POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Analysis of the Resilience of Monitoring Services in Smart Grid

Supervisors

Prof. FULVIO RISSO

Candidate

SEBASTIANO LA TERRA

October 2022

Summary

In recent times, more and more industries require monitoring and control systems based on a reliable IT infrastructure. Recently, the traditional electrical grid is facing similar requirements, as the introduction of renewable sources (e.g., solar panels, wind farms, and more) drives the need for enhanced monitoring. Since control systems depend on real-time data to function, it is unacceptable to work with data that may be obsolete before it reaches the control systems. These issues are the focus of the edge/fog computing paradigms, which place services close to the new data sources to enable analysis and computation as soon as the data is produced.

A scenario where centralized computations shift to the edge requires an in-depth analysis of a geographically distributed infrastructure, in addition to heterogeneous hardware as well as physically insecure locations. We also must consider that resources at the edge are less abundant than in a cloud environment and that network partitioning events that isolate one or more sites are a possibility. Hence, each site should therefore be resilient to internal failure and able to resist node, control plane, and storage failures.

Additionally, another crucial factor is the creation of a fault-tolerant system, which is realized using redundant components. In addition, increased grid observability is crucial both to enable a series of mission-critical applications such as congestion control and fault detection and to balance the power supply and the demand. As a result, the need for smart power grids has become more and more prominent because they enable to: (*i*) use tools to monitor the distributed energy generation; (*ii*) share data from sensors and meters; (*iii*) collect and process data to control the electrical power system.

Delivering resilient monitoring and computing services that use real-time data requires strong data resilience. For performing data analysis for statistical meaning or post-incident analysis, historical data durability is definitely essential too. To withstand hardware and network failure, mechanisms for performing consistent disk or volume backups and data replication are required.

This thesis shows a possible approach to build a resilient database cluster and, consequently, an analysis on resilience in data persistence, on the scalability aspects

of the entire infrastructure, and on the orchestrator timing involved in a solution that guarantees self-healing, in the face of the issues mentioned above.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past 70 years, the power system architecture consisted of big power plants – such as fossil-fuelled, nuclear power, or hydropower – capable of producing up to 1000MW and based on centralized control. There was an interaction between the production system and the transport system in order to ensure always the same value of frequency and to receive the required amount of energy. While this portion of the power system had an automatized control, the traditional electrical distribution systems, responsible of delivering energy to end users by gradually reducing voltage levels, was almost completely passive, with only local real-time monitoring and control for the largest loads, but no further interactions between the power system and the loads were carried out. [1].

The climate changes leading to global warming, driven by the human emissions of greenhouse gases, required the reduction of the produced CO2. 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 $\circ$ higher than the baseline 1951-1980 mean [3]. The solution thought by the European Union was to decarbonize energy system by using new renewable, green and clean sources of energy, 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 introduction of new power sources has made the production system distributed and, for this reason, it is no longer tolerable to have a centralized distribution system.

An increased grid observability is crucial both to enable a series of mission-critical applications such as congestion control and fault detection [5] and both to balance the power supply and the demand. In our case, the approach was to deploy a network of sensors, called Phasor Measurement Units, capable of providing information about the physical world. Specifically, they are well-known devices that produce accurate and synchronized voltage/current phasors with sampling

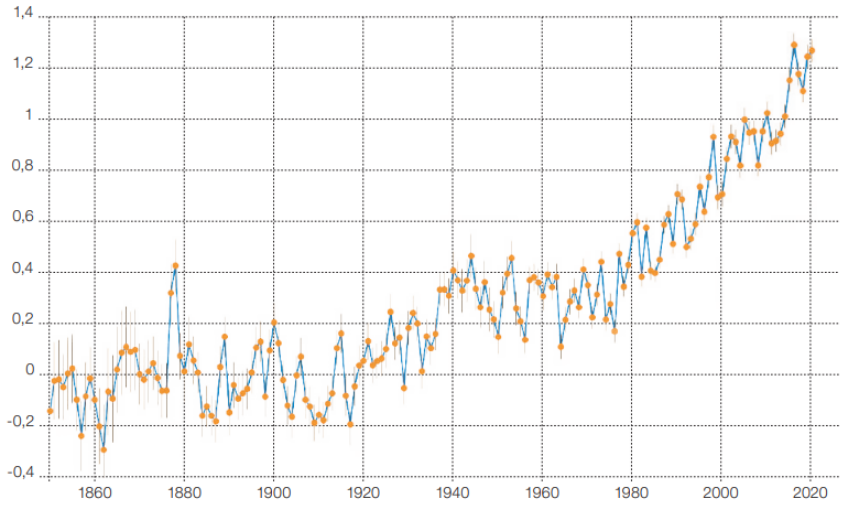


Figure 1.1: Temperature anomaly in C ◦ from 1850 to 2020 to respect the common baseline 1951-1980 mean

