to successfully ping the target node, it will not be immediately removed from the group membership list. Instead, it will be marked as suspicious. If the suspected node is not able to prove that it is alive after a certain number of intervals, the suspicion will be confirmed, and the node will be removed from the group membership list.

Three state changes can be performed to the internal group membership lists: marked as alive, marked as suspicious and confirmed the suspicion, i.e., marked as dead. The dissemination mechanism is in charge of transmitting these state changes to the rest of the nodes. In order to do so, all three above-described messages (ping, ping request and acknowledgement) will contain state change information attached. Each state change will be transmitted several times. State changes received from other nodes will also

be transmitted, achieving the infectious style spread that grants the logarithmic propagation time.

As several state changes related to the same node can be spread simultaneously, a priority needs to be defined. State changes include an incarnation number and those with higher values take priority. If both state changes have the same incarnation number, marking as dead takes priority over both other types, and marking as suspicious takes priority over marking as alive. Therefore, the incarnation number only needs to be increased when a node wants to claim that a suspicion made by another node was incorrect.

The target node at each interval is extracted from a queue. This queue is initially filled with some members, known as seeds. Once this queue is empty, it is populated with all the nodes in the group membership list in a random order. This ensures that each time the queue is fully consumed, all nodes have been pinged, i.e., it guarantees that they are all checked in a time-bounded fashion despite being in a pseudo-random order.

There are several parameters that need to be adjusted to each use case: the interval, the ping acknowledge deadline, the ping request acknowledge deadline, the number of nodes to delegate the ping in case of direct ping failure, the number of times a state change is gossiped by each node, and the number of intervals before a node is removed from the group after it is marked as suspicious.

In order to evaluate the scalability, the number of interactions between nodes and its relationship with the total number of nodes needs to be considered. In heartbeat-based protocols, each node interacts with every node in the cluster every period (2.2). $M^{heartbeat}$

denotes the number of exchanged messages in a heartbeat-based protocol, and N the number of nodes in the cluster.

$$M^{heartbeat}(N) = N(N-1) = N^2 - N \longrightarrow O(N^2) \qquad (2.2)$$

SWIM reduces the number of node interactions to one per interval. These interactions may require multiple messages but the relationship with the number of nodes is linear as it can be seen in equation (2.3). M_{max}^{SWIM} denotes the maximum number of messages exchanged in SWIM, and k a single-digit integer constant that depends on the parameterization of the protocol.

$$M_{max}^{SWIM}(N) = kN \longrightarrow O(N) \qquad (2.3)$$

The main drawback of the reduction of interactions is the increment in failure propagation time. It measures how long it takes for the whole cluster to detect a node failure. In heartbeat-based protocol where nodes interact with every node in the cluster, failures are detected in constant time as shown in equation (2.4), where $t^{heartbeat}$ is the propagation time, T the heartbeat period, and d the number of periods that nodes will wait to prevent false positives.

$$t^{heartbeat}(N) = dT \longrightarrow O(0) \qquad (2.4)$$

As demonstrated by the authors in [23], the infection-style propagation mechanism of SWIM achieves a logarithmic failure propagation time relationship with the number of nodes. This relationship is shown in equation (2.5), where t^{SWIM} is the propagation

time, and λ is a parameter related to the number of times each state change needs to be gossiped to the rest of the nodes.

$$t^{SWIM}(N) = \lambda \log N \longrightarrow O(\log N) \tag{2.5}$$

SWIM achieves a better scalability regarding the number of exchanged messages by sacrificing propagation time compared to heartbeat-based protocols. The propagation time is not as relevant for network monitoring as it is for group membership. Therefore, the improvement offered by SWIM compared to heartbeat protocols is even more relevant in the desired context.

2.5.2 Network State Monitoring mechanism

SWIM fulfills the low resource utilization, lightweight messages, and scalability requirements; but it does not offer network measuring capabilities. However, an extended protocol called SWIM-NSM (Network State Monitoring) has been designed to allow capturing network metrics for every link [24].

While SWIM provides the failure detector and dissemination mechanisms, SWIM-NSM extends these with a third monitoring mechanism that is in charge of network state monitoring. This mechanism performs latency, jitter, and packet-loss ratio calculations. Bandwidth is not measured by SWIM-NSM, as it would require more intrusive measuring techniques.

Directly computing the latency of two nodes requires knowing the time a message was sent and the time it was received in the same time reference system. As each of these timestamps are measured in different nodes, a single time reference cannot be guar-

anteed. Therefore, computing the latency would require knowing the difference between both time systems (δ^t), as seen in equation (2.6). t is used to refer to the sending and t' to the receiving node's time references. Hence, it would require a time synchronization mechanism.

$$latency = t_{rec} - t_{sent} = t'_{rec} - t_{sent} + \delta^t \tag{2.6}$$

An alternative solution is to approximate the latency as half of the round-trip time (RTT). Despite being in two different time system references, δ^t can be simplified as shown in equation (2.7). D is the time elapsed between the ping reception in the ping receiving node until the acknowledge was sent, and it is specially important for ping request delegation as another ping is performed before the corresponding acknowledgement message is sent. This duration needs to be provided to the ping sending node to calculate the RTT.

$$\begin{aligned} RTT &= t_{rec}^{ping} - t_{sent}^{ping} + t_{rec}^{ack} - t_{sent}^{ack} \\ &= t_{rec}^{\prime ping} - t_{sent}^{ping} + \delta^t + t_{rec}^{ack} - t_{sent}^{\prime ack} - \delta^t \\ &= t_{rec}^{ack} - t_{sent}^{ping} + t_{rec}^{\prime ping} - t_{sent}^{\prime ack} \\ &= t_{rec}^{ack} - t_{sent}^{ping} + D \approx 2 * latency \end{aligned} \tag{2.7}$$

In order to monitor the network state metrics described above, each node keeps track of the results of every ping and ping request interactions with every other node during the time window of interest. This includes if the message exchange succeeded or failed, and in case it succeeded, the latency of that transmission. The failure

detection mechanism of the SWIM protocol needs to be modified to enable the monitoring mechanism.

In SWIM, when a ping request message is sent, the receiving node only answer with an acknowledge message in case it was able to successfully ping the target node. This does not allow distinguishing which node failed, the one receiving the ping request or the target of that ping request. However, SWIM-NSM tracks which messages failed, so it needs to differentiate those two cases.

For this purpose, a fourth message type is introduced: the forward acknowledge. This new message differs from the acknowledge message by a single additional flag. In SWIM-NSM, when a node is requested to ping a target, it will always answer with a forward acknowledge message instead of only answering in case it could successfully communicate with the target of the ping request. The outcome of the requested ping is encoded in the additional flag.

Additionally, computing the RTT (as described in equation 2.7) requires a new duration field to be added to both acknowledge and forward acknowledge messages. Duration is encoded as a 4-byte unsigned integer representing the number of microseconds elapsed. The maximum value of this field is 4 294 967 295 µs, i.e., 1 h 11 min 34 s 967 ms 295 µs.

With these modifications to the failure detection mechanism, the monitoring mechanism is able to obtain the raw metrics required to compute the latency, jitter, and packet-loss ratio metrics.

On the one hand, on a successful ping transmission between nodes N_i and N_j (figure 2.1 A), a success state and the latency are stored in the source node N_i. The target node N_j cannot update any metric as the acknowledge message could have failed

and cannot compute the RTT either way. On the other hand, on a failed ping transmission (figure 2.1 B), node N_i will detect the lost packet after a certain timeout, and it can therefore store a fail state. Node N_j will not be able to update any metric, as it either failed to receive the ping in the first instance, or its acknowledgement was lost but it is unable to detect it.

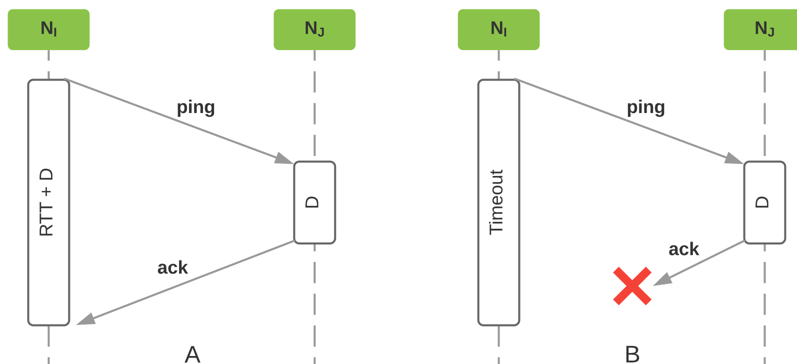


Figure 2.1: SWIM-NSM ping interaction.

Ping request and forward acknowledge message pairs also provide the opportunity to store metrics similar to ping and acknowledge pairs as depicted in figure 2.2. Node N_i, after failing a ping transaction with node N_j, requests nodes N_{k1}, N_{k2}, and N_{k3}. Node N_{k1} is able to ping successfully node N_j, and thus can store a success state and a latency value. Node N_{k2}, however, is unable to ping N_j and therefore only stores a fail state. Both answer the ping request with a forward acknowledge, allowing N_i to store a success state and a latency for each of those links. However, the ping request transmission with node N_{k3} was not successful, and therefore a fail state is stored for this link.

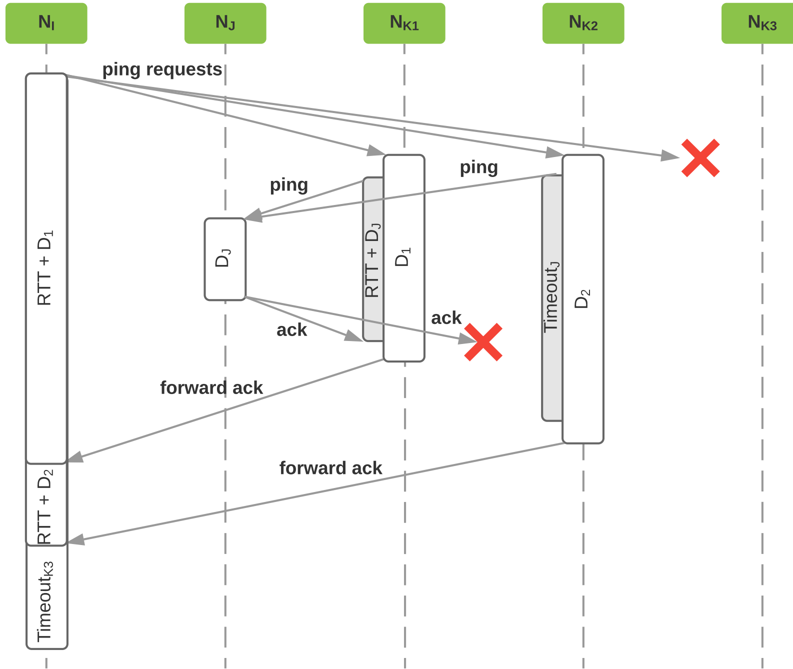


Figure 2.2: SWIM-NSM ping request interaction.

With these raw latency, success state and fail state measurements; the latency, jitter, and packet-loss rate properties for every link can be calculated.

The full specification of the wire protocol for SWIM-NSM v1.0 is detailed in appendix A.

2.5.3 Implementation of SWIM-NSM

A SWIM-NSM protocol library over TCP has been developed in Go. This library creates the background processes that implement the protocol (SWIM-NSM daemons). They are deployed in every node in the cluster. Each of these daemons requests confirmations

periodically from the rest of them and it also answers the confirmation requests from other daemons.

Raw latency, success state and fail state measurements are stored for a moving time window. The raw measurements during the validity time window are used to compute the latency, jitter, and packet-loss ratio of each link.

When creating a daemon using this library, the means to provide the input and obtain the outputs need to be adapted for each use case. As an input, it requires the seeds to start building the internal list. A full list of members is not required as the gossip mechanism will be able to build this full list from just some starting seeds. The outputs are the latency, jitter, and packet-loss ratio measurements for every link. For example, the daemon used for this protocol assessment (following section) is executed from the console, so the input is provided as a terminal argument and the output is printed to the console.

Additionally, the library provides several configuration parameters for the daemons in order to facilitate a better adaptation to every network. The most relevant parameters are the following ones:

- Period: interval between the daemon ping messages. On the one hand, a higher period imposes a lower load to the network. On the other hand, a lower period achieves a shorter failure detection and propagation time, and increases the network state monitoring sample size.

- Ping timeout: deadline for direct pings. A higher timeout slightly delays the failure detection, but it reduces the number of false positives in busy networks.

- Ping request timeout: deadline for ping request. As a ping interaction has to be made, a sensible value is double or triple the ping timeout.

- Number of ping requests: number of nodes requested for delegated pings. A value of 3 was considered as optimal by [25] for thousands of nodes. Lower values can be used for smaller groups. A function based on the number of nodes can be used.

- Gossip transmission times: number of times each state change is transmitted by each node. A higher number increases the mean size of each message, but it provides a more stable start for the dissemination messages. A function based on the number of nodes can be used.

- Suspicion confirmation periods: number of intervals that a node will wait after marking another node as suspicious before it is finally removed from the group. A higher value delays failure detection time, but it reduces false positives. A function based on the number of nodes can be used.

- Metrics buffer characteristics: both the size and the valid time window for the buffers that store network state metrics. Metrics are automatically discarded after the time window has elapsed. If a new measurement cannot be stored because the buffer is full, the oldest measurement will be discarded.

2.5.4 Assessment of SWIM-NSM

The protocol was assessed on a testbed of 9 Raspberry Pi 3 Model B (table 2.3), connected through a high-latency office Wi-Fi network. The conditions of the network differ from the ideal ones. This choice was made on purpose to assess the behavior of the protocol in a more hostile environment.

Table 2.3: Raspberry Pi 3 Model B characteristics.

Property	Value
Processor	Quad Core ARM Cortex A35
Processor speed	1.2 GHz
RAM	1 GB
OS	Raspbian (November 17)
SD card	16 GB class 10

Daemons were created with the SWIM-NSM library with a period of 200 ms. 500 ms and 1500 ms were parameterized for the ping timeout and ping request timeout respectively. In case of a node being unable to ping its target, 2 additional nodes were requested to ping it. Each gossip was transmitted 5 times by each node, and 10 periods were waited before confirming as dead a node that was suspicious. The metrics buffers were limited to ten minutes, and any older measurement was dropped.

The test included several forced scenarios to check the behavior of the daemons under those circumstances. The steps below were followed:

1. A first daemon was initiated.
2. Seven of the remaining daemons were initiated immediately afterwards with the address of the first one.

3. The last of the daemons was delayed to verify that nodes could join an already running group seamlessly.

4. One of the running daemons was forcefully shut down to verify that the cluster was able to recognize that this node left the group.

5. The "failed" node was started again to verify that a node that was removed from the cluster could join the group again.

6. The daemons were shut down after an hour.

The protocol was able to maintain an up-to-date list of the active members in the group under all of the above circumstances. Latency, jitter, and packet-loss rate properties were being monitored successfully. The exact numeric values are not indicative of the performance of the protocol because they are related to the state of that network in that moment.

In order to evaluate the scalability, the number of exchanged messages by SWIM-NSM per period is compared to heartbeat-based protocols, both with acknowledge and without it. The exact number of messages exchanged by SWIM-NSM depends on the number of successful pings, therefore a lower and an upper bound are provided, corresponding to the best and the worst-case scenarios. Figure 2.3 shows these number of messages up to 300 nodes. A zoom of the first 20 nodes is also provided, showing that even for a small number of nodes in the cluster, SWIM-NSM requires less messages; and this difference greatly increases with the number of nodes. Once eleven or more nodes are part of the cluster, even the worst-case scenario for SWIM-NSM requires less exchanged messages than a heartbeat-based protocol.

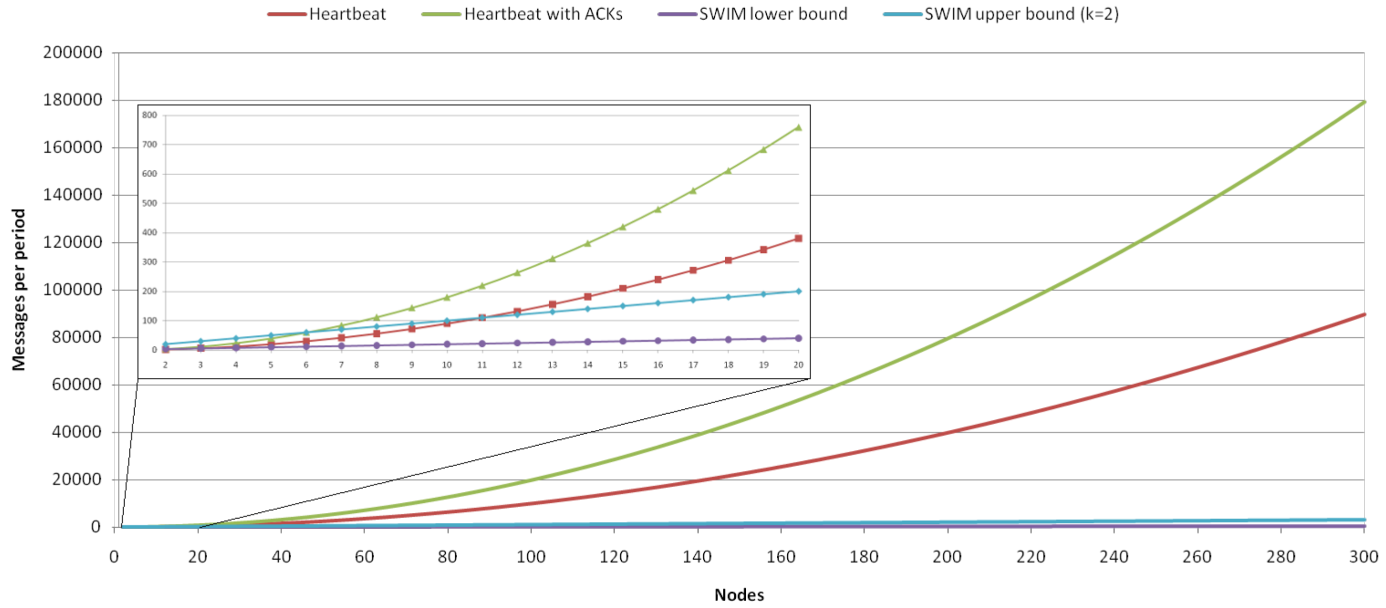


Figure 2.3: Number of messages required by SWIM-NSM.

2.6 Conclusions

SWIM-NSM offers a distributed, scalable, and lightweight network state monitoring approach oriented at Edge to Cloud continuum clusters.

The proposed protocol extends SWIM by enabling the recollection of latency, and success or fail state for each link of the network. These raw metrics can be used to compute three of the properties that characterize a link: latency, jitter, and packet-loss ratio. Bandwidth, the fourth property, cannot be collected with SWIM-NSM as it would require a more intrusive probing mechanism that would use more network resources.

Computing the properties from the raw metrics captured by the protocol requires very simple operations, thus requiring very low CPU time. A small 4-byte field was added to some of the messages, keeping the messages as lightweight as possible. The distributed approach of this protocol provides a scalability advantage compared with the approaches studied in the related work. The reduced number of messages exchanged (linear relationship to the number of nodes instead of quadratic in heartbeat-based protocols) reduces the impact on the network even in Edge to Cloud continuum scenarios with a high number of nodes. The daemons required for SWIM-NSM only need to be deployed in the nodes that are part of the orchestration architecture, without requiring to affect any device that is not owned by the company. These characteristics enable SWIM-NSM to fulfill all the identified requirements for network state monitoring in the Edge to Cloud continuum.

The SWIM-NSM protocol has been assessed in a test-bed cluster, properly capturing the network state metrics while nodes join

and leave the cluster seamlessly. The daemons were able to successfully keep an updated group membership list, while latency, jitter and packet-loss rate properties for each link were being measured by the network state monitoring mechanism.

CHAPTER 3

Application-Centric Orchestration Architecture

"Have no fear of perfection - you will never reach it."

- Salvador Dali

3.1 Introduction

An orchestration architecture is responsible for the execution life cycle of the application components, from their deployment to their execution end. Therefore, the architecture needs to be aware of the hardware resources that it has available (infrastructure model) and the applications to be executed (workload model).

The infrastructure model characterizes the nodes that take part in the cluster managed by the orchestration architecture. These nodes provide the hardware resources where the applications will be executed. The architecture needs to know, additionally to the characteristics of these nodes, their resource usage.

The workload model defines the applications that the orchestration architecture manages. This model should describe not only the applications and all their constraints, but also their QoS requirements. Furthermore, the current state of the applications execution also needs to be stored in this model.

In order to deploy these applications in the infrastructure nodes, several system components work together to perform all the management tasks of the orchestration architecture.

In this chapter, the set of requirements for an Edge to Cloud continuum-oriented orchestration architecture are identified. Next, different centralized and distributed scheduling schemes are analyzed, highlighting their main advantages and drawbacks. From this analysis, the most appropriate scheme for a new orchestration architecture is selected. The related work is then evaluated against the identified requirements in order to assess their viability. Finally, ACOA (Application-Centric Orchestration Architecture) is presented. It is a novel orchestration architecture designed for the

Edge to Cloud continuum, fulfilling the identified requirements. The models used by ACOA to represent the infrastructure and the workload are introduced and the components that enable the orchestration of applications in the target scenario are described.

3.2 Requirement identification

The Edge to Cloud continuum introduces a new set of challenges. Identifying these challenges is needed to evaluate the validity of existing orchestration approaches for this scenario.

As the target is the Edge to Cloud continuum, the nodes that are used to form this kind of infrastructures are very heterogeneous: from powerful nodes in Cloud datacenters to resource-constrained devices distributed geographically. The orchestration architecture needs to support this infrastructure heterogeneity to take advantage of the characteristics that each of these nodes offers (**R1 - Heterogeneous infrastructure**).

Edge to Cloud continuum-oriented applications are generally formed by components that are executed in the infrastructure nodes. These components work together to perform the application tasks. Therefore, the most common processes that need to be orchestrated are expected to be running continuously, as opposed to batch jobs (**R2 - Long-lived components**).

The diversity of the infrastructure in the Edge to Cloud continuum enables its use for multiple vertical domains: from smart factories and logistics to smart homes and buildings. These applications have very different QoS requirements. Scheduling each

of these applications needs to be tailored to their needs (**R3 - Scheduling customization**).

All the above requirements, defining applications and components that need to be distributed among several nodes and with different QoS requirements, increase the complexity of the system. This complexity needs to be abstracted from the end user. The applications definition should be provided in a human-readable format (**R4 - Ease of use**).

Managing the infrastructure is also a complex task. New nodes need to be able to join the cluster seamlessly, as well as leaving it. The orchestration architecture should provide enough tools to manage the whole distributed infrastructure in a simple fashion (**R5 - Infrastructure management**).

