POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Delivering Resilient Virtualized Services in Smart Grid Environments

Supervisors

Prof. Fulvio RISSO

Candidate

Claudio USAI

October 2021

Summary

Cloud computing is currently a hot topic and the ICT industry is witnessing migration of legacy application to the cloud. Its advantages such as theoretical limitless scalability and high reliability of deployed services looks appealing to every industry that needs an IT infrastructure. The electrical industry is not an exception, monitoring and control systems profoundly rely on a resilient and reliable IT infrastructure. In this industry, latency matters since control systems rely on real time data to work. It is therefore not acceptable to work with data that might be already old when it reaches the control systems and the cloud might not be near enough to where measurements are produced to guarantee an efficient control loop. Edge/Fog computing are paradigms that aim at solving these very issues: bringing critical services as near as possible to the edge of the network. Placing services near to the source of data enables earlier analysis and computation, as well as earlier decisions. In this context technologies like Kubernetes looks very promising but need to be extended or supported by other tools to allow workloads to be run on possibly cheap hardware and making possible to manage thousands of nodes or even clusters that are geographically distributed.

The goal of this thesis work is to present a possible approach to build a resilient infrastructure that is able to run and orchestrate virtualized services on thousands peripheral sites. Multi-cluster solution have been analyzed as well as more specific edge oriented solutions to meet the special needs of the scenario. Particularly, eventualities such as hardware or software failure and network partitioning have been considered as critical and possible issues that the infrastructure should withstand.

Acknowledgements

I would like to thank Prof. Fulvio Risso for all he gave me during the last year as teacher, supervisor and sometimes mentor. It has been a pleasure to work with his support and guidance.

I would also like to thank my family that has always supported me throughout my studies, especially my parents Franco and Cecilia and my sister Giulia.

Another special thanks to my friends and colleagues who have always been there during my ups and downs in the last years and especially in the last few months. Thanks Marco, Giorgia, Leonardo, Michele, Sara, Valentina, Emanuela, Stefano, Giulia, Gaia, Federico, Davide, Luca, Francesco, Claudio.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The classical electrical power system architecture, developed over the past 70 years, had a centralized control. There were big power plants (fossil-fuelled, nuclear power, or hydropower), producing up to 1000MW. The production system interacted with the transport system in order to ensure always the same value of frequency and to receive the required amount of energy. This portion of the power system had an automatized control while the distribution system was almost completely passive, with only local real-time monitoring and control for the largest loads, but no additional interactions between the loads and the power system were performed [1].

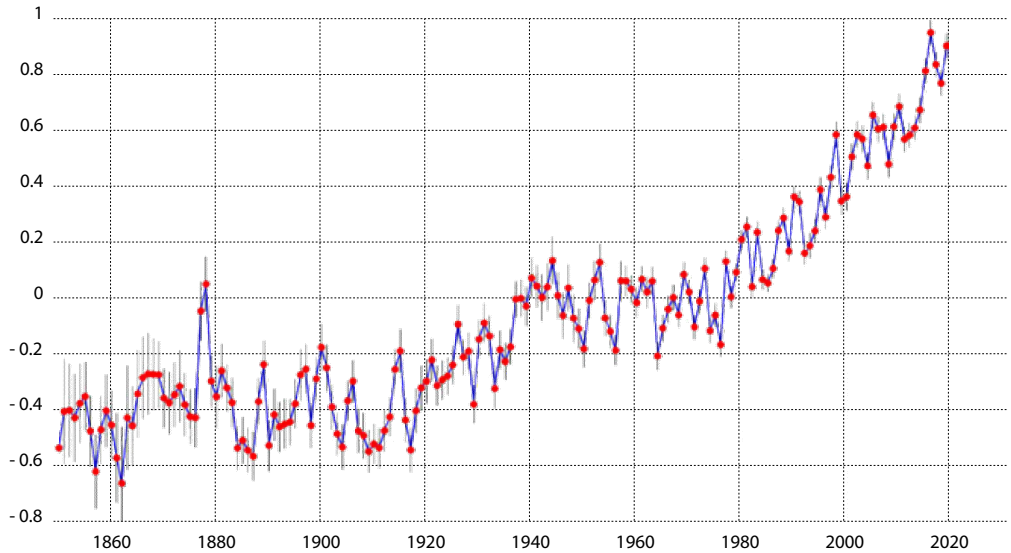


Figure 1.1: Temperature anomaly in C° from 1850 to 2019 to respect the common baseline 1951-1980 mean

From Legambiente climate report 2020 [2] - Berkeley Earth data combined with sea data from UK Hadley center.

The climate changes leading to global warming, driven by the human emissions of greenhouse gases, required the reduction of the produced CO_2. According to NASA, 2020 tie with 2016 was the warmest year on record, with a long-term record of the last seven years, when recorded temperatures were, on average, 1,02 C° higher than the baseline 1951-1980 mean [3]. In order to reduce the carbon footprint, new renewable, green and clean sources of energy were introduced, with some consequences in terms of power grid management. For example, the EU with the *Clean Energy Package* set the target for the 32% for renewable energy sources in the EU's energy mix by 2030, and the goal of carbon neutrality by 2050 [4]. Indeed, the centralized control of the power grid was not enough for a power system where production was not centralized anymore. There was the need to increase the grid observability via a network of sensors providing information about the physical world [5] and allowing the power grid to balance the power supply and the demand. Thanks to the increase of grid observability, new perspectives of automatized control, even in the distribution system, are possible [1]. The usage of the ICT technologies in order to share data from sensors and meters, collect and process it to control the electrical power system is the concept of *smart grid*. However, nowadays, the concept of *smart grid 2.0* [6] has been introduced. It refers to a new design of the smart grid, based on electricity sharing via a plug & play approach. This means that as soon as a new portion of the grid is attached to the main grid, it starts exchanging electricity with the rest of the grid, injecting or absorbing power [7].

1.1 Power grid resiliency with micro-grids

At this point, the concept of *micro-grid* comes into play, as a portion of the grid with loads, accumulation systems, and production systems, able to work attached to the main grid, or *as an island*, which means autonomously, isolated from the rest of power grid. The concept of micro-grid is not new, and in the past, intended as a way of bringing light to remote communities or as a backup system of the main grid. However, the difference lies in how they are powered. While in the past micro-grids relied on fossil fuels, the introduction of renewable sources of energy not only allowed a reduction of costs, but the energy production at the edge of the power grid, improved the reliability [8]. Coming back to the concept of smart grid 2.0. Each micro-grid can be plugged to the main grid, and it can exchange electricity, supporting the main power grid, injecting power, or requiring electricity, if needed. In any case, if the micro-grid detaches from the rest of the grid, on purpose or because of unintended events, it can survive, go on working even though it is isolated. Resiliency, indeed, is nowadays a crucial aspect for power grids, considering the increase of extreme weather events, due to climate changes. The

increase of the average temperatures causes a reduction in rainfall, but a consequent rise of floods, storms, and hydrogeological risk [2, 9]. Figure 1.2 shows the increase of extreme weather events in Italy for each year. It is evident how the world is changing, extreme climate events are becoming much more frequent, and people are called to get used to this new normality. Human infrastructures need to be redesigned for this new world, to be resistant to the weather pattern of the future. Even the electrical power system, indeed, should be able to predict, react and survive these extreme events, and it is crucial for the design of the smart grid 2.0.

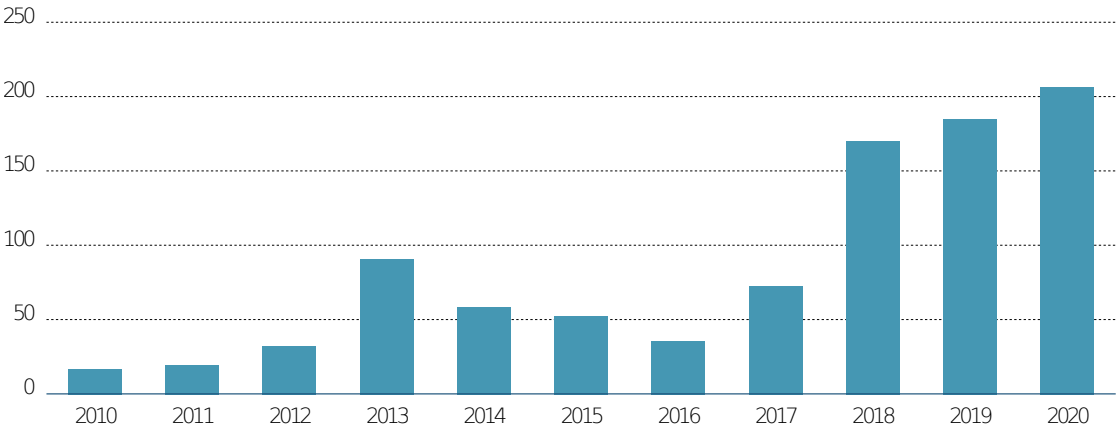


Figure 1.2: Extreme weather events in Italy for each year
From Legambiente climate report 2020 [2] - Osservatorio Cittá Clima, Legambiente 2020

1.2 ICT resiliency in a smart grid 2.0

Smart grid 2.0 requires real-time control for more than 50% of the power demand, requiring the monitoring of great amount of data coming from sensors and devices. The analysis and processing of this huge quantity of data and the control of the grid require the deployment of smart IT technologies and usage of big data analysis techniques [7]. A practical example of the need for data and their utility is the post incident analysis carried out after the August 9th 2019 transmission system frequency event in Great Britain [10], where the authors used phasor data available to try to reconstruct the different stages of the incident and to learn a lesson from it. Moreover, the resiliency of the power grid should be provided by a robust ICT infrastructure.

- Services have to be monitored to launch them again, in case of a failure of the application itself or the node where it was running.
- As the power grid should be able to support partitioning, even the ICT must support the partitioning of the infrastructure. Extreme weather events,

accidental events, or network failures might isolate one or more sites of the electrical power system. When the site is isolated, its ICT infrastructure should be able to react, go on working, even though the connection with the centralized control has been lost.

The ICT of the power grid should handle the complexity due to the widely geographically distributed infrastructure. The solution scalability is crucial since it handles hundreds of thousands of sites, and this number can easily grow over time. Therefore, the solution must allow new sites to seamlessly join the rest of the infrastructure, according to the concept of plug&play electrical grid. A huge quantity of devices and sensors of different nature, all over the power grid, some of them with low computational power, produce data over different physical media. The role of the ICT of the power grid is allowing this huge amount data to safely reach all the consumers, according to their requirements in terms of QoS. All the services running over the smart grid should be able to produce and consume data, transparently moving across the nodes of the ICT infrastructure, if needed. Data should be produced and consumed with an asynchronous approach in order to improve the scalability, maintainability, and simplicity of the applications, still keeping latencies under control, supporting real-time applications.

1.3 Overview of service resiliency on power grids

This thesis work will focus on the aspect of resilient services in smart grid environments, leaving the aspects related to resilient communications to be explored in another thesis work. Chapter 2 presents an overview of the ICT architecture in power grid environments, this is necessary to give background context to understand the motivation of choices presented later on. In chapter 3 several technologies are considered, all based on Kubernetes, to tackle the problem of resiliency at the edge. First Kubernetes distributions are presented and then multi cluster solutions and edge oriented solutions. An architecture is then defined in chapter 4 taking into account the various sides of the problem, including service communication and latency issues, and explaining the choices made. The developed implementation is presented in chapter 5, here practical choices are explained as well as the reason for choosing some technologies or pattern over others. In chapter 6 numerical results are presented, focusing on resource consumption and comparing results with equivalent deployment in different conditions and devices. Chapter 7 sums up the work and points out possible aspects worth to be further explored.

Chapter 2

ICT architecture in an electrical power grid

The aim of this chapter is to provide an overview of the ICT infrastructure in the electrical power grid. The models to be used in the exchange of information with distributed energy resources are defined by the IEC-61850 standard. The electrical grid can be divided into three slices, each of them having a different role:

- *Production system*: where the electricity is produced, converted with the right values of current, voltage and frequency and finally introduced in the transmission system. In the past, this was mostly done in huge production plants (e.g., hydroelectric, coal), while in recent years this is being integrated with many small-size production plants (e.g., solar power).
- *Transmission system*: in charge of collecting the electricity from the power plants and transporting it to the distribution systems (i.e. Terna in Italy).
- *Distribution system*: in charge of bringing the electricity to the final users, typically this part of the network is in charge of the energy providers.

Nowadays, the production system is not anymore the only source of energy, due to the presence of many small producers closer to the user, such as solar panels, wind farms and more. This means that even in the distribution systems there is the need to replicate the mechanism present in the production system, not having anymore the possibility to have a completely centralized control, but it was needed to move this control even at the edge.

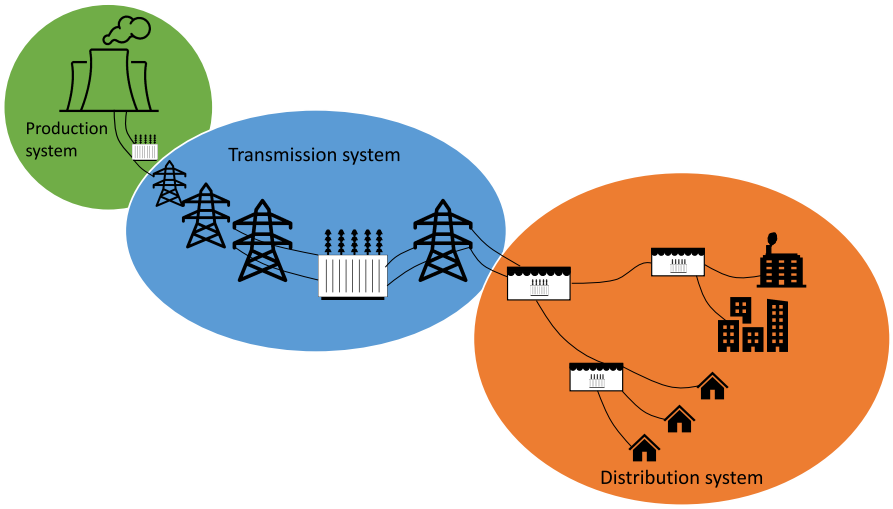


Figure 2.1: Electrical hierarchy overview.

2.1 Production system

The production system is made by energy producers, who produce electricity and collect it in the transmission system by means of some transformers, regulating voltage and intensity of the electricity. Producers need to regulate the frequency of the produced energy according to the values provided by the national controller, so that generators can always keep the same value of frequency and provide the required amount of electricity in the electrical grid.

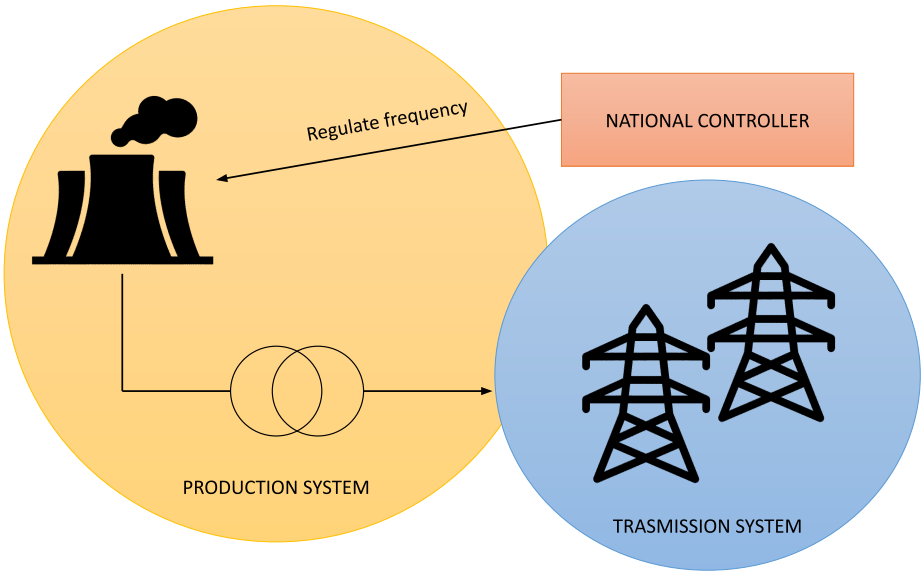


Figure 2.2: Production systems work in synergy with the transmission system.

2.2 Transmission system

The transmission system is made of electrical towers of 380kV, 220kV and 130/150kV all connected with the others, forming an unique grid covering the entire national surface. This system is controlled by some stations with a set of transformers converting the ultra-high voltage to the high voltage. Electric substations (often abbreviated SSE) are located near a production plant, at the point of delivery to the end user and at the interconnection points between the lines: they therefore constitute the nodes of the electricity transmission network. Substations perform one or more of the following functions:

- interconnect multiple High Voltage power lines at the same voltage level, creating a network node (via crossbars);
- interconnect several HV power lines with each other at different voltage levels (through transformers);
- re/phase the apparent power of the network (by means of capacitor banks or power factor correction inductors, also called "reactors" as they absorb reactive power);
- convert the voltage from AC to DC and vice versa (conversion substations). [11]

Even these transformers have some sensors and actuators, the latter are controlled by devices called IED (Intelligent Electronic Device). All the devices running locally, e.g.,in a substation, are connected to each other by means of an Ethernet LAN, which also includes a Station controller, e.g., a server with the proper controlling software. Logically, the station controller is connected with the Regional controller, which is further (logically) connected to a National controller.
The physical network connection between each station and the rest of the ICT network is usually achieved with dedicated links; in the past, this infrastructure was completely under the control of the Electrical company (i.e., ENEL), which was then spinned-out at around 1990-2000 when the Italian telecommunication market was open to competition, leaving to the creation of the Wind telecommunication company. Nowadays, the above physical network connections are in part still under the control of the Electrical company, while others are simply links bought from a telecommunication provider.
Electricity cannot be stored, therefore there is the need to guarantee the balance between the produced energy with the demand. This operation is a real time control called dispatching and it is under the responsibility of the National controller, which acquires data from a large number of players operating both in production and demand, performs forecasts about the national electricity requirements and interacts with producers and remote management centers in order to module the supply and structure of the grid as require. [12]
In this case, the network is based on optical fibers running through the overhead protection cables, but still having a satellite network as backup.