From Legambiente climate report 2021 [2] - Berkeley Earth data combined with sea data from UK Hadley center

rates up to 60 measurements per second and are frequently used in transmission systems. However, there is an increasing interest in PMUs for distribution networks as well, in order to implement novel schemes for better control and fault location. As a result, the need for smart power grids has become more and more prominent because they enable to:

- use tools such as remote reading and load shedding in emergency cases in order to monitor the distributed energy generation [1];
- share data from sensors and meters;
- collect and process data to control the electrical power system.

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 enters in the picture. A micro-grid is a local interconnected network for electricity delivery from producers to consumers with defined electrical boundaries. So, it is a local network of energy production

and distribution, that normally can work both attached to the main grid, or as an *island*, which means autonomously, isolated from the rest of power grid.

The core functionalities for micro-grid technologies are: (*i*) power generation; (*ii*) power storage; (*iii*) control, manage and measure; (*iv*) convert and consume [8].

The current concept of micro-grid is different from the past, wherein it was intended as a backup system of the main grid and it mostly relied on fossil fuels. Indeed, nowadays micro-grids rely on renewable sources of energy (such as solar panels or wind turbines) and this innovation enables the improvement of reliability and the reduction of both the environmental impact and costs.

Since the number of extreme weather events is increasing, as mentioned above, the resiliency of the power grids is becoming the main requirement. This is one of the main features of smart grid 2.0. In this case each micro-grid can cooperate with the main grid supporting it by means of injecting power or exchanging/requiring electricity as needed. In any case, even if the micro-grid detaches from the main grid, on purpose or because of unintended events, it can continue to work independently.

The increase of the average temperatures causes a reduction in rainfall, but also an increase in intense sporadic rainfall with a consequent rise of floods, storms, and hydro-geological risk [9]. Figure 1.2 shows the increase of extreme weather events in Italy for each year. It is evident how, in general, climate change causes an increase in the frequency and intensity of extreme weather phenomena [9], and people are inevitably obliged to get used to this new normality. For this reason, it is essential to redesign human infrastructures according to the current climate changes in order to make them resilient to the weather consequences of this tragic trend. This updates involve also the electrical power system, which should be able to predict, react and survive these extreme events. These arrangements are crucial for the new design of the smart grid 2.0.

1.2 ICT resiliency in a smart grid 2.0

More than 50% of the power consumption in the smart grid 2.0 must be controlled in real-time, necessitating the computing and the monitoring of a large amount of data from sensors and devices. The deployment of smart IT technologies and the use of big data analysis techniques are required for the analysis and processing of this enormous amount of data as well as the control of the grid [7]. The post incident analysis conducted after the August 9th, 2019 transmission system frequency event in Great Britain provides a practical illustration of the need for data and their usefulness [10]. The authors used the phasor data at their disposal to try to reconstruct the different stages of the incident and to draw a lesson from it. Furthermore, a robust ICT infrastructure should offer the power grid resilience. Two of the key features that must be present in a resilient infrastructure are:

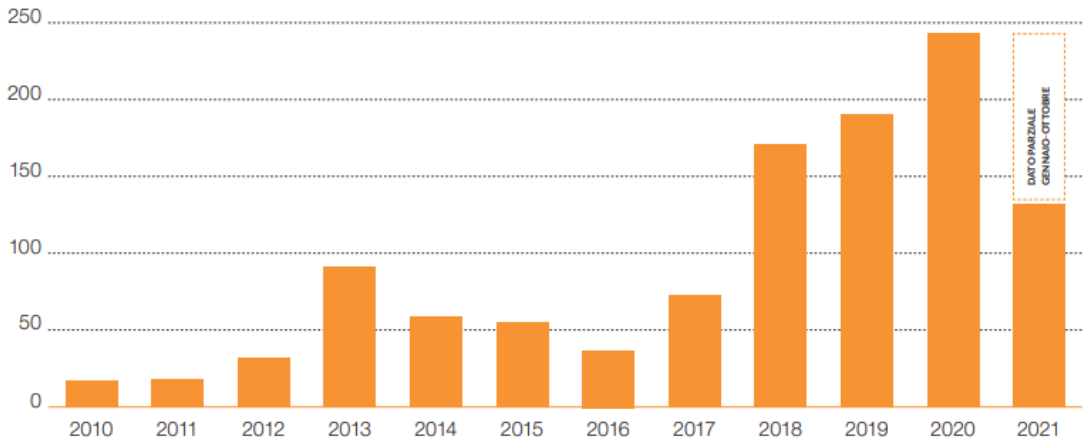


Figure 1.2: Extreme weather events in Italy for each year
From Legambiente climate report 2021 [2] - Osservatorio Cittá Clima, Legambiente 2021