Additionally, nodes need to be monitored continuously (**R6 - Infrastructure monitoring**). Detecting node failures allows the orchestration architecture to move the components that were being executed in them to other nodes. It also allows measuring different metrics, such as available memory or disk space, that are relevant for scheduling.

Similarly, network metrics also need to be monitored (**R7 - Network monitoring**). Applications may have QoS requirements that rely on the network state, such as end-to-end response time or reliability constraints. Latency between the nodes where an application is being executed directly impacts the end-to-end response time. The packet loss rate may also impact the reliability of an application.

Scheduling algorithms need to take into account both node and inter-node metrics, as they may impact the QoS of Edge to Cloud

continuum-oriented applications. This means that scheduling decision can no longer be made on a per component basis, the whole application needs to be scheduled as the relative position of components affects the QoS of the application (**R8 - QoS awareness**).

The complexity of these algorithms, added to the increasing number of nodes that these architectures need to support, makes scalability a real concern for these orchestration architectures. Large infrastructures and workloads should not affect the system performance (**R9 - Scalability**).

All the above identified requirements can be summarized in the following list:

1. **Heterogeneous infrastructure:** support Edge to Cloud continuum infrastructures and the diversity of nodes that can be part of them.
2. **Long-lived components:** support long-lived components as the building block for Edge to Cloud continuum-oriented applications.
3. **Scheduling customization:** scheduling needs to be tailored to different applications from different vertical domains.
4. **Ease of use:** workload definition in a human-readable format.
5. **Infrastructure management:** nodes dynamically join/leave the cluster.
6. **Infrastructure monitoring:** detect node failures and monitor node resources (CPU usage, available memory, disc space, ...).

7. **Network monitoring:** monitor network metrics (latency, jitter, packet-loss rate).

8. **QoS awareness:** scheduling decisions must take into account the QoS requirements of Edge to Cloud continuum-oriented applications.

9. **Scalability:** handle large infrastructure and workload without considerable penalty to system performance.

3.3 Scheduling schemes

The scheduling process is one of the core orchestration architecture tasks. It allocates each of the application components to an infrastructure node where it will be executed, based on the component specification extracted from the user-provided application definitions.

Different scheduling algorithms can be implemented; the higher the complexity, the longer they take to make a decision. The scheduling process will observe the current system state, but it may change during the time required to run the scheduling algorithm. Nodes leaving the cluster or node failures may result in a scheduling decision that assigns a component to a non-available node. Multiple schedulers running concurrently can also make conflicting decisions which are denoted as scheduling collisions. These conflicting scheduling decisions need to be processed again, despite they happen rarely.

Different schemes have been used historically, with different degrees of decentralization. The following section describes them and lists their main advantages and drawbacks. The capability of

running long-lived components (R2), taking into account applications' QoS (R8) and the scalability (R9) that each of these schemes presents will be evaluated, as these are the requirements that are most directly affected by the chosen scheduling scheme.

3.3.1 Monolithic scheme

The monolithic scheme is a fully centralized approach depicted in figure 3.1. A single scheduler process handles all the scheduling and needs to implement all the policies supported by the system. The components that need to be scheduled are queued and handled sequentially.

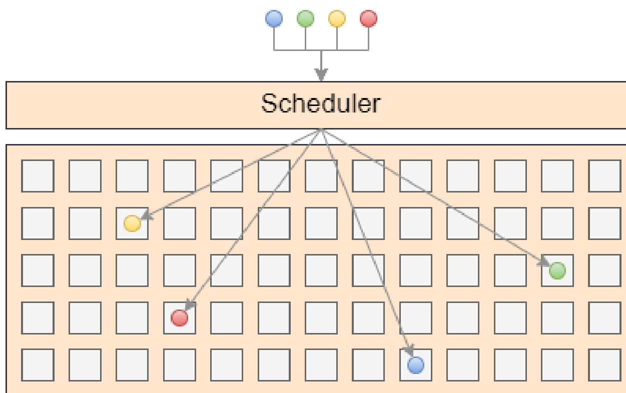


Figure 3.1: Monolithic scheduling scheme.

The monolithic approach is the most used scheduling scheme among both High-Performance Computing (HPC) systems and Container Orchestration Engines (COEs). Some examples of this approach are Borg [26], one of the first COE used internally by Google, or Hadoop's early version schedulers. Kubernetes [27] and Docker Swarm [28] also follow this scheduling scheme.

This scheme does not present scheduling collision, as all the scheduling decisions are made in a centralized process sequentially. The only state changes that may happen during the scheduling algorithm execution are the infrastructure changes related to nodes leaving the cluster or node failures.

The centralized scheduler is in charge of all the infrastructure resources. Therefore, the whole infrastructure cluster is electable as a target node by the monolithic scheduler. The best node among all the infrastructure can be selected based on the scheduling algorithm and configured policies, achieving optimal placement.

Regarding the scalability, a single centralized process evaluates the scheduling algorithm sequentially. This is known as HoL (Head of Line) blocking. Scheduling becomes the bottle neck of the COE, limiting the number of applications that can be handled in larger clusters without affecting the performance. This is the main drawback that this scheduling scheme presents.

3.3.2 Partitioned scheme

The partitioned scheme splits the infrastructure in different subsets of nodes. These subsets are called partitions and are managed by a separate scheduler. The components to be scheduled are routed to one of the partition schedulers by a centralized process as illustrated in figure 3.2. The partitioning and routing policies can be very diverse, including geographical distribution, partition load balancing, etc. An example of a partitioned scheme can be found in Dryad's Quincy [29].

On the one hand, the partitioned scheme does not present collisions even with multiple schedulers executed concurrently. Each

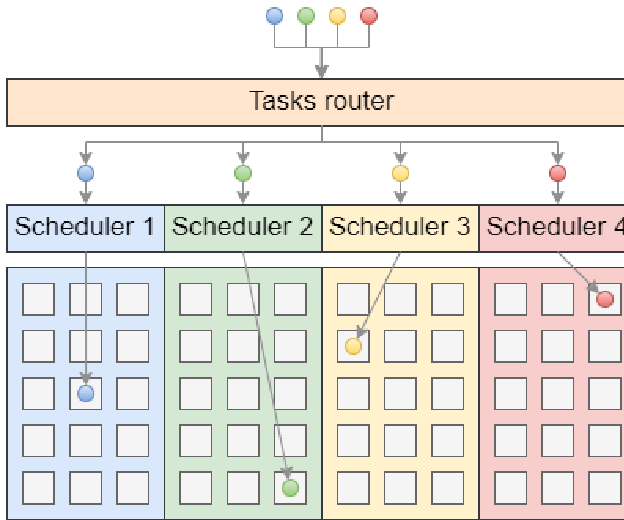


Figure 3.2: Partitioned scheduling scheme.

scheduler owns its infrastructure subset, and no other scheduler can schedule in that partition. Similar to the monolithic scheme, only infrastructure related changes may result in conflicting scheduling decisions.

On the other hand, each scheduler can only select some nodes as target for their scheduling components. While a node in a different partition may be a better fit for certain condition, a node with worse conditions for this specific component will be selected from the nodes in the corresponding partition. This may lead to sub-optional placement of application components in the whole infrastructure. This drawback is more evident with a higher number of partitions of the infrastructure.

The scalability issue is tackled by the partitioned approach. As the number of partitions increases, this scheme will be able to

handle a larger number of applications. Moreover, HoL blocking is reduced as several concurrent schedulers can make decisions.

Although the main drawback of the partitioned approach is the sub-optimal placement mentioned above, another issue is its lower adaptability, as partitions need to be defined statically.

3.3.3 Two-level scheme

The two-level scheduling scheme tackles the adaptability issue from the previous approach. Similar to the partitioned scheme, multiple schedulers handle separate subsets of the infrastructure nodes, but these nodes are not defined statically. A new system component, called resource manager, is in charge of assigning nodes to each scheduler, as seen in figure 3.3.

Different approaches for the resource manager are followed. In Mesos [30], the resource manager offers a set of resources to the scheduler, which picks among them. YARN (Yet Another Resource Negotiator) [31] can also be configured to follow the two-level scheme, where each scheduler negotiates resources with the centralized resource manager.

Similar to the partitioned scheme, there are no scheduling collisions in a two-level architecture, as resources are managed in a centralized component. Once the resources have been allocated to a scheduler, it owns them until they are released. Infrastructure related state changes can still result in conflicting scheduling decision, as in previous schemes.

While the resources of each scheduler are no longer static partitions, they still have access to a limited subset of the infrastructure. Despite scheduling components in the best possible node for that

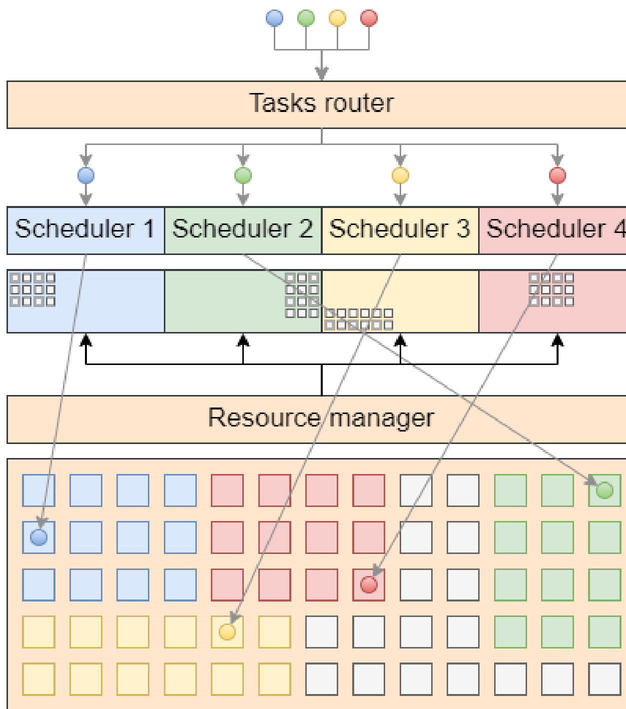


Figure 3.3: Two-level scheduling scheme.

subset, it still may not be the best possible node in the whole infrastructure. Therefore, this scheme also suffers from sub-optimal placement.

HoL blocking is solved in a similar way as in the partitioned scheme. Multiple concurrent schedulers considerably improve the scalability capability of the two-level scheme.

The two-level approach solves the statically defined partitions issue from the partitioned scheme, but its main drawback still remains: sub-optimal placement.

3.3.4 Shared state scheme

The previous two schemes only make scheduling decisions on nodes they own. Thus, they are certain that no scheduling collision can happen. The shared state scheme uses optimistic placement instead. Every scheduler places its components in any node in the cluster according to the state when the scheduling algorithm started. In case the state changes due to a scheduling decision from one of the concurrent schedulers, a collision may happen. This requires a new component, the collision solver, as seen in figure 3.4. The collision solver will make sure that the conflicting scheduling decisions are re-scheduled.

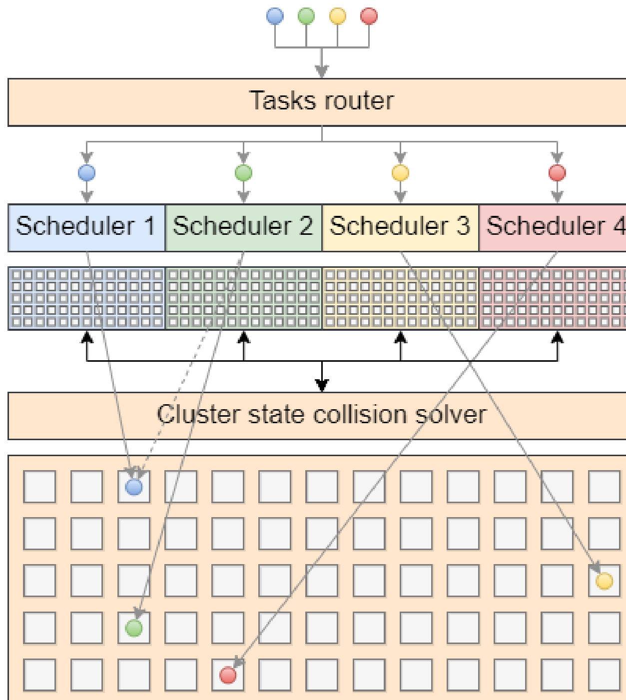


Figure 3.4: Shared-state scheduling scheme.

This scheme is followed by Omega [32], a COE used at Google internally, Apollo [33], a COE used by Microsoft internally, and Nomad [34], a container scheduler by HashiCorp.

Unlike the previous ones, this scheme may present scheduling collisions. Multiple concurrent schedulers may result in a set of scheduling decisions that exceed the available resources of some of the target nodes. This requires re-evaluating the scheduling algorithm.

Regarding the infrastructure, this scheme uses a single cluster without partitions. Any scheduler may assign components to any node in the cluster, therefore not suffering from the sub-optimal placement issue of the previous approaches.

Similar to the other distributed approaches, the HoL blocking issue is solved. The scalability of this scheme can be enhanced by increasing the number of concurrent schedulers.

The shared state solves both the HoL blocking from the monolithic approach and the sub-optimal placement issue from the partitioned and two-level schemes. Its main drawback is the introduction of scheduling collisions. These collisions require to run the scheduling algorithm again.

3.3.5 Distributed scheme

A fully distributed approach, as the one illustrated in figure 3.5, is also used by schedulers such as Sparrow [35]. Instead of solving scheduling collisions, components scheduled in a node without enough resources are queued and executed when enough resources become available.

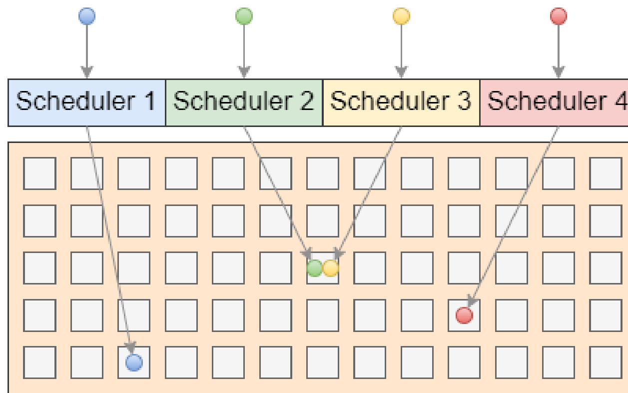


Figure 3.5: Distributed scheduling scheme.

This is only possible if the components that are being scheduled are short-lived. It relies on the fact that, even if more short-lived components are scheduled to a node than what it can handle concurrently, the components will eventually finish their job, allowing the rest of the scheduled tasks.

HoL blocking, sub-optimal placement or the need to solve scheduling collisions are issues that are not found in this scheme. Its main drawback is its limitation to short-lived components.

3.3.6 Comparison of scheduling schemes

The requirement fulfillment of each scheme has been summarized in table 3.1. The distributed scheme does not allow scheduling long-lived processes (R2). Therefore, this kind of scheme cannot be used for Edge to Cloud continuum-oriented applications.

The HoL blocking issue of the monolithic scheme negatively affects the scheme's scalability (R9). Furthermore, scheduling algorithms that are QoS-aware (R8) increase the complexity, and

consequently, the processing time required. This further exacerbates the HoL blocking issue of this scheme.

Both partitioned and two-level schemes split the infrastructure in different subsets, either statically or dynamically. This limits the nodes that can be selected as targets for each of the components of an application. This sub-optimal placement issue affects the QoS-awareness of the scheduling decisions (R8).

Table 3.1: Scheduler schemes requirement fulfillment.

Scheme	R2	R8	R9	Drawbacks
Monolithic	✓	✗	✗	HoL blocking
Partitioned	✓	✗	✓	Static partitions & sub-optimal placement
Two-level	✓	✗	✓	Sub-optimal placement
Shared state	✓	✓	✓	Scheduling collisions
Distributed	✗	✓	✓	Short-lived components only

The shared state scheme introduces scheduling collisions. These collisions need to be solved requiring to process the scheduling algorithm an additional time. However, this scheme enables the use of multiple concurrent schedulers with a single infrastructure partition. Therefore, the shared state scheme arises as the best fit for orchestration architectures that target the Edge to Cloud continuum, in order to achieve the long-lived components (R2), QoS-awareness (R8) and scalability (R9) requirements.

3.4 Related work

Several orchestration architectures with similar goals can be found in the literature. The most relevant ones are described in this section.

An orchestration architecture for the Fog using a hybrid approach is proposed in [36]. It highlights the heterogeneity of the infrastructure and divides it into three different layers: the Cloud level, the Fog level and the IoT level. It also provides some system components in charge of managing and monitoring this infrastructure, as well as mechanisms to customize orchestration in a per-application basis. It does not provide any network monitoring mechanism, nor considers these metrics when selecting the deployment, despite mentioning it as future work.

Another orchestration architecture for the Fog is presented in [37]. The heterogeneous infrastructure is characterized by the resources each node has. It provides different components to manage and monitor the infrastructure. However, scheduling decisions cannot be customized per application nor take into account network metrics.

A Fog computing orchestration framework is proposed in [38]. The infrastructure is partitioned in different "domains", each one with its own controller. Controllers monitor the latency among them, and these metrics are used in the offloading of segments of the applications to other clusters. However, it does not take this into account inside each "domain".

A similar approach that only considers network metrics among datacenters is used for the algorithm presented in [39]. Its goal is to optimize the number of applications that can be deployed in the infrastructure based on the required throughput of each application. It divides the infrastructure in a single Cloud and multiple Fog datacenters, and the applications in a Fog and a Cloud "chunk". While this approach targets network-related application quality of service, it lacks the generality in both the infrastructure and the

workload model required for an Edge to Cloud orchestration architecture.

Fogernetes, a Fog-oriented orchestration architecture based on Kubernetes, is presented in [40]. It uses a qualitative description of the infrastructure through labels, which specify information such as the location of the node, the layer it belongs to, or the capabilities of the node. These labels are also set as part of the requirements of each application component, and a simple match is performed to deploy these components. However, these labels are not being updated dynamically as the state of the infrastructure and workload evolves.

Another orchestration architecture based on Docker Swarm is proposed in [41]. An agent called OpenIoTFog is installed in every node in the system to provide infrastructure management and monitoring capabilities. However, this approach does not consider network metrics.

A parallel genetic algorithm to orchestrate Fog applications is proposed in [42]. It tackles scalability by partitioning the infrastructure. These partitions are merged and split again dynamically based on monitored data. It does not provide any detail on which metrics are being monitored or considered during scheduling, nor provides any mean to customize them in a per-application basis. It is compared with a non-parallel genetic algorithm, which it outperforms. However, it claims to still have scalability issues when increasing the number of total components.

A QoS-aware algorithm for the Fog is designed in [43]. It provides an infrastructure and workload model to characterize both nodes and the applications. The algorithm uses these models' data

to provide a set of potential deployments that accomplished their specified requirements. A tool is developed to execute this algorithm, but it does not manage the infrastructure nor monitors any of the metrics required for the provided models in order to be considered a full architecture.

Two more algorithms are developed in [44]. It describes the different system components required to orchestrate services in the Fog and provides the workload model to characterize the services. The algorithms use this model to orchestrate services taking into account the different QoS requirements. The performance of these algorithms is then evaluated.

A microservice orchestration for the Cloud-Edge continuum called Nautilus is described in [45]. Based on the application definitions, it groups components so that they are deployed in the same node to minimize the data transfer between them. These component groups are then deployed based on the available node resources. It also dynamically migrates these components from busy nodes to idle ones to guarantee QoS requirements. However, it does not consider the network state among nodes in the scheduling decision.

A capillary orchestration architecture is proposed in [46]. The infrastructure is divided in three layers: Cloud, Fog, and Edge. The main goal of the architecture is to offload containers from the Edge nodes to Fog or Cloud nodes when the first ones do not have the required resources available. For this purpose, it uses infrastructure and network metrics to select the Cloud Fog node where the container should be moved to. The scalability of the different system components is not considered.

An agent-based architecture is presented in [47]. The infrastructure is divided in three layers, where the Edge nodes offload their services to Fog or Cloud nodes based on each service's QoS requirements. This architecture is specifically designed for standalone services, and does not take into account the interrelations of the components in each application. The deployment decision is made by negotiation between the different agents.

All the above research efforts handle some of the identified requirements but none of them solves them all. The fulfillment of the identified requirements is summarized in table 3.2.

3.5 Application-Centric Orchestration Architecture

A novel orchestration architecture called ACOA, which stands for Application-Centric Orchestration Architecture, is designed to fulfill the identified requirements for an Edge to Cloud continuum-oriented architecture. It follows a modified shared state scheme, where the infrastructure is divided in two different planes: the control plane and the execution plane. The execution plane is where the different application components are deployed, while the control plane is in charge of managing the entire system. The control plane is further divided in two different layers: the system control layer and the application control layer. The application control layer is dedicated to application scheduling, while the rest of the system management tasks are performed in the system control layer, including state data storage or handling the communication among system components and with the users.

Table 3.2: Identified requirements fulfillment by related work.

Authors	**Ref.**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**
K. Velasquez et al.	[36]	✓	✓	✓	-	✓	✓	✗	✗	-
M. S. de Brito et al.	[37]	✓	✓	✗	✓	✓	✓	✗	✗	-
Y. Jiang et al.	[38]	✓	✓	✗	✓	✓	✓	✗	✓	-
F. Faticanti et al.	[39]	✗	✓	✓	✗	-	-	-	✓	-
C. Wöbker et al.	[40]	✓	✓	✓	✓	✓	✗	✗	✓	-
S. Hoque et al.	[41]	✓	✓	-	✓	✓	✓	✗	✗	-
Z. Wen et al.	[42]	✓	✓	✗	-	-	-	-	-	✗
A. Brogi et al.	[43]	✓	✓	✓	✓	✗	✗	✗	✓	-
J. Tsai et al.	[44]	✓	✓	✓	✓	✗	✗	✗	✓	✓
K. Fu et al.	[45]	✓	✓	✗	-	✓	✓	✗	✗	-
S. Taherizadeh et al.	[46]	✓	✓	✓	-	✓	✓	✓	✓	✗
Z. Nezami et al.	[47]	✓	✗	✓	-	✓	✓	✓	✓	-

When a node joins the cluster, it specifies which planes and layers it is part of (R5). There is no limitation on the number of planes or layers a single node can belong to. Big infrastructures with a large number of nodes will use specialized nodes for each plane and layer, while smaller infrastructures with a more limited amount of nodes will perform system management and execution tasks in the same node. This division in planes and layers allows for a better control on the resources used for each task type: system management, application scheduling and application execution.