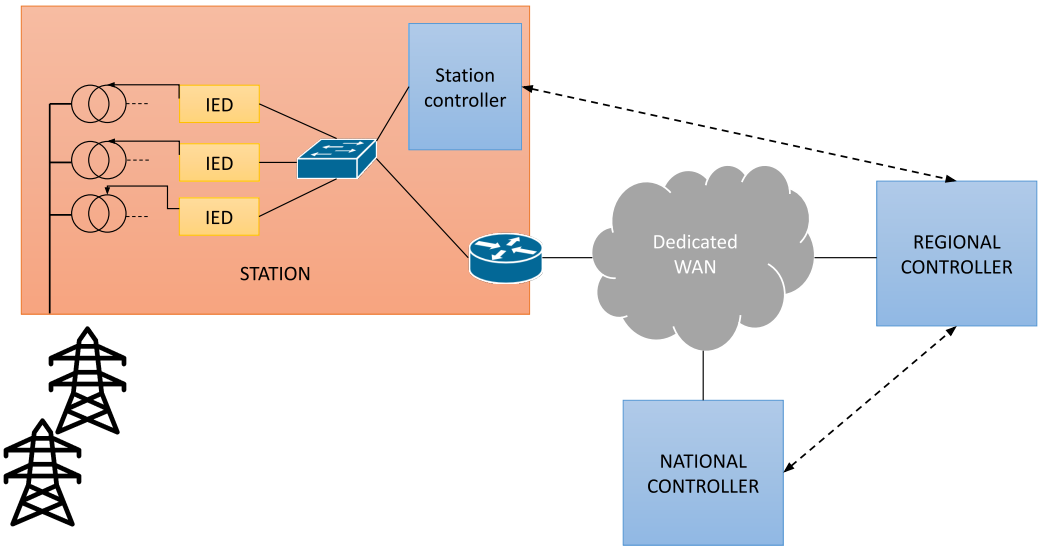


Figure 2.3: ICT network architecture of the transmission system.

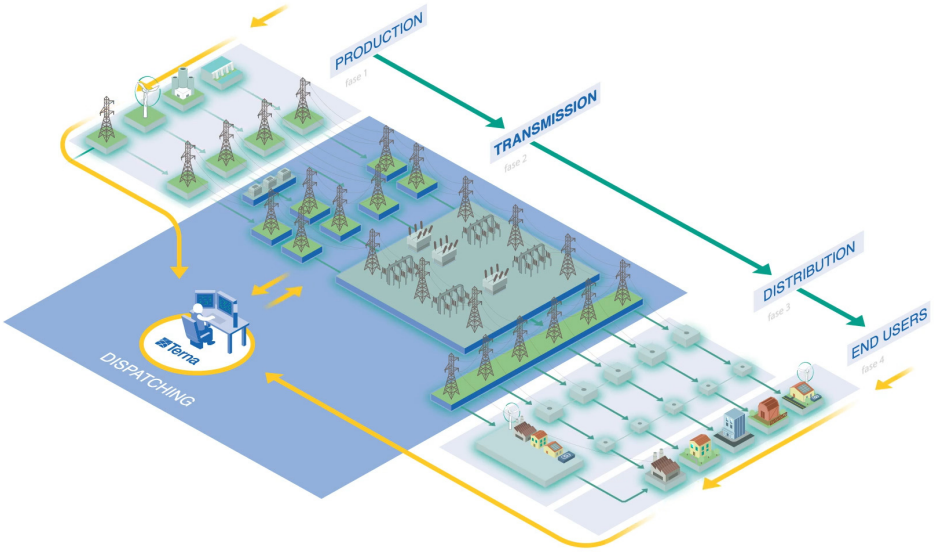


Figure 2.4: Terna's transmission system.

2.3 Distribution system

The distribution system starts from the primary substations where the high voltage electricity is converted into medium voltage. Here a set of measurement systems are used in order to track the state of the transformers, and to perform some adjustments opening and closing them, changing the transformation ratio, in order to keep the correct working point of the electrical grid. This is the starting point for the medium voltage lines, each controlled by a switch. These lines arrive at the secondary substations, where the medium voltage is converted to low voltage. These are the starting point for the low voltage lines, connected to user loads, electricity generation systems, accumulation systems, which could also have some meters and sensors providing information about their working status.
Inside kiosks sensors and actuators are connected via Ethernet LAN, while data from loads, electricity generation systems, accumulation systems coming from the outside could reach the controller in charge of handling it, using GSM, 4G or powerline. The distribution system has three main levels of control:

- Primary Substations: here there could be data coming from the inside of the substation, but also for the outside world. This data is sent to the station controller in charge of performing a local control.
- Area control centers: the station controllers of the substations exchange data with the area control center of the geographical area where they are located, which could be an entire city or a portion of it.
- ICT control center: it is the remote monitoring center for the ICT of the electricity provider, its role is configuring all the devices, monitoring the state of the infrastructure, checking for anomalies, such as failures or intrusions, trying to recover it from the effects of an incident. This component is also present in the transmission system.

Typically, data flowing between the control centers is carried over a dedicated WAN network, which might be made of fiber or equivalent technology. Each section of the network has a firewall, filtering incoming and outcoming traffic. The overall ICT system is also protected by an access point which performs some encryption to the incoming and outcoming traffic. That's because different distributors and the ICT of the transmission system, uses different keys, in order to keep them independent. This means that all the outcoming traffic should be decrypted with the internal key, then encrypted with the key shared with the destination, and then decrypted again and encrypted with the key of the destination. The same thing should happen with communications between the transmission systems of two different nations, this is needed because, since Europe runs on a single frequency, a variation of the frequency or a failure of a part of the grid, might affect all the other nations, which

should properly react. This makes evident how a good ICT system is crucial for a properly working power grid. The ICT, indeed, should be in charge of monitoring the working status of each component and tuning each of them in order to provide the desired state, but it has also the role of protecting it, for example detaching from the grid a power plant which goes out of frequency. Even though both control and protection allow to keep the correct working status of the power grid, they are totally independent, since they have different requirements even in terms of reaction time.

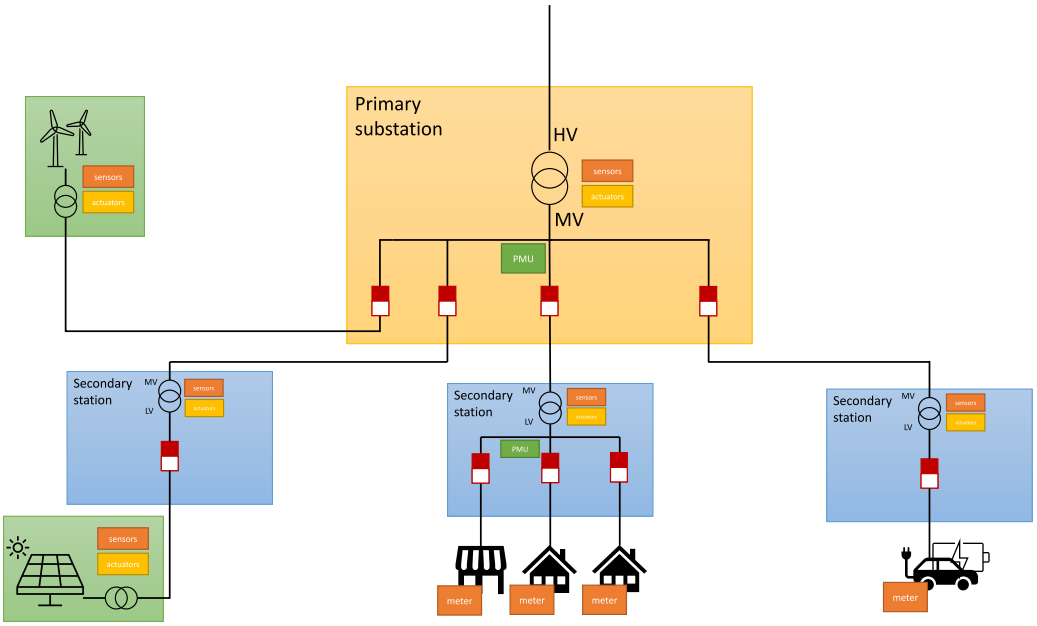


Figure 2.5: Electrical architecture of the distribution system.

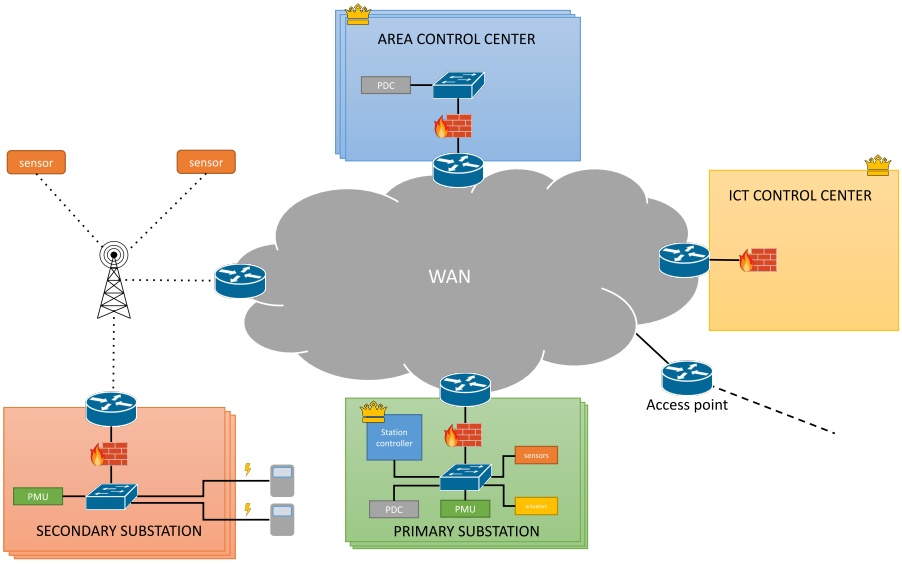


Figure 2.6: ICT network architecture in the distribution system.

Chapter 3

Related work

3.1 Kubernetes

During the last decade application development and deployment have seen a major change with the arrival of Docker and mainstream container technologies that rapidly took over standard virtual machines due to their ease of use, build, deploy and to their lightweight nature. Along with that, microservices based architectures has become increasingly popular and nowadays applications may include hundreds of them in their architecture. The need for scalability required the ability to manage thousands of instances in order to keep up with customer's request and made hardly feasible to operations engineers to manually manage these huge numbers. The necessity for this new kind of automation, management and monitoring lead to the rise of orchestrators, with Kubernetes becoming the de facto standard due to its being open source and strongly supported by an active community that is still growing. Kubernetes was originally developed by Google and then made open source in 2014, inspired by a decade of experience deploying scalable distributed web applications and with the reliable technology of linux containers. Its being open source and vendor agnostic are the main reason of its popularity along with the possibility for the community to always bring new features both as custom extension or as feature in the main codebase. Kubernetes allows to define application deployments in a declarative way, giving administrators the possibility to store configurations in yaml and json files that can be deployed with few commands and even in full automation using DevOps CI/CD practices. A brief overview of the basic concepts of Kubernetes will be given including core resources and some of its main components. [13]

3.1.1 Basic concepts

A Kubernetes *cluster* is composed of *nodes*. A node is a resource that abstracts the concept of hardware (it may be a physical server, a VM or even just fictitious) and can be a *master* node or a *worker* node.
A *master* node is where the Kubernetes control plane runs, and exposes the *API server* which can be used by an administrator to get cluster information or for applying changes to the cluster (i.e. deploy a new application) as well as by operators, custom software, to watch resources and apply changes to them.
Kubernetes is made for orchestrating containers, however a user cannot actually just run a container the same way it is done with docker. The smallest deployable unit is a *Pod*, a resource that represents a group of pods and its configuration. For example, in pod's manifest can be declared labels to identify the resource, volumes to be mounted, environment variables, open ports, container restart policy, etc. Kubernetes will take care of making sure that the app is started once volumes are available, container images are pulled and other prerequisites.
Configuration in a container environment cannot always be know at time of deployment, may change during time and the same configuration might be used many times. For this reason *ConfigMaps* and *Secrets* offer the possibility to define simple key-value pairs that can be either used as strings in pods templates or mounted as files in pods' volumes.
However, pods usually are not deployed "as is", applications are mainly deployed as *deployments*, a resource that provides a way to specify the number of replicas. The deployment controller will provide self-healing, making sure the number of running replicas stays constant, restarting containers or reinstantiating pods, if necessary.
In kubernetes exposing applications can be easily done using *services*, an abstraction used to make a pod reachable either at cluster level or from outside the cluster. A declared service must specify a label selector that should match the target pods, the ports to be exposed and the target ports of the container in the pods. In case an application is to be reached only by inside the cluster the type ClusterIP must be set, otherwise NodePort and LoadBalancer types are available.
Persistence is managed through the *PersistentVolume* and *PersistentVolumeClaim* resources, the first represent the volume that resides in physical drive whereas the latter represent the request for a persistent volume made by a user. [14]

3.1.2 Core modules

As previously said, a cluster is composed of master nodes, where runs the control plane, and worker nodes. The control plane, in turn, is composed of several modules:

- *kube-apiserver*, the component that exposes the kubernetes API. Several instances can be run in order to perform load balancing.

- *etcd* is the backing store of kubernetes, it is a consistent and highly available key-value store. Its high availability mode enables cluster to have more than one master and reduce outages due to master failures.

- *kube-scheduler*, a component that watches the pod resources and when a new one is created takes care of choosing a suitable node and schedule it, based on taints, affinities and other kind of configurable constraints.

- *kube-controller-manager*, a component that runs the different controller processes. Each controller is, logically, a separate process but simplicity they are compiled into a single binary. Example of controllers are: Node controller, deployment controller, Service Account and Token controllers.

There are then a set of components that are common to both master and worker nodes:

- kubelet, an agent that is responsible of making sure that containers declared in a Pod are running. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet manages only containers created by Kubernetes.

- kube-proxy, a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. Kube-proxy maintains network rules on nodes that network communication to Pods from network sessions inside or outside of the cluster. It leverages the operating system packet filtering layer if any, otherwise forwards the traffic itself.

- CNI, Container Network Interface is a CNCF project that defines a specification and libraries for developing plugins used to configure network interfaces in Linux containers. If a CNI network plugin is installed in a cluster, it is automatically done in each node.

- CSI, Container Storage Interface like CNI is a specification for developing storage drivers plugins used in linux containers. A CNI might bring features such as volume replication, snapshotting and backup.

- CRI, Container Runtime Interface is defaulted to containerd, but it is possible to use also Docker (deprecated) or CRI-O.

3.2 Kubernetes for the edge

3.2.1 k3s

K3s is a certified Kubernetes distribution that aims at simplifying the setup and management as well as reduce the overall load on the host machines. In fact it has been introduced as the "lightweight kubernetes" due to the removal of several lines of code from the codebase that are needed only when Kubernetes is run on a cloud provider environment. As a consequence k3s suits well for deployments on bare metal with relatively low computational resources (compared to data center hardware) and especially for development environment and the edge. Due to some differences on the codebase, especially the lack of `kubeadm`, setting up a hybrid cluster with nodes running vanilla Kubernetes and others running k3s is not supported, even though it is undocumented it should still be possible. K3s comes packed with a set of dependencies,

- containerd
- CoreDNS
- a local path provisioner CNI is present to give the possibility to mount volumes from the node's filesystem without the need to manually install plugins.
- a default ingress controller (traefik [15])
- an embedded load balancer
- an embedded network policy controller
- a default lightweight CNI Flannel [16]

Each of the previous plugins can be disabled in order to give the user the possibility to customize the cluster behaviour.
The installation script provided gives also the possibility to set all of the options available on the upstream Kubernetes as well as some specific flags that target k3s only behaviours. [17]
Its resource consumption has been analyzed in *Bohm and Wirtz* [18] where a comparison with the microk8s, which will be presented in the next pages, and with vanilla Kubernetes. The researchers used four Ubuntu 20.04 Virtual Machines (VMs) with 2 vCPUs, 4 GB memory and an SSD with a capacity of 50 GB each. The results showed very similar resource utilization for CPU and Memory, with k3s that spared the most in disk usage with respect to the two others.

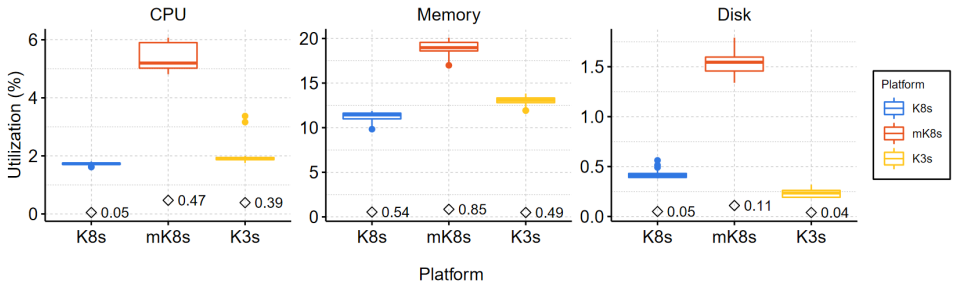


Figure 3.1: Resource utilization for K8s, MicroK8s, and K3s.

3.2.2 MicroK8s

MicroK8s is a production ready Kubernetes distribution that targets edge and IoT applications as well as CI/CD and development environment. It is based on the upstream Kubernetes codebase, with minor changes aimed at reducing the memory footprint, and comes with an additional command line interface `microk8s` that can be used to enable its additional features. With respect to vanilla Kubernetes, Microk8s offers interesting features that can be enabled out of the box through the cli:

- Self-healing high availability clusters, MicroK8s automatically chooses the best nodes for the Kubernetes datastore. When a cluster database node is lost, another node is automatically promoted, never leaving the cluster without at least one master.