- services must be watched over in order to restart them if the application or the node on which it was running fails;

- even the ICT must support the partitioning of the infrastructure, just as the electrical grid should be able to. Extreme weather conditions, accidental events, or network failures could isolate one or more electrical power system sites. When a location is isolated, its ICT infrastructure should be able to respond and continue operating despite the loss of communication with the centralized control.

The complexity brought on by the infrastructure's large geographic distribution should be handled by the power grid's ICT. Since it manages hundreds of thousands of sites, and this number might quickly increase over time, the solution's scalability is essential. As a result, the solution must support the plug-and-play electrical grid concept, allowing additional locations to seamlessly join with the rest of the infrastructure. Numerous various types of devices and sensors, some of which have limited computing power, are dispersed throughout the power grid and generate data over a variety of physical media. This enormous amount of data must safely reach all users in accordance with their QoS needs, and that is the function of the power grid's ICT. All services operating over the smart grid should be able to generate and consume data, moving transparently between ICT infrastructure nodes as necessary. To increase the scalability, maintainability, and simplicity of the applications while still keeping latencies under control and supporting real-time applications, data should be produced and consumed using an asynchronous approach.

1.3 Overview of service resiliency and scalability on power grids

This thesis work will focus mainly on the analysis of the infrastructure developed using PMUsim, OpenPDC and MySQL services, analyzing resilience in data persistence, the scalability aspects of the entire infrastructure and the timing involved in a solution that guarantees self-healing. Chapter 2 provides an overview of ICT architecture in power grid systems, which is required to provide context for understanding the motivation of later choices. Chapter 3 shows a description of the technology used, namely Kubernetes, with a subsequent presentation of the edge-oriented solution, K3s. In chapter 4, an architecture is defined, taking into account the various aspects of the problem, such as service communication, latency issues, and data resiliency. Chapter 5 shows several solutions that ensure redundancy between different replicas of the database, highlighting the main advantages and disadvantages to understand the reason for the choice made. A brief description of the developed implementation is presented in chapter 6, focusing mainly on the services used in the infrastructure. It is also explained reason for choosing some technologies over others. Finally, there are also results that show resource consumption with this current implementation. In Chapter 7 the concept of self-healing is mainly deepened with greater attention on the response time of the orchestrator in the face of possible failures. Chapter 8 presents the simulator used for a further deepening of the scalability of the infrastructure with the respective results of the analyses carried out. Chapter 9 sums up the work and points out possible aspects worth to be further explored.

Chapter 2

ICT architecture in an electrical power grid

An overview of the ICT infrastructure in the electrical power grid is what this chapter aims to do. 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:

- the production system refers to the process of producing electricity, converting it using the proper current, voltage, and frequency values, and then introducing it into the transmission system. This was primarily carried out at large production facilities in the past (such as those that produced coal or hydroelectricity), but in recent years, many smaller production facilities have begun to incorporate it (e.g., solar power);

- the transmission system is in charge of transferring electricity from power plants to distribution systems (i.e. Terna in Italy);

- the distribution system is responsible for getting electricity to the ultimate customers; normally, the energy providers are in control of this area of the network.

As mentioned above, nowadays, due to the existence of numerous small producers closer to the consumer, such as solar panels, wind farms, and more, the production system is no longer the only source of energy. This means that even in distribution systems, where it is no longer possible to have a completely centralized control, it was necessary to move this control even to the edge, such as mechanism similar to that found in the production system.