This section starts by providing an infrastructure and workload model. The infrastructure model describes the nodes, and therefore the resources that ACOA will have available. The workload model characterizes the applications that will be orchestrated. ACOA will use this information to assign resources to each of the applications. All the system components needed for this orchestration are also listed, detailing their tasks and the interactions among them.

3.5.1 Infrastructure model

The infrastructure model depicted in figure 3.6 aims to represent nodes throughout the Edge to Cloud continuum (R1). It provides a generic definition that enables nodes to join the system by providing these data (R5). The whole infrastructure is modeled as a **cluster**, which contains several **nodes** and **links**.

Each **node** is identified by a unique name within the cluster, and is defined by a set of static and dynamic properties. On the one hand, static properties represent characteristics that do not change or, at least, do not change often, e.g., the operating system, whether a disk is encrypted, or the datacenter which the node

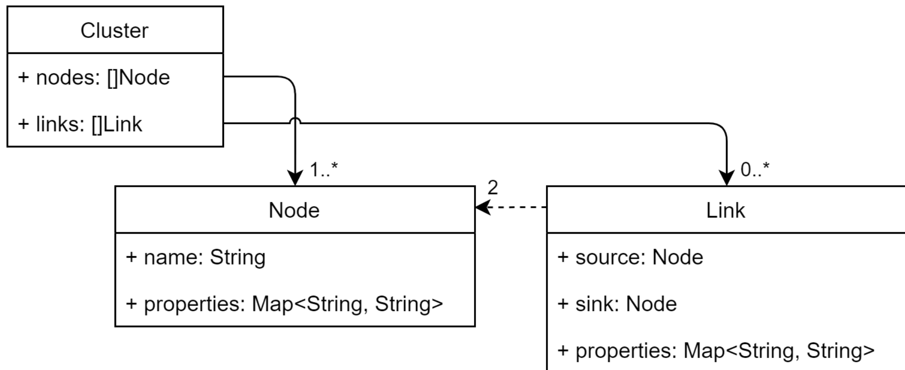


Figure 3.6: Infrastructure model.

belongs to. On the other hand, dynamic properties represent characteristics that experiment variations over time, such as the available memory or the CPU load.

The network is modeled by a set of network **links** between a source node and a sink node. The need of a **link** model is raised by the inclusion of the network state. The direction of the link is important to represent the network asymmetry, thus the total number of **links** is two times the square of the number of nodes. Each network **link** has a set of properties, e.g., latency, jitter, or the transmission success rate.

Dynamic properties, for both **nodes** and network **links**, need to be updated continuously to ensure that their values are up to date (R6 & R7). All these properties are required as inputs to the scheduling algorithm in order to make the optimal deployment decision.

3.5.2 Workload model

The workload model provides a full definition of the applications (R4) that will be executed in the infrastructure. It has been designed taking into account the heterogeneous applications that can be executed in the Edge to Cloud continuum, and their QoS requirements (R8). Figure 3.7 shows the **workload** model, a collection of **applications** that will be executed by ACOA.

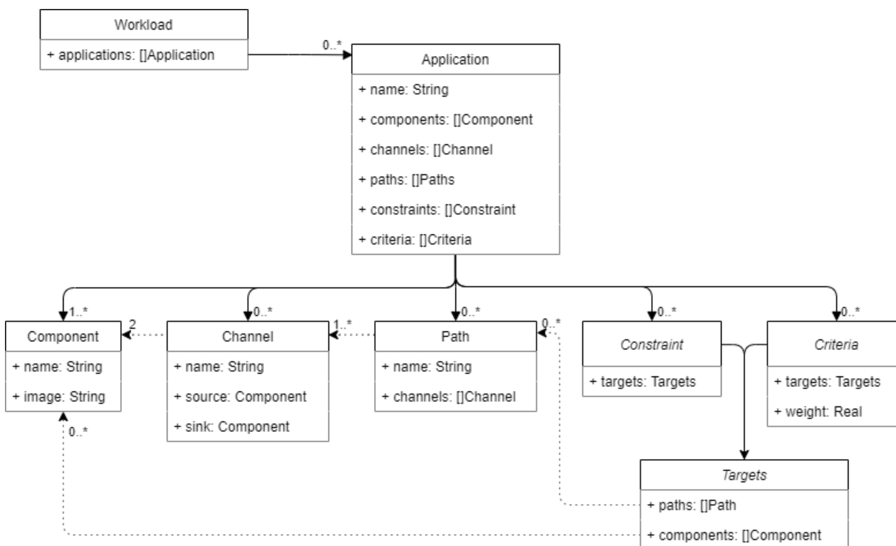


Figure 3.7: Workload model.

Multiple **applications** will be run by ACOA, so they need a unique name that will act as an identifier. They are composed of **components** which represent the executable pieces that will perform the application's tasks. The communication between two components is defined via **channels**. When the components are deployed to nodes, channels indicate which links are relevant for the application, as these links will be used to send messages. These channels are grouped in **paths** that will be used as targets by the

scheduling policies (**constraints** and **criteria**) in order to fulfill the application QoS requirements.

Components, **channels**, and **paths** require a name that will be used as a unique identifier. Additionally, **components** are implemented as containers, thus, the container image they use needs to be specified. **Channels** identify the source and the sink components, defining the direction of the messages, as the network cannot be assumed to be symmetric. **Paths** are characterized by the **channels** that join a set of **components**. Both **paths** and the whole **application** can be represented as DGs (Directed Graphs), where **components** are the graph's vertices and **channels** are the edges.

Two different kinds of policies are considered in order to customize the application scheduling (R3): **constraints** and **criteria**. **Constraints** filter the potential nodes where each component can be deployed. Several types of constraints have been defined: requiring a certain hardware (sensor/actuator) or software (database) component, or a certain amount of free disk space. Optimization **criteria** implement application QoS requirements by configuring how the exact node from the set of potential targets is chosen. Multiple optimization criteria can be provided with different weights to achieve the desired deployment behavior for each application separately. Several optimization criteria types have been defined: prioritizing the nodes with the smaller number of containers or the highest percentage of free memory, minimizing the end-to-end (e2e) response time for a certain path, or maximizing the message transmission success rate. As the architecture evolves, new policies, both constraints and optimization criteria, will be defined to represent other types of applications' QoS. Both **constraints** and

optimization **criteria** need to define their **targets**. They can be applied in a per-component basis or to subsets of the application (e.g., minimizing the e2e response time of an application critical path needs to define the critical path).

3.5.3 Architecture Components

The goal of ACOA is to use the resources characterized in the infrastructure model to execute the desired applications defined in the workload model. This goal is achieved by the combined effort of several system components. These system components can be divided in three categories:

- System control components: deployed in every node that joins the homonymous layer. There are three system components belonging to this category: the API server, the state database, and the system scheduler.
- Application control components: deployed in the corresponding layer as required by the desired workload. Application schedulers are the only system component that belongs to this category.
- System daemons: deployed in every node that joins the cluster, independently of which layers it is part of. There are two types of system components that belong to this category: node daemons and monitoring daemons.

Figure 3.8 illustrates these components and their location among ACOA's planes and layers. Only one instance of the three system control components is needed, while multiple application control

components (one scheduler per application) and system daemons will be present. In cases where greater resilience is desired, High Availability (HA) setups can be used. These setups specify multiple nodes as part of the system control layer, and therefore multiple instances of the system control components will be present in the system, as depicted with dashed lines in the figure.

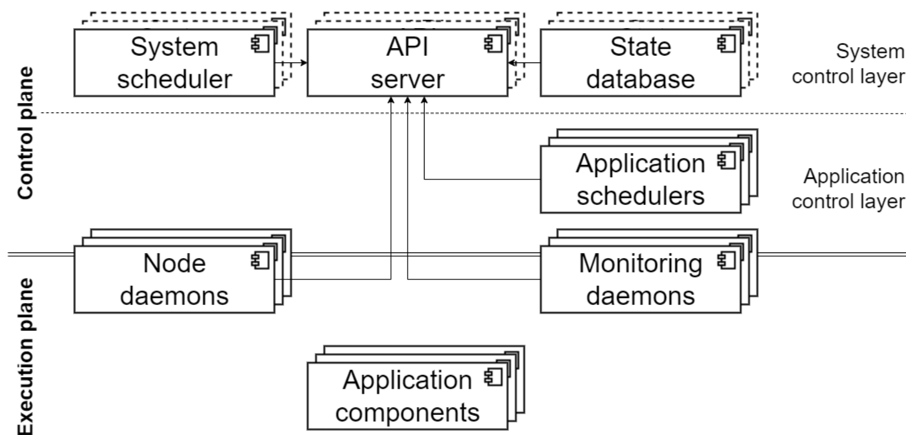


Figure 3.8: ACOA architecture.

The application definition from the users is stored in the state database through the API server according to ACOA's workload model. The data gathered by the system daemons is also stored in the state database through the API server. Node daemons provide static data about their corresponding nodes as part of the infrastructure model, as well as the status of the application software being executed in them. Monitoring daemons provide the dynamic data of the infrastructure models, both for nodes and links. System and application schedulers obtain the required information from the state database in order to perform the orchestration. The data flow among system components is illustrated in figure 3.9. All

these system components, and their contributions to ACOA's final goal, are detailed in the following sections.

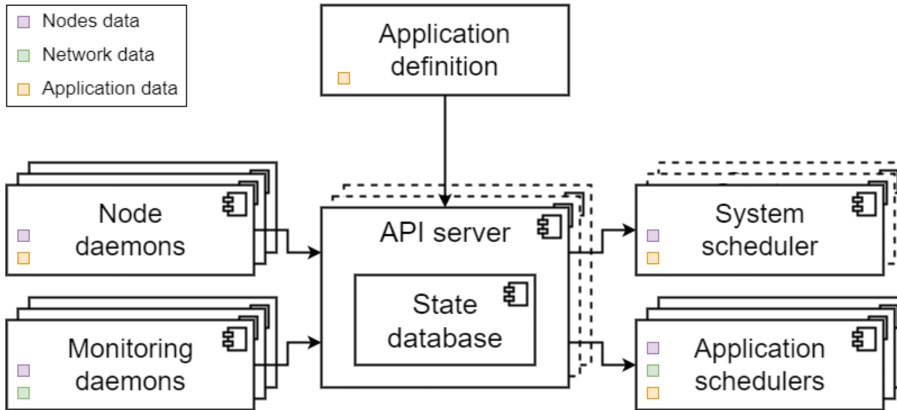


Figure 3.9: ACOA data flow.

3.5.3.1 API Server

The API server is a system control component that exposes a REST (Representational State Transfer) API used by the rest of the system components to interact among them. This interface is also used by the users to register their applications according to the workload model.

When the system control layer is implemented by several nodes, each of them will have an instance of the API server component. Placing a load balancer in front of all these API server instances will not only improve the resilience of the architecture, but also increase the scalability of this component (R9).

3.5.3.2 State database

All the system's data will be stored in the state database. This information comprises both the current state of the cluster as re-

ported by different daemons (R6 & R7), as well as the user-provided application definitions that represent the desired state. The end goal of the whole architecture is to reconcile both states, i.e., perform all the necessary actions so that the current state of the system complies with the desired state obtained from the user input. All this information is stored according to both the infrastructure and workload model previously defined.

If several nodes are part of the system control layer, multiple instances of the state database component will be present. By using a distributed database in order to store the state of the system, ACOA benefits from the failure tolerance and scalability (R9) guarantees that these databases offer.

3.5.3.3 System scheduler

The system scheduler's task is to determine the location of the application schedulers among the application control layer nodes. The application scheduler location does not require complex scheduling algorithms because it does not directly affect the QoS of the target application.

When multiple instances of the system scheduler are present in the cluster, only one of them is active while the rest are idle, ready to take the lead if the active one fails. This component is not a scalability bottleneck. It is only needed when new applications are registered by the user, and it only needs to schedule a single component, the application scheduler, among a limited set of nodes in the application control layer.

When the system scheduler is initialized, it requests the API server to be notified whenever the workload definition is changed.

Changes to the workload definition are applied through the API server. The API server identifies which changes need to be performed based on the workload state stored in the database and it updates the state database accordingly. These changes may consist of new applications being created or existing ones being modified or deleted. When an application is created, it requires a new application scheduler. This application scheduler is removed when the application is deleted. These two processes are illustrated in figure 3.10. Modifications to already existing applications do not require to create or remove the application scheduler.

If a new application scheduler needs to be created, the API server notifies the system scheduler. Based on the data of the state database, the system scheduler will select the target node (among the application control layer nodes) for the application scheduler. The system scheduler informs the API server about the selected node and the API server sends the corresponding order to the node's daemon which runs the corresponding container.

If the application scheduler needs to be removed, the API server sends the corresponding order to the selected node's daemon which stops the corresponding container.

3.5.3.4 Application schedulers

Application schedulers execute the scheduling algorithm customized for each application (R3). They use the information from the state database to execute any kind of scheduling algorithm, tailored to the QoS requirements of that specific application (R8).

Application schedulers are part of the application control layer; therefore, they are distributed among the application control layer

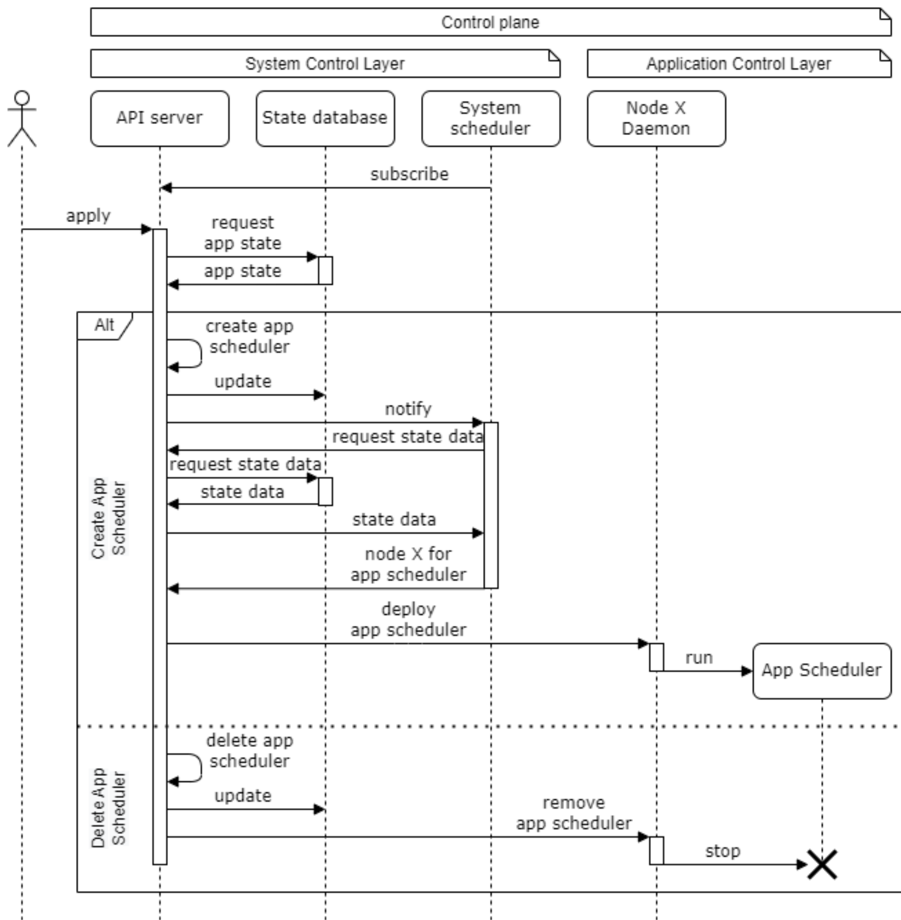


Figure 3.10: Management of application schedulers.

nodes. This enables a shared-state scheduling approach that improves the scalability of the architecture (R9). One of these schedulers will be running for each application. If considered necessary, multiple instances of each application scheduler can be used, but only one per application will be active while the rest remain idle. This should not be needed as application schedulers benefit from the resilience of any other deployed component. If the node where the application scheduler fails, all the components that were being

executed in the node, including the application scheduler, will be orchestrated again.

When the application scheduler is initialized, it requests the API server to be notified when its application definition is changed. Changes to its application definition are applied through the API server. The API server identifies which changes need to be performed based on the application state stored in the database and it updates the state database accordingly. These changes may consist of new components being created or existing ones being modified or deleted. When an application is created, all its components are created. When the application is deleted, all its components are removed. Modifications to already existing applications may need to create certain components that were previously not present and delete other components that are no further required. These processes are depicted in figure 3.11.

If a new application component needs to be created, the API server notifies the application scheduler. Based on the data of the state database, the application scheduler will select the target node (among the execution plane nodes) for the application component. The application scheduler informs the API server about the selected node and the API server sends the corresponding order to the node's daemon which runs the corresponding container. The scheduling algorithm is not executed again if the application state has not changed, since the algorithm selects target nodes for all the components in an application in a single execution.

If an application component needs to be removed, the API server sends the corresponding order to the selected node's daemon which stops the corresponding container.

3.5.3.5 Node daemons

Node daemons are in charge of managing the containers of their corresponding node. They are notified by the API server when new containers need to be executed or already existing ones need to be removed (R2). They also monitor these containers and report back their state to the API server, that stores them in the state database.

Additionally, node daemons also provide the characteristics of the node as part of the infrastructure model. This includes information like the OS, architecture, total RAM, or disk space. It also reads a configuration file that allows to specify additional data like the availability of a certain hardware sensor or actuator.

Node daemons are executed in every node in the cluster, independently of which plane they are part of. The system components are also containers, even if they do not need to be orchestrated, and therefore control plane nodes also need node daemons to manage their own containers.

3.5.3.6 Monitoring daemons

These daemons monitor the state of the system and report their measurements to the API server that stores them in the state database. This allows keeping an up-to-date representation of the current state of nodes and the surrounding network. Node state is defined by properties, such as the amount of available memory or the CPU load of a node (R6). Network state is collected by the SWIM-NSM protocol, providing latency, jitter, and packet-loss rate measurements for every pair of nodes (R7).

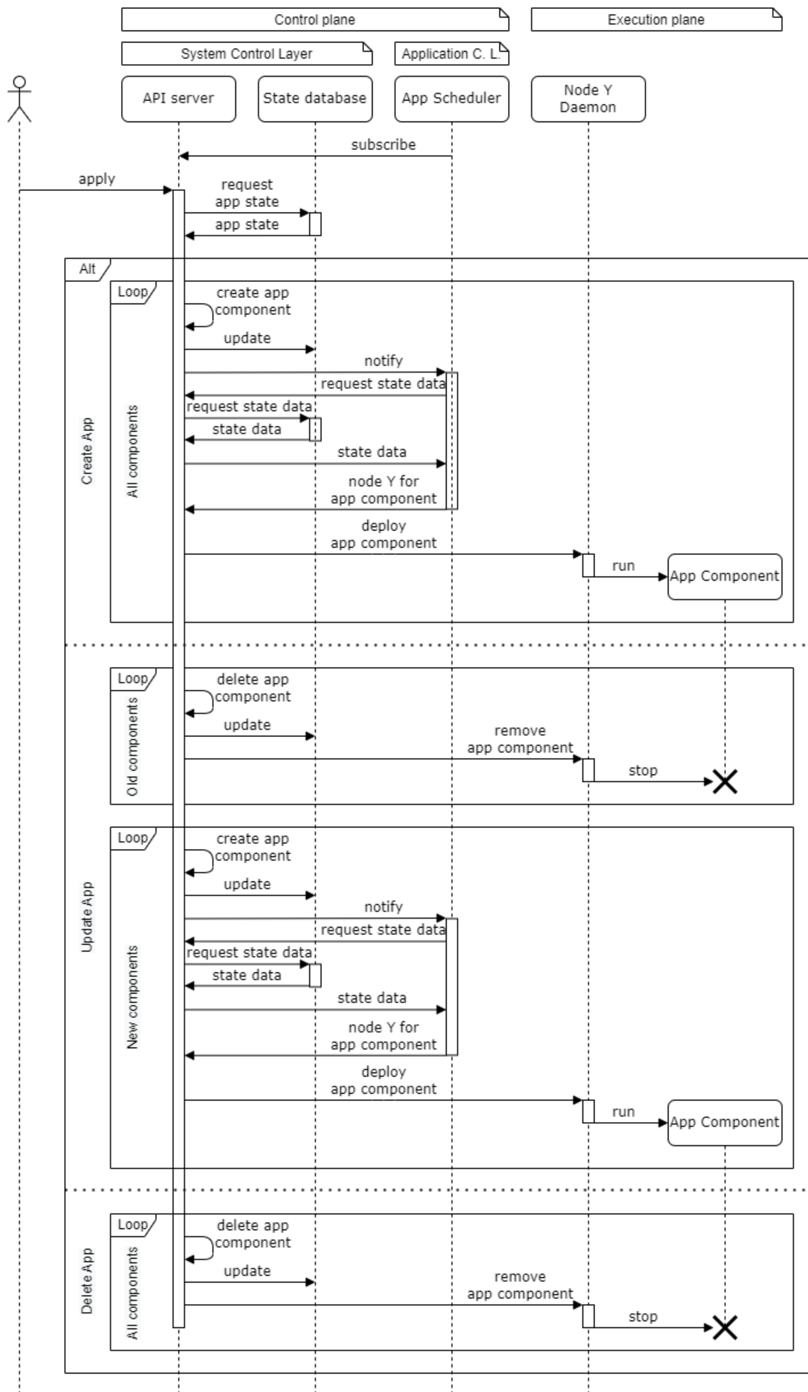


Figure 3.11: Management of application components.

Similar to the previous component, monitoring daemons are deployed in every node in the system, independently of which plane they are part of. All the nodes, and their surrounding network, need to be monitored.

3.6 Conclusions

ACOA is an orchestration architecture targeted to the Edge to Cloud continuum and its challenges. It follows the shared-state scheme, but the schedulers are dynamically deployed as new applications are deployed. It also provides an infrastructure and a workload model in order to characterize Edge to Cloud continuum nodes and applications.

The infrastructure model provides a single generic node model that can represent any node (R1), instead of several more specialized models as found in the related work. This enables any node to seamlessly join the cluster (R5). Dynamic properties enable the monitoring of both nodes (R6) and the network (R7).

The workload model allows the users to define the components, relations, and QoS requirements of each application (R3 & R4). This application definition will be used by the scheduling algorithm in conjunction with the monitored data from the infrastructure model to determine the optimal deployment for each component (R8).

The system components perform the system management tasks needed for the correct behavior of the orchestration architecture. Additionally, they also contribute to fulfill some of the identified requirements. The API server and the state database are repli-

cated on all the system control layer nodes in order to achieve a better scalability (R9) for HA setups. Scalability is also enabled by the shared state scheduling approach followed by application schedulers. These schedulers allow a per application customization (R3) and take into account the corresponding QoS requirements (R8). Their scheduling algorithms make use of the node and network metrics available in the state database. These metrics are kept up to date by the monitoring daemons (R6 & R7). Once orchestrated, the node daemon is in charge of executing the different application components (R2).