- dqlite: instead of using etcd as backing store dqlite has been chosen offering more flexibility that is needed by the dynamic promotion of nodes to master.

- LXD: possibility to run microk8s as immutable container.

- configuration defaults: MicroK8s defaults to the most widely used Kubernetes options, providing also networking, storage and standard services with standard configuration out of the box.

- multi-vendor compatibility: just like Kubernetes, it is compatible with most popular cloud provider

- bundled for production: it is already bundled with tracing, metrics, service mesh and registry add-ons.

3.2.3 FLEDGE

Devices at the edge are powerful enough to be able to run containerized microservices and keeping a low size and power makes them suitable to be deployed almost anywhere. They may however not be able to run an orchestrator due to their high resource requirements. In *Extending Kubernetes Clusters to Low-resource Edge Devices using Virtual Kubelets* [19] the authors present FLEDGE, a low-resource container orchestrator which is capable of directly connecting to Kubernetes clusters by using Virtual Kubelets, a VPN to secure traffic between nodes and *containerd* as container runtime.

FLEDGE aims at being compatible with most Kubernetes API so that other tools can still be able to extract data as from a standard Kubelet. The API calls supported by a Virtual Kubelet consist of pod management, pod status, node status, logging and metrics. Networking is handled differently than in Kubernetes, there is no CNI installed since the FLEDGE agent takes care also of the networking. The assumption is that on such devices the number of pods deployed will be relatively low and it is therefore more appropriate to develop a simple and naive pod networking handler. Moreover FLEDGE still uses the CIDR defined in Kubernetes, this approach does not influences the networking in the rest of the cluster. The paper also analyze the problem of the Virtual Kubelet placement (cloud vs edge) and, in case of a high number of FLEDGE nodes, points out that the it could saturate the maximum number of pod deployable in each Kubernetes nodes which is 200.

FLEDGE's resource usage has been evaluated and compared to Kubernetes and k3s, showing improvement on the storage and memory footprint, particularly on ARM64 architectures. A very strong upside is that, even if the comparison with k3s does not highlight relevant differences, there is possibility to join K8s clusters.

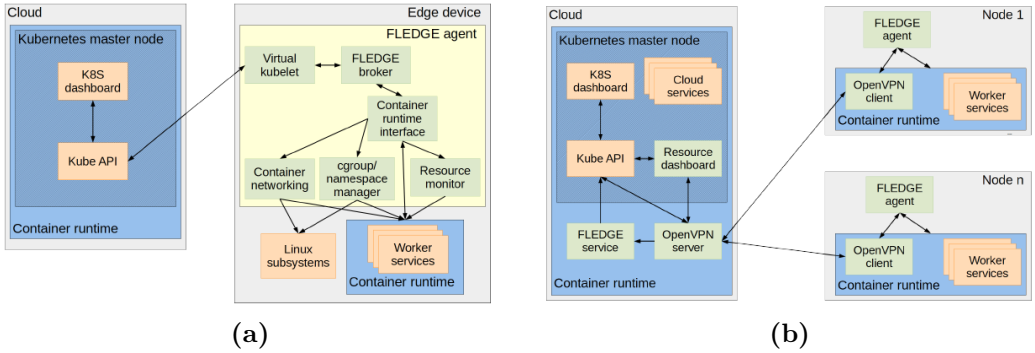
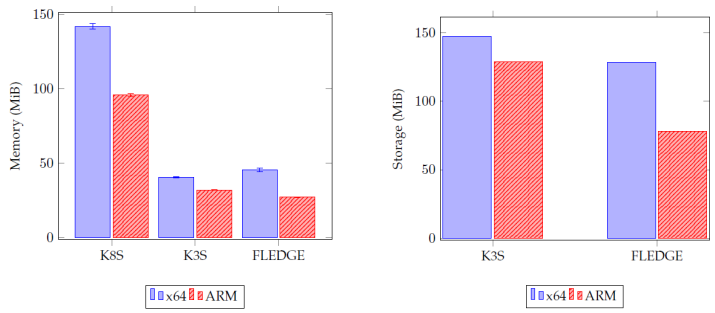


Figure 3.2: Virtual Kubelet use in FLEDGE 3.2a, Network traffic example 3.2b



(a) **(b)**

Figure 3.3: Memory footprint comparison 3.3a, Storage footprint comparison 3.3b

3.2.4 KubeEdge

KubeEdge is a Kubernetes native edge computing framework, it is not a kubernetes distribution and aims at providing a set of tools to run workload at the edge. The architectures cornerstone is the CloudCore component, which runs as a workload on a Kubernetes cluster on the cloud and is responsible of managing the various edge nodes. Each edge node runs an instance of the EdgeCore componenent. EdgeCore can run containerized workload on demand by the CloudCore component, using either Docker (default), containerd, CRI-O or kata-containers as container runtime, and provides different features such as local storage interface, networking, configmaps and secrets replications.

- Edged: an agent that runs on edge nodes and manages containerized applications.
- EdgeHub: a web socket client responsible for interacting with the CloudHub component. This includes syncing cloud-side resource updates to the edge and reporting edge-side host and device status changes to the cloud.
- CloudHub: a web socket server responsible for watching changes at the cloud side, caching and sending messages to EdgeHub.
- EdgeController: a controller that manages edge nodes and sets pods metadata so that the worload can be targeted to a specific edge node.
- EventBus: an MQTT client to interact with MQTT broker (Mosquitto [20]), offering publish and subscribe capabilities to other components and external IoT devices.

- DeviceTwin: responsible for storing device status, syncing device status to the cloud and providing query interfaces for applications.
- MetaManager: the message processor between edged and EdgeHub. It is also responsible for storing/retrieving metadata to/from a SQLite database.

[21]

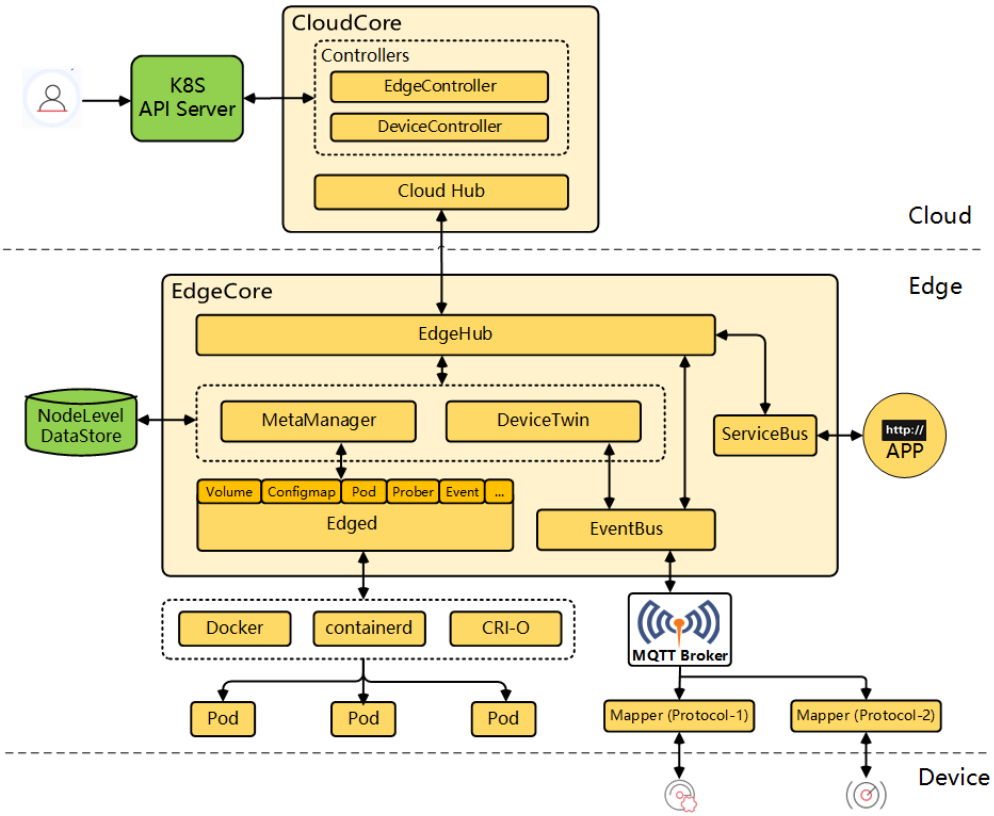


Figure 3.4: KubeEdge architecture

3.3 Multi-Cluster

As the cluster topology grows in complexity, different approaches are needed in order to manage and provide resiliency to the system. In the case of kubernetes clusters, a possibility is leveraging multi-cluster technologies to manage multiple clusters as a whole and possibly defining policies for workload offloading and scheduling. In this section a selected subset of multi-cluster solution will be briefly analyzed to give the reader an overall view of the possible ways that have been considered to tackle the problem of managing such a high number of clusters.

3.3.1 KubeFed

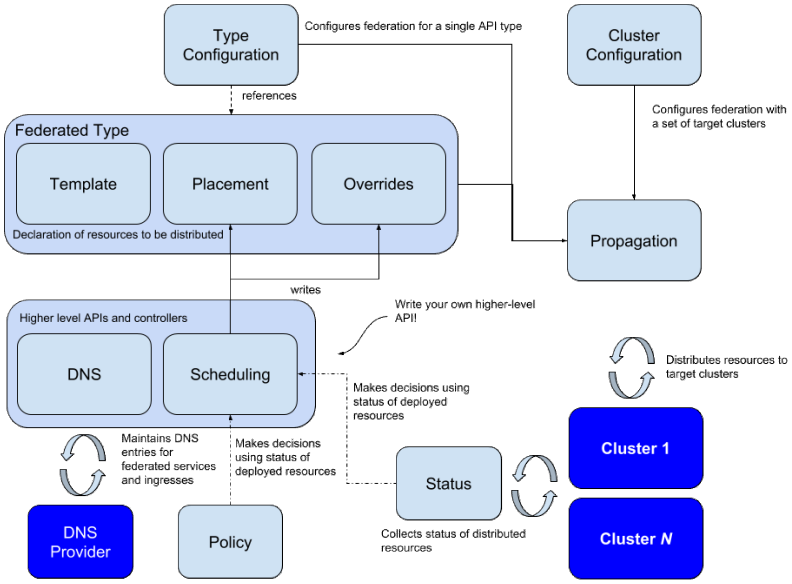


Figure 3.5: KubeFed's workflow [22]

KubeFed (Kubernetes Cluster Federation) allows to coordinate the configuration of multiple Kubernetes clusters from a single set of APIs in a *host cluster* that will then propagate the configuration to *member clusters*. The concept of *federation* is intended as creating a common interface to a pool of clusters, or to types, which can be used to deploy Kubernetes applications across those clusters. The project aims at providing low level mechanism to be used as foundations to build more complex multi-cluster solutions to use cases such as deploying multi-geo applications and disaster recovery.
KubeFed uses *federated types* to define which types should be *federated* across specified clusters of the pool (i.e. configmaps, namespaces, deployments). A *federated type* will include the actual resource inside its template as well as the clusters where it is desired to be placed, listed in the placement spec.
These previous abstractions provide building blocks that can be used by higher-level APIs:

- Status, which collects the status of resources distributed by KubeFed across all federated clusters
- Policy, which determines which subset of clusters a resource is allowed to be distributed to

- Scheduling, which refers to a decision-making capability that can decide how workloads should be spread across different clusters

[23]

3.3.2 A decentralized control plane

Devices at edge are by definition geographically dispersed and the number of devices managed by a single administrative authority could reach thousands. For this reason KubeFed might not fit well with the scenario. Particularly, having a host cluster, which can be considered a pro from an administrative perspective, it is a con since is also a single point of failure. Moreover the project targets clusters that are distributed across different world regions defined by cloud providers and therefore targets infrastructure composed of dozens of cluster, whereas in an edge scenario hundreds or even thousands would be a more realistic reference number.

In *Larsson et al.* [24], the authors describe a possible approach still based on KubeFed but with important modifications on its backend, using CRDTs. Conflict-free Replicated Data Types (CRDTs), are a class of distributed data types with mathematically provable conflict free properties [25] and that find practical application in distributed databases.

The main problem of KubeFed is that the host cluster act as "truth" while member clusters are only passive participants of the federation since is the federation controller, deployed in the host cluster, that constantly watch a updates resources in member clusters. The solution proposed is to keep the KubeFed API, in order to maintain compatibility with what has already been defined, but deploy different controllers in each cluster that watch a CRDT database storing the distributed resources. The database analysed in the paper is Riak KV, which implements CRDTs and has clustering deployment possibility that might span acrosso multiple Kubernetes clusters. Riak KV gives up consistency in favor of availability in the case of network partitions and to achieve low latency [26]. The proposal uses than a "local cluster" backend based on etcd and the the "multi-cluster" backend based on Riak KV, the federation controller in each cluster would then watch the distributed resources in Riak and act accordingly in case of any discrepancy i.e. scheduling new pods. The work explains possible examples of use and also proposes solutions to problem that might arise in a possible implementation, however the proposal still remains without an actual implementation.

3.3.3 Fog-Atlas

Fog-Atlas is a framework whose objective is to manage a geographically distributed and decentralized infrastructure that provides computational resources, storage and network services following the Fog Computing paradigm. The framework proposes

two different deployment modality: the conventional way with a single cluster with nodes that span across the geographic region and a multi-cluster way,based on kubefed, where a cluster act as *host cluster* while others act as *member clusters*. Obviously the latter approach is much more effective in delivering resiliency to the distributed infrastructure, even though in some cases it might perform well due to its low complexity.
The concept behind the federated approach is similar to the one used in vanilla KubeFed, the administrator can apply the desired deployments directly on the host cluster and the *fadepl-controller* (Fog Atlas Deployment Controller) will take care of propagating the desired configuration, apps and services to the specified clusters. Additionally the framework defines its CRDs that are used to configure the federated resources and additional constraints that model a distributed infrastructure: *Region*, *Link*, *ExernalEndpoint*. FogAtlas defines also a set of CRDs that describe the federated application deployment in order to allow the controller to perform the application placement based on requirements declared in these resources: *FADepl* (Fog Atlas Deployment), *FedFAApp* (federated application). Trough the Fadepl is possible to specify the region where an app or service is needed and other requirements such as the need for a GPU, a given CPU architecture as well as geographic based constraints and data flow to/from other microservices (*FADeplDataFlow*). In case of network partitioning (i.e. a member cluster becomes isolated from the others) the member clusters maintain their own autonomy and resilience. However, at the moment of writing, the reinstantiation of a remote service that is no longer available due to the isolation is not performed automatically by the framework and must still be performed manually by an administrator. This limit can be overcame leveraging the operator pattern and writing the software that takes care of this specific task. Although, the framework might still lack of some additional information embedded in the CRDs needed to identify which services should be reinstantiated locally (i.e. some kind of application dependency graph and priorities) and it would mean that additional complexity is left to the user of the framework. [27] [28] [29]

3.3.4 Liqo

Liqo is an open source project started at Politecnico of Turin that allows Kubernetes to seamlessly and securely federate multiple clusters, sharing resources and services to run workloads across them in a *liquid computing* fashion.
In contrast with the approaches presented previously, Liqo does not use the KubeFed framework to achieve cluster federation. Instead it is based on the concept of Virtual Node, an approach that has already been used by many important projects and proved to be effective.

Virtual Kubelet is an open source Kubernetes kubelet implementation that

masquerades as a kubelet for the purposes of connecting Kubernetes to other APIs. The project has been made public by Microsoft and it is known to be used in cloud providers' serverless contexts to allow running workloads on a dynamic environment that might not be a Kubernetes node but instead an entire pool of servers orchestrated by some external API. Another use case, as in Liqo, is to masquerade another cluster to allow cluster federation, this way another cluster can be seen as a (virtual) node from the home cluster point of view, and workloads can be easily scheduled on them.
Liqo's main features are:

- *dynamic discovery*: Liqo can dyanmically discover other liqo clusters in the same LAN and start a peering
- *resource reflections*: resources are automatically reflected to the other clusters participating in the peering. For example a Pod will be seen by the control plan of each cluster but only one will actually be running, the others will be just ghost resources.
- *multi-cluster networking*: Liqo's *network fabric* takes care of setting up the proper rules to enable traffic redirection to the offloaded workloads. This is done through a wireguard tunnels between cluster and through an overlay network inside single clusters.
- *ease in setup*: the project is focused also on making it simple to deploy and being ready to peer in minutes.

Liqo relies on different CRDs and operators to integrate its custom logic into Kubernetes. In Figure 3.6 is depicted an example deployment within two cluster, showing its core components:

- *liqo network manager*: manages the creation of the `networkconfigs` CRDs for the remote clusters and processes the ones received from the remote clusters. In case remote networks overlaps with any of the address spaces used in the local cluster, it takes care of remapping them to avoid collisions. When the `networkconfigs` have been exchanged between the two clusters the Liqo Network Manager creates a new `tunnelendpoints` resource, which describes the interconnection between the two clusters.
- *liqo gateway*: is made up by several operators and is responsible for establishing secure tunnels to other peering clusters and inserting NAT rules for the remote pod and external CIDRs. The tunnel operator, given a `tunnelendpoints` resource, creates a vpn tunnel to the cluster described by the CR and adds a static route for the remote cluster. The NAT Mapping Operator, given the customer resource *natmappings*, configures NAT rules to send the incoming

traffic, destined to an external CIDR IP address, to the right workload. The *liqo gateway* is deployed in High Availability, a labeler operator takes care of making sure that only the elected gateway will have the proper label that enable traffic redirection to it.

- *liqo route*: is also made up by several operators and run on each node of the cluster. It is responsible for creating the overlay network used to communicate between nodes and to set network rules that redirect outgoing traffic to the active liqo gateway.