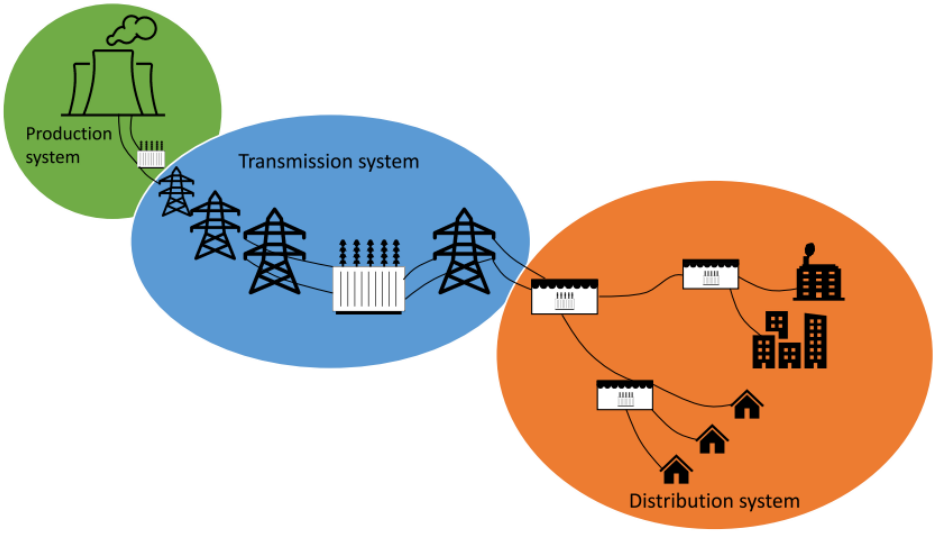


Figure 2.1: Electrical hierarchy overview.

2.1 Production system

Energy producers build the production system by generating electricity, collecting it in the transmission system using a few transformers, and controlling the voltage and intensity of the electricity. In order for generators to maintain the same value of frequency and supply the necessary amount of electricity in the electrical grid, producers must control the frequency of the produced energy in accordance with the values provided by the national controller. Electromechanical generators, which are primarily powered by heat engines powered by combustion or nuclear fission, are the most common way that electricity is produced at a power plant. In alternative, as shown in Figure 2.4, electricity can be generated by the kinetic energy of wind and water flow, solar photovoltaic panels and geothermal power and wind turbines.

2.2 Transmission system

Electricity transmission is the stage between producing electricity and distributing it to consumers. It is carried out with the use of a network architecture comprised of 380 kV, 220 kV, and 130/150 kV electrical towers connected to each other, forming a single grid that spans the entire country. 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

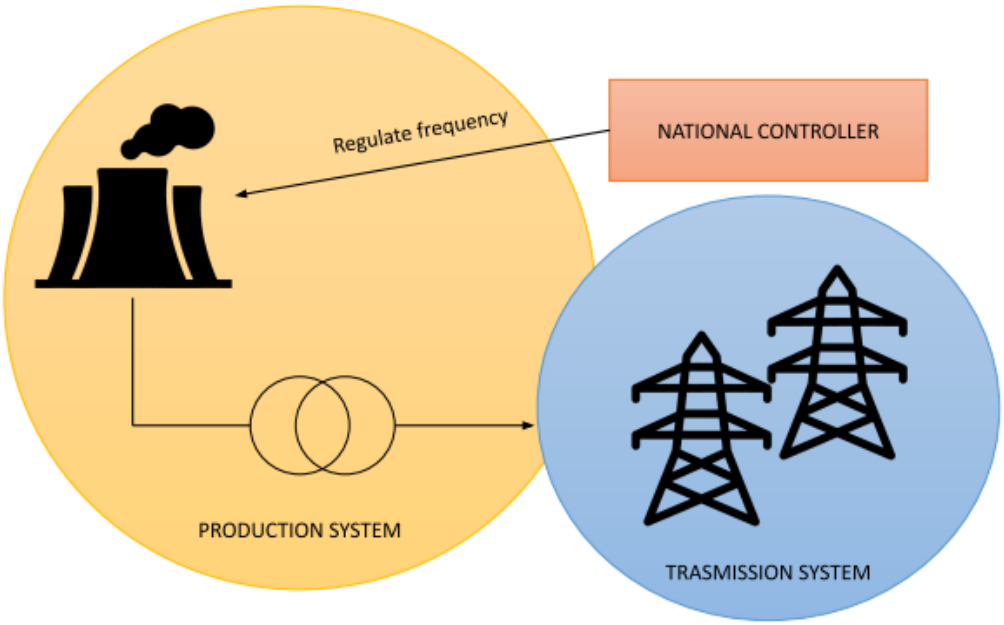


Figure 2.2: Production systems work in synergy with the transmission system.