The participation of each model and component on the fulfillment of these requirement is summarized in table 3.3. The system scheduler, despite not participating directly in any of the requirements, it is needed to deploy the application schedulers.

Table 3.3: ACOA requirement fulfillment.

Requirement	1	2	3	4	5	6	7	8	9
Infrastructure model	✓				✓	✓	✓		
Workload model			✓	✓				✓	
API server									✓
State database						✓	✓		✓
System scheduler									
Applications scheduler			✓					✓	✓
Node daemon		✓							
Monitoring daemon						✓	✓		

CHAPTER 4

Implementation of ACOA

"Tell me and I forget.
Teach me and I remember.
Involve me and I learn."

- Benjamin Franklin

4.1 Introduction

In order to implement ACOA, already existing COEs will be considered first. The use of one of these engines as a base for ACOA's implementation allows benefiting from already implemented and thoroughly tested features. Therefore, the most used COEs have been reviewed in order to evaluate which one offers a set of features that better aligns with ACOA's requirements.

Once an already existing COE is selected, it needs to be extended to fully implement ACOA architecture. This includes extending both the infrastructure and workload models, and developing new system components. These new components have been implemented in Go, a modern programming language developed by Google whose first version was released in 2012.

4.2 State of the technology

Several COEs have been developed throughout the last decade. Kubernetes and Docker Swarm are the ones with a higher traction from the community [48, 49]. Therefore, these two COEs will be considered as a basis for ACOA.

Kubernetes [27, 50], usually shortened to K8s, is a COE that was initially developed by Google. It is based on previously container management tools used internally by the company called Borg and Omega. Departing from the more classic programming languages, Kubernetes was entirely developed using Go. The first version of K8s was released on July 21, 2015, as part of the Cloud Native Computing Foundation initiative.

Being a completely open-source project, Kubernetes' GitHub repository quickly became one of the most popular ones in the platform. This community contribution, in addition to the traction of several companies such as Google, IBM or Red Hat, has made Kubernetes a very flexible and customizable framework capable of handling a wide variety of workloads. Kubernetes is offered by all mayor Cloud providers such as Google (Google Kubernetes Engine, GKE [51]), Amazon Web Services (AWS Elastic Kubernetes Service, EKS [52]) or Azure (Azure Kubernetes Service, AKS [53]). The ecosystem around Kubernetes includes other COEs that are based on it, such as RedHat's OpenShift, or that can be integrated with it, e.g., Twitter's Mesos.

Docker Swarm [28] is the COE developed by the company of the most used container virtualization technology, Docker. This COE was released in 2016 and was also programmed in Go language.

Docker Swarm is also an open-source project. In comparison to Kubernetes, it aims to be easier to use. It is integrated with Docker and uses a similar user interface, simplifying the learning process for those that already used Docker to run containers in a single machine.

The following sections will provide a general overview of how Kubernetes and Docker Swarm work. Both share conceptual similarities, but they are implemented through different workload and infrastructure models, and a different set of system components. The workload model is the abstract representation for the user applications that need to be executed. The infrastructure model represents the physical nodes in the cluster. The system components are in charge of performing all the orchestration related tasks. These two COEs will be evaluated against the identified require-

ments in order to use one of them as a basis for the implementation of ACOA.

4.2.1 Kubernetes

Kubernetes does not provide explicitly separated infrastructure and workload models. All its objects are defined by five fields. The first two, `apiVersion` and `kind`, identify the type of the object. The third one contains `metadata` information, i.e., the name of the object. The last two fields, `spec` and `status`, are specific to each type of object, and they contain the characterization and runtime information of the object respectively. In order to create any object, the four first fields need to be provided, while the last one is created and updated by Kubernetes internally.

Kubernetes provides several abstractions that compose its workload model. The first of these abstractions is called Pod. Pods are a light wrapper over containers and are the scheduling unit used by Kubernetes. A Pod represents one or several containers that will be deployed to one of the nodes in the infrastructure. Aside from specifying which containers are part of it, additional information that needs to be taken into account by the scheduler can be specified, like node requirements.

These Pods are not usually directly defined by the end users. Instead, higher abstraction layers are provided by Kubernetes, which will be turned into Pods internally. These abstractions will have a `pod specification` field inside their own `spec` field, which will be used as the `spec` field for the Pods they create. The three most common abstractions are called ReplicaSets, Deployments and DaemonSets, as it can be seen in figure 4.1.

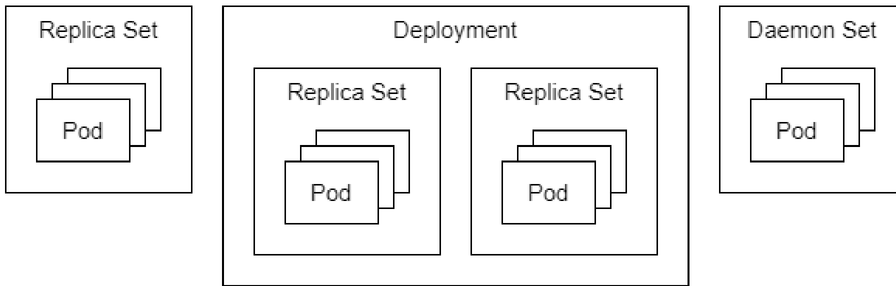


Figure 4.1: Kubernetes Replica Set, Deployment and Daemon Set.

ReplicaSets allow specifying the number of replicas of a certain Pod type that there should be present in the system. Kubernetes will create as many Pods as required and each one will be handled separately.

Deployments are an additional layer of abstraction over ReplicaSets, therefore requiring both the number of replicas and the pod specification. Deployments will create a ReplicaSet, but when a modification is performed on the pod specification, they will not modify that ReplicaSet. They will create a second one with the new specification and start migrating replicas from one to the other. The way they migrate can be configured, preventing downtime as the replicas are not stopped simultaneously.

DaemonSets define Pods that should be run in every node in the system. The pod specification must be provided. The Pods created from DaemonSets already have a fixed node where they should be run, so they do not require to be further scheduled.

Additional abstractions are provided to run containers as finite jobs. This aspect increases Kubernetes flexibility by enabling batch processing patterns to be executed, additionally to the more traditional microservice-oriented patterns followed by this engine.

Regarding the infrastructure model, Kubernetes only specifies a Node object. It represents a physical node in the cluster and its current state. Node objects are created and removed automatically when nodes join and leave the cluster.

Kubernetes divides the infrastructure into two separate planes: control and execution. The control plane contains the different system components required for the proper operation of the architecture. The execution plane is in charge of executing the workload that is deployed on the system. Each of these planes comprises several components that are distributed among several nodes. Figure 4.2 shows all the system components of both planes.

The execution plane, also called data plane, is formed by all the nodes that can be selected to run containers corresponding to the user defined workload.

The control plane is formed by a minority of the nodes, but their tasks are fundamental for the proper functioning of the whole architecture. Nodes that are part of this plane are usually dedicated to running the system components. However, for setups with a smaller number of nodes, these control plane nodes can also take part in the execution plane.

At least a single node is required to be part of the control plane, but HA setups with multiple nodes are recommended for bigger systems. Additional control plane nodes provide increased performance and resilience to the system components. Figure 4.3 illustrates an example of a HA setup with three control plane nodes.

There are four types of system components that are executed in the control plane: etcd database, API server, scheduler, and controllers. One instance of each of these components is executed in

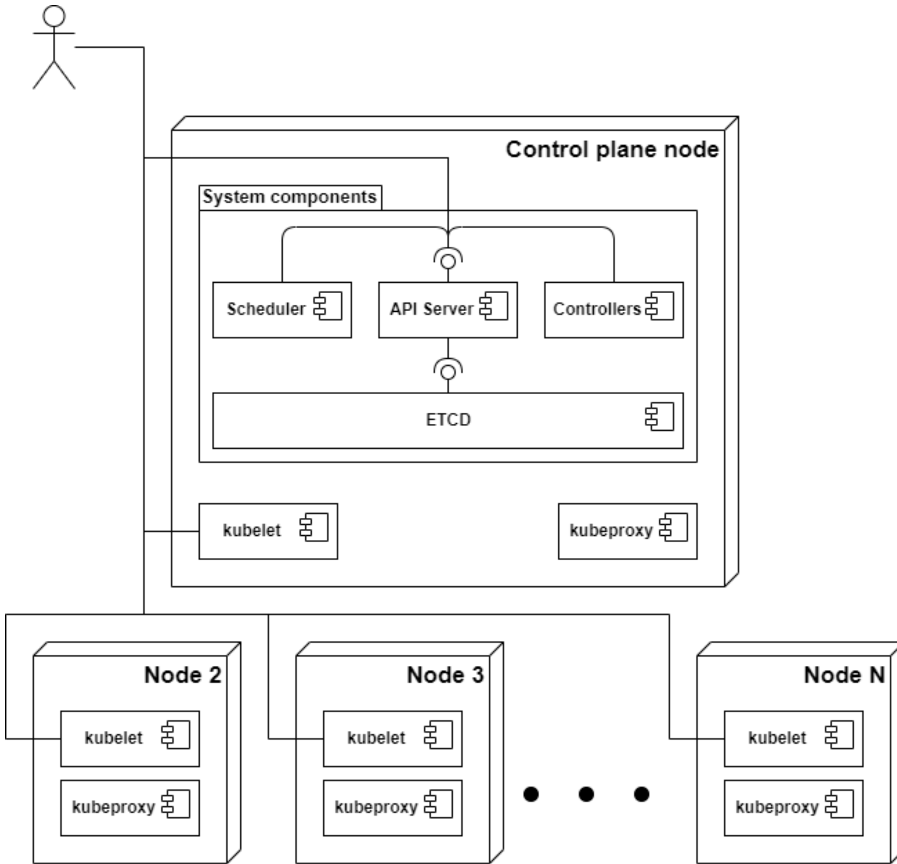


Figure 4.2: Kubernetes components.

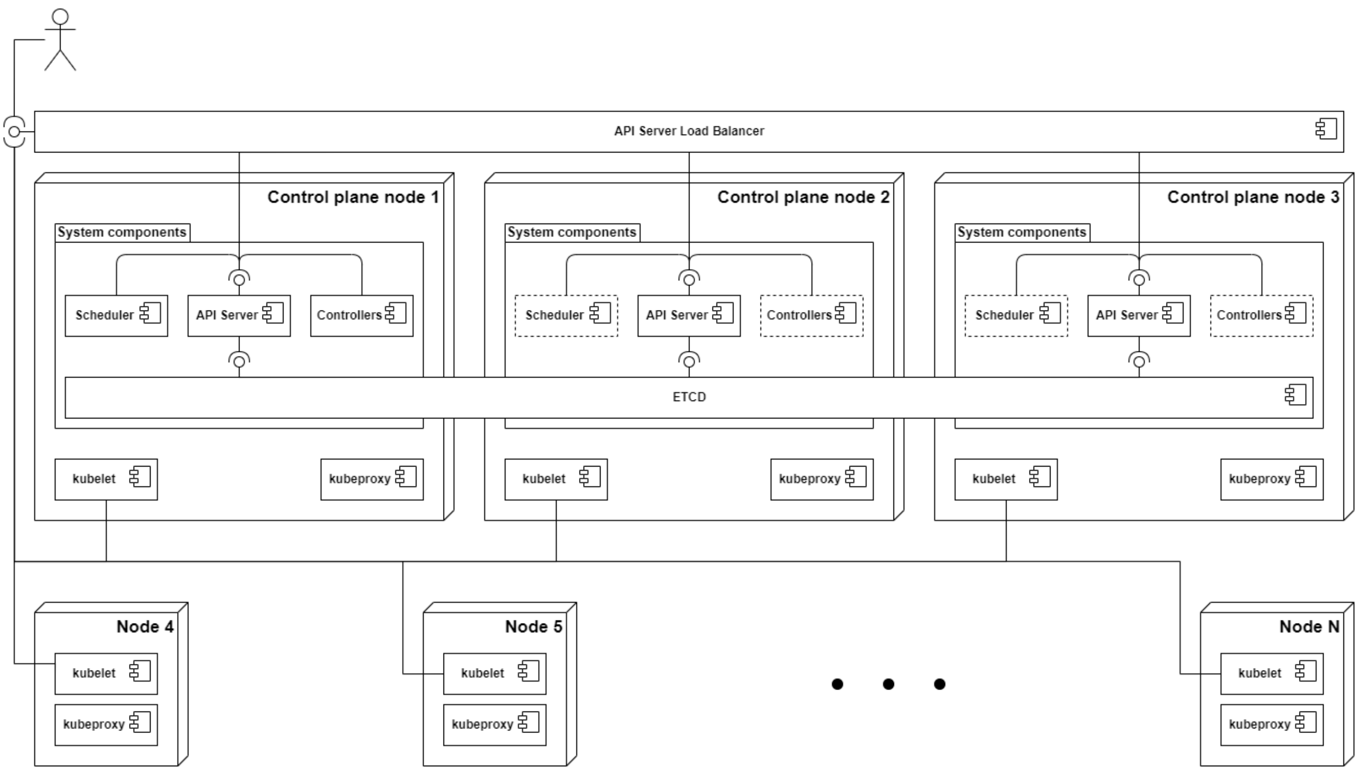


Figure 4.3: Kubernetes HA setup components.

each of the control plane nodes. HA setups may have an additional component: a load balancer in front of the API servers.

The etcd database is a distributed database that stores all the state data that K8s needs for its operation. It contains information about the workload that the user wants to execute in the system, the workload that is currently being executed or the different nodes that are part of the infrastructure. While etcd database is used by default, it can be replaced by other distributed databases. In setups with multiple control plane nodes, the multiple database instances allow the system to benefit from the capabilities of a distributed database, improving the resilience of the system.

The API server exposes the Kubernetes API. It handles the requests, both from users and other system components, and updates the state of the etcd database accordingly. Placing multiple API server instances behind a load balancer in HA setups increases not only the resilience of the system, but also its performance, as it can handle an increased number of simultaneous requests.

The scheduler determines the location where each Pod will be executed. In order to make these decisions, the default scheduler can take into account the available resources or the rest of the components that are also being executed in each node. If more complex scheduling algorithms are required, alternative schedulers can be developed. Kubernetes follows a centralized scheduling process, and uses a leader selection algorithm so that only one instance is running in HA setups. Additional instances from other control plane nodes will be in an idle state, ready to take the leadership if the current active instance can no longer perform its tasks. This

increases the resilience of the system, but it does not increase the performance of the scheduling process in HA setups.

The controllers are in charge of implementing the Kubernetes workload model. There is one controller per abstraction in Kubernetes model. They process the different objects that the users declare as part of their workload definition and transform them to Pods that the scheduler is able to understand. In order to extend this workload model, additional controllers can be developed. Similar to the scheduler component, controllers are centralized, using a leadership process to determine which instance is active and which are idle in HA setups with multiple instances of each controller. This results in a resilience increment without increasing the performance.

There are two additional system components that run in every node of the system, independently of which plane they belong to: the kubelet and kube-proxy.

The kubelet is in charge of supervising that its node is executing the Pods that were assigned to it by the scheduler. The scheduling decisions with all the required information to execute a container is sent from the scheduler to each kubelet through the API server. After that, it is the responsibility of the kubelet to supervise the proper functioning of each container, informing the control plane components of any non-compliance.

The kube-proxy task is to implement the virtual network that workload containers will use to communicate among themselves. This virtual network allows for easy communication among containers even for very complex infrastructure setups with geograph-

ically distributed nodes or over different cloud providers' data centers.

The collaboration of all these system components enables a declarative approach: instead of specifying which actions need to be performed, the user defines the desired workload, and the system is able to determine which actions need to be performed.

4.2.2 Docker Swarm

Docker Swarm does not provide explicit infrastructure or workload models. The definition of each object type requires specific fields.

Its workload model is based on tasks. A task is a light wrapper around a container and is the scheduling unit used by this COE. These tasks are not expected to be defined directly by the end users, Docker Swarm provides a higher-level abstraction called services. Two different service types can be defined by the user: replicated and global services. Replicated services allow specifying the number of instances of each container that need to be executed, and the policy to use when updating them. Global services represent containers that need to be run in every node in the cluster.

Regarding the infrastructure model, Docker Swarm provides a node object. These objects are created and deleted automatically. They contain all the relevant information about their physical counterparts. This COE also includes a swarm object, which represents the whole set of nodes that are part of the infrastructure.

In Docker Swarm, the infrastructure is divided in two types of nodes: manager and worker nodes. Manager nodes perform the system related tasks. Worker nodes execute the workload that is deployed on the system. By default, manager nodes also act

as worker nodes. The whole set of nodes is denoted as a swarm. Figure 4.4 illustrates these node types.

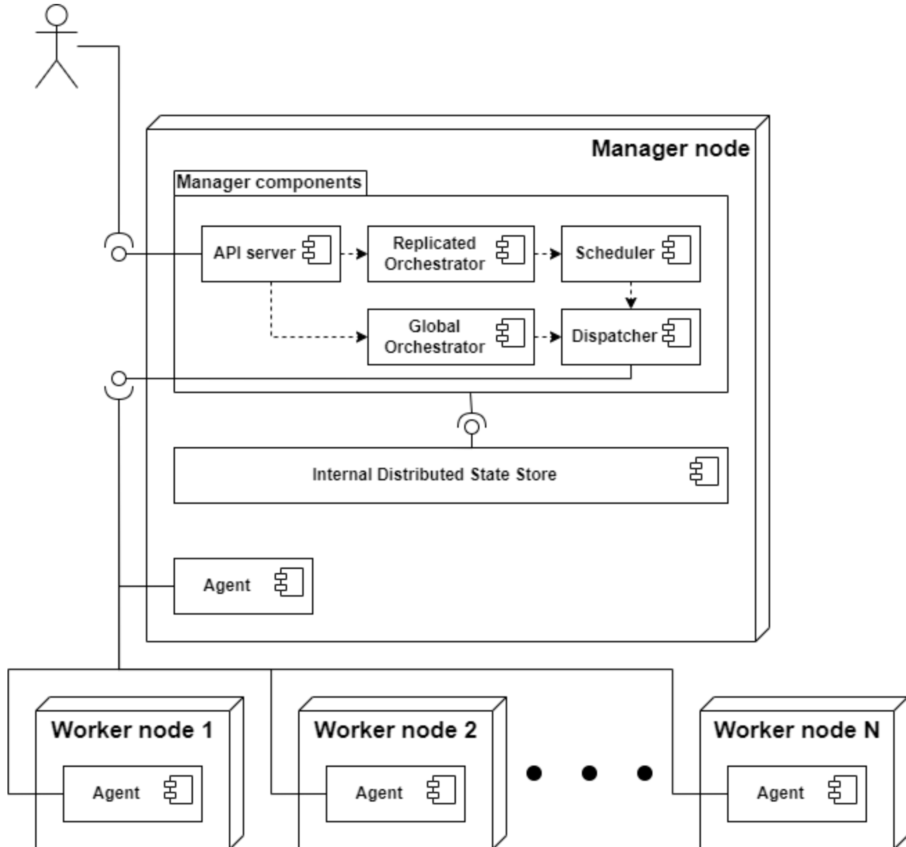


Figure 4.4: Docker Swarm components.

A swarm without any manager node alive still is able to run the ongoing workload, but it is unable to modify this workload or recover from failures. Deploying multiple manager nodes provides resilience as the system would tolerate the loss of some of the manager nodes without losing its capabilities. A swarm with multiple manager nodes uses a consensus protocol to elect a leader that will perform all the orchestration tasks. The rest of the nodes will act

as proxies for incoming requests and one of them will be elected as the new leader in case of a failure of the previous one. As a consensus is required, the optimal amount of manager nodes should be odd. The official documentation recommends keeping the number of manager nodes lower or equal to seven, as having multiple manager nodes hinders the performance of the system. Figure 4.5 depicts a swarm with three manager nodes. Communication with each worker node is done through a single manager node. When one of the manager nodes fails, the orphaned worker nodes are divided among the remaining manager nodes.

Five main component types are executed in manager nodes: state store, API server, one orchestrator per service type, scheduler, and dispatcher. HA setups may have an additional component: a load balancer in front of the API servers. As Docker Swarm uses a leader election mechanism among the available manager nodes, only the instances of these components in the leader are active, except for the state store and the API server. This increases the resilience of the architecture, but it does not affect the performance.

An internal distributed state store is maintained throughout all the manager nodes of a swarm based on a consensus protocol. This state store contains information about the desired workload as well as the current state of the system. As all the state is stored in every manager node, any manager node can take the place of a failing leader manager node, increasing the resilience of the system.

The API server exposes Docker Swarm's API to the users and other system components. In swarms with multiple manager nodes, the non-leader API servers act as proxies that forward the requests to the leader manager node, not affecting the performance of the system.

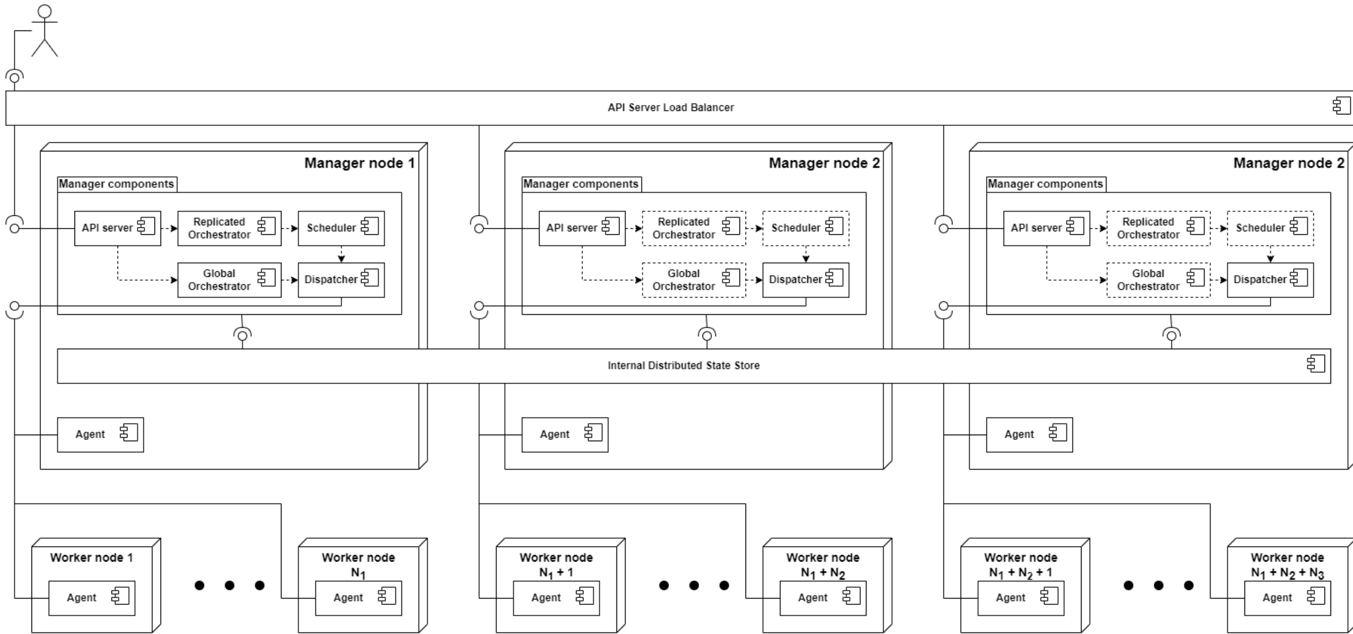


Figure 4.5: Docker Swarm HA setup components.