[30] [31]

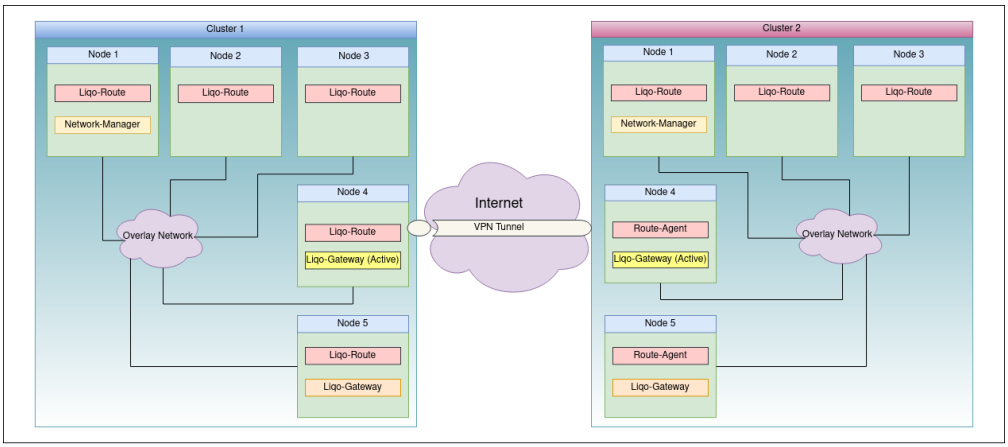


Figure 3.6: Liqo's components in an example deployment

3.3.5 Tensile-Kube

Tensile-Kube is a an implementation of Virtual Kubelet provider that enables multi-cluster management using a centralized approach. It has been developed by Tencent Games for sharing hardware resources between different clusters in a data center environments. The overall architecture distinguish a *upper cluster* and possibly several *lower clusters*. It uses the virtual node approach, hiding each lower cluster behind a virtual node, a virtual kubelet pod running in the upper cluster that advertise itself to the upper cluster as a worker node and takes care instantiate and manage the resources in the related lower cluster. The project includes, aside from the virtual kubelet provider, a *multi-cluster scheduler*, a *descheduler* and a *webhook*. The multi-cluster scheduler watches all of the lower clusters' capacity and call a filter function while scheduling pods, the descheduler is responsible of enforcing affinities and other placement policies, they can be both deployed or only the latter in case of very high overall number of nodes. The webhook converts some

fields into annotations that would otherwise create instabilities at scheduling time. Tensile-Kube was born with the purpose of sharing computing resource from different clusters in the same data center. For this reason it does not take any responsibility in the networking side of the infrastructure since the various nodes. In the example deployment presented in the CNFC blog post, all clusters use a Flannel CNI using as backend the same etcd instance, making pods and services addresses automatically synchronized between the different API servers. In a real scenario this can be an issues since it is not always possible to have a single etcd backend for all the clusters' CNI, particularly in an edge environment. [32]

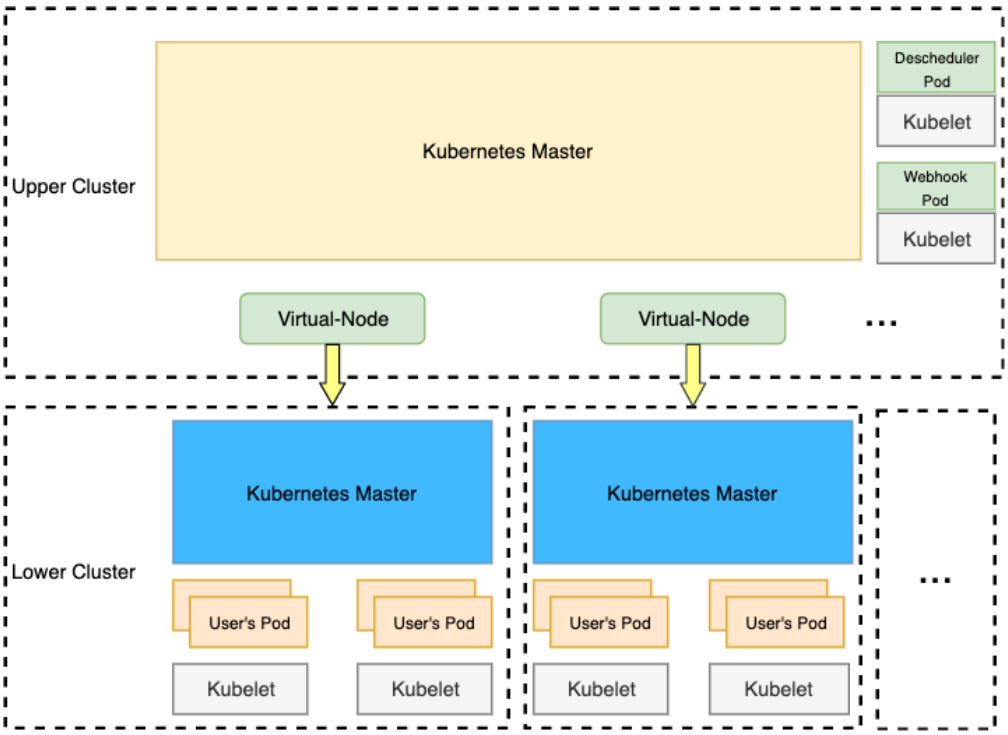


Figure 3.7: Tensile Kube architecture.

3.4 Edge device orchestration

With the rise in popularity of paradigms such as edge and fog computing, an increasingly number of challenges appeared when tried to be put in place. Edge devices are per-definition not under strict control of the owning companies, they are geographically distributed, they might be diverse in hardware and specification, they must be reliable enough to guarantee a certain level of availability of the provided services. Application deployed on such hardware should not be aware of these

challenges and business logic should stay somehow the same as if the distributed system would run on a cutting edge hardware in a data center. Kubernetes alone cannot offer this kind of features, especially if the infrastructure is comprised of thousands of nodes. This need has been identified and in the last few years different approaches have been developed. In this section will be given an overview of such technologies, in particular *StarlingX* and *Eve-OS* try to respond to the need for a common foundation where to deploy application on the edge. In addition Rancher Fleet, even if quite different from the previous, might as well address some of these problems in its own way.

3.4.1 StarlingX

StarlingX is a project supported by the Open Infrastructure Foundation, it is a software stack that aims at providing an OS bundled with all the necessary software needed to deploy an edge cloud on up to 100 servers. Its key features include:

- single package that includes an optimized version of the linux distribution CentOS, storage and networking components, and all the cloud infrastructure needed to run edge workloads.
- predefined configurations to meet a variety of edge cloud deployment needs.
- optimized for security, ultra-low latency, extremely high service uptime, and streamlined operation.
- fault management and service management capabilities, which provide high availability for user applications.

StarlingX puts its focus on the importance of low latency in edge computing and the ability to run any kind of workload. Its use cases range from mini data-centers near telecommunication operator towers to on premise clouds for industrial automation, hospitals and transportation. The project does not state official hardware requirements but it is clear that is expected to run on servers and not on low end hardware.
StarlingX relies on Kubernetes to run workload on containerized environments and on a containerized OpenStack to run workloads on VMs in case of hard requirements on kernel version or OS.
Different deployments configurations are possible depending on the edge site dimension:

1. All-In-One server running control plane, storage provisioning and workloads.
2. Highly Available Multi-Server with some servers dedicated to the control plane, some to storage and other to running workloads.

3. Distributed Cloud: a *central cloud* and several autonomous *subclouds*.

The latter implements the OpenStack Edge Computing Groups's MVP Edge Reference Architecture, specifically the "Distributed Control Plane" scenario [33]. This scenario is comprised of a centralized datacenter where monitoring and critical applications are placed and several medium to large sized edge sites called subclouds where latency critical application are placed. Central cloud and subclouds are connected through an L3 network, the connection to the central cloud must be configured at setup time. Subclouds can belong to groups to simplify management in case several subclouds have configuration or other common traits. The central cloud provides a *RegionOne* region for managing the physical platform of the central cloud and the *SystemController* region for managing and orchestrating over the *subclouds*. Each subcloud is deployed as a set of servers running StarlingX Kubernetes control plane, each edge site's control plane is independent of the central cloud. The result are autonomous subclouds that can schedule and resiliently manage workload even in case of network partitioning and isolation from the centralized cloud. The StarlingX distributed cloud configuration offers a set of open REST API that allows to retrieve and update configuration of subclouds and groups. The API are currently used by the custom cli and the Horizon dashboard, but it's obviously possible to build software logic to leverage the API with a custom controller.

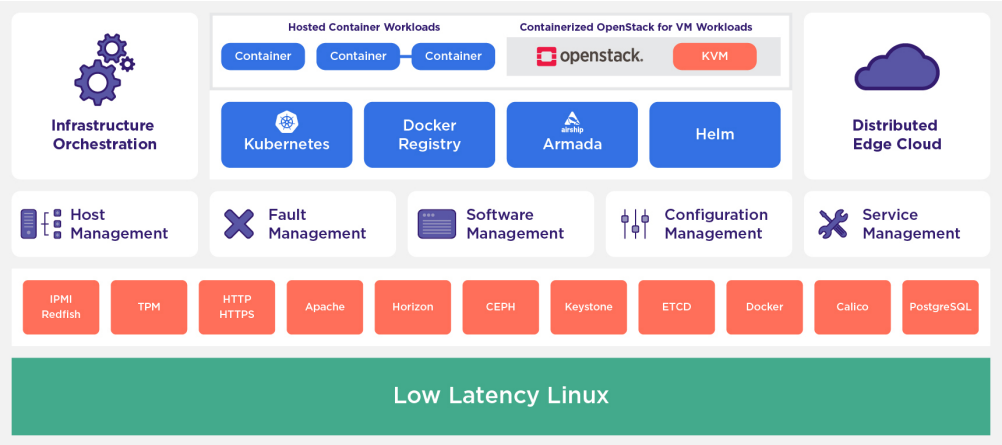


Figure 3.8: StarlingX's major components

3.4.2 Eve-OS

Eve stands for Edge Virtualization Engine and is a project backed by the Linux Foundation Edge. Eve's objective is to be the "Android" for the edge, an open and vendor-agnostic platform to be used as foundation to application deployment

at the edge. It is in fact an open Linux-based operating system for distributed edge computing that exposes a set of APIs and security features out of the box. The project is based on the assumption that application deployed on the edge should not take care of the additional problems that it involve. Security and device management should be responsibility of the platform and Eve aims at taking care of all these additional challenges leveraging cutting edge kernel versions, hardware root of trust, signed images for EVE-OS and applications, TLS, hypervisors for strong isolation and distributed firewall for every app.
A device running Eve is identified by a X.509 certificate generated by the TPM at first boot and used for the management of traffic to the controller [34]. The controller is identified by its own certificate but each devices is imprinted with the root CA to verify controller's identity, At start up Eve automatically connect to the controller to pull its configuration and or upgrades. The devices targeted by Eve are smart edge devices such as CPEs and IoT devices but also edge data center servers, so a wide variety of hardware and architectures is supported.
Eve takes care of virtualizing and separating workloads, at the time of setup, an hypervisor can be chosen (KVM, Xen or ACRN) which will be used to run separately each instance. The project tries to generalize the concept of workload and defined a specification for Edge Containers: an extension of linux containers that can also describe VM instances. Eve supports both containers and VMs through Edge Containers and is also compatible with kubernetes workloads, in fact it is possible to run k3s to orchestrate them. Including all these orchestration tools and being based on an eventual consistency model, the system continues to run in the current state if it loses connection to the centralized orchestration service being able to withstand network partitioning and temporary isolation from the cloud controller.
Eve's security features are some of its main strength as zero touch and zero trust security concept have been applied to the project. This requires an eventual consistency model in which edge nodes The OS is configured at the moment of flashing the device storage memory, its initial configuration will include the address of the cloud controller and its trusted certificate, the device will automatically try to connect at startup to get its configuration. Eve can avoid physical intrusions disabling IO ports (i.e. USB ports or network interface) or reserve some of them only to specific workloads in order to guarantee separation between the different processes.
A device running Eve is inaccessible, the only way to apply changes to the system is through the cloud controller, no ssh or remote connection is allowed. Another important feature is the ability to supporting autonomous operation and remote management with risk free updates from the cloud, since a series of test will be performed after the upgrade and in case issues appears it will revert back to the previous version of the system.

As said earlier, the APIs are completely open and they span from all the different monitoring features to device configuration and application deployment. The APIs are also extensible so that anyone can build its own custom controller. However, since the project is quite young and the community is at an early stage, at the moment there are only an enterprise (closed source) controller and an open source example controller named Adam. [35] [36]

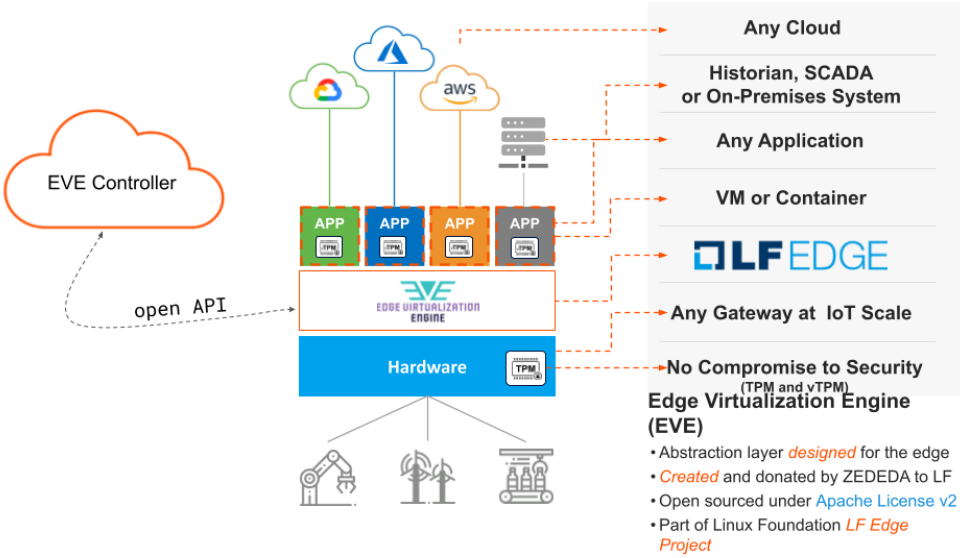


Figure 3.9: Eve OS deployment example

3.4.3 Rancher Fleet

Fleet is a set of Kubernetes custom resource definitions (CRDs) and controllers to manage GitOps for a single Kubernetes cluster or a large scale deployments of Kubernetes clusters (up to one million). Even though is not a device orchestration solution its behaviour is somehow similar to what happens in the previously presented projects: a set of clients have a single source of truth for their state (running applications) and either constantly poll it o receive updates. Rancher Fleet offers this same mechanism with the simplicity of git, making possible to deploy applications to thousands of clusters without having to worry about each single one. With respect to the previous ones the lack of OS backed device security is certainly an important downside, however its simplicity make it a good candidate for managing such a high number of deployments. Fleet's core component are:

- *Fleet Manager*: The centralized component that orchestrates the deployments of Kubernetes resources from git. In a multi-cluster setup this should be a

dedicated Kubernetes cluster. In a single cluster setup the Fleet manager will be running on the same cluster you are managing with GitOps.

- *Fleet controller*: The controller running on the Fleet manager orchestrating GitOps.
- *Fleet agent*: Every managed downstream cluster run an agent that communicates with the Fleet manager. The agent is a set of Kubernetes controllers running in the downstream cluster.
- *Bundle*: An internal unit used for the orchestration of resources from git. When a git repo is scanned it produces one or more bundles, a collection of resources that get deployed to a cluster. *Bundle* is the fundamental deployment unit used in Fleet, its content may be Kubernetes manifests, Kustomize configuration, or Helm charts. Regardless of the source they are dynamically rendered into a Helm chart by the agent and installed into the downstream cluster as a helm release.
- *BundleDeployment*: When a Bundle is deployed to a cluster an instance of a Bundle is called a BundleDeployment.It represents the state of that Bundle on a given cluster with its cluster specific customizations. The Fleet agent is only aware of BundleDeployment resources that are created for its cluster.

To install Fleet in a cluster with Helm, certificates from the upstream cluster will be required since agents will need to access the API server. [37] Then downstream clusters can be registered using a *cluster registration token* and a client-id or some cluster labels. Clusters can then be grouped by labels, this feature is useful in case of high number of cluster and if some of them have very similar or same deployment targets. [38]

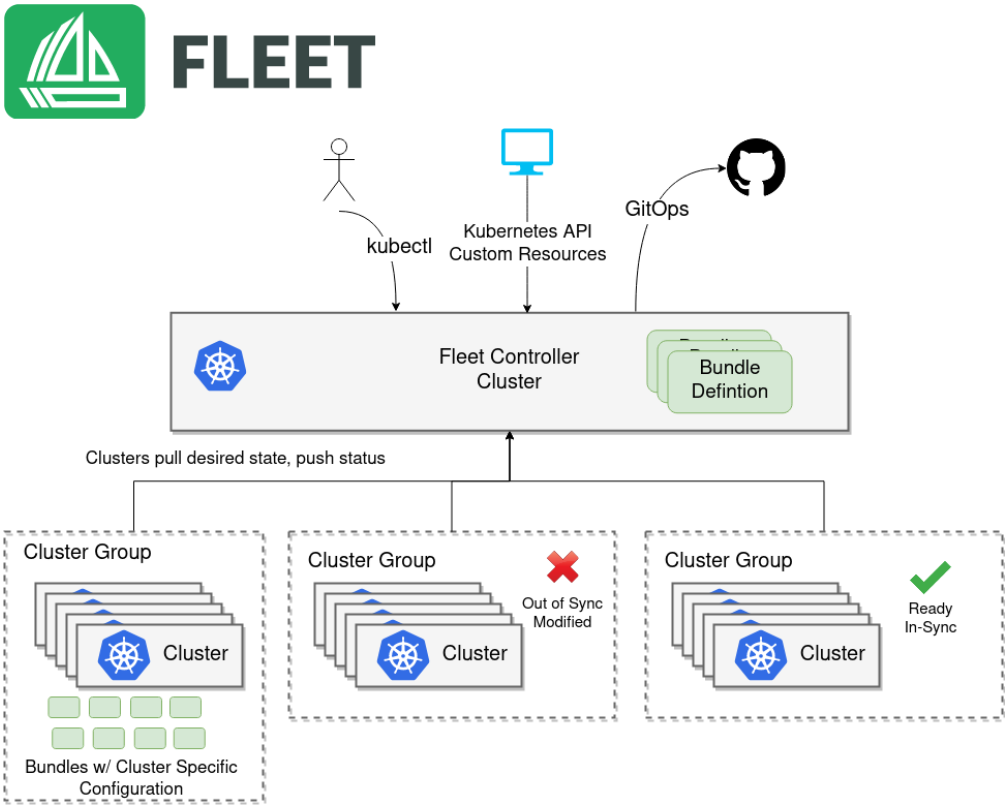


Figure 3.10: Rancher Fleet workflow

3.5 Considerations

The technologies and approaches mentioned above represents different kind of solutions to problems similar to the scenario considered or very closely related. However not every solution presented can actually be used for reason already cited, such as community support, available documentation, target scenario and scalability.