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. From this station there is a hierarchy of controllers. First it is connected

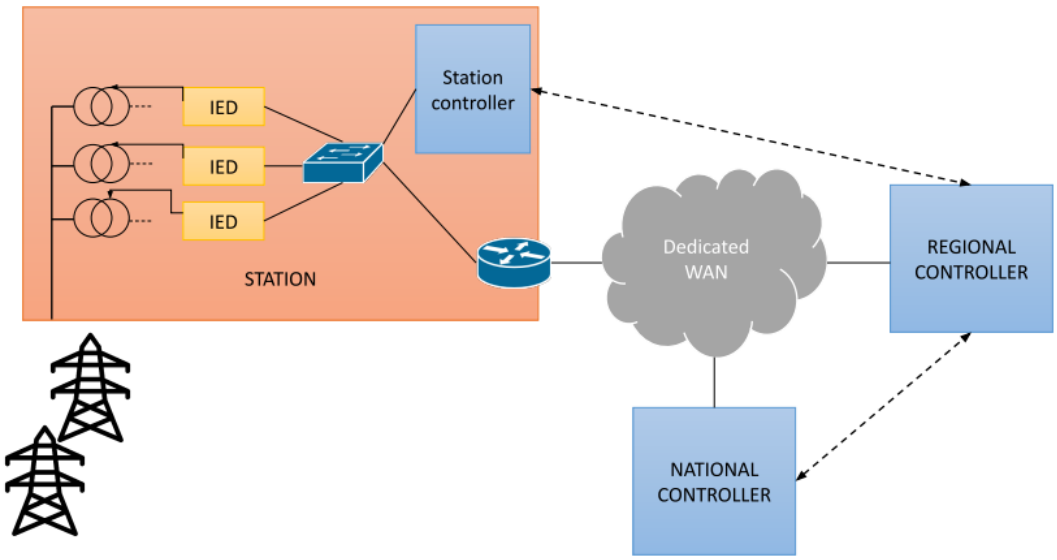


Figure 2.3: ICT network architecture of the transmission system.

to a regional controller, which in turn 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.

The control of the entire infrastructure is fundamental because electricity cannot be stored, therefore there is the need to guarantee the balance between the produced energy with the demand. The National Controller is in charge of this dispatching operation, which is a real-time control. The National Controller collects data from numerous players involved in both supply and demand, makes predictions about the country's electricity needs, and communicates with producers and remote management centres to module the grid's supply and structure as necessary [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.

2.3 Distribution system

The primary substations, where high voltage electricity is transformed into medium voltage, are where the distribution system gets its start. Here, a variety of measurement systems are used to monitor the condition of the transformers and make necessary corrections opening and closing them, changing the transformation ratio, in order to maintain the electrical grid's proper operating point. The medium voltage lines, each of which is controlled by a switch, begin here. The medium voltage

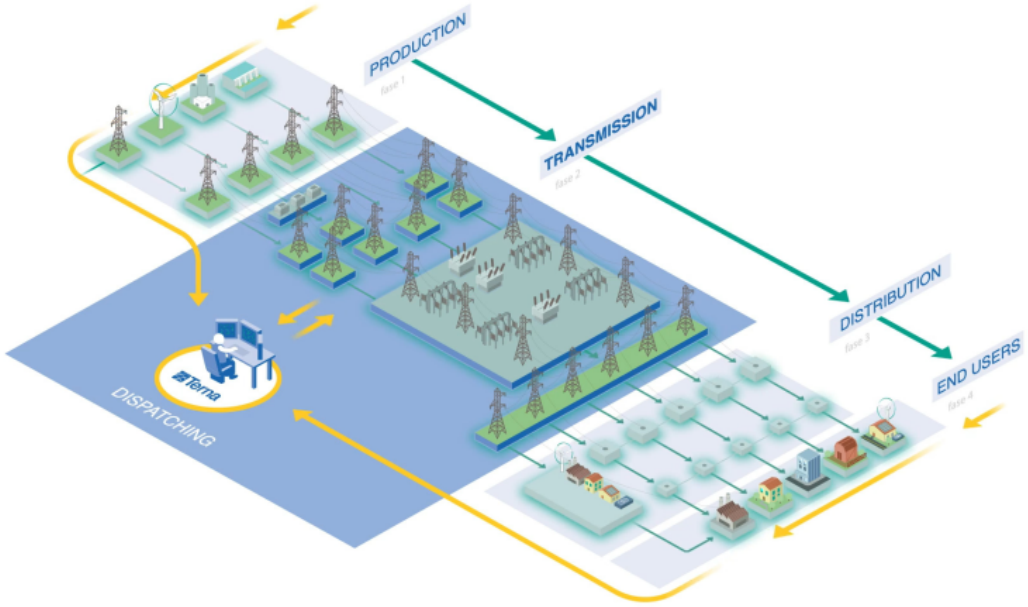


Figure 2.4: Terna's transmission system [12].

is changed to low voltage when these lines reach the secondary substations. These serve as the entry point for low voltage lines that connect user loads, electricity generation systems, and accumulation systems. These lines may also be equipped with sensors and metres that can provide data on their operational status. While data from loads, electricity generation systems, and accumulation systems coming from the outside could reach the controller in charge of handling it using GSM, 4G, or powerline, sensors and actuators inside kiosks are connected via Ethernet LAN. 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 local control.
- Area control centers: the station controllers of the substations communicate with the area control center of the geographic area in which they are situated. This area may include all or part of a city.
- ICT control center: this is the remote monitoring center for the ICT of the electricity provider. Its responsibilities include setting up all the devices, monitoring the state of the infrastructure, looking for anomalies like failures or intrusions, and attempting to recover from incidents. The transmission

system also contains this component.