There are two different orchestrators, one for each service type. The replicated orchestrator creates all the different replicas of each container and prepares them to be scheduled. The global orchestrator also creates all the corresponding replicas.

The scheduler selects the worker node that executes each of the workload components defined by the user through replicated services. After filtering the nodes that have available the required resources, it selects the one with the smaller number of running containers among them. This way, it spreads the containers evenly among all worker nodes. In contrast with the replicated services, the global services are assigned a node by the orchestrator, thus not requiring to be processed by the scheduler.

The dispatcher is in charge of interacting with the worker nodes. It sends all the required containers' information to the nodes where they should be executed.

Every node, including managers and workers, require one additional system component: the agent. It receives the information from the dispatcher, executes the corresponding containers, and notifies any status update to the manager node.

All these system components handle the distributed execution of the workload defined by the user in the swarm.

4.2.3 Kubernetes vs Docker Swarm

Both Kubernetes and Docker Swarm provide similar tools as COEs. They are both open-source projects with a lot of traction from the community.

Kubernetes and Docker Swarm are supported on a wide variety of hardware systems (R1). Nodes can seamlessly leave and join the clusters (R5) and they will be monitored without any additional configuration (R6).

Both platforms support long-lived processes natively (R2). They both define their workload in YAML (Yet Another Markup Language) files. YAML is a markup language that focuses on having a human-readable format (R4), while still being specific enough for computers to process fast.

Kubernetes' scheduling algorithm can be customized by executing separate schedulers and specifying if these alternative schedulers should be used in a component basis (R3). Docker Swarm does not offer this customization level.

None of them monitors the network state connecting all the infrastructure nodes (R7), and therefore scheduling decisions cannot take into account the network state to optimize the QoS of the applications (R8).

Both platforms allow HA setups. These setups can handle a considerable number of nodes and applications. However, none of them takes this scalability requirement (R9) into consideration with the scheduling scheme they use, which is important in Edge to Cloud scenarios. They both use a monolithic scheduling scheme where a single process handles all the scheduling decisions, as the other ones are idle.

Kubernetes can be customized in multiple ways: from the underlying virtual network plugin that nodes use to communicate among them, to the database used to store the state. This high level of customization increases the complexity of the COE. In com-

parison, Docker Swarm focuses on ease of use as one of its main goals, reducing the customization capabilities of this platform.

Table 4.1 contains the identified requirement fulfillment of these two COEs. There are two aspects in Kubernetes that are specially relevant for the needs of ACOA: customizable scheduling and extendable infrastructure and workload models. Each pod definition can specify which scheduler should determine where the corresponding container should be executed. Additionally, the infrastructure and workload models of Kubernetes can be extended by developing new controllers which can implement abstractions such as applications or their components. The customization capability makes Kubernetes a better fit that Docker Swarm as a basis for ACOA's implementation, using all the deployment-grade tested features of this platform and focusing on developing the new ones required for the Edge to Cloud continuum.

Table 4.1: Kubernetes and Docker Swarm requirement fulfillment.

Requirement	Kubernetes	Swarm
R1 - Heterogeneous infrastructure	✓	✓
R2 - Long-lived components	✓	✓
R3 - Scheduling customization	✓	✗
R4 - Ease of use	✓	✓
R5 - Infrastructure management	✓	✓
R6 - Infrastructure monitoring	✓	✓
R7 - Network monitoring	✗	✗
R8 - QoS awareness	✗	✗
R9 - Scalability	✓/✗	✓/✗

4.3 ACOA implementation over Kubernetes

ACOA has been implemented on top of Kubernetes, by extending its features in order to fulfill the identified requirements. This includes:

1. Extending Kubernetes infrastructure model with the link abstraction.
2. Extending Kubernetes workload model with the application and component abstractions.
3. Extending Kubernetes architecture by splitting Kubernetes' control plane into two different layers and introducing new system components.

4.3.1 Extended Kubernetes infrastructure model

Kubernetes already provides a Node object, that can be used to implement ACOA's node model. Node's `metadata` already contains a `name` field that can be mapped to ACOA's node name. It also contains a `labels` field, which is a map of strings to strings that can be used to store ACOA's node properties. Node instances are automatically created and deleted when new nodes join or leave the system.

However, Kubernetes' infrastructure model does not provide a Link object, so it needs to be extended for this purpose. Links will also be automatically created when nodes join or leave the system.

Their `spec` field will store both the source and sink nodes, while the same `labels` field mentioned for nodes will be used to store its properties.

4.3.2 Extended Kubernetes workload model

The abstraction level provided by Kubernetes is insufficient to describe ACOA's workload model. It needs to be extended to follow the application-centric approach of ACOA. This requires the introduction of two new abstractions: application and component.

When an application is defined, it will create a Pod for its application scheduler and as many Components as needed. Components are not expected to be created by the user directly, they will be created automatically from the application definitions. Components will also create their corresponding Pod. The rest of the elements from ACOA's workload model do not require specific abstractions, but they need to be considered as part of the application properties. The extended workload model is depicted in figure 4.6, where the new introduced abstractions are shadowed in green.

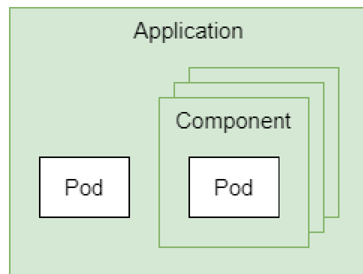


Figure 4.6: ACOA abstractions.

Applications are defined in YAML files (R4) with the syntax illustrated in code snippet 4.1. It follows Kubernetes syntax, where

the two first fields, `apiVersion` and `kind`, specify which kind of object is being created, and the next two, `metadata` and `spec`, provide the definition of the object.

Code snippets 4.1: Example YAML definition of an application in ACOA.

```yaml
apiVersion: acoa/v1
kind: Application
metadata:
  name: application-1
spec:
  scheduler:
    # ...
  components:
    # ...
  channels:
    # ...
  paths:
    # ...
  constraints:
    # ...
  criteria:
    # ...
```

Kubernetes `metadata` is common to all objects [54]. The only required field is `name`, a unique identifier. Other fields include `namespace`, to provide virtual separation between objects; or `labels`, a key-value pairs intended to specify identifying attributes. However, most of the fields inside `metadata` are updated by the system and are read-only from a user perspective, e.g., `creationTimestamp`, which contains the time when that object was created.

Each kind of object contains a customized `spec` field. There are six different fields in an application's `spec`: `scheduler`, `components`, `channels`, `paths`, `constraints`, and `criteria`. The first one allows providing configuration parameters for the application sched-

uler, while the rest are arrays and relate to ACOA's workload model.

The `scheduler` field allows to parameterize the application schedulers (code snippet 4.2). The implemented application scheduler reviews the selected deployment periodically (specified in seconds). If a new deployment with a better score is found, it modifies the previous deployment. In order to avoid constantly moving applications for minor improvements, a threshold is set. If the score of the new deployment does not improve the current one by that percentage, the application deployment will not be modified. The default values for these two parameters are 300 seconds and 5%.

Code snippets 4.2: Example YAML definition of an application scheduler configuration in ACOA.

```yaml
scheduler:
  period: 300
  threshold: 5
```

The `components` field is an array of `PodTemplateSpec` [55] (code snippet 4.3). It contains its own `metadata` field with a `name` and its own `spec` field with an array of `containers`. There are also other optional fields that can be used to further configure the pod that will be created for each component.

Code snippets 4.3: Example YAML definition of an application components in ACOA.

```yaml
components:
  - metadata:
      name: component-1
    spec:
      containers:
        - image: comp-1-container:1.0.0
  - metadata:
      name: component-2
```

```yaml
      spec:
        containers:
          - image: comp-2-container:1.0.0
    - metadata:
        name: component-3
      spec:
        containers:
          - image: comp-3-container:1.0.0
```

The `channels` array (code snippet 4.4) contains a list of objects with three fields, all of them mandatory: `name`, `source`, and `sink`. The first one is a unique identifier. The last two are the names of the component that sends the message and the one that receives it respectively. The next field, `paths` (code snippet 4.5), groups these channels by name. Defining the paths makes the system aware of the message flow that will happen at runtime, as required by some of the scheduler policies.

Code snippets 4.4: Example YAML definition of an application channels in ACOA.

```yaml
channels:
  - name: channel-1-2
    source: component-1
    sink: component-2
  - name: channel-2-3
    source: component-2
    sink: component-3
```

Code snippets 4.5: Example YAML definition of an application paths in ACOA.

```yaml
paths:
  - name: path-1-3
    channels:
      - channel-1-2
      - channel-2-3
```

The `constraints` and `criteria` arrays allow customizing the behavior of the scheduler process (code snippet 4.6). Both have a `target` field that can be either an array of `components` or `paths`. They also have a `type` field to specify which kind of constraint or criteria should be applied and an optional `value` field that can contain further information specific to each kind. Additionally, criteria have a `weight` field to specify the importance of each of them. The following types of constraints and criteria have been implemented, and can be further expanded as needed:

- `node` (constraint): the value field specifies the name of the node where the target should be executed. The target is usually a single component, but multiple components or paths can also be provided, in which case it will be applied to all the related components.

- `require-label` (constraint): nodes that contain the specified label are considered electable to execute the specified target. If an optional value is provided, the label's value must also match it.

- `avoid-label` (constraint): nodes that contain the specified label are filtered out from the electable nodes. If an optional value is provided, the label's value must also match it to be filtered out.

- `e2e-response-time` (criteria): expects a single path that forms a DAG as the target, and optimizes the latency between the nodes containing those components.

- `e2e-reliability` (criteria): expects a single path that forms a DAG as the target, and optimizes the success rate between the nodes containing those components.

Code snippets 4.6: Example YAML definition of an application constraints and criteria in ACOA.

```
constraints:
  - targets:
      components:
        - component-1
    type: node
    value: node-1
criteria:
  - targets:
      paths:
        - path-1-3
    type: e2e-response-time
    weight: 1.0
```

4.3.3 Extended Kubernetes architecture

ACOA's infrastructure is divided into two planes: control and execution. The control plane is further divided into two layers: system and application. Kubernetes also has the two planes separation, but the control plane layers are specific of ACOA. This layer separation improves the scalability of the scheduling process, which is one of the requirements that Kubernetes does not fulfil (R9).

Some of Kubernetes system components directly relate with ACOA system components, but other components need to be implemented. This relationship is summarized in table 4.2.

- The **API server** is implemented by Kubernetes API server and controllers. Controllers for link, application and compo-

nents have been developed to extend Kubernetes' infrastructure and workload models with these new abstractions.

- Kubernetes' etcd database is equivalent to the **state database** in ACOA.
- Kubernetes' scheduler is used as the **system scheduler** in ACOA.
- **Application schedulers** do not exist in Kubernetes. They will be dynamically deployed by ACOA as Pods in the application control layer nodes.
- Kubernetes' kubelets act as the **node daemons** in ACOA.
- Kubernetes' kubelets also perform **monitoring daemons'** tasks by monitoring the node. Additional monitoring daemons are deployed in all nodes to measure the network state using SWIM-NSM.

Table 4.2: Kubernetes extended components for ACOA.

ACOA	Kubernetes	New
API server	API server and controllers	Additional controllers
State database	etcd	-
System scheduler	Scheduler	-
Application schedulers	-	Deployed dynamically
Node daemon	kubelet	-
Monitoring daemons	kubelet	SWIM-NSM daemon

All these components are illustrated in figure 4.7. The new components, identified by the green background, have been imple-

mented in Go, and the corresponding container images have been created. The corresponding ACOA's components have been shadowed in blue, and planes and layers have also been delimited with dashed lines.

Schedulers and controllers in Kubernetes use a subscription mechanism. They inform the API server which abstraction they are watching, and when any of the instances of these abstractions are changed, they get notified. They then determine if any action needs to be taken. This is the approach followed by the link controller, application controller, application schedulers and component controller.

The link controller subscribes to changes to nodes, which include their creation and deletion. When a new node is created, the required link instances are created. This includes links with all the nodes in the system, including itself, and for both directions. When a node in deleted, all the links that have it as a sink or a source are automatically deleted.

The application controller subscribes to changes to applications. When a new application instance is created, the application controller creates the pod instance for the corresponding application scheduler. It constrains this pod to the application control layer nodes. The system scheduler will notice this pod and schedule it accordingly. The application controller is also in charge of updating the corresponding component instances when an application is created, modified, or deleted.

The application schedulers subscribe to changes in their corresponding components. It executes the customized scheduling algorithm for each application based on the metrics measured by the

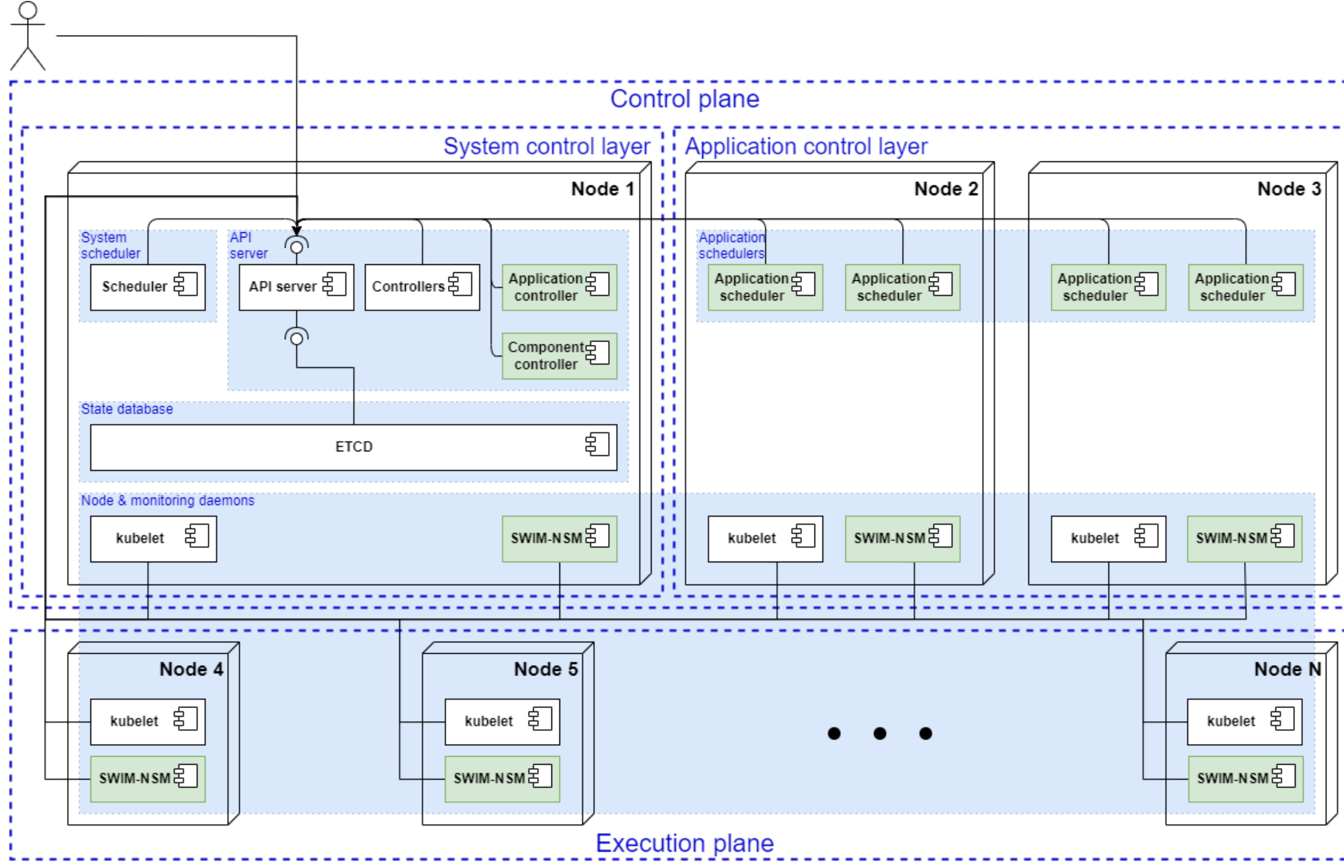


Figure 4.7: UML component diagram of ACOA architecture over K8s.

monitoring daemons (R8). It modifies the component instances to include the target node location.

The component controller also subscribes to changes in components. Once component instances are assigned a target node by the application schedulers, it creates the corresponding pod instances with the target node already defined. These pods do not require the intervention of the system scheduler as they were already scheduled by the corresponding application scheduler.

The above-described process needs to take into account application QoS (R8), which requires the collection of node and network metrics. Node metrics are already collected by the kubelet, while network metrics are collected by a new monitoring daemon. It is generated by the SWIM-NSM library described in a previous chapter, keeping an up-to-date list of the latency, jitter, and success rate of all the links in the cluster (R7).

4.4 Conclusion

Kubernetes and Docker Swarm both offer thoroughly tested features that fulfill some of the requirements identified for the Edge to Cloud continuum. They lack network monitoring capabilities (R7) and QoS-aware scheduling algorithms (R8). They both offer HA setups, but they do not consider scalability (R9) for the scheduling process.

ACOA has been implemented on top of Kubernetes, as it offers more customization capabilities than Docker Swarm. Kubernetes infrastructure model has been extended to make it network aware. The workload model has also been extended in order to improve the

scheduling customization capabilities (R3) and to provide a higher abstraction level for the users (R4). Three new abstractions (link, application, and component), and their corresponding controllers, have been introduced for this purpose.

The separation of the control plane in two layers and the inclusion of new system components allow fulfilling the remaining requirements. The inclusion of the daemon implementing the SWIM-NSM protocol allows monitoring the network state of the system (R7). The application schedulers make use of these metrics to implement scheduling algorithms that take into consideration the QoS of the applications (R8). The shared state approach of these application schedulers in the application control layer provides a better scalability of the scheduling algorithm for the Edge to Cloud continuum (R9).

CHAPTER 5

ACOA assessment

"You need to assess yourself on a yearly basis and see how far you have gone and what you still need to work on."

- Sunday Adelaja

5.1 Introduction

ACOA has been designed in order to fulfill the nine identified requirements. Most of them can only be validated in a qualitative manner exception for the scalability requirement (R9). This chapter evaluates ACOA and compares its behavior with a non-application-centric approach.

First, the scalability improvements brought by the use of a shared-state scheme are evaluated. The mean deployment time required for hundreds of software components (containers) by different number of application schedulers is analyzed.

Next, a use case in the transportation vertical domain is used to assess ACOA's suitability to orchestrate Edge to Cloud continuum-oriented applications. ACOA is compared to an orchestration architecture that does not have into account the application QoS requirements. Kubernetes has been used for this comparison, but similar results are expected for other COEs that are not application QoS aware.

Finally, the influence of the number of components in a path and the network variability in the improvement provided by ACOA is theoretically obtained. These theoretical bounds are compared with the empirically obtained data from the above use case.

5.2 Shared-state scheme validation

ACOA uses a shared-state orchestration scheme where multiple schedulers orchestrate the workload simultaneously, one per application. This enables the application-centric approach of ACOA by providing customization capabilities in a per application basis.

Additionally, the use of multiple schedulers prevents the HoL blocking issue present in architectures with a single scheduler, resulting in better scalability. In order to assess that scalability improvement, the time required to orchestrate and deploy several containers has been measured. This time interval is considered since the containers are defined by the user until they are assigned the node where they will be executed.

This assessment has been carried out for one, five and ten schedulers. Aside from the number of schedulers, there is no other difference. The same scheduling algorithm is used to deploy exactly identical containers, so that the obtained results can be used to validate if there is a time reduction and if it is relevant.

The mean deployment time for different amounts of containers has been measured. Figure 5.1 illustrates the reduction in the mean time required to deploy each component when using multiple concurrent schedulers. The ten-scheduler case presents a reduction of the mean deployment time compared to the single scheduler case between 37% and 67%.

The existence of multiple concurrent schedulers also introduces the possibility of collisions. A collision happens when two schedulers make parallel decisions that cannot be satisfied at the same time. A collision requires one of the affected schedulers to make a new decision. The frequency of these collisions during the deployment was also measured. The resource utilization of each of the deployed components was fixed so that the cluster utilization in each test was 85%, 90% and 95%. For a total of 1500 components deployed, no collision was found for the 85% utilization case, a single collision (0.06%) was found for the 90% utilization case with ten schedulers and the 95% utilization case with five schedulers,

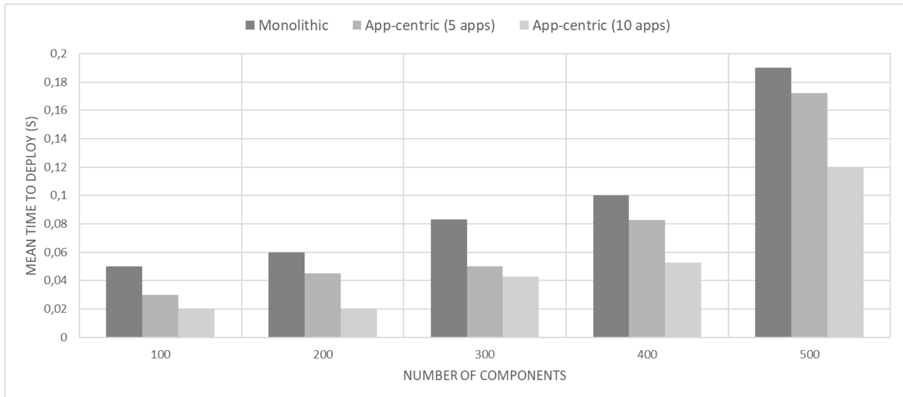


Figure 5.1: Deployment time depending on the number of schedulers.

and two collisions (0.13%) were found for the 95% utilization case with ten schedulers, as it can be seen in table 5.1.