There exists no such thing as *the* Kubernetes for the edge and all of the technologies analysed have their own strengths. However, k3s stands out for its simplicity in deployment on field, as well as microK8s. Fledge is an experimental project that still brings some relevant features but is surely not thought for being extensively used. KubeEdge is still a promising framework, however its limitations about the lack of independent edge sites makes it not fit for our scenario.

Multi cluster administration and management are a complex tasks, and each different situation requires a special approach. The solutions analysed revealed their strengths in the situation they are born for and the two approaches (KubeFed and Virtual Kubelet) emerged. On one side KubeFed brings with it all the pros

of being the official solution, although scalability might become an issue. On the other hand Virtual Kubelet based solution demonstrate to be more flexible even though they bring more complexities. Of all the solutions, only Liqo takes care of inter-cluster networking and this appears as an important upside in a geographically distributed scenario.

At the edge, devices are not always accessible and this makes upgrading and management an issue. The solutions analysed takes care of different aspects of the problem. EVE-OS seems the most promising one, even though not yet mature. StarlingX packages a lot of features and it also seems hungry of resources, this makes it not suitable for the scenario where devices might be not powerful enough. Rancher Fleet, even though not oriented to device orchestration, aims at managing a high number of cluster taking care of updates and selective deployments, very important features that allow for a centralized configuration while keeping sites independent, using the same concept as in project EVE.

Chapter 4

Orchestrated architecture for the power grid

4.1 Service and infrastructure resiliency

In a scenario where centralized computations migrate to the edge, the complexity of a geographically distributed infrastructure comes into play. The infrastructure may be comprised of heterogeneous hardware and possibly physically insecure sites. Moreover the number of sites can easily grow and this needs a scalable solution that gives the possibility to join new sites to the group as seamlessly as possible.

Running workloads at the edge brings in availability problems that were already solved in a centralized architecture. For example in a cloud environment, if a physical server has some failures the applications and VMs that were running on that server will be re-instantiated in another server and it is even possible that customers will not even notice the incident due to already running replicas. At the edge resources are not as abundant as in cloud environment and network partitioning events that isolates one or more sites are a possibility that must be taken into account. Therefore each site needs to withstand network partitioning and isolation from the cloud and clearly a fully centralized control plane cannot be the solution. Kubernetes can be of great help in orchestrating workloads and is considered as foundation of the architecture presented in this chapter. However k8s alone cannot be the solution, as will be showed in this section, and in the next chapter the implementation problems will also be taken into account.

4.1.1 Geographically distributed clusters

Table 4.1 shows the number of primary and secondary stations across the years reported in the *Development plan 2020-2022* of *e-distribuzione* [39], a company

inside the Enel group operating in the electrical distribution sector. The plan presents an increasing number of stations (secondary and primaries) in their distribution grid during the last ten years. As reported in the table, hundreds of thousands of peripheral sites are involved, and each of them should be independent from a centralized control, since they should be able to go on working even if they are isolated. Having a unique big cluster is certainly not the right choice. Therefore, it might be possible to split the distribution grid into areas following the energy grid hierarchy. In order to minimize the impact of network partitioning, each area cannot be managed as a unique cluster, but each site of the area will be a cluster.

Year / Site	2011	2012	2013	2014	2015	2016	2017	2018	2019
Primary stations	2.134	2.144	2.159	2.168	2.188	2.195	2.199	2.203	2.200
Secondary stations	432.074	436.204	438.359	439.558	441.056	442.418	443.774	445.159	446.410

Table 4.1: Number of primary and secondary station over years 2011-2019 [39].

Having a cluster per site strongly makes each site autonomous, since each of them has a control plane guaranteeing that all the deployed services will stay up, going on performing their computation even if isolated from the rest of the grid. Each cluster should also be resilient to internal failures, withstand node failures, control plane failures, storage failures. It is expected a sufficient number of nodes and replicas of services and data to guarantee that the services will be kept running even in case of the previously cited failures.

Managing such a high number of clusters is not a trivial and multi-cluster approaches, originally designed for multi-cloud clusters, might not be the best choice. Even though a centralized control plane is not convenient, the services to be deployed are known a priori, since they depend on the kind of site and area. Scheduling of services inside clusters is in charge of its local control plane. However, since the services to be deployed and configurations are known a priori, they can be retrieved from a single source of truth in cloud, via a push or pull fashion (fig 4.1).

4.1.2 Services

The services considered are PMUs and PDCs:

- PMUs (Phasor Measurement Unit) require physical hardware and, being measurements units (data producers), their location is bounded. Therefore their placement is considered fixed and might be in each site (Secondary and Primary stations as well as production sites).
- PDCs (Phasor Data Concentrator) are services that can act both as data producer and consumer. They are defined in IEEE C37.247 as a set of functions

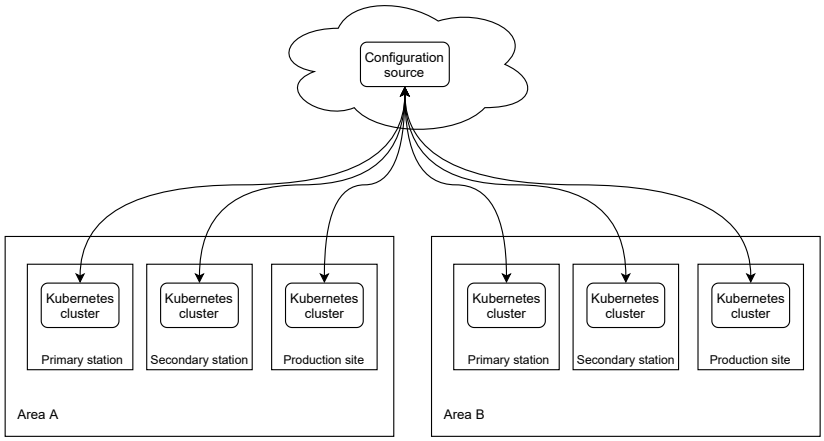


Figure 4.1: Distributed clusters with single source of truth for configuration

that produce an order output of the syncrophasors collected by the PMUs. Each instance can be connected to several PMUs to collect data and can produce one or more output that can be used by other applications or PDCs as input. Being software services with no special hardware needs, they can then be placed anywhere they are needed.

Since services are to be deployed in separate clusters, some of them, data producers, are going to be exposed to the external network in order to be reachable from services in other sites. In the case considered, PDCs are the components to be exposed since they elaborate the data produced by PMUs so that can be used by consumers.

4.1.3 Data resiliency

Data resiliency is a key requirement for delivering resilient monitoring and computing services that work with real time data. Historical data persistence is obviously critical for performing data analysis for statistical meaning or post incident analysis. What is needed then are mechanism to perform regular backups of disks or volumes and replication of data in order to withstand hardware and network failure.
A first level of data resiliency should be achieved at cluster level, so that data are not tied to a single node but rather replicated across different nodes. Another level of data resiliency should be achieved at bigger scope, performing regular backup, and possibly incremental, and pushing them to the cloud so that can data can be accessed by analysts.

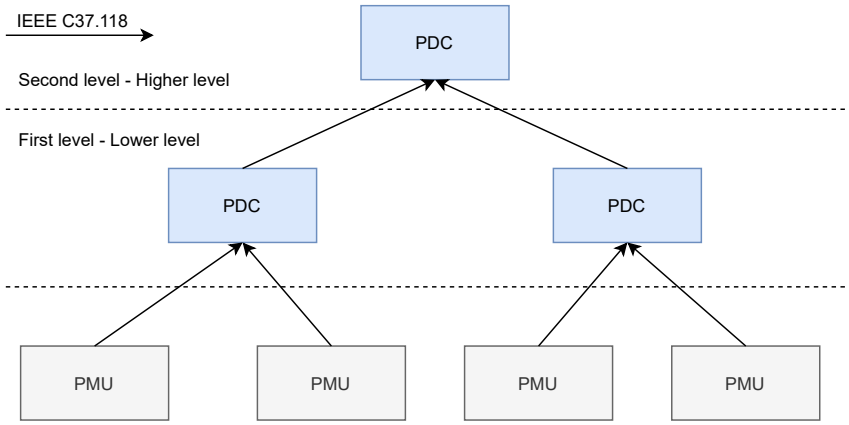


Figure 4.2

4.2 Data flow and communication resiliency

The introduction of renewable energy and new kind of loads, such as electric vehicles, make the power system a dynamic environment. [5] Its orchestration can be facilitated via smart grids, allowing the integration of the efforts of the main actors of the power system (generators, carriers and consumers) [40]. This is translated in the need of increasing the observability of the power system [5], which can be achieved via a huge network of sensors, interacting among them and with the infrastructure in order to provide information about physical world. This data can be later stored, processed, analysed in order to control the behaviour of the grid through intelligent actuators [41] or for offline analysis [42]. It is clear that smart grid networks should manage a great amount of data, delivered over different physical media, coming from many different types of devices, some of them with limited computational power, and with different requirements in terms of QoS [41].

4.2.1 Reducing distances with the Point of Presence

Some applications, especially the ones related to the control and stability of the power system, have strict requirements in terms of data delivery latency [42]. Reducing the distance between the interacting parts could be a way for reducing latencies. Differently from the transmission systems, where the network follows the power grid topology, is private, owned and self-managed by the transmission grid operator, this is not always true with the distribution system, where the electrical companies might need to rely on telecommunication providers. This means that, it is likely that the sites of the distribution system are not directly connected through

dedicated links, but reaching a site from another requires a transit in the network of the telecommunication provider.

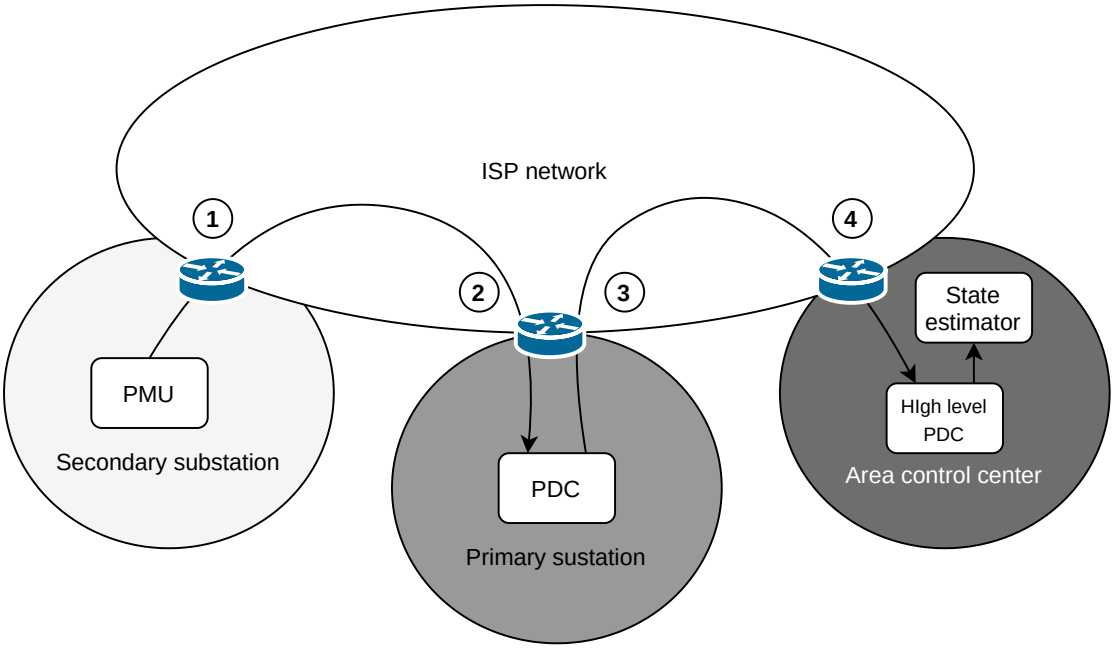


Figure 4.3: Data enters and exits from the power provider network four times

Knowing the path of traffic is extremely important for optimizing it and reducing latency. For example, let's imagine having a set of PMUs producing data and sending it to the PDC located in the closest primary station. The output stream is then sent to the *state estimator* inside the area control center. Looking at figure 4.3 and following the traffic, it might be clear how the exchanged data has to enter and exit from the power provider network to the ISP network and vice versa multiple times:

1. PMU data exits the electricity provider network and enters the ISP network.
2. From the ISP network, it is routed again to the power provider network in order to be delivered to the PDC.
3. The output stream of the PDC needs to go back to the ISP network.
4. Finally, the stream reaches the state estimator entering into the electricity provider network for the second time.

This configuration is clearly not convenient since we are, at least, doubling the round-trip time. We would like to have a geographical area, in the middle between

all these services allowing to terminate the traffic before, without entering and exiting the electricity provider network. A *Point Of Presence* might be the perfect place where to locate the critical services: it stays in the ISP network and there is a great number of PoPs spread in a geographical area. Therefore, it would be the closest hop from any site of a distribution system area.

At this point it might be possible to consider a lower level PDC for each site of the power grid, collecting the data of the local PMUs. Their output stream, is then sent to the higher level PDC, located inside the ISP, where the ouput can be sent as input of the local state estimator or other applications performing data processing or storage of historical data. Figure 4.4 shows the resulting topology. In this case the output stream of the PDC exists from the private network of the site, reaching the Point of Presence where the PDC and the local estimator are placed. This allows to reduce the path followed by the traffic, and consequently even the transmission latency of the data. That is why we decided to physically move the services of the *area control center* to the PoP.

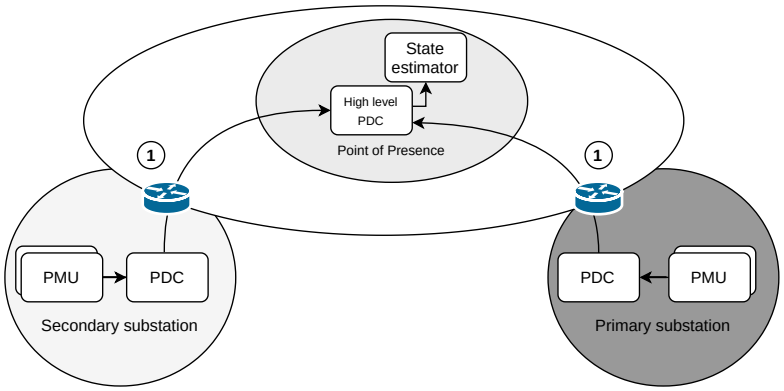


Figure 4.4: Data exits a single time from the electricity provider network reaching the PoP

It is important to notice that even though the ICT network of the distribution system might use the telecommunication provider network, the traffic of the distribution system flows on top of an overlay network managed by the ISP or by the power distribution operator, so that clusters, services and machines of the electricity provider cannot be reached through the Internet.

4.2.2 A data-centric architecture

A smart grid might be seen as a huge distributed system, with devices and applications of different natures, producing, consuming, processing data and interacting in order to provide a coherent service, which is a *resilient power grid*. In this context a point-to-point interaction represents a too rigid approach, applications are difficult

to be written, the interactions between the components is fixed and introducing new components or changing the existing ones could represent a problem [43]. The main idea would be having a *data-centric* architecture, where the actors don't need to know who is in charge of producing some data, where it should retrieved and who is consuming it. This can be achieved via the publish-subscribe paradigm, where producers and consumers need only to know how to reach an intermediary broker, and contacting it in order to consume or produce data. The data should be accessible from any point of the grid without creating any dedicated channel.

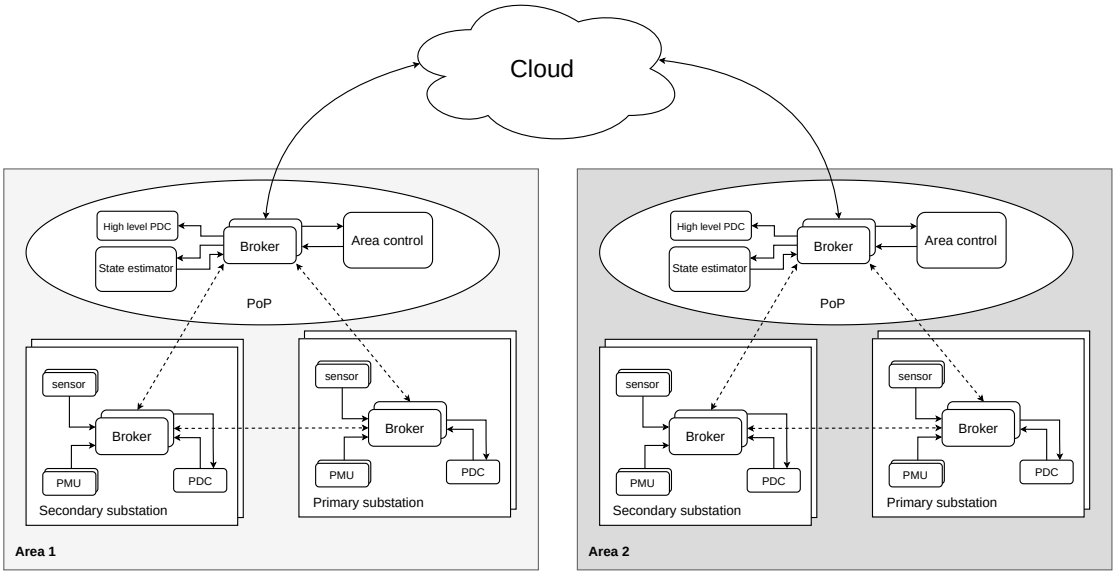


Figure 4.5: Distribution system as a data-centric architecture

A solution might be having a cluster of brokers for each area of the distribution system. Brokers should be replicated for multiple reasons:

- **Scalability**: due to the great amount of expected traffic, a single instance of the broker would not be enough;
- **Latency**: brokers should be located as close as possible to the data and the consumers of that data;
- **Resiliency**: multiple instances of the broker allow to improve the availability of the service; if any of the broker instances fail, there always will be another instance able to accept the requests and deliver the data to the subscribers.