Typically, a dedicated WAN network, which may be composed of fibre or a similar technology, is used to carry data travelling between the control centres. A firewall protects each area of the network, filtering both incoming and outgoing traffic. An access point that encrypts both the incoming and outgoing traffic further protects the entire ICT system. This is due to the fact that various distributors and the ICT of the transmission system utilise various keys to maintain their independence. 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 should hold true for communications between transmission systems of two different countries. This is necessary because, since Europe uses a single frequency, any changes to that frequency or failures of individual grid components could affect all other countries, requiring appropriate responses. This demonstrates how important a good ICT system is for a functioning power grid. The ICT should be in charge of ensuring that each component is functioning properly and tuning each of them to achieve the intended state, but it also has a protective role to play, such as disconnecting a power plant from the grid when it gets out of frequency. Even while both control and protection help to maintain the power grid's proper operational status, they are completely separate from one another because they have different requirements, even in terms of response time.

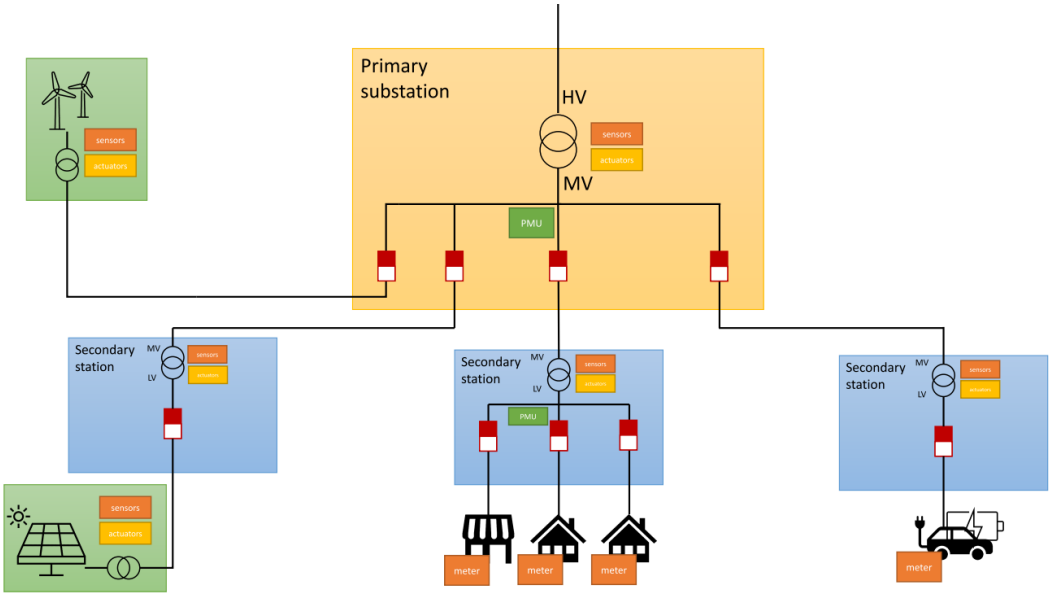


Figure 2.5: Electrical architecture of the distribution system [13].