Table 5.1: Collisions during component deployment.

Cluster utilization	**85%**		**90%**		**95%**	
Schedulers	**5**	**10**	**5**	**10**	**5**	**10**
100 components	-	-	-	-	-	-
200 components	-	-	-	-	-	-
300 components	-	-	-	-	1	-
400 components	-	-	-	1	-	2
500 components	-	-	-	-	-	-
Percentage	0%	0%	0%	0.06%	0.06%	0.13%

5.3 Railway use case

The transportation vertical domain has been selected as the use case scenario to assess ACOA, more specifically, the railway sector.

The railway sector presents a geographically distributed infrastructure with different size nodes that offers a good example

of which kinds of sectors would benefit from an Edge to Cloud continuum-oriented architecture. This infrastructure comprises from resource constrained nodes on board of each train or in every station to powerful devices in the company's headquarters or in Cloud providers' datacenters. Combining all these resources into a single cluster provides a good example of an Edge to Cloud continuum infrastructure.

The applications in this sector are also very heterogeneous: from simple applications on board of every train to complex algorithms that take into account the data from the whole architecture. Closed doors verification, smoke detection, ticket and identity verification, or train speed profiling are some examples of these applications. Each of these applications has different QoS requirements that can be considered by ACOA. For this use case, a simple smoke detection application inside a train and a more complex speed profiling one have been selected to demonstrate the viability of ACOA.

The following sections describe the specific infrastructure and workload used for this use case. Finally, the deployment configuration results with ACOA are compared to an orchestration architecture that does not take into account the application QoS requirements.

5.3.1 Infrastructure description

The railway sector has computing devices that can be part of an Edge to Cloud continuum cluster spread among several locations. Trains, stations, central headquarter offices and even Cloud providers can contribute to the cluster infrastructure. Therefore, the infrastructure is spread among a large geographical area. Even

in local railway transportation, nodes are distributed throughout a city, and long-distance railway networks can extend one or even several countries.

Inside a train, several devices can be found: the visualization dashboards for the driver, and telemetry or security data acquiring nodes are some examples. These nodes are usually resource constrained, but they can answer the application demands very quick as they are located in the Edge part of the continuum.

Nodes can also be found in stations, such as ticket validation or dispensing machines, schedules dashboards or computers in information/security offices. The computing resources available in stations usually are higher than those found in a train, but they are still placed close to the physical world. From the train applications point of view, they can be considered as Fog devices placed towards the middle of the Edge to Cloud continuum.

The most powerful devices owned by the company are placed in the central company headquarters. They can execute complex algorithms and store high amount of data, being positioned towards the most distant part of the Edge to Cloud continuum. In some cases, the ownership of these datacenters may be delegated to Cloud providers, removing the need to own and maintain them.

For the following use case, a down-scaled version of such an infrastructure has been used. A cluster of 20 ACOA nodes has been provisioned for these applications. Each of them is running k3s, a certified lightweight implementation of Kubernetes 'built for IoT & Edge computing" [56]. These 20 nodes have been used to simulate a multi-datacenter setup with 5 different locations, as illustrated in table 5.2.

Table 5.2: Emulated railway infrastructure.

Location	Nodes
Headquarters	1–6
Cloud provider	7–12
Train 1	13–15
Train 2	16–18
Station	19–20

The first six nodes represent the devices in the headquarters and the following six are virtual machines from a Cloud provider. Nodes 13-15 and 16-18 are placed on two different trains, while the last two nodes are placed in a station. The location of each of the nodes in each train is also relevant for the current use case. The first node in each train represents the security data gathering node (13 & 16), the driver's dashboard is represented by the second node (14 & 17), and the third node is in charge of gathering the telemetry data (15 & 18). These nodes have access to specific hardware needed for the applications of this use case. All nodes have been appropriately labeled in the configuration file that node daemons read, according to table 5.3, so that application schedulers are aware of their capabilities.

Table 5.3: Train node labels.

Label	1-12	13	14	15	16	17	18	19-20
train-1		✓	✓	✓				
train-2					✓	✓	✓	
fast-processor	✓							
database	✓							
security-sensors		✓			✓			
dashboard			✓			✓		
telemetry				✓			✓	

Nodes 1-3 are configured as part of the system control layer, while nodes 4-6 are part of the application control layer. All nodes, including these first six, are part of the execution plane. This setup guarantees that all layers have enough resources to perform their tasks while they still provide resilience to the system control components.

Despite simulating a multi-datacenter setup, the 20 provisioned nodes are physically connected to the same Ethernet switch. Therefore, the actual measurements of the SWIM-NSM daemon do not properly characterize the desired simulated datacenters. Instead, a simulated network monitoring system has been used for the purpose of this assessment. The simulated latency and success rate values are randomly generated, but taking into consideration which location each node in the link belongs to. The lower and upper bounds for these generated values have been selected based on previous execution of the SWIM-NSM protocol, and are summarized in table 5.4. As components of the same application can be placed in the same node, the metrics of a node with itself must also be taken into account. A very small latency and perfect success rate are considered for this case, as message exchanges do not even need to reach the network. A slightly higher latency and a good success rate are considered for nodes that belong to the same location, e.g., two nodes in the same train. A higher latency and lower success rate are used for links that span nodes in two different locations.

5.3.2 Workload characterization

Smoke monitoring and speed profiling applications have been executed in the previously described infrastructure. The two selected applications have very different characteristics in order to repre-

Table 5.4: Network metric bounds.

Target	Latency	Success Rate
Same node	0.25–0.35 ms	100%
Same location	0.8–1.2 ms	95–100%
Different location	15–25 ms	85–95%

sent the heterogeneity of the applications that can be deployed in the Edge to Cloud continuum. They present different QoS requirements that are considered by ACOA during the orchestration process.

This section describes both applications, and provides the YAML definition used to characterize them.

5.3.2.1 Smoke monitoring application

The smoke monitoring application takes smoke concentration measurements from the first train. These measurements are used to raise an alarm in the driver's dashboard if they go over a certain threshold, and they are also stored in a database after being processed. Despite being a simple application, it includes preprocessing, historic data storage in the Cloud and short deadlines for alarm triggering, illustrating some of the requirements of applications targeted towards the Edge to Cloud continuum.

The application has been implemented with five components as depicted in Figure 5.2. The first component (*measurement*) reads the measurements of the smoke concentration from the sensor for the next two components. Component two (*batching*) groups multiple measurements and computes deltas to reduce the memory footprint required to store the data. The third component (*level*

detection) triggers events when the values go above or below certain thresholds. Component four (*storage*) stores the data and the events into a database. Finally, the fifth component (*alarm*) fires an alarm to inform that the configured threshold was surpassed.

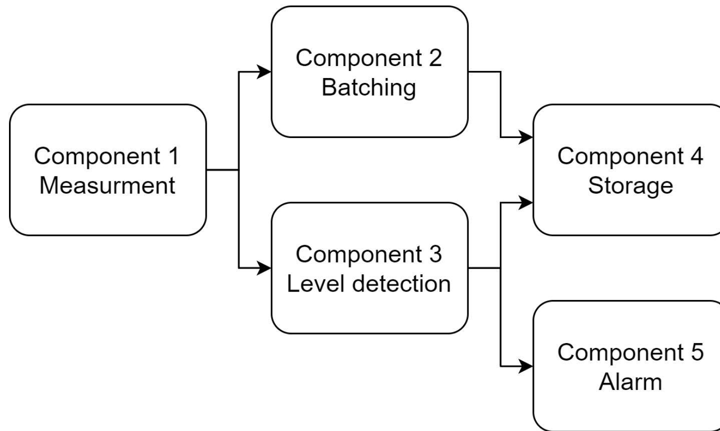


Figure 5.2: Smoke monitoring application components.

This application is defined in a YAML file according to ACOA's workload model and then registered through the API server. Code snippet 5.1 provides the basic skeleton of this file, where each fragment is being detailed hereunder. The default period and threshold parameters for the application scheduler have been used, so that fragment has been omitted.

Code snippets 5.1: YAML file structure for the smoke monitoring application.

```
apiVersion: acoa/v1
kind: Application
metadata:
  name: smoke-monitoring
spec:
  components:
    // ...
```

```
channels:
  // ...
paths:
  // ...
constraints:
  // ...
criteria:
  // ...
```

The five different components that take part in this application need to be defined in their corresponding fragment. The workload model requires each component to specify a unique name and the image of the container that implements it. Code snippet 5.2 provides these values for this application.

Code snippets 5.2: YAML file fragment for the components definition of the smoke monitoring application.

```
components:
  - metadata:
      name: measurement
    spec:
      containers:
      - image: smoke-monitoring-measurement:1.0.0
  - metadata:
      name: batching
    spec:
      containers:
        - image: smoke-monitoring-batching:1.0.0
  - metadata:
      name: level-detection
    spec:
      containers:
        - image: smoke-monitoring-level:1.0.0
  - metadata:
      name: storage
    spec:
      containers:
        - image: smoke-monitoring-storage:1.0.0
  - metadata:
      name: alarm
```

```yaml
spec:
  containers:
    - image: smoke-monitoring-alarm:1.0.0
```

The workload model also requires defining the exchanged messages between the components, by providing channels with a unique name, and source and sink components. The messages exchanged in this application follow the arrows present in figure 5.2, and this information is specified as seen in code snippet 5.3.

Code snippets 5.3: YAML file fragment for the channels definition of the smoke monitoring application.

```yaml
channels:
  - name: channel-1-2
    source: measurement
    sink: batching
  - name: channel-1-3
    source: measurement
    sink: level-detection
  - name: channel-2-4
    source: batching
    sink: storage
  - name: channel-3-4
    source: level-detection
    sink: storage
  - name: channel-3-5
    source: level-detection
    sink: alarm
```

These channels are further grouped in three paths that will be used as targets for the different scheduling policies. The *data storage* path comprises the *measurement*, *batching* and *storage* components; the *event storage* path includes the *measurement*, *level detection* and *storage* components; and finally, the *alarm* path includes the *measurement*, *level detection* and *alarm* components. A unique name for these paths and the list of channels that are part

of each of them need to be provided according to the workload model. These paths have been characterized in code snippet 5.4.

Code snippets 5.4: YAML file fragment for the paths definition of the smoke monitoring application.

```
paths:
  - name: data-storage
    channels:
      - channel-1-2
      - channel-2-4
  - name: event-storage
    channels:
      - channel-1-3
      - channel-3-4
  - name: alarm
    channels:
      - channel-1-3
      - channel-3-5
```

Some of the application components impose some constraints that limit the nodes where they can be deployed. The *measurement* component requires the node to have access to the smoke sensor in order to obtain the data, and the *alarm* component requires the node to have access to the driver's dashboard in order to emit the alarm, both from the first train. Additionally, the *storage* component requires the node to be able to access a database. These constraints are implemented with the "require-label" constraint type, using as values the labels previously set to the infrastructure nodes as depicted in code snippet 5.5.

Code snippets 5.5: YAML file fragment for the constraints definition of the smoke monitoring application.

```
constraints:
  - targets:
      components:
        - measurement
```

```
        - alarm
    type: require-label
    value: train-1
  - targets:
      components:
        - measurement
    type: require-label
    value: security-sensors
  - targets:
      components:
        - alarm
    type: require-label
    value: dashboard
  - targets:
      components:
        - storage
    type: require-label
    value: database
```

The other kind of policy, criteria, is used to optimize the placement of each component attending to the application QoS. On the one hand, the most critical path for the smoke monitoring application is the *alarm* path. Alarm triggering needs to be performed in a timely and reliable manner, as it is one of the security applications of each train. Therefore, policies for minimizing the latency and maximizing the reliability of this path are set. On the other hand, both storage paths (*data storage* and *event storage*) are not time sensitive. As the event information can be reconstructed from the measured data, only the *data storage* path will be configured with a reliability policy. This reliability has a weight four times smaller than the *alarm* path due to the relative importance in comparison. The target, type, and weight of each of these policies have been characterized in code snippet 5.6 according to ACOA's workload model.

Code snippets 5.6: YAML file fragment for the criteria definition of the smoke monitoring application.

```
criteria:
  - targets:
      paths:
        - data-storage
    type: e2e-reliability
    weight: 0.25
  - targets:
      paths:
        - alarm
    type: e2e-response-time
    weight: 1.0
  - targets:
      paths:
        - alarm
    type: e2e-reliability
    weight: 1.0
```

5.3.2.2 Speed profiling application

Speed profiling application's goal is to determine which is the optimal speed for a train based on all the information available. This includes, among other data, the expected arrival time, and the position and speed of other surrounding trains. Computing these speed profiles can prove to be resource demanding for nodes in the train and it is offloaded to Cloud nodes.

The application has been implemented with seven components as depicted in figure 5.3. Four different types of components take part in this application, but some of them are replicated for every train. The first component type is the *telemetry* acquisition component that reads the GPS coordinates and train speed, needing one instance per train. A *message broker* component is in charge of routing all the collected data into the speed profiling algorithms,

as every algorithm requires the data from all the trains. The algorithm for each train is executed in the corresponding *profiler* component instance. The computed speed profiles are displayed in the train dashboards for the driver by the *visualization* components.

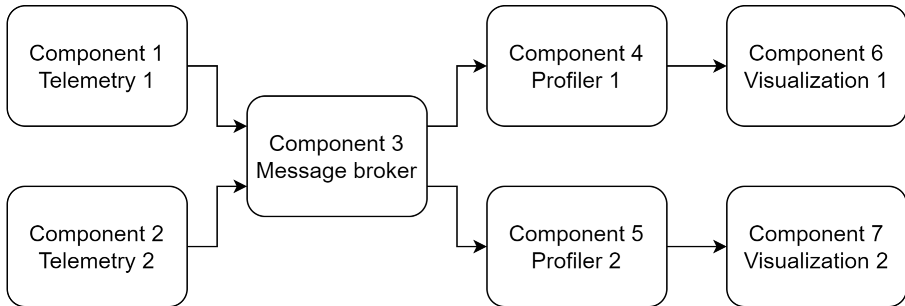


Figure 5.3: Speed profiling application components.

Similar to the smoke monitoring application, code snippet 5.7 provides the basic skeleton of the YAML definition file, and all the relevant information is included afterwards.

Code snippets 5.7: YAML file structure for the speed profiling application.

```
apiVersion: acoa/v1
kind: Application
metadata:
  name: speed-profiling
spec:
  components:
    // ...
  channels:
    // ...
  paths:
    // ...
  constraints:
    // ...
  criteria:
    // ...
```

The components field will contain the previously described components. All the necessary components are named and provided with the image that implements their container, as seen in code snippet 5.8.

Code snippets 5.8: YAML file fragment for the components definition of the speed profiling application.

```
components:
  - metadata:
      name: telemetry-1
    spec:
      containers:
      - image: speed-profiling-telemetry:1.0.0
  - metadata:
      name: telemetry-2
    spec:
      containers:
      - image: speed-profiling-telemetry:1.0.0
  - metadata:
      name: message-broker
    spec:
      containers:
        - image: speed-profiling-message-broker:1.0.0
  - metadata:
      name: profiler-1
    spec:
      containers:
        - image: speed-profiling-profiler:1.0.0
  - metadata:
      name: profiler-2
    spec:
      containers:
        - image: speed-profiling-profiler:1.0.0
  - metadata:
      name: visualization-1
    spec:
      containers:
        - image: speed-profiling-visualization:1.0.0
  - metadata:
      name: visualization-2
    spec:
```

```
containers:
  - image: speed-profiling-visualization:1.0.0
```

The messages exchanged in this application follow the arrows presented in figure 5.3. The presence of a *message broker* component simplifies the message flow, as *telemetry* components publish their data to the broker and *profiler* components subscribe to it, instead of requiring each *telemetry* component to send messages to every *profiler*. The message exchanged among all the components are represented by channels, which need to be provided to ACOA by defining their names, as well as the source and sink components, as seen in code snippet 5.9.

Code snippets 5.9: YAML file fragment for the channels definition of the speed profiling application.

```
channels:
  - name: channel-1-3
    source: telemetry-1
    sink: message-broker
  - name: channel-2-3
    source: telemetry-2
    sink: message-broker
  - name: channel-3-4
    source: message-broker
    sink: profiler-1
  - name: channel-3-5
    source: message-broker
    sink: profiler-2
  - name: channel-4-6
    source: profiler-1
    sink: visualization-1
  - name: channel-5-7
    source: profiler-2
    sink: visualization-2
```

These channels follow two different paths, one for each train, and they will be used as targets for the different scheduling policies. These paths have been characterized in code snippet 5.10.

Code snippets 5.10: YAML file fragment for the paths definition of the speed profiling application.

```
paths:
  - name: train-1
    channels:
      - channel-1-3
      - channel-3-4
      - channel-4-6
  - name: train-2
    channels:
      - channel-2-3
      - channel-3-5
      - channel-5-7
```

Some of these components impose some constraints that limit the nodes where they can be deployed. The *telemetry* components require the node to have access to the train's telemetry data, the *profiler* components require a Cloud node with high processing power and the *visualization* components need to have access to the driver's dashboard. These constraints are implemented with the "require-label" constraint type, using as values the labels previously set to the infrastructure nodes, and they are depicted in code snippet 5.11.

Code snippets 5.11: YAML file fragment for the constraints definition of the speed profiling application.

```
constraints:
  - targets:
      components:
        - telemetry-1
        - visualization-1
    type: require-label
```

```
    value: train-1
  - targets:
      components:
        - telemetry-2
        - visualization-2
    type: require-label
    value: train-2
  - targets:
      components:
        - telemetry-1
        - telemetry-2
    type: require-label
    value: telemetry
  - targets:
      components:
        - visualization-1
        - visualization-2
    type: require-label
    value: dashboard
  - targets:
      components:
        - profiler-1
        - profiler-2
    type: require-label
    value: fast-processor
```

Aside from these constraints, additional policies have been defined to improve the application's QoS. No reliability policy is used in this case because the publisher–subscriber pattern followed by the *message broker* already provides it. Each of the paths has been configured with a policy minimizing the e2e response time in order to provide a quick response to unexpected changes, such as delays in other trains. These optimization criteria have been described in code snippet 5.12.

Code snippets 5.12: YAML file fragment for the criteria definition of the speed profiling application.

```
criteria:
  - targets:
```

```
      paths:
      - train-1
    type: e2e-response-time
    weight: 1.0
  - targets:
      paths:
        - train-2
    type: e2e-response-time
    weight: 1.0
```

5.3.3 Deployment evaluation

In order to compare the behavior of the proposed architecture, the same applications have been deployed in the aforementioned infrastructure using both base Kubernetes and ACOA.

Kubernetes is not aware of the application abstraction, so the above applications need to be transformed into a set of pods. The list of components in code snippets 5.2 and 5.8 are `PodTemplateSpec` instances, so they can be used as the pod spec fields. The label-based constraints present in code snippets 5.5 and 5.11 can also be defined in Kubernetes, despite using a different syntax. An example of a pod definition for the smoke monitoring application *measurement* component can be seen in code snippet 5.13, where K8s syntax for defining the constraints can be seen.

Code snippets 5.13: Pod declaration for the smoke monitoring application measurement component.

```
apiVersion: v1
kind: Pod
metadata:
  name: smoke-monitoring-measurement
spec:
  containers:
  - image: smoke-monitoring-measurement:1.0.0
  affinity:
```

```
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchExpressions:
      - key: train-1
        operator: Exists
      - key: security-sensors
        operator: Exists
```

As ACOA's scheduling decision is based on the optimization criteria, the deployment is deterministic for a given network state snapshot. However, the channel, path and optimization criteria concepts cannot be included in Kubernetes. Therefore, the base K8s scheduler selects one of the possible nodes at random from the ones that fulfill the specified constraints. The number of potential deployment configurations for an application is the product of the number of possible nodes for each component, after applying their constraints.

ACOA's default application scheduler grades each of the criteria defined for each application. The weighted mean of all these criteria scores is used to select the best deployment configuration. This means that all criteria scores need to be normalized. The accumulated success rate, obtained by multiplying the success rates along the whole path, is already normalized. However, the accumulated latency, obtained from the sum of the latencies through the whole path, is not normalized. In order to normalize it, the accumulated latency of the best deployment for each snapshot, i.e., the minimum accumulated latency ($l_m in$), is divided by the accumulated latency of each deployment configuration (l_i) to obtain its score (s_i), as seen in equation 5.1.

$$s_i = \frac{l_{min}}{l_i} \tag{5.1}$$

The best and worst potential deployments for the smoke monitoring application, as graded by the aforementioned policies, have been depicted in table 5.5. ACOA selects the best graded deployment. Having 4800 potential deployments, the chance that the default Kubernetes scheduler chooses the same deployment as ACOA is 0.02%.

Table 5.5: Smoke monitoring deployment nodes selection.

Component		Possible Nodes	Best Case	Worst Case
1	Measurement	13	13	13
2	Batching	1–20	8	6
3	Level detection	1–20	13	10
4	Storage	1–12	8	7
5	Alarm	14	14	14
	Score		98%	43%

As expected, the *measurement* and *alarm* components are fixed to nodes 13 and 14, respectively, as they are dependent on some hardware that is only accessible by those nodes. The *storage* component is deployed in one of the headquarter or cloud provider nodes as those are the ones that can support a database. The difference between the best and worst cases relies on the deployment of the *batching* and *level detection* components. On the one hand, the best-case scenario (figure 5.4) places those components with other components in their corresponding paths in the same node.

On the other hand, the worst-case scenario (figure 5.5) does not only place them in different nodes, but from different locations. This means extra hops on higher latency and lower success rate links for the worst-case scenario, which explains the low score for this case.

Similarly, the best and worst potential deployments for the speed profiling application have been depicted in table 5.6. ACOA always selects the best graded deployment. The number of potential deployments is a bit lower, 2880, resulting in a 0.03% chance of Kubernetes choosing the same scenario as ACOA.

Table 5.6: Speed profiling deployment nodes selection.