Any measure, processed or aggregated data published in a broker inside a cluster, should be reachable from any broker belonging to the same cluster in

a completely transparent way. This enables the services belonging to the same area, to freely exchange data, and to deploy new applications or data processing algorithms without changing what already exists. In some cases, different areas of the distribution system might need to interact, for example, that's the case of the *inter control center* communication between the adjacent areas [42] or other data useful for offline analysis. Typically this kind of data does not have strict requirements in terms of latency. The idea is using services aimed to export a part of the data published in the local cluster of brokers, in some cases performing some aggregation or local processing, in order to reduce the dimension. Finally this data is sent to the cloud. From here, it is possible to perform further processing or aggregation, data can be stored, given to wide-area controllers, or delivered to other areas of the distribution system or to any other component of the power grid which might require it.

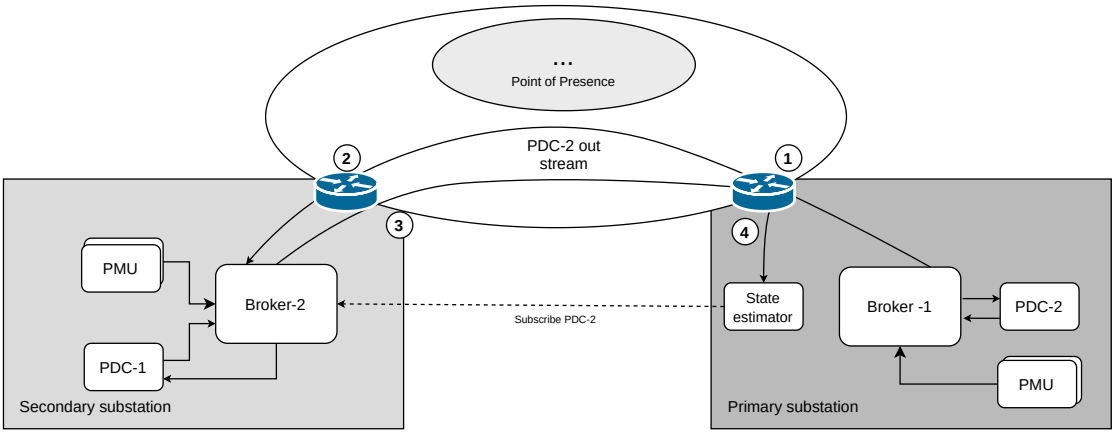


Figure 4.6: Unoptimized traffic caused by subscription to a wrong broker instance

Another aspect to take into consideration is how the load is balanced over the different instances of the brokers located over the area. We would like to be able to connect the client to the closest broker, this has great importance in terms of optimization of performance. For example, let's consider the case of a PDC placed inside a primary substation, this PDC produces the output stream which is sent to the an instance of the broker located in that primary substation. In the same location, the state estimator connects to a broker and subscribes to the output stream of the PDC. However, we are not managing where the clients should connect, therefore it connects to an instance of a broker in a secondary substation. Following the path of the data in figure 4.6, the broker located in the secondary substation should first retrieve the data from the instance of the broker in the primary substation, where the PDC is sending the data, and then it can forward it to the state estimator. This shows how selecting the right instance of the broker is

crucial when latency matters, otherwise we might have completely unoptimized paths, with a negative impact on performance.

Chapter 5

Implementation

In this chapter a description of specific components and their used configuration will be provided. In addition it will be stated the reason for such components and how they have been adapted to be run as container, if necessary.

5.1 Infrastructure

5.1.1 Orchestrator

The **Kubernetes distribution** chosen for the edge is *k3s*, as presented in *Böhm and Wirtz* [18], it demonstrated performances very close to vanilla Kubernetes bringing advantages such as setup simplicity and a reduced disk usage footprint. It is to be noted that new Kubernetes distribution are emerging and k3s, vanilla Kubernetes and microK8s do not represent the only choices available. The need for a reliable and easy to setup distribution has been the decisive factors since the three analysed perform very closely in functionalities and resource need.
K3s has been configured to have master redundancy using the embedded etcd option. This allowed to have a highly available control plane capable of withstanding a master node failure due to the takeover of the lead by another one in stand by. Moreover the default configuration has been changed to reduce the interval between a node failure detection and the start of new instances of the services deployed in the failed node. As default, Kubernetes sets this interval to 5 minutes. In our case it has been considered too long and set to 20 seconds, however it can be further reduced but keeping in mind that below 10 seconds also other parameters should be changed and may impact performances. The API server options used are:

- `default-not-ready-toleration-seconds` set to 20. This configures the default annotation that is placed in every pod that defines how much time should be tolerated for a pod to be in node in a `NotReady` state.

- `default-unreachable-toleration-seconds` set to 20. As the previous one sets the interval tolerated for a pod to be in a node in a `Unreachable` state.

Experimental results will be shown in the next chapter but it is to be noted that in Kubernetes a node goes into the NotReady state after 40 seconds of being unreachable or, better said, at the fourth time that a node's kubelet is polled and no response is received. This interval is also configurable and taking also this one into account brings the restart of pods, in case of node failure, to around a minute after the failure.

5.1.2 Multi-cluster

To address the **multi-cluster management** issue, two main approaches have been considered: a federated multi-cluster architecture based on either Virtual Kubelets or on KubeFed, a centralized approach with a single source of truth (Eve-OS, Rancher Fleet).
The first approach gives a single point of management for multiple clusters and a two-level control plane, one at host/higher cluster level and one at member/lower cluster level. A single point for managing the cluster from the administrative perspective is needed because of the high number of clusters. A two-level control plane is somehow necessary if areas are thought as single "big" clusters, the first level one is needed to provide the administration entry point, the second one to provide resiliency to each single site. However, the need for a single "big" cluster per area exists only if an area level scheduling of workload is needed. In the scenario we are considering, services are latency critical and need to be close to the source of the data to be analyzed. Moreover, it is not expected to run dynamic workloads on edge nodes since services may be tied to hardware requirements (connected to meters, actuators) and they are expected to be long running workloads rather than single execution jobs. For the previous reason a two-level scheduling has not been considered crucial to the scenario considered and source of unnecessary complexity on the management of the infrastructure.
The second approach still presents the single point of management either as git repository (Rancher Fleet) or as cloud controller (Eve OS). The latter is considered a very promising solution given its integration with k3s,its security-first design and the possibility to manage also different parts of the device at low level. However the framework is still evolving and not fully documented, creating a new controller would take a considerable effort since at the moment the only example controller is still lacking of features. Rancher Fleet appears to be a good compromise providing the advantages of a single source of truth and still being designed to handle thousands of clusters. Clusters can be then grouped by area and kind of site (i.e. primary or secondary station) in order to differentiate the services to be deployed and the related specifications (configuration, number of replicas).

Rancher Fleet has been used with the multi cluster configuration: a cluster in the cloud runs the fleet manager and watches the git repository containing the configuration while fleet agents, running in peripheral clusters, connect to the fleet manager to get updates. The configuration considered uses the agent initiated cluster registration, meaning that the peripheral cluster will poll the central fleet manager allowing to traverse NATs in case peripheral clusters are in a LAN and not directly exposed to the public internet. The scenario considered only one region, divided in several area, although it would be possible to extend the pattern to several regions too. Each cluster is given the appropriate labels to identify the region, area and kind of site, so that rancher Fleet can be configured to push only the configuration for the selected cluster. For example, it is known that every secondary station will have a PDC, so every cluster matching the label `site: secondary-station` will be selected to deploy a PDC. The configuration can be set selecting configmap and secrets based on region and area. This approach is however limited due to possible configuration issues, in fact configmap and secret should be specifically defined for each cluster since configuration might be not only area specific but also cluster specific. While rancher fleet itself does scale up to 1 Million of clusters [38], it is obvious that this approach has issues since configuration should be declared manually and one file per cluster could pollute the repository. A possible approach would be to have a separate centralized repository that clusters use to pull their specific configuration dynamically and periodically

5.2 Services

The main components are OpenPDC, PMUsim and MySQL.

- *OpenPDC* is an open source implementation of the IEEE C37.247-2019 standard which describes the specifications for PDC functions. It is written in C# and developed by the Grid Protection Alliance.
- *PMUsim* is an open source PMU simulator written in C that provides a GUI to configure the parameters for the output to be produced
- *MySQL* is a well known DBMS, in this case it is needed since OpenPDC can store its configuration in DBs. The lightweight option is sqlite, however, being file based, it would not allow for remote configuration. This reason forced to go the DBMS way and MySQL has been chosen.

5.2.1 OpenPDC

OpenPDC is a software written in C# but, despite the language ability to produce cross platform binaries, its main target is the Windows OS. Some versions are also

POSIX compatible and for this work, version 2.4 has been built to be run on Linux systems using mono and inside docker containers.
The database used by OpenPDC to get its configuration needs an initial setup. In non-cloud environment this would be carried out by a system administrator using the *Configuration Setup Utility* provided by the Windows installation. However in a cloud environment this phase should be automated and for this reason an *init container* has been built. The *openpdc-init* container takes care of initialize the proper user and tables in the MySQL database and parameters can be set at container startup using environment variables. The configuration of OpenPDC regarding database connection, node id and exposed ports is done via an XML file that the application reads at startup and creates a default one if none is present. This configuration is carried out by a shell script that expects environment variables to be set and use them to generate a configuration file from a template.

Configuring the PDC connection to PMUs cannot be done declaratively as done with the database. In fact the *OpenPDC manager* helper application gives this possibility through a user interface although available only on Windows systems. As a consequence, while the PDC application can be started automatically, the PDC still have to be configured remotely using the Manager and ssh to tunnel the traffic to the remote instance since the application connects to localhost by default. An important advantage given by the openPDC Manager is the possibility to visualize real time graphs of the measurements received by the PMUs connected and even replay historical data, if connected to an historian.

5.2.2 PMUsim

PMUsim is an open-source, C-based, IEEE C37.118-complaint PMU simulator tool, allowing to generate random synchrophasors. It comes with the iPDC set of tools and it provides a graphical interface useful for the PMU configuration. Unfortunately, even though the GUI makes configuration easier when users directly interact with the tool, this is not true in a cloud environment. However, PMUsim allows to store a configuration in a binary file so that it is be possible to reload the same configuration once the application is restarted, or to have multiple configurations that can be loaded when needed. The idea was performing some modifications to the application source code, in order to be able to load a configuration file, at the application startup, without interacting with GUI.

When the PMUsim application starts, the main process forks, and the newly created process starts the *PMU-server*, the component in charge of connecting to the PDC and generating the random synchrophasors. The role of the main process, instead, is the one of drawing the GUI and interacting with the PMU-server process, via signals, in order to apply the configurations performed by the user via the GUI. We stripped off the GUI from the main process and we added a parameter

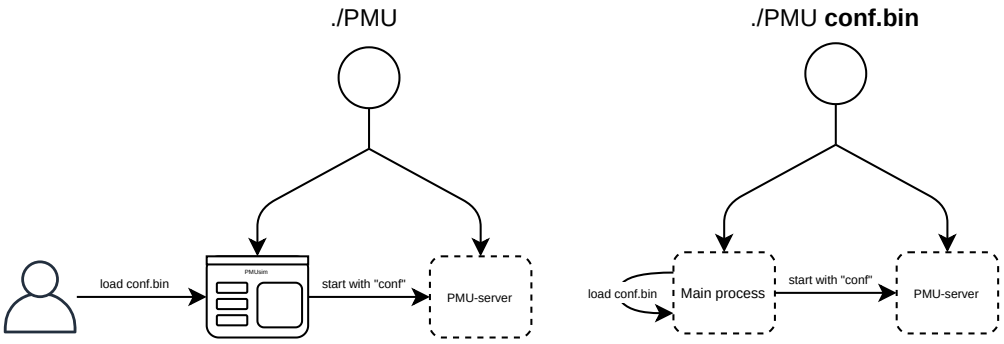


Figure 5.1: Loading of a configuration file before and after the GUI removal

to the main function, containing the path of the configuration file to be loaded at application startup. When the application starts, it reads the configuration file passed as parameter, parses it and sends a signal to the server process, in order to inform it that a new configuration is ready to be loaded. At this point the server can apply the configuration provided by the user inside the file. This makes really simple executing the application inside a container, since no graphical interaction with the application is needed, but only mounting to the container the desired configuration file.

5.2.3 MySQL

MySQL is a well known open source Database Management System developed, distributed, and supported by Oracle Corporation. In this case it has not been used to store data (measurements), instead as said previously it is needed by OpenPDC software to read and store its configuration.A lightwheight option would have been *sqlite*, however being file based makes remote connection, and therefore PDC configuration, difficult to automate.
In our scenario an instance of MySQL database is run in every cluster so that PDCs can easily reach their configuration. In the broader scenario a database cluster can be setup, where a master replica is run in the cloud and on each peripheral site runs a slave and read-only replica. Again the single source of truth is used, since the master replica is the only one with writing privileges and the data is replicated to the edge as soon as a connection to the cloud is available.
An important remark is that the database in this case has only be deployed because of this peculiar PDC implementation and the storage of the data gathered by the monitoring infrastructure is taken in consideration with the use of MQTT brokers. The approach presented appears to be still valid for cases similar to this where data at the edge should only be read.

5.2.4 Longhorn for resilient data persistency

Persistence in Kubernetes is managed through built-in abstractions such as *PersistentVolumes* and *PersistentVolumeClaims*, however the default driver provides only basic features and does not take into account data replication. To overcome this limitation, CSIs are available as plug-in, in this case the CNI Longhorn offered what was needed in our scenario: data replication, ability to perform regular backups and a very simple and straightforward setup process. A *StorageClass* has been defined to define the policies to be applied to persistence data such as number of replicas and scheduled backups. An important option specified in the StorageClass is `defaultDataLocality` set to `best-effort`, it enforces a policy so that Longhorn tries to keep a replica on the same node as the attached volume.
Moreover, additional configuration have been used to allow a quick instantiation of services using a persistent volume in case of node failure. Particularly, the option `nodeDownPodDeletionPolicy` has been set to `delete-both-statefulset-and-deployment-pod` enabling the force deletion of StatefulSets and Deployments in a failed node, so that the volume can be attached to a new running instance. The Longhorn CNI is not the only one that offers these functionalities, however being already compatible with k3s and having a simple configuration showed to be a good match for the use case presented.

5.3 Demo

A short demo have been set up to show the lifecycle of application ported in Kubernetes and to understand its behavior in case of the previously cited failure. A cluster of 4 nodes has been set up with 3 masters in *high availability* mode and the fourth node set up as worker. This kind of configuration has been chosen to show the behavior of the orchestrator in its high availability configuration as well as the additional overhead introduced by it, the worker has been introduced to show its lower resource usage and the possible different behaviour, with respect to master nodes, in case of failures.

In the demo, two PDCs have been deployed (1st and 2nd level), 3 PMU simulators and a MySQL instance used by the two PDCs. The simulated faults are:

- stateless service failure: a stateless service (pdc/pmu) fails, the orchestrator restart it or creates a new instance of the service.
- stateful service failure: a stateful service (MySQL) fails, the orchestrator restart or creates a new instance attaching it to the same persistent volume and thus preserving data.
- worker node failure: a worker node is isolated from the rest of the cluster, the

control plane notice that and reinstantiate the services on another healthy node.

- master node failure: a master node is isolated from the rest of the cluster, another master node is elected as leader and services on the isolated node are reinstantiated on other healthy nodes.

This setup has then been used as basic module to provide multi cluster deployment using Rancher Fleet, selecting deployments on label basis. For example PMUs and low level PDC deployed only on secondary-station labelled clusters, high level PDC only on primary-station labelled clusters. The use of Rancher Fleet allowed to seamlessly update the deployments with the only action of pushing new files to a git repository, with the possibility to specify the rollout strategy.

Chapter 6

Results

In this chapter an evaluation of the proposed implementation will be presented. In the next section an overview of the evaluation method will be given and then a section will be dedicated to each group of analyzed results. Particularly, resource consumption (CPU, memory and disk usage) and reaction times in case of the considered faults have been evaluated.

6.1 Evaluation method

Since the workloads are intended to run in containerized environments, only linux based operating systems have been considered. To keep the results consistent and to allow to compare them fairly, Ubuntu 20.04 has been used as base OS for every measurements taken. The architecture considered are x86 and ARM, both 64-bit and will be referred respectively as x64 and arm64 in the next pages. Further information are available in table 6.1.

Moreover complex container orchestration tool such as Kubernetes is not just a thin software: orchestration means constantly talking with the container runtime backend, keep track of running resources and and either run the control plane or communicate with it. Therefore understanding the cost of running the workloads using an orchestrator is a critical part of the evaluation.

CPU and memory consumption metrics have been collected using the *sysstat* tool which automatically collect information using linux primitives and using a cronjob with a default frequency of 10 minutes. Sysstat docs suggests not to set the interval below 30 seconds, otherwise it might impact performances. Therefore, instead of the default 10 minutes interval, it has been set to 1 minute. Every test case have been set up in a close to real environment, i.e. PDCs connected to a remote PMU producing an output stream.