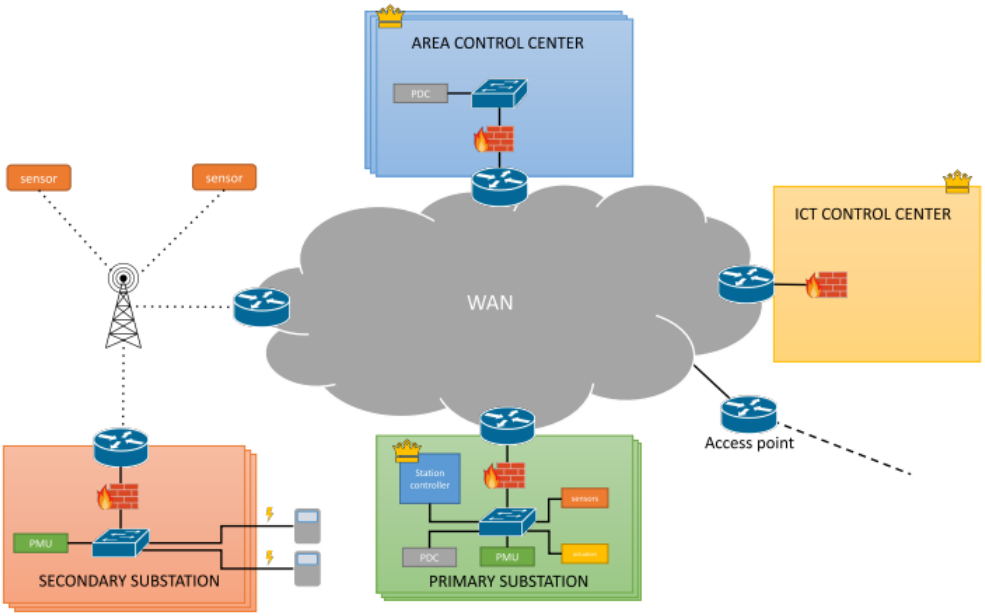


Figure 2.6: ICT network architecture in the distribution system [13].

Chapter 3

Kubernetes

In the last decades some paradigms have been born like CI/CD (Continuous Integration and Continuous Delivery) and more in general the DevOps model. These paradigms were born to optimize the software management process, at all stages of its life cycle.

A first solution, prior to the advent of Docker, was the introduction of virtualization technologies for software lifecycle management. All the various virtualization technologies include virtualization of physical hardware, through the use of a Hypervisor. Each virtual machine is a machine equipped with all the components: a virtual copy of the hardware that the operating system requires to run, the applications and their associated libraries and dependencies and also the operating system (called Guest OS).

However, while they offer a high degree of isolation (each VM is actually a separate drive) on the other hand one of the main limitations, which makes modern applications incompatible with virtualization itself, is the significant waste of computing resources. This limitation results from the overhead of resources due to the need to execute the guest OS, as well as the applications it runs, thus causing a degradation of the overall performance of the system. As a result, there was a need to have a system that maintained the same benefits of virtualization (scalability, isolation) but with a lower resource consumption.

From this derives the birth of containers that do not virtualize the underlying hardware, but rather virtualize the operating system (usually Linux), so that each container includes only the application and its libraries and dependencies. Since the guest OS is no longer present, containers are lighter and, therefore, faster and portable than VMs. As a result, Docker and mainstream container technologies, which quickly took over standard virtual machines, has resulted in a significant change in application development and deployment during the previous decade.

In Figure 3.1 we can see the evolution of application deployment, from traditional deployment to container-based deployment.

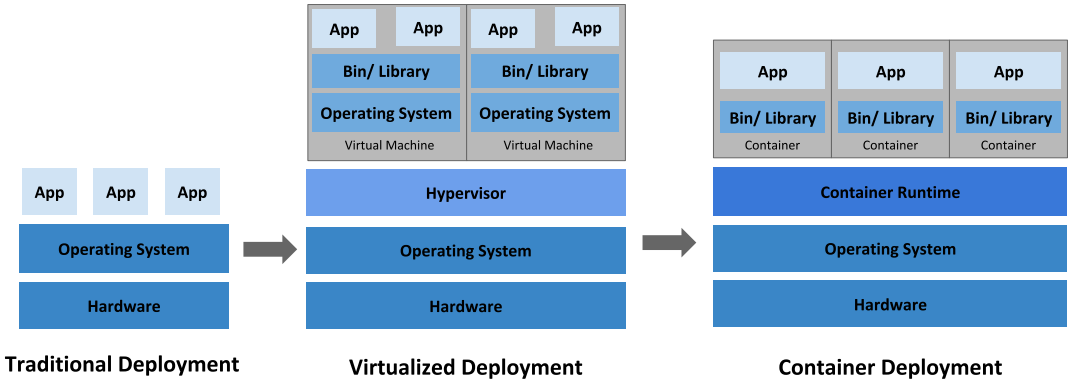


Figure 3.1: The various eras of deployment [14].

Container-based applications have become increasingly popular, and the need to make these applications scalable and resilient has led to the emergence of a framework that can automate and manage the lifecycle of thousands of containers. This system is usually an Orchestrator and in our case we use Kubernetes.

A brief overview of the basic concepts of Kubernetes will be given including core resources and some of its main components.

3.1 Basic Concepts

Key concepts of Kubernetes include:

1. Completely declarative specification: the idea is to describe, using Kubernetes API objects, the state we want to get and we don't have to worry about how we get it;

2. Control Loop-oriented approach: the control plane implements a control loop, observing the desired state and making changes with the aim of moving the current state towards the desired one.