Component		**Possible Nodes**	**Best Case**	**Worst Case**
1	Telemetry 1	15	15	15
2	Telemetry 2	18	18	18
3	Message broker	1–20	6	12
4	Profiler 1	1–12	5	2
5	Profiler 2	1–12	6	3
6	Visualization 1	14	14	14
7	Visualization 2	17	17	17
	Score		98%	62%

As expected, the *telemetry* and *visualization* components are fixed to nodes 15, 18, 14, and 17, respectively. The difference between both cases is that the best-case scenario (figure 5.6) deploys all of them in the same location, while the worst-case scenario (figure 5.7) deploys the message broker in a cloud-provider node and the two profilers are placed in the headquarter nodes. This requires

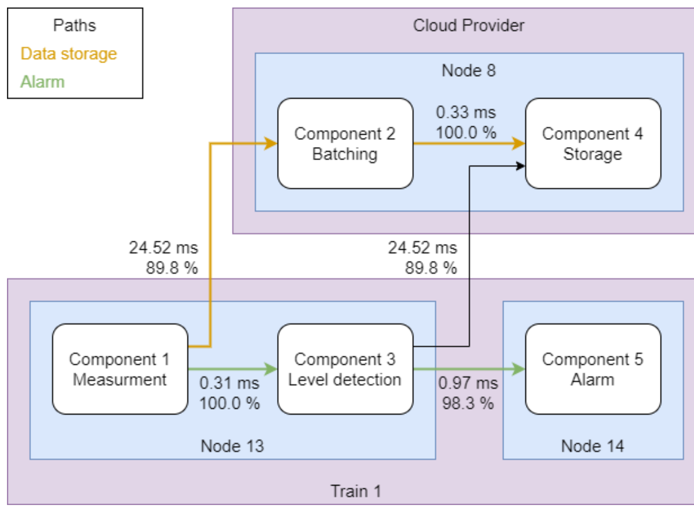


Figure 5.4: Smoke monitoring application best-case deployment.

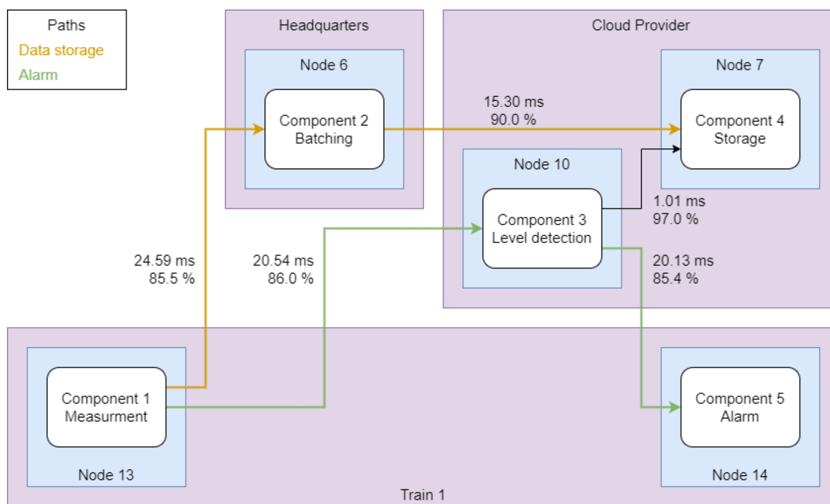


Figure 5.5: Smoke monitoring application worst-case deployment.

an additional hop between datacenters that is more expensive time-wise.

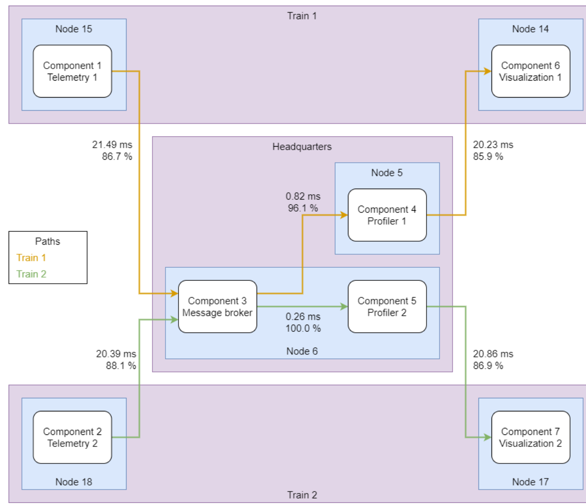


Figure 5.6: Speed profiling application best case deployment.

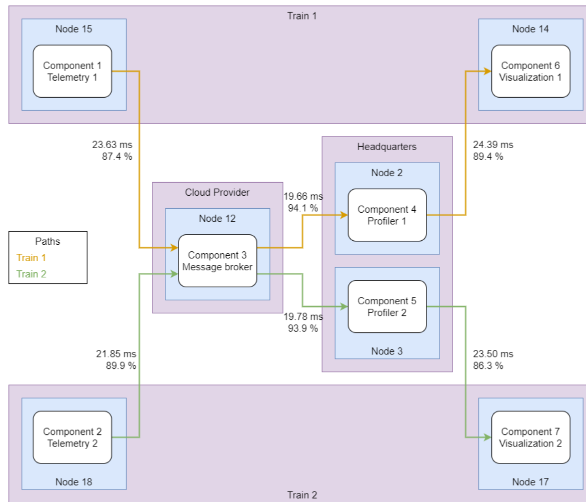


Figure 5.7: Speed profiling application worst case deployment.

5.4 Theoretical bounds for e2e response time scores

From the above railway use case, regarding e2e response time scores, it can be concluded that the main difference between the

best and the worst possible deployment configurations relies on the number of hops that messages need to perform between locations. This effect is due to the fact that latencies between locations are an order of magnitude higher than latencies among the same location.

There are other factors (e.g., the number of hops between components) that also affect the final score of a deployment configuration, but its effect cannot be so easily appreciated from the above results. In order to see the influence of this additional factors, the theoretical lower bound for the score of an e2e response time criterion can be calculated.

The latencies between nodes in the system are simplified to two different values: one for nodes in the same location (l_{same}) and another one for links joining two different locations (l_{diff}). The ratio of these two values (l_{ratio}) can be computed as depicted in equation 5.2. This ratio depends on the infrastructure, and, for the railway use case above, it is around 20.

$$l_{ratio} = \frac{l_{diff}}{l_{same}} \tag{5.2}$$

Another important factor is how many hops are forced between different locations (h_{diff}), and how many hops are allowed in the same location (h_{same}), which add up to the total number of hops (h). These parameters depend on the workload. The event storage path of smoke monitoring application has a total of two hops and zero forced ones between locations, while both paths of the speed profiling application have a total number of three hops, two of them forced between locations.

The score of an e2e response time criterion for a specific deployment (s_i) is computed as the smallest accumulated latency of

all the potential deployments (l_{min}) divided by the accumulated latency of the specific deployment (l_i), as seen in equation 5.1. Theoretically, the smallest accumulated latency only has the enforced hops between locations, while the rest are inside the same location, which can be expressed as in equation 5.3.

$$l_{min} = h_{diff} * l_{diff} + h_{same} * l_{same} \tag{5.3}$$

The worst-case scenario, i.e., the highest potential value for a deployment configuration's accumulated latency (l_{worst}), implies that all hops span multiple locations, as expressed in equation 5.4.

$$l_{worst} = hl_{diff} \tag{5.4}$$

Therefore, the formula for the worst potential score can be simplified according to equation 5.5. This score has two terms: one constant based on the workload characterization (h_{diff}/h), and the other depends on the ratio between the latencies among nodes of same or different locations.

$$s_{worst} = \frac{l_{min}}{l_{worst}} = \frac{h_{diff}}{h}\frac{l_{diff}}{l_{diff}} + \frac{h_{same}}{h}\frac{l_{same}}{l_{diff}} = \frac{h_{diff}}{h} + \frac{h_{same}}{h}\frac{1}{l_{ratio}} \tag{5.5}$$

Figure 5.8 shows the score for the worst potential deployment configuration. The horizontal axis is the latency ratio in a logarithmic scale, while the vertical axis represents the score. The orange line is the constant term (h_{diff}/h), while the blue line is the whole score. A latency ratio of one means that there is no difference between the two latencies, and therefore the score would

be 100%. As the difference between the two latencies increases, the score rapidly approaches the constant term, which is just based on the workload characterization.

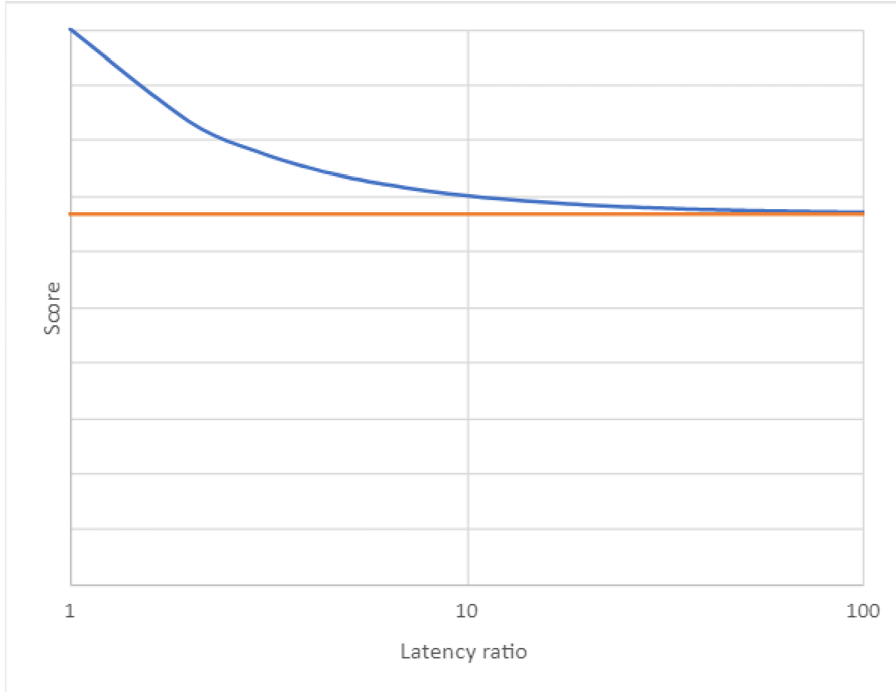


Figure 5.8: Theoretical worst-case score.

For the speed profiling application, the theoretical worst-case score can be computed with the above provided formula. The result is 68% (as seen in equation 5.6), which is higher than the score obtained in the previous section for the worst-case deployment. This happens because one of the simplifications was to reduce all latencies to just two values.

$$s_{worst} = \frac{2}{3} + \frac{1}{3}\frac{1}{20} = 0.68 \tag{5.6}$$

Without this simplification, the last step in equation 5.5 cannot be performed. Latencies l_{diff} and l_{same} would take smaller values for the best accumulated latency (l_{min}) than for the worst-case scenario (l_{worst}). Considering the biggest and lowest latencies for links that have node 15 or 18 as the source, and links that have node 14 or 17 as the sink, the theoretical bound is placed at 55% (as seen in equation 5.7). The latency variability of the network further reduces the theoretical lower bound.

$$s_{worst} = \frac{2}{3}\frac{20.13}{24.78} + \frac{1}{3}\frac{0.80}{24.78} = 0.55 \tag{5.7}$$

It can be concluded that both the number of hops that are not forced to be to different locations, and the latency variability of the network, contribute to reducing the theoretical lower bound for the score. This smaller bound increases the improvement achieved by ACOA over orchestration architectures that do not consider application QoS requirements.

5.5 Conclusions

Several tests have been performed to assess the validity of ACOA and its fulfillment of the identified requirements.

The impact of the shared-state orchestration scheme used for ACOA has been validated [57]. The required mean deployment time has been measured for different number of components and schedulers, showing an improvement for every studied scenario. The appearance of collisions during these tests has also been measured. Their frequency has been proved to be orders of magnitude lower than the mean deployment time reduction, proving that the

use of the shared-state scheme improves the scalability of the architecture (R9).

A use case in the railway domain has been used to validate ACOA. An infrastructure of 20 nodes spread among five different locations has been used, simulating high-end nodes in headquarter or cloud-provider datacenters as well as resource-constrained nodes in trains and stations (R1). Two applications with different QoS requirements (R3) have been defined in a human readable YAML file (R4). Based on the node and network data gathered by the architecture (R6 & R7), ACOA has been able to select the best deployment configuration among thousands of possibilities to optimize the specified application QoS requirements (R8).

This use case highlights the importance of considering application QoS requirements during the scheduling algorithm, by illustrating the difference between the best possible deployment configuration and the worst case. The influence that some workload parameters (such as the number of messages exchanged in each path) and infrastructure metrics (like the latency variability) have also been analyzed. The theoretical results obtained by this method match the empirical results obtained during the use case assessment.

CHAPTER 6

Conclusions and future lines

"There is no real ending. It's just the place where you stop the story."

- Frank Herbert

6.1 Conclusions

The present work contributes with an orchestration architecture for the Edge to Cloud continuum that is aware of the application QoS requirements. ACOA manages distributed containerized applications at runtime and dynamically reconfigure the deployment of their components over the infrastructure nodes to satisfy QoS requirements.

As Edge to Cloud continuum infrastructures comprise a large number of nodes that present very different characteristics, an infrastructure model is proposed to characterize the whole cluster. It includes both nodes, which represent their corresponding physical device, and links that define the network state.

Furthermore, a workload model is also proposed to organize all the required information about each application in order to take into account the QoS requirements for the scheduling and deployment algorithms.

The information related to these two models is gathered, stored, and consumed by the different system components in order to perform the orchestration tasks. *Node daemons* manage the containers in each node and *monitoring daemons* capture the state of the whole cluster, including the network. The data obtained by these daemons and the application definitions provided by the users are stored in the *state database* through the *API server*. The *system scheduler* consumes this information to deploy the *application schedulers*, which also consume it when executing their QoS-aware scheduling algorithms. These *application schedulers* (one per application) review the selected deployment periodically to dynamically reconfigure if necessary.

The increased complexity due to application QoS requirements requires a more scalable approach than the current centralized ones. Therefore, an orchestration scheme with multiple concurrent schedulers has been used. This approach not only increases the scalability of the orchestration architecture, but it also enables customizing the scheduling algorithm to deploy each application based on its non-functional requirements.

The dynamic scheduling and deployment algorithms require to know the current network state. Latency, jitter, and packet-loss ratio measurements for each link in the cluster are provided by the *monitoring daemons*. They implement SWIM-NSM, a new protocol designed for this purpose with lightweight-ness and scalability as its main goals.

ACOA has been implemented on top of Kubernetes. For this purpose, Kubernetes has been extended to include: 1) link abstraction for the infrastructure model to make it network-aware, 2) applications and components for the workload model to provide a higher abstraction level for the users and to improve the scheduling customization capabilities, and 3) new system components to implement the distributed monitoring and scheduling, as well as the above specified abstractions.

Several tests have been carried out to assess the suitability of ACOA for the Cloud to Edge continuum. First, the performance of the multiple scheduler approach is evaluated. Next, a use case scenario with two different applications from the railway vertical domain is used to compare ACOA with base Kubernetes scheduling. Finally, a theoretical analysis to consider the impact of the number of nodes and the network variability in the score used to select the best deployment is also performed.

The results of this work have been published in several international conferences and indexed magazines with renown in the research field:

[24] Adrián Orive et al. "Passive Network State Monitoring for Dynamic Resource Management in Industry 4.0 Fog Architectures". In: *Fourteenth International Conference on Automation Science and Engineering (CASE)*. Munich, Germany: IEEE, 2018, pp. 1414–1419. DOI: 10.1109/COASE.2018.8560475.

[57] Adrián Orive et al. "Novel orchestration architecture for Fog computing". In: *Seventeenth International Conference on Industrial Informatics (INDINN)*. Helsinki, Finland: IEEE, 2019. DOI: 10.1109/INDIN41052.2019.8972087.

[58] Adrián Orive et al. "Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications". In: *Sensors* 22.5 (2022). ISSN: 1424-8220. DOI: 10.3390/s22051755.

6.2 Future lines

ACOA offers several important advantages compared to existing approaches, but there are some topics that still need to be further researched.

The present work has not considered security and privacy, which is a major concern in the related field. Kubernetes provides basic authorization mechanisms to prevent non-allowed users to deploy potentially harmful applications. Application implementations should also use secure communications between their components. However, studying the security and privacy subject from

an architecture point of view is still an open research topic from which ACOA would benefit.

The dynamic redeployment being used in ACOA is based on a configurable period, and a score improvement threshold to prevent constant changes with minimal impact. These values require domain knowledge from the user as different applications or domains require different values. This value tuning process could be automated with Artificial Intelligence (AI) approaches. These approaches could adapt these configurations parameters based on the variability of the network or the criticality of the application being deployed.

Regarding the network monitoring protocol, measuring bandwidth in a non-intrusive manner is still an open research topic. Adding this fourth metric to the already existing latency, jitter and packet-loss ratio would fully characterize each link. However, all the existing methods are based on overflowing the link with packets, which would affect the performance of the applications being executed in ACOA.

APPENDIX A

SWIM-NSM wire protocol

A.1 Introduction

Short	Protocol	Version
SWIM-NSM	SWIM Network State Monitoring	v1.0

This appendix describes the wire-format of the messages exchanged by SWIM-NSM clients in order to keep an up-to-date list of active nodes in the group and compute the network metrics of the whole mesh.

Messages can be categorized into detection and dissemination messages. Every package transmission will be composed of a single detection message and any number of dissemination messages that will allow achieving the infectious-style gossiping behavior of SWIM. Additionally, they will also add a protocol identification header.

A.2 Header

The packet header is composed of a mandatory version block and additional blocks defined by each version.

A.2.1 Version block

The goal of this block is to identify the SWIM-NSM protocol version so that the messages in this package can be decoded accordingly. A variable-length field is used for this purpose, where the most significant bit in each byte indicates if an additional byte is required to represent the version.

After removing the most significant bit of every byte, the remaining bits are divided in two bit-arrays of the same length, with

the binary representation of the major and minor version numbers respectively. In case of an odd number of bits, the bit-array corresponding to the major version will be one bit longer than the one corresponding to the minor version.

Table A.1: Maximum major and minor version values per block size.

Block size	Max. major version	Max. minor version
1 byte	15	7
2 bytes	127	127

The smallest version block size that fits the protocol version must be used. The maximum supported values of the major and minor versions for each header size is represented in table A.1. For the version described in this appendix (v1.0), the header is shown in table A.2.

Table A.2: Version block for v1.0.

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	1	0	0	0

A.2.2 Additional blocks

The version described in this appendix (v1.0) does not define any additional header block.

A.3 Detection message

A single detection message is encoded after the header. This message can be one of the four following types:

- Ping

- Ping request
- Ack
- Forward ack

These messages are composed of a detection message header block, a token block, and additional blocks defined by each message kind.

A.3.1 Common detection message blocks

A.3.1.1 Detection message header block

The detection message header block's length is fixed to 1 byte. The four most significant bits are used for type-specific flags, while the two less significant bits are used to identify the type of detection message. The remaining two bits are reserved for future use and should always be unset. Table A.3 shows the bit representation of the headers of each detection type.

Table A.3: Detection message header block.

Message type	b7	b6	b5	b4	b3	b2	b1	b0
Ping	*	*	*	*	0	0	0	0
Ping request	*	*	*	*	0	0	0	1
Ack	*	*	*	*	0	0	1	0
Forward ack	*	*	*	*	0	0	1	1

A.3.1.2 Token block

The token block is a two byte fixed length integer used to identify which detection message is being answered to. Ping and ping request messages generate a unique (per client) 2-byte token and

encode them in this block, and ack and forward ack messages use the same token as the ping or ping request they are answering to.

Note: *the easiest way to implement this is by using an incremental integer, but this behavior is not enforced by the protocol definition.*

A.3.2 Ping detection message

This detection message is sent to check the status of another node. In this context, source refers to the client sending this packet.

A.3.2.1 Header block

The ping detection message defines two flags in the most significant bits of the detection message header block. The two remaining bits must be unset. Table A.4 shows the bit representation of the ping detection message header block.

Table A.4: Ping detection message header block.

b7	b6	b5	b4	b3	b2	b1	b0
Source IP flag	Source port flag	0	0	0	0	0	0

- The source IP flag determines if the provided source IP uses IPv4 (unset) or IPv6 (set) format.
- The source port flag determines if a non-default port is being used by the source client.

A.3.2.2 Token block

As described in the common detection message blocks section (A.3.1.2).

A.3.2.3 Source IP block

This block defines the IP address of the client that sent this message.

This block is mandatory and its length can be either 4 bytes (source IP flag unset) or 16 bytes (source IP flag set).

A.3.2.4 Source port block

This block specifies the non-default port that is being used by the client that sent this message.

This block is optional (only present if the source port flag was set) and its length is 2 bytes.

A.3.3 Ping request detection message

This message is sent to request another node to check the status of a third one. In this context, source refers to the client sending this message, while target refers to the client that needs to be pinged.

A.3.3.1 Header block

The ping request detection message defines four flags in the most significant bits of the detection message header block. Table A.5 shows the bit representation of the ping request detection message header block.

Table A.5: Ping request detection message header block.

b7	b6	b5	b4	b3	b2	b1	b0
Src. IP	Src. port	Tgt. IP	Tgt. port	0	0	0	1

- The source IP flag determines if the provided source IP uses IPv4 (unset) or IPv6 (set) format.
- The source port flag determines if a non-default port is being used by the source client.
- The target IP flag determines if the provided target IP uses IPv4 (unset) or IPv6 (set) format.
- The target port flag determines if a non-default port is being used by the target client.

A.3.3.2 Token block

As described in the common detection message blocks section (A.3.1.2).

A.3.3.3 Source IP block

This block defines the IP address of the client that sent this message.

This block is mandatory and its length can be either 4 bytes (source IP flag unset) or 16 bytes (source IP flag set).

A.3.3.4 Source port block

This block specifies the non-default port that is being used by the client that sent this message.

This block is optional (only present if the source port flag was set) and its length is 2 bytes.

A.3.3.5 Target IP block

This block defines the IP address of the client that needs to be pinged by the receiver of this message.

This block is mandatory and its length can be either 4 bytes (target IP flag unset) or 16 bytes (target IP flag set).

A.3.3.6 Target port block

This block specifies the non-default port that is being used by the client that needs to be pinged by the receiver of this message.

This block is optional (only present if the target port flag was set) and its length is 2 bytes.

A.3.4 Ack detection message

This message is sent to answer a ping detection message.

A.3.4.1 Header block

The ack detection message defines no flag. The four remaining bits must be unset. Table A.6 shows the bit representation of the ack detection message header block.

Table A.6: Ack detection message header block.

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	1	0

A.3.4.2 Token block

As described in the common detection message blocks section (A.3.1.2).

A.3.4.3 Duration block

The duration block specifies the elapsed time between receiving the ping detection message and sending the ack detection message in microseconds (µs).

This block is mandatory and its length is 4 bytes.

Note: *the precision and length of this block establish a practical upper bound to the protocol's timeout at 4 294 967 295 µs (i.e., 1 h 11 min 34 s 967 ms 295 µs).*

A.3.5 Forward ack detection message

This message is sent to answer a ping request detection message.

A.3.5.1 Header block

The forward ack detection message defines a single flag in the most significant bit. The three remaining bits must be unset. Table A.7 shows the bit representation of the forward ack detection message header block.

Table A.7: Forward ack detection message header block.

b7	b6	b5	b4	b3	b2	b1	b0
Fail flag	0	0	0	0	0	1	1

- The fail flag specifies if the ping requested has succeeded (unset) or failed (set).