CPU usage values represent the time in which the CPU is not in idle state and

the system did not have an outstanding disk I/O request. The values considered are the average value of all CPUs for given moment and are presented as box plots to show the range of values that have the highest frequency and distinguish them from outliers. Memory usage values simply represent the non-free memory at a given time.

Every test case have been carried out for at least 4 hours since the applications demonstrated to have a consistent behaviour over time and in order to collect enough data to perform a basic statistical analysis identifying significant average values and outliers.

Disk usage values have been taken considering the size of images in case of docker containers whereas, in case of installation as plain linux applications the size of the dependencies needed and the size of the software has been considered.

Reaction times tests have been carried out using helper bash scripts to poll the system for the events considered and store their timestamp for later analysis. Each single case will be explained more in depth in the dedicated section.

Architecture	x86 (64-bit)	arm (64-bit)
Machine	VM	Raspberry Pi 4B
linux kernel	5.4.0-48-generic	5.4.0-1042-raspi
CPU model	Intel Xeon (Cascadelake)	Cortex-A72
CPU cores	4	4
CPU frequency	2.2 GHz	1.5 GHz
Memory size	8 GB	4/8 GB
Disk size	15 GB (SSD)	15 GB (micro SD)
OS	Ubuntu 20.04.3 LTS	Ubuntu 20.04.3 LTS

Table 6.1: Relevant specifications of the machine used to carry out the tests.

6.2 Containerization overhead

The first step of the evaluation was to collect values about the initial state of the system, that means cpu and memory consumption of an Ubuntu "vanilla" machine with no other tool or services installed apart from the ones bundled with the OS. Another important initial step was to collect measurements on a system running the docker daemon in order to have reference values to evaluate the consumption of the containerized version of the applications. Image sizes are reported in tables 6.4 and 6.6 for both x64 and arm64 architectures, base image sizes have been given for reference in order to highlight the effective size of applications and their dependencies.

Then the applications have been tested both as apps installed on the OS and as running container. Containers have been deployed individually, as first step, and then in two different deployments that should reflect possible situations:

- *Normal deployment*, which is comprised of an instance of MySQL, a PMUsim and a PDC instance.
- *Whole deployment*, which represents the whole demonstration: a MySQL instance, 3 PMUsim and 2 PDCs instances (1st and 2nd level).

Table 6.2: Image sizes (ubuntu and alpine given for reference as bae image), base image in parenthesis.

Table 6.3

container	total size
ubuntu	72.7 MB
alpine	5.61 MB
mysql	556 MB
openpdc-init (alpine)	40.2 MB
openpdc (ubuntu)	633 MB
openpdc (alpine)	408 MB
pmu	264 MB

Table 6.4: Image sizes for x64

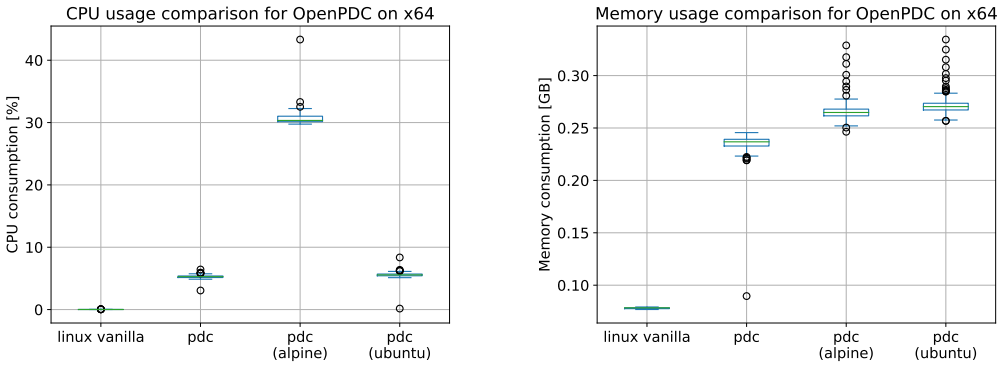
Table 6.5

container	total size
ubuntu	65.6 MB
alpine	5.34 MB
mysql	497 MB
openpdc-init (alpine)	40.2 MB
openpdc (ubuntu)	595 MB
openpdc (alpine)	406 MB
pmu	251 MB

Table 6.6: Image sizes for arm64

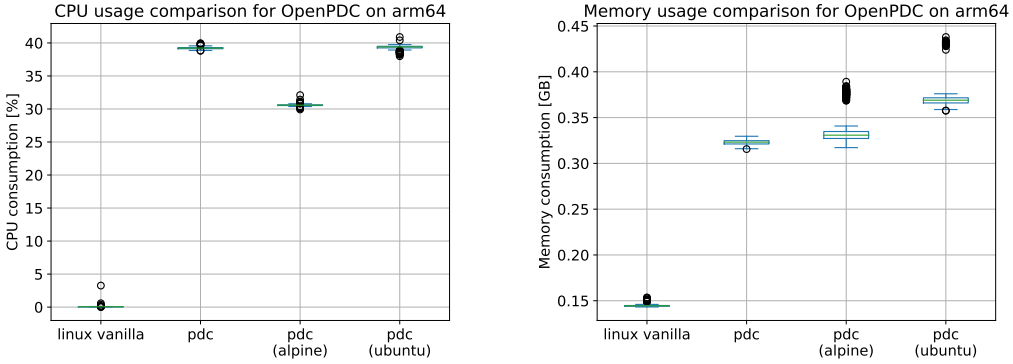
OpenPDC has been containerized in two different images, one using alpine as base image and the other using ubuntu. This has been done since performance issues arose depending on architecture and base image. In figures 6.1 and 6.2 CPU and memory usage is shown, particularly CPU shows an additional usage on alpine x64 and for ubuntu based images on arm64. For this reason in the following environments, the alpine image has been used for arm64 and the ubuntu image for x64.

In figures 6.3 and 6.4 a comparison of apps installed as standard applications with their containerized version is given. The CPU overhead given by the containerized environment is negligible in case of x64 and arm64 (apart from the case of the pdc implementation), whereas highlights and additional $\tilde{5}3$ MB memory usage in case of x64 and $\tilde{4}0$ MB in case of arm64. More in depth data about deltas with respect to vanilla apps are shown in table 6.7.



(a) **(b)**

Figure 6.1: Comparison pdc containerized versions (x64).

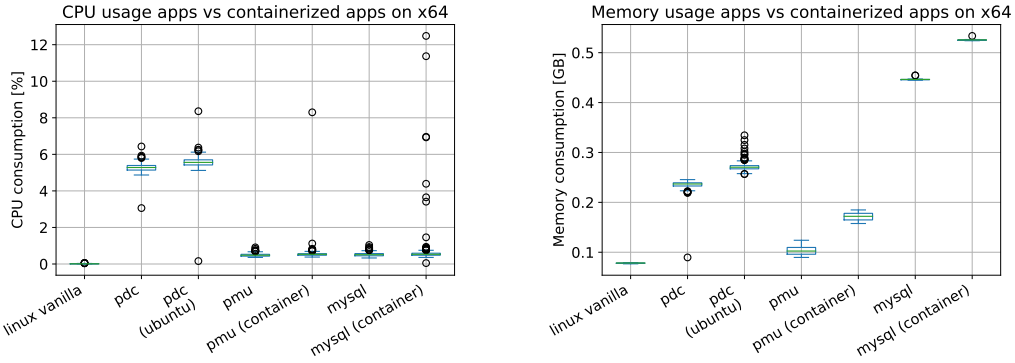


(a) **(b)**

Figure 6.2: Comparison of pdc containerized versions (arm64).

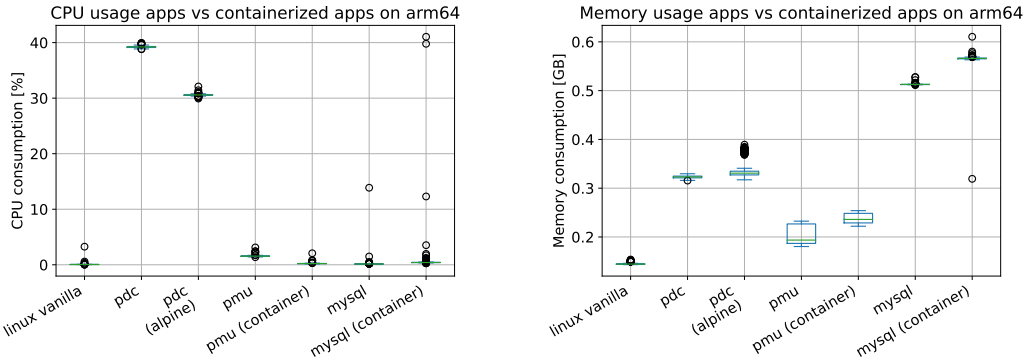
Application	arch	Δ CPU	Δ Memory
openpdc (ubuntu)	x64	0.29 %	49 MB
	arm64	0.11 %	17 MB
openpdc (alpine)	x64	25.36 %	15 MB
	arm64	-8.64 %	15 MB
pmu	x64	0.07 %	35 MB
	arm64	-1.34 %	35 MB
mysql	x64	0.12 %	52 MB
	arm64	0.47 %	40 MB

Table 6.7: Resource usage deltas with respect to apps regularly installed in the OS.



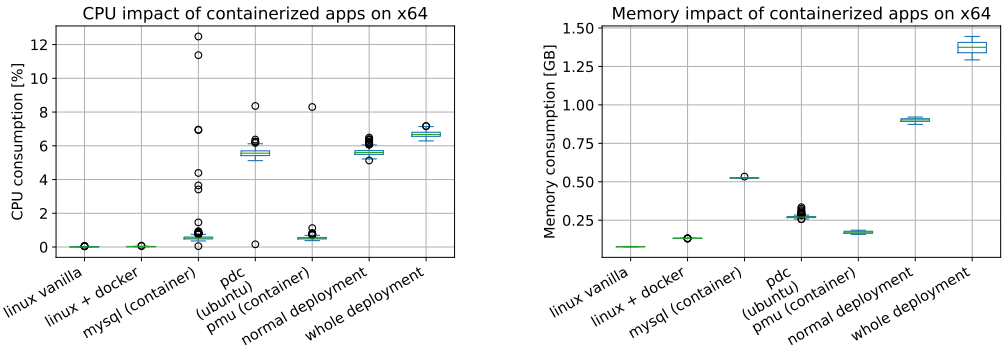
(a) **(b)**

Figure 6.3: Comparison of apps against their containerized versions (x64).



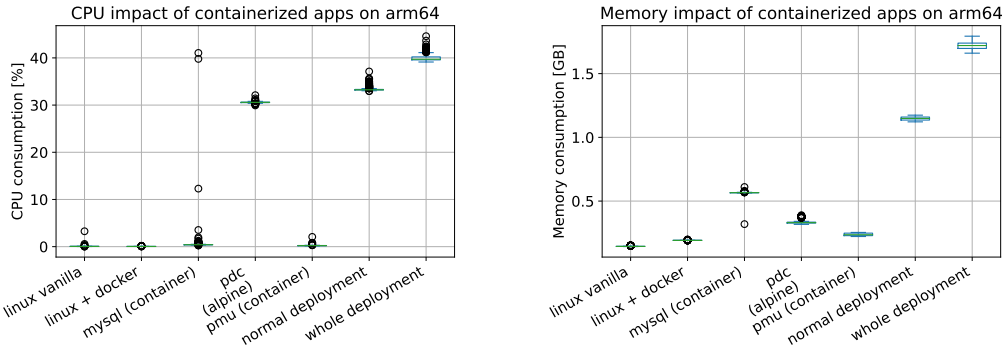
(a) **(b)**

Figure 6.4: Comparison of apps against their containerized versions (arm64).



(a) **(b)**

Figure 6.5: Resource usage of containerized apps, singularly and as a group (x64).



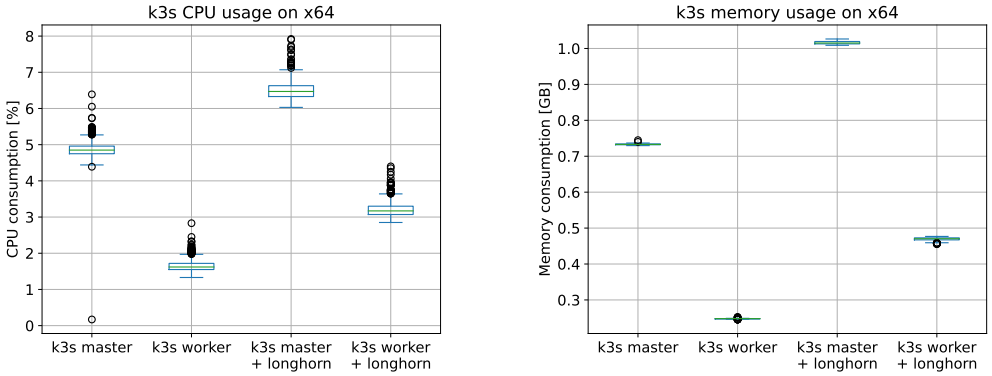
(a) **(b)**

Figure 6.6: Resource usage of containerized apps, singularly and as a group (arm64).

6.3 Orchestration overhead

Kubernetes' impact on computational resources have been studied in case of master and worker nodes and then with the installation of the Longhorn CSI. As specified in the implementation, k3s has been used as Kubernetes distribution and it have been used during these test cases. In the first case, a k3s cluster of a master and a worker have been setup without any workload. In the second case Longhorn have been installed, again without additional workloads. An important note is that Longhorn runs a set of services that in case of a 2 nodes cluster results in a higher load than one experienced in a 3+ nodes cluster, since not all of them are per-node.

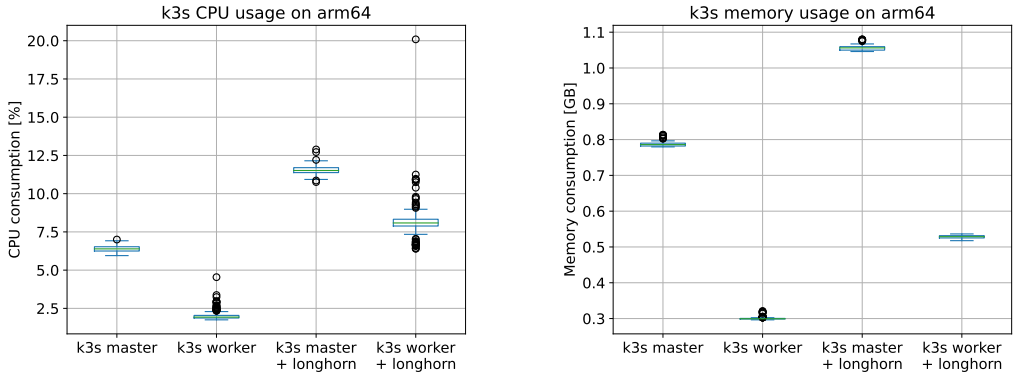
In table 6.8 a numerical comparison shows the deltas of the different setups. The deltas shown represents the differences between the mean values in the setup considered and a VM running only the Ubuntu OS. As expected master nodes result in a higher cpu and memory footprint, the use of Longhorn CSI further increases the footprint that becomes particularly significant in master nodes on arm devices (11 % of CPU and 910 MB of memory).



(a) **(b)**

Figure 6.7: k3s resource usage in mater, worker configuration with/without longhorn (x64).

A general evaluation of the demo presented in section 5.3 is shown in figures 6.9 and 6.10. Resource usage clearly shows a higher usage in master nodes, system pods were distributed across all nodes both the ones related to Longhorn and to Kubernetes system. An important notice is that the node running the low level PDC, shows the highest CPU usage in both architectures and this result in a particularly high usage on the Raspberry Pi deployments. However, since the number of services deployed is relatively low and also the resource usage, the control plane first seems to take into account the number of pods running on each node and tries to even that, this is shown in tables 6.9 and 6.10.

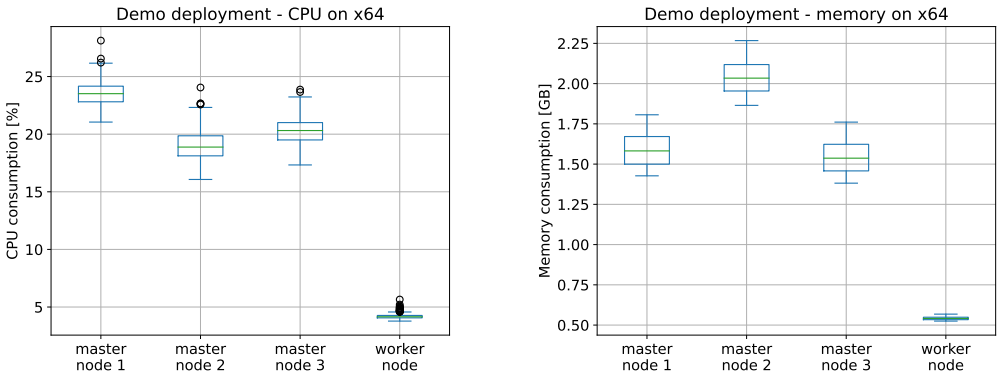


(a) **(b)**

Figure 6.8: k3s resource usage in master, worker configuration with/without longhorn (arm64).

Setup	arch	Δ CPU	Δ Memory
k3s master	x64	4.84 %	330 MB
	arm64	6.33 %	641 MB
k3s worker	x64	1.63 %	84 MB
	arm64	1.93 %	154 MB
k3s master + Longhorn	x64	6.5 %	470 MB
	arm64	11.47 %	910 MB
k3s worker + Longhorn	x64	3.2 %	200 MB
	arm64	8.11 %	383 MB

Table 6.8: Resource usage deltas with respect to Ubuntu vanilla VMs.



(a) **(b)**

Figure 6.9: Resource usage of a demo deployed in a 4 node cluster (x64).