The basic deployable execution of a Kubernetes application is a Pod, which is a group of one or more containers. A Pod's contents are always co-located and co-scheduled, and executed in a shared context. In the pod's manifest we can define labels to identify the resource, open ports, container restart policy, volumes to be mounted etc.

In a container environment, configuration cannot always be known at the time of deployment, may vary over time, and may be used again. ConfigMaps and Secrets both provide the option to build straightforward key-value pairs that can be used as strings in pod templates or mounted as files in pods' volumes. However, pods

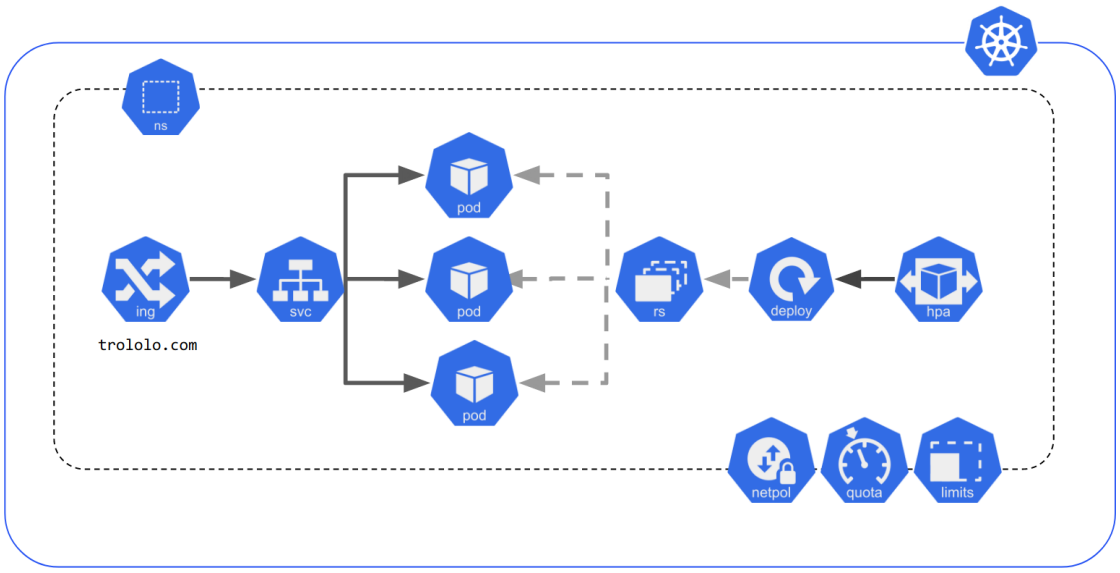


Figure 3.2: Architecture Kubernetes [15].

usually are not deployed "as is", applications are mainly deployed as *deployments*. If the Pods need to track state, consider the *StatefulSet* resource.

Deployment is a resource that permits to create stateless Pod and to define the number of replicas. It creates ReplicaSet in the background and it, in turn, creates Pod. It permits to scale up and down the number of replicas of a specific Pod. The deployment controller will enable self-healing by ensuring that the number of running replicas remains constant and, if necessary, restarting containers or reinstantiating pods. These things are useful for stateless application, where there is no persistent data. StatefulSets, instead, are valuable for application that require persistent data. In this case each pod has a specific storage volume. Persistence is managed through the PersistentVolume and PersistentVolumeClaim resources, the first represent a piece of storage in the cluster, that resides in physical drive whereas the latter represent the request for a persistent volume made by a user, so it tells that a pod wants to access to a given piece of storage. Since the pods can be updated, be deleted and be rescheduled changing their address, in Kubernetes there is a way to expose applications in a stable way. It is the Service and it is used to make a pod reachable either at cluster level or from outside the cluster. In the Service's manifest we 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. Basically we have three type of services: (1) ClusterIP, which permits to expose an application only inside the cluster; instead, Node Port and Load Balancer permit to expose application outside the cluster.

3.2 Core Modules

A Kubernetes cluster is composed of nodes and there are two types of nodes: master node and worker node.

Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods [16].

Kubernetes control plane consists of a collection of processes running on the cluster:

1. The Master Node is a collection of four processes that run on a single node in the cluster, which is designated as the main node. They are:

   (a) kube-apiserver: it is the front-end for the Kubernetes control plane, so it is the component that exposes the kubernetes API [17];

   (b) 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 [17];

   (c) kube-scheduler, a component that decides in which node to schedule a new pod, based on the pod resources, taints, affinities and other kind of configurable constraints [17];

   (d) kube-