A.3.5.2 Token block

As described in the common detection message blocks section (A.3.1.2).

A.3.5.3 Duration block

The duration block specifies the elapsed time between receiving the ping request detection message and sending the forward ack detection message in microseconds (µs).

This block is mandatory and its length is 4 bytes.

Note: *the precision and length of this block establish a practical upper bound to the protocol's timeout at 4 294 967 295 μs (i.e., 1 h 11 min 34 s 967 ms 295 μs).*

A.4 Dissemination messages

Any number of dissemination messages may be encoded after the detection message, in order to spread gossips. These messages can be one of the three following types:

- Alive
- Suspect
- Confirm

These messages are composed by a dissemination message header block, some dissemination message type specific blocks and an incarnation block.

A.4.1 Common dissemination message blocks

A.4.1.1 Dissemination message header block

The dissemination message header block's length is fixed to 1 byte. The four most significant bits are used for type-specific flags, while the two less significant bits are used to identify the type of dissemination message. The remaining two bits are reserved for future use and should always be unset. Table A.8 shows the bit representation of the headers of each dissemination type.

Table A.8: Dissemination message header block.

Message type	b7	b6	b5	b4	b3	b2	b1	b0
Alive	*	*	*	*	0	0	0	0
Suspect	*	*	*	*	0	0	0	1
Confirm	*	*	*	*	0	0	1	0

A.4.1.2 Incarnation block

The incarnation block is a variable length incremental integer number that is used to determine which dissemination message should take precedence when multiple contradictory dissemination messages are being gossiped simultaneously.

A variable amount of leading set bits, followed by a single unset bit, is used as a prefix. The number of leading set bits is the number of additional bytes that are used to encode this variable-length integer. After loading the additional bytes, the prefix is removed, and the remaining bit array is interpreted as an unsigned integer.

Several examples using this variable-length integer representation are provided down below:

Table A.9 shows the bit representation of number 1. It is composed of a single bit prefix (0) indicating that no extra bytes are required, and the number representation in the remaining 7 bits (000 0001).

Table A.9: Variable length encoding of 1.

Byte number	b7	b6	b5	b4	b3	b2	b1	b0
Byte 1	0	0	0	0	0	0	0	1

Table A.10 shows the bit representation of number 127. It is composed of a single bit prefix (0) indicating that no extra bytes are required, and the number representation in the remaining 7 bits (111 1111).

Table A.10: Variable length encoding of 127.

Byte number	b7	b6	b5	b4	b3	b2	b1	b0
Byte 1	0	1	1	1	1	1	1	1

Table A.11 shows the bit representation of number 128. It is composed of a two bit prefix (10) indicating that one extra byte is required, and the number representation in the remaining 14 bits (00 0000 1000 0000).

Table A.11: Variable length encoding of 128.

Byte number	b7	b6	b5	b4	b3	b2	b1	b0
Byte 1	1	0	0	0	0	0	0	0
Byte 2	1	0	0	0	0	0	0	0

Table A.12 shows the bit representation of number 72 057 594 037 927 936. It is composed of a nine bit prefix (1111 1111 0) indicating that eight extra bytes are required, and the number representation in the remaining 71 bits (0000000 00000001 followed by 7 fully unset bytes).

A.4.2 Alive dissemination message

This message is sent to specify that a node is alive. It can only be generated by itself, but can be gossiped by all the nodes in the system.

Table A.12: Variable length encoding of 72 057 594 037 927 936.

Byte number	b7	b6	b5	b4	b3	b2	b1	b0
Byte 1	1	1	1	1	1	1	1	1
Byte 2	0	0	0	0	0	0	0	0
Byte 3	0	0	0	0	0	0	0	1
Byte 4	0	0	0	0	0	0	0	0
Byte 5	0	0	0	0	0	0	0	0
Byte 6	0	0	0	0	0	0	0	0
Byte 7	0	0	0	0	0	0	0	0
Byte 8	0	0	0	0	0	0	0	0
Byte 9	0	0	0	0	0	0	0	0
Byte 10	0	0	0	0	0	0	0	0

A.4.2.1 Header block

The alive dissemination message defines two flags in the most significant bits of the dissemination message header block. The two remaining bits must be unset. Table A.13 shows the bit representation of the alive dissemination message header block.

Table A.13: Alive dissemination message header block.

b7	b6	b5	b4	b3	b2	b1	b0
IP flag	Port flag	0	0	0	0	0	0

- The IP flag determines if the provided IP uses IPv4 (unset) or IPv6 (set) format.
- The port flag determines if a non-default port is being used by the client.

A.4.2.2 IP block

This block defines the IP address of the client that claims to be alive.

This block is mandatory and its length can be either 4 bytes (IP flag unset) or 16 bytes (IP flag set).

A.4.2.3 Port block

This block specifies the non-default port that is being used by the client that claims to be alive.

This block is optional (only present if the port flag was set) and its length is 2 bytes.

A.4.2.4 Incarnation block

As described in the common dissemination message blocks section (A.4.1.2).

A.4.3 Suspect dissemination message

This message is sent to mark a node as suspicious of being down.

In this context, source refers to the client starting this suspicion, while target refers to the client that is marked as suspicious.

A.4.3.1 Header block

The suspect dissemination message defines four flags in the most significant bits of the dissemination message header block. Table A.14 shows the bit representation of the suspect dissemination message header block.

Table A.14: Suspect dissemination message header block.

b7	b6	b5	b4	b3	b2	b1	b0
Src. IP	Src. port	Tgt. IP	Tgt. port	0	0	0	1

- The source IP flag determines if the provided source IP uses IPv4 (unset) or IPv6 (set) format.
- The source port flag determines if a non-default port is being used by the source client.
- The target IP flag determines if the provided target IP uses IPv4 (unset) or IPv6 (set) format.
- The target port flag determines if a non-default port is being used by the target client.

A.4.3.2 Source IP block

This block defines the IP address of the client that started the suspicion.

This block is mandatory and its length can be either 4 bytes (source IP flag unset) or 16 bytes (source IP flag set).

A.4.3.3 Source port block

This block specifies the non-default port that is being used by the client that started the suspicion.

This block is optional (only present if the source port flag was set) and its length is 2 bytes.

A.4.3.4 Target IP block

This block defines the IP address of the client that is being marked as suspicious.

This block is mandatory and its length can be either 4 bytes (target IP flag unset) or 16 bytes (target IP flag set).

A.4.3.5 Target port block

This block specifies the non-default port that is being used by the client that is being marked as suspicious.

This block is optional (only present if the target port flag was set) and its length is 2 bytes.

A.4.3.6 Incarnation block

As described in the common dissemination message blocks section (A.4.1.2).

A.4.4 Confirm dissemination message

This message is sent to confirm that a node is down.

In this context, source refers to the client making the claim, while target refers to the client that is down.

A.4.4.1 Header block

The confirm dissemination message defines four flags in the most significant bits of the dissemination message header block. Table A.15 shows the bit representation of the confirm dissemination message header block.

Table A.15: Confirm dissemination message header block.

b7	b6	b5	b4	b3	b2	b1	b0
Src. IP	Src. port	Tgt. IP	Tgt. port	0	0	1	0

- The source IP flag determines if the provided source IP uses IPv4 (unset) or IPv6 (set) format.

- The source port flag determines if a non-default port is being used by the source client.
- The target IP flag determines if the provided target IP uses IPv4 (unset) or IPv6 (set) format.
- The target port flag determines if a non-default port is being used by the target client.

A.4.4.2 Source IP block

This block defines the IP address of the client that made the claim that a node was down.

This block is mandatory and its length can be either 4 bytes (source IP flag unset) or 16 bytes (source IP flag set).

A.4.4.3 Source port block

This block specifies the non-default port that is being used by the client that made the claim that a node was down.

This block is optional (only present if the source port flag was set) and its length is 2 bytes.

A.4.4.4 Target IP block

This block defines the IP address of the client that is being marked as down.

This block is mandatory and its length can be either 4 bytes (target IP flag unset) or 16 bytes (target IP flag set).

A.4.4.5 Target port block

This block specifies the non-default port that is being used by the client that is being marked as down.

This block is optional (only present if the target port flag was set) and its length is 2 bytes.

A.4.4.6 Incarnation block

As described in the common dissemination message blocks section (A.4.1.2).

References

[1] Ed Sperling. “Computing’s Swinging Pendulum”. In: *Forbes* (Mar. 2010). URL: https://www.forbes.com/2010/03/05/cloud-computing-management-technology-cio-network-data.html.

[2] Gordon Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (Apr. 1965).

[3] Gordon Moore. “Excerpts from A Conversation with Gordon Moore: Moore’s Law”. In: (2005). URL: http://large.stanford.edu/courses/2012/ph250/lee1/docs/Excepts_A_Conversation_with_Gordon_Moore.pdf.

[4] Edward Ashford Lee. “Cyber Physical Systems: Design Challenges”. In: *International Symposium on Object-Oriented Real-Time Distributed Computing*. Orlando, FL, USA: IEEE, 2008. DOI: 10.1109/ISORC.2008.25.

[5] Ian Fuat Akyìldìz et al. “Wireless sensor networks: A survey”. In: *Computer Networks* 38.4 (Mar. 2002), pp. 393–422. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(01)00302-4.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A survey”. In: *Computer Networks* 54.15

(Oct. 2010), pp. 2787–2805. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.05.010.

[7] Internet Engineering Task Force. *IETF*. [Online; accessed 13-September-2022]. URL: https://www.ietf.org/.

[8] Harald T. Alvestrand. *A Mission Statement for the IETF*. RFC 3935. Oct. 2004. DOI: 10.17487/RFC3935. URL: https://www.rfc-editor.org/info/rfc3935.

[9] Guy Almes, Sunil Kalidindi, and Matthew Zekauskas. *A One-way Delay Metric for IPPM*. RFC 2679. Sept. 1999. DOI: 10.17487/RFC2679. URL: https://www.rfc-editor.org/info/rfc2679.

[10] Guy Almes et al. *A One-Way Delay Metric for IP Performance Metrics (IPPM)*. RFC 7679. Jan. 2016. DOI: 10.17487/RFC7679. URL: https://www.rfc-editor.org/info/rfc7679.

[11] C. Demichelis and P. Chimento. *IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)*. RFC 3393. Nov. 2002. DOI: 10.17487/RFC3393. URL: https://www.rfc-editor.org/info/rfc3393.

[12] Guy Almes, Sunil Kalidindi, and Matthew Zekauskas. *A One-way Packet Loss Metric for IPPM*. RFC 2680. Sept. 1999. DOI: 10.17487/RFC2680. URL: https://www.rfc-editor.org/info/rfc2680.

[13] Guy Almes et al. *A One-Way Loss Metric for IP Performance Metrics (IPPM)*. RFC 7680. Jan. 2016. DOI: 10.17487/RFC7680. URL: https://www.rfc-editor.org/info/rfc7680.

[14] Joseph Ishac and Phil Chimento. *Defining Network Capacity*. RFC 5136. Feb. 2008. DOI: 10.17487/RFC5136. URL: https://www.rfc-editor.org/info/rfc5136.

[15] Stanislav Shalunov et al. *A One-way Active Measurement Protocol (OWAMP)*. RFC 4656. Sept. 2006. DOI: 10.17487/RFC4656. URL: https://www.rfc-editor.org/info/rfc4656.

[16] René Serral, Albert Cabellos, and Jordi Domingo. "Network performance assessment using adaptive trafficsampling". In: *Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet, 7th International IFIP-TC6 Networking Conference*. Singapore: Springer, 2008, pp. 252–263. ISBN: 978-3-540-79548-3. DOI: 10.1007/978-3-540-79549-0_22.

[17] Han Hee Song and Praveen Yalagandula. "Real-time End-to-end Network Monitoring in Large Distributed Systems". In: *2nd International Conference on Communication Systems Software and Middleware*. Bangalore, India: IEEE, 2007. ISBN: 1-4244-0613-7. DOI: 10.1109/COMSWA.2007.382612.

[18] Alaknantha Eswaradass, Xian-He Sun, and Ming Wu. "Network Bandwidth Predictor (NBP): A System for Online Network performance Forecasting". In: *6th International Symposium on Cluster Computing and the Grid*. Singapore: IEEE, 2006. ISBN: 0-7695-2585-7. DOI: 10.1109/CCGRID.2006.72.

[19] Ramana Rao Kompella et al. "Every Microsecond Counts: Tracking Fine-Grain Latencies with a Lossy Difference Aggregator". In: *SIGCOMM Computer Communication Review* 39.4 (2009), pp. 255–266. ISSN: 0146-4833. DOI: 10.1145/1594977.1592599.

[20] Myungjin Lee, Nick Duffield, and Ramana Rao Kompella. “Not All Microseconds Are Equal: Fine-Grained per-Flow Measurements with Reference Latency Interpolation”. In: *SIGCOMM Computer Communication Review* 40.4 (2010), pp. 27–38. ISSN: 0146-4833. DOI: 10.1145/1851275.1851188.

[21] Myungjin Lee, Nick Duffield, and Ramana Rao Kompella. “MAPLE: A Scalable Architecture for Maintaining Packet Latency Measurements”. In: *Internet Measurement Conference*. Boston, Massachusetts, USA: Association for Computing Machinery, Inc., 2012, pp. 101–114. ISBN: 9781450317054. DOI: 10.1145/2398776.2398788.

[22] Donald Brown, James Leth, and James Vandendorpe. “Fault recovery in a distributed processing system”. U.S. Patent 4710926A.

[23] Abhinandan Das, Indranil Gupta, and A. Motivala. “SWIM: scalable weakly-consistent infection-style process group membership protocol”. In: *Proceedings International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE, 2002. DOI: 0.1109/DSN.2002.1028914.

[24] Adrián Orive et al. “Passive Network State Monitoring for Dynamic Resource Management in Industry 4.0 Fog Architectures”. In: *Fourteenth International Conference on Automation Science and Engineering (CASE)*. Munich, Germany: IEEE, 2018, pp. 1414–1419. DOI: 10.1109/COASE.2018.8560475.

[25] Shane Snyder et al. “A Case for Epidemic Fault Detection and Group Membership in HPC Storage Systems”. In: *International Workshop on Performance Modeling, Benchmark-*

ing and Simularion of High Performance Computer Systems. New Orleans, LA, USA: Springer, 2014, pp. 237–248. DOI: 10.1007/978-3-319-17248-4_12.

[26] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *European Conference on Computer Systems (EuroSys)*. Bordeaux, France: Association for Computing Machinery, Inc., 2015, pp. 1–17. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964.

[27] The Kubernetes Authors. *Kubernetes*. [Online; accessed 02-January-2023]. URL: https://kubernetes.io/.

[28] Docker Inc. *Docker Swarm*. [Online; accessed 02-January-2023]. URL: https://docs.docker.com/engine/swarm/.

[29] Michael Isard et al. "Quincy: Fair Scheduling for Distributed Computing Clusters". In: *Symposium on Operating System Principles (SOSP)*. Big Sky, MT, USA: Association for Computing Machinery, Inc., 2009, pp. 261–276. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629601.

[30] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Symposium on Networked System Design and Implementation (NSDI)*. Boston, MA, USA: USENIX, 2011, pp. 295–308. DOI: 10.5555/1972457.1972488.

[31] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN: yet another resource negotiator". In: *Symposium on Cloud Computing (SOCC)*. Santa Clara, CA, USA: Association for Computing Machinery, Inc., 2013, pp. 1–16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633.

[32] Malte Schwarzkopf et al. "Omega: flexible, scalable schedulers for large compute clusters". In: *European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic: Association for Computing Machinery, Inc., 2013, pp. 351–364. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465386.

[33] Eric Boutin et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *Symposium on Operating System Design and Implementation (OSDI)*. Broomfield, CO, USA: USENIX, 2014, pp. 285–300. ISBN: 978-1-931971-16-4. DOI: 10.5555/2685048.2685071.

[34] HashiCorp. *Nomad scheduling*. [Online; accessed 02-January-2023]. URL: https://developer.hashicorp.com/nomad/docs/concepts/scheduling.

[35] Kay Ousterhout et al. "Sparrow: distributed, low latency scheduling". In: *Symposium on Operating System Principles (SOSP)*. Farminton, PA, USA: Association for Computing Machinery, Inc., 2013, pp. 69–84. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522716.

[36] Karima Velasquez et al. "Service Orchestration in Fog Environments". In: *International Conference on Future Internet of Things and Cloud (FiCloud)*. Prague, Czech Republic: IEEE, 2017, pp. 329–336. ISBN: 978-1-5386-2074-8. DOI: 10.1109/FiCloud.2017.49.

[37] Mathias Santos de Brito et al. "A service orchestration architecture for Fog-enabled infrastructures". In: *International Conference on Fog and Mobile Edge Computing (FMEC)*. Valencia, Spain: IEEE, 2017, pp. 127–132. ISBN: 978-1-5386-2859-1. DOI: 10.1109/FMEC.2017.7946419.

[38] Yuxuan Jiang, Zhe Huang, and Danny H. K. Tsang. "Challenges and Solutions in Fog Computing Orchestration". In: *IEEE Network* 32.3 (Nov. 2017), pp. 122–129. ISSN: 0890-8044. DOI: 10.1109/MNET.2017.1700271.

[39] Francescomaria Faticanti et al. "Throughput-Aware Partitioning and Placement of Applications in Fog Computing". In: *IEEE Transactions on Network and Service Management* 17 (Sept. 2020), pp. 2436–2450. ISSN: 1932-4537. DOI: 10.1109/TNSM.2020.3023011.

[40] Cecil Wöbker et al. "Fogernetes: Deployment and Management of Fog Computing Applications". In: *Network Operations and Management Symposium (NOMS)*. Taipei, Taiwan: IEEE, 2018. DOI: 10.1109/NOMS.2018.8406321.

[41] Saiful Hoque et al. "Towards Container Orchestration in Fog Computing Infrastructures". In: *Annual Computer Software and Applications Conference (COMPSAC)*. Turin, Italy: IEEE, 2017, pp. 294–299. DOI: 10.1109/COMPSAC.2017.248.

[42] Zhenyu Wen et al. "Fog Orchestration for Internet of Things Services". In: *IEEE Internet Computing* 21.2 (Mar. 2017), pp. 16–24. ISSN: 1089-7801. DOI: 10.1109/MIC.2017.36.

[43] Antonio Brogi and Stefano Forti. "QoS-Aware Deployment of IoT Applications Through the Fog". In: *IEEE Internat of Things Journal* 4.5 (Oct. 2017), pp. 1185–1192. ISSN: 2327-4662. DOI: 10.1109/JIOT.2017.2701408.

[44] Jen-Sheng Tsai et al. "QoS-Aware Fog Service Orchestration for Industrial Internet of Things". In: *Transactions on Services Computing* 15.3 (2020), pp. 1265–1279. ISSN: 1939-1374. DOI: 10.1109/TSC.2020.2978472.

[45] Kaihua Fu et al. "Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum". In: *IEEE Transactions on Parallel and Distributed Systems* 33 (Nov. 2021), pp. 1825–1840. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3128037.

[46] Salman Taherizadeh, Vlado Stankovski, and Marko Grobelnik. "A Capillary Computing Architecture for Dynamic Internet of Things: Orchestration of Microservices from Edge Devices to Fog and Cloud Providers". In: *Sensors* 18.9 (2018). ISSN: 1424-8220. DOI: 10.3390/s18092938.

[47] Zeinab Nezami et al. "Decentralized Edge-to-Cloud Load Balancing: Service Placement for the Internet of Things". In: *Access* 9 (2021), pp. 64983–65000. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3074962.

[48] Rafael Fayos-Jordan et al. "Performance comparison of container orchestration platforms with low cost devices in the fog, assisting Internet of Things applications". In: *Journal of Network and Computer Applications* 169 (2020). ISSN: 1084-8045. DOI: 10.1016/j.jnca.2020.102788.

[49] Valeria Cardellini et al. "Self-adaptive Container Deployment in the Fog: A Survey". In: *International Symposium on Algorithmic Aspects of Cloud Computing (ALGOCLOUD)*. Munich, Germany: Springer, Cham, 2019, pp. 77–102. ISBN: 978-3-030-58628-7. DOI: 10.1007/978-3-030-58628-7_6.

[50] Zeineb Rejiba and Javad Chamanara. "Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches". In: *ACM Computing Surveys* 55.7 (2022), pp. 1–37. ISSN: 0360-0300. DOI: 10.1145/3544788.

[51] Google. *Google Kubernetes Engine*. [Online; accessed 06-October-2022]. URL: https://cloud.google.com/kubernetes-engine.

[52] Amazon. *Amazon Elastic Kubernetes Service*. [Online; accessed 06-October-2022]. URL: https://aws.amazon.com/eks/.

[53] Microsoft. *Azure Kubernetes Service*. [Online; accessed 06-October-2022]. URL: https://azure.microsoft.com/en-in/services/kubernetes-service/.

[54] Kubernetes. *Kubernetes ObjectMeta v1*. [Online; accessed 06-October-2022]. URL: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/%5C#objectmeta-v1-meta.

[55] Kubernetes. *Kubernetes PodTemplateSpec v1*. [Online; accessed 06-October-2022]. URL: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/%5C#podtemplatespec-v1-core.

[56] Cloud Native Computing Foundation. *K3s*. [Online; accessed 13-September-2022]. 2019. URL: https://k3s.io.

[57] Adrián Orive et al. "Novel orchestration architecture for Fog computing". In: *Seventeenth International Conference on Industrial Informatics (INDINN)*. Helsinki, Finland: IEEE, 2019. DOI: 10.1109/INDIN41052.2019.8972087.

[58] Adrián Orive et al. "Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications". In: *Sensors* 22.5 (2022). ISSN: 1424-8220. DOI: 10.3390/s22051755.

Glossary

A

ACOA Application-Centric Orchestration Architecture

API Application Programming Interface

C

COE Container Orchestration Engine

CPS Cyber-Physical System

CPU Central Processing Unit

D

DG Directed Graph

E

EoS Economies of Scales

H

HoL Head of Line

HPC High-Performance Computing

I

IETF Internet Engineering Task Force

IoT Internet of Things

M

MAPLE Measurement Architecture for Packet LatEncies

N

NBP Network Bandwidth Predictor

O

OS Operating System

Q

QoS Quality of Service

R

RAM Random Access Memory

REST Representational State Transfer

RFC Request For Comment, standard published by the IETF

RTT round-trip time

S

SWIM Scalable, Weakly-consistent, Infection-style, processes group Membership protocol

SWIM-NSM Network State Monitoring extension for the SWIM protocol

T

TCO Total Cost of Ownership

U

UGV Unmanned Ground Vehicle

W

WSN Wireless Sensor Network