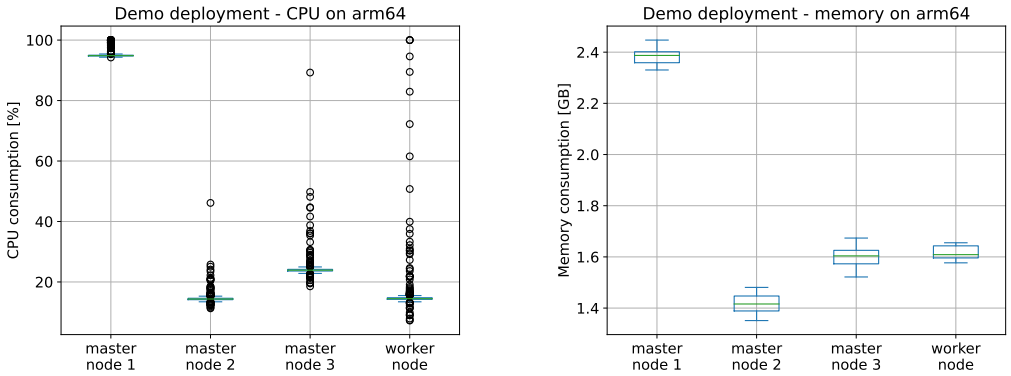


Figure 6.10: Resource usage of a demo deployed in a 4 node cluster (arm64).

Node	running pods	demo pods	system pods
node 1 (master)	12	2	10
node 2 (master)	12	2	10
node 3 (master)	13	1	12
node 4 (worker)	12	1	11

Table 6.9: Number of pods running on each node (x64).

Node	running pods	demo pods	system pods
node 1 (master)	12	2	10
node 2 (master)	12	1	11
node 3 (master)	13	2	11
node 4 (worker)	12	1	11

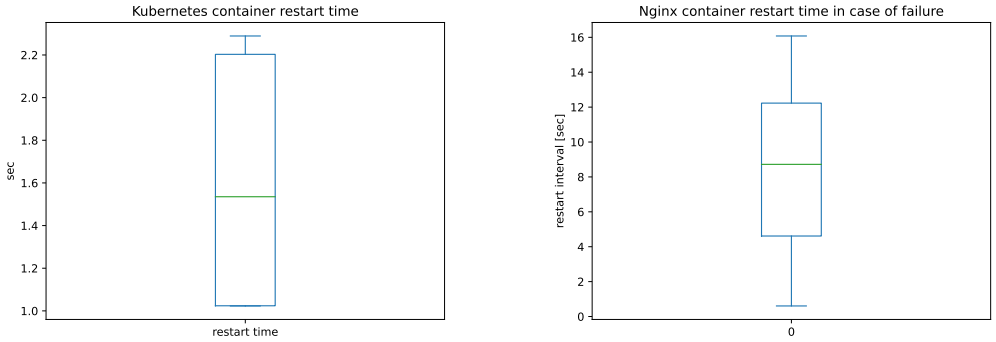
Table 6.10: Number of pods running on each node (arm64).

6.4 Orchestrator reaction times

Measuring the reaction times of the orchestration is not an easy task. The objectives were to determine an average reaction time in two specific cases:

1. Container restart after its failure.
2. New Pod instantiation in a healthy node after a node either fails or becomes unreachable.

For the first case, a Pod running an nginx has been setup and, to simulate the crash, a bash script that killed the nginx process has been set to run automatically after 15 seconds the container start. From outside the cluster another bash script have been written to open a tcp connection to the webserver's port on a busy loop, whenever a change in status (up/down) was detected it was saved in log file along with the timestamp. However Kubernetes has a CrashLoopBackoff status that is activate whenever a container continuously fails, this status enables an exponential backoff timeout, capped at 5 minutes. The Kubelet will wait the timeout before restarting the container again, if it fails the timeout continues to grow.



(a) **(b)**

Figure 6.11: 6.11a Restart time interval before activation of the backoff. 6.11b restart time with backoff.

For the second case two bash scripts have been used. One defined the firewall rules (`iptables`) rules in the node (VM) that was going to be isolated from the others, printing also the timestamp of the event. The second was run in another node to repeatedly contact the Kubernetes api, checking the status of the node to register when the failure was detected and checking the pods to register the new instances of the pods in the isolated node were instantiated. In 6.12 the three critical reaction intervals are shown: interval for a node to get into the

NotReady/Unreachable status, interval for the creation of new pods in place of the ones running in the failed node, the total interval. While the first might seem irrelevant at first sight, it is directly related to the second one, since pods are re-created only if a node is in the NotReady or Unreachable status for the specified amount of time. Thus an evaluation of the former was necessary to effectively evaluate the whole reaction time of the orchestrator in the case considered.

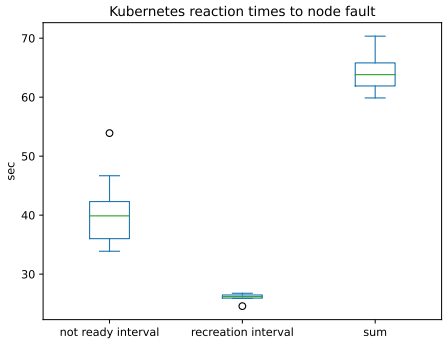


Figure 6.12

6.5 Further analysis

An additional view is now presented to evaluate the overall costs of virtualization and orchestration. As show in section 6.2 the overhead caused by containerization is in line with the expectations, below 1 % for CPU (apart from some edge cases) usage and between 20 and 50 MB depending on the application. This shows that lightweight virtualization can bring several advantages with low computational overhead.

In section 6.3 the computational cost is shown, however to actually evaluate it a further analysis is needed. In both the hardware setups evaluated CPUs had 4 cores, even though the x86 (64-bit) still outperforms the arm64 due to its high end range (Intel Xeon) and optimizations that are currently available on x86 machines. Moreover the use of arm64 is not intended to show the differences between architectures but rather for showing the differences between a general purpose server or VM and a low end device such as a Raspberry Pi 4. That said the overhead given can be evaluated in terms of vCPU to define how much the orchestrator makes use of resources and whether its cost is so significant with respect to applications deployed. Given the fact that each setup used 4 core CPUs and the value representing the CPU usage is the mean of the usage on each of the 4 cores at given time, we can translate this data in number of vCPU used by doing:

$$vCPU_{usage} = \Delta CPU \times N_{cpucores} = \Delta CPU \times 4$$

The result is given in table 6.11. In every case considered the vCPUs needed is well below 1. In case of master node, and particularly with the use of Longhorn, the vCPUs needed is not negligible and thus a cluster should be designed according to these constraints (i.e. more powerful nodes dedicated to masters or a number of workers adequate to the workload foreseen). In case of worker nodes, the overhead is very low but again, when using longhorn an additional resource usage comes into play and this could be a bottleneck especially on Raspberry Pi 4 devices where the usage appears to be still important.

Setup	arch	vCPUs	Δ Memory
k3s master	x64	0.19	330 MB
	arm64	0.25	641 MB
k3s worker	x64	0.06	84 MB
	arm64	0.07	154 MB
k3s master + Longhorn	x64	0.25	470 MB
	arm64	0.45	910 MB
k3s worker + Longhorn	x64	0.12	200 MB
	arm64	0.32	383 MB

Table 6.11

Chapter 7

Conclusions and future work

The work of this thesis analysed several possible approaches to the orchestration of geographically distributed services and to deliver the resiliency needed in the industry specific scenario. The definition of Kubernetes clusters on a one per site fashion still seems to be the best choice, due to the inherent resiliency provided by local master nodes. The use of Liqo to share critical resources, such as MQTT brokers, that have to be reachable from sites belonging to the same area allows for seamless communication inside an area. This simplifies the infrastructure management and setup, an important upside with the considered number of sites. The real management of geographically distributed clusters is carried out by Rancher Fleet that proves to be a valuable component of the architecture, allowing to join new clusters and classify them with labels and groups so that services can be declared based on the sites feature and location, instead of declaring services per each site. This cannot be said for the configuration too, since at the moment each cluster is expected to receive its own specific configuration (ConfigMaps, Secrets) and this still needs them to be defined per-site.

The demo produced focused on single cluster resiliency but still carried out preliminary experimental porting of application to Kubernetes in order to both prove that could be ported and that lightweight virtualization could fit this scenario with a limited impact on resource usage. Storage resiliency has been taken into account and considered critical part of the architecture. The evaluation focused on the overhead brought by lightweight virtualization and especially by the orchestrator, as well as storage resiliency and the time taken by Kubernetes to react to simulated faults. The results brought up relevant information necessary to design edge clusters both in hardware resources and number of nodes. The analysis on reaction times gives a general view of the behavior of Kubernetes and highlights where the tuning

should be carried out in order to improve the reactions in case of faults.

7.1 Future work

This thesis explored different possibilities for dealing with the problems of the industry when moving towards edge computing. However it leaves many options open for future development. Some of the solutions used in the implementation are not thought for this specific use case and might be adapted in the future to fully satisfy the needs. In this work, Longhorn CSI has been considered because of its features and its simplicity, more complex solutions might be more suitable to this use case and possibly even to the general multi cluster scenario. Liqo has also this ambition and the feature might come in future releases, however composed solutions are also possible with the use of CSI such as Rook-CEPH, Minio and OpenEBS.

The multi cluster management might be carried out and evaluated in different ways. For example, the pattern of having a centralized repository for configuration might be brought to next phase when project EVE becomes more mature. The use of EVE-OS and the development of a custom centralized controller looks like a natural continuation of this work. Moreover, Liqo can not only be used for resource sharing between peering clusters, like in our case for the MQTT broker, but also to schedule dynamic workloads since the central cluster will have the ability to schedule them in the peered clusters. This could mean that a given workload could be scheduled, generically speaking, in an area and actually scheduled on clusters with low loads without having to worry about where the service is nor about its management.

Another possible continuation could be a work on multi cluster dynamic scheduling, particularly following what has been presented in 3.3.2. An integration could be developed both with KubeFed as presented in [24] or with Liqo, even though, in the latter, changes to the framework might be necessary.

All the above mentioned possible paths should then be evaluated in terms of resource requirements to give an idea to the costs that such additional features would bring to the systems.

Bibliography

[1] R. Bayindir, I. Colak, G. Fulli, and K. Demirtas. «Smart grid technologies and applications». In: *Renewable and Sustainable Energy Reviews* 66 (2016), pp. 499–516. ISSN: 1364-0321. DOI: https://doi.org/10.1016/j.rser.2016.08.002. URL: https://www.sciencedirect.com/science/article/pii/S1364032116304191 (cit. on pp. 1, 2).

[2] Legambiente. *Il clima é giá cambiato*. Tech. rep. Nov. 2020 (cit. on pp. 1, 3).

[3] *2020 Tied for Warmest Year on Record, NASA Analysis Shows | NASA*. https://www.nasa.gov/press-release/2020-tied-for-warmest-year-on-record-nasa-analysis-shows. (Accessed on 09/28/2021) (cit. on p. 2).

[4] *Clean energy for all Europeans package | Energy*. https://ec.europa.eu/energy/topics/energy-strategy/clean-energy-all-europeans_en. (Accessed on 09/28/2021) (cit. on p. 2).

[5] Konstantinos V. Katsaros, Binxu Yang, Wei Koong Chai, and George Pavlou. «Low latency communication infrastructure for synchrophasor applications in distribution networks». In: *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. Venice, Italy: IEEE, Nov. 2014, pp. 392–397. ISBN: 9781479949342. DOI: 10.1109/SmartGridComm.2014.7007678. URL: http://ieeexplore.ieee.org/document/7007678/ (visited on 08/14/2021) (cit. on pp. 2, 37).

[6] Junwei Cao and Mingbo Yang. «Energy Internet – Towards Smart Grid 2.0». In: *2013 Fourth International Conference on Networking and Distributed Computing*. 2013, pp. 105–110. DOI: 10.1109/ICNDC.2013.10 (cit. on p. 2).

[7] Hossein Shahinzadeh, Jalal Moradi, Gevork B. Gharehpetian, Hamed Nafisi, and Mehrdad Abedi. «IoT Architecture for Smart Grids». In: *2019 International Conference on Protection and Automation of Power System (IPAPS)*. 2019, pp. 22–30. DOI: 10.1109/IPAPS.2019.8641944 (cit. on pp. 2, 3).

[8] IRENA. *Innovation Outlook: Renewable Mini-grids*. 2016 (cit. on p. 2).

[9] Terna. *Piano di Sviluppo 2021 (Development plan 2021)*. https://download.terna.it/terna/Piano_Sviluppo_2021_8d94126f94dc233.pdf. 2021 (cit. on p. 3).

[10] Callum MacIver, Keith Bell, and Marcel Nedd. «An analysis of the August 9th 2019 GB transmission system frequency incident». In: *Electric Power Systems Research* 199 (2021), p. 107444. ISSN: 0378-7796. DOI: https://doi.org/10.1016/j.epsr.2021.107444. URL: https://www.sciencedirect.com/science/article/pii/S0378779621004259 (cit. on p. 3).

[11] *Wikipedia - Sottostazione Elettrica*. URL: https://it.wikipedia.org/wiki/Sottostazione_elettrica (cit. on p. 7).

[12] Terna. *Italian National Grid*. URL: https://www.terna.it/en/about-us/business/italian-national-grid (cit. on p. 8).

[13] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes Up and Running*. O'Reilly, 2019 (cit. on p. 13).

[14] *Kubernetes docs*. URL: https://kubernetes.io/docs/concepts/ (cit. on p. 14).

[15] *Traefik kubernetes ingress*. URL: https://doc.traefik.io/traefik/v2.5/routing/providers/kubernetes-ingress/ (cit. on p. 16).

[16] *Flannel CNI*. URL: https://github.com/flannel-io/cni-plugin (cit. on p. 16).

[17] *k3s*. URL: https://k3s.io/ (cit. on p. 16).

[18] Sebastian Boehm and Guido Wirtz. «Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes». In: Mar. 2021 (cit. on pp. 16, 43).

[19] Tom Goethals, Filip DeTurck, and Bruno Volckaert. «Extending Kubernetes Clusters to Low-resource Edge Devices using Virtual Kubelets». In: *IEEE Transactions on Cloud Computing* (2020), pp. 1–1. DOI: 10.1109/TCC.2020.3033807 (cit. on p. 18).

[20] *Mosquitto MQTT Broker*. URL: https://mosquitto.org/ (cit. on p. 19).

[21] *Kube Edge official website*. URL: https://kubeedge.io/en/docs/kubeedge/ (cit. on p. 20).

[22] *KubeFed*. URL: https://github.com/kubernetes-sigs/kubefed/blob/master/docs/images/concepts.png (cit. on p. 21).

[23] *KubeFed*. URL: https://github.com/kubernetes-sigs/kubefed (cit. on p. 22).

[24] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. «Decentralized Kubernetes Federation Control Plane». In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020, pp. 354–359. DOI: 10.1109/UCC48980.2020.00056 (cit. on pp. 22, 63).

[25] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. «Conflict-Free Replicated Data Types». In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3 (cit. on p. 22).

[26] *Riak KV, docs*. URL: https://riak.com/products/riak-kv/resiliency/index.html?p=10906.html (cit. on p. 22).

[27] F. Faticanti, D. Santoro, S. Cretti, and D. Siracusa. «An Application of Kubernetes Cluster Federation in Fog Computing». In: (Mar. 2021) (cit. on p. 23).

[28] *Fog-Atlas*. URL: https://fogatlas.fbk.eu/ (cit. on p. 23).

[29] *Fog-Atlas CRDs*. URL: https://github.com/fogatlas/crd-client-go (cit. on p. 23).

[30] *Liqo*. URL: https://github.com/liqotech/liqo (cit. on p. 25).

[31] *Liqo Official Documentation*. URL: https://doc.liqo.io (cit. on p. 25).

[32] *Tensile-Kube*. URL: https://github.com/virtual-kubelet/tensile-kube (cit. on p. 26).

[33] *Edge ReferenceArchitecture*. URL: https://wiki.openstack.org/wiki/Edge_Computing_Group/Edge_Reference_Architectures#Distributed_Control_Plane_Scenario (cit. on p. 28).

[34] *TPM - Trusted platform module library*. URL: https://www.iso.org/standard/66510.html (cit. on p. 29).

[35] *Eve OS*. URL: https://github.com/lf-edge/eve (cit. on p. 30).

[36] *Eve OS wiki on LF-EDGE*. URL: https://wiki.lfedge.org/display/EVE (cit. on p. 30).

[37] *Helm - the package manager for Kubernetes*. URL: https://helm.sh/ (cit. on p. 31).

[38] *Rancher Fleet*. URL: https://fleet.rancher.io (cit. on pp. 31, 45).

[39] E-distribuzione. *Piano di Sviluppo 2020-2022 (Development plan 2020-2022)*. https://www.e-distribuzione.it/content/dam/e-distribuzione/documenti/e-distribuzione/Piano_di_Sviluppo_2020_22_30giu2020.pdf. 2020 (cit. on pp. 34, 35).

[40] Nick Jenkins, Janaka Ekanayake, and Goran Strbac. *Distributed Generation*. en. Institution of Engineering and Technology, Jan. 2010. DOI: 10.1049/PBRN001E. URL: https://digital-library.theiet.org/content/books/po/pbrn001e (visited on 08/14/2021) (cit. on p. 37).

[41] Agustin Zaballos, Alex Vallejo, and Josep Selga. «Heterogeneous communication architecture for the smart grid». In: *IEEE Network* 25.5 (Sept. 2011), pp. 30–37. ISSN: 0890-8044. DOI: 10.1109/MNET.2011.6033033. URL: http://ieeexplore.ieee.org/document/6033033/ (visited on 08/14/2021) (cit. on p. 37).

[42] Prashant Kansal and Anjan Bose. «Bandwidth and Latency Requirements for Smart Transmission Grid Applications». In: *IEEE Transactions on Smart Grid* 3.3 (2012), pp. 1344–1352. DOI: 10.1109/TSG.2012.2197229 (cit. on pp. 37, 41).

[43] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. «The many faces of publish/subscribe». en. In: *ACM Computing Surveys* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/857076.857078. URL: https://dl.acm.org/doi/10.1145/857076.857078 (visited on 08/14/2021) (cit. on p. 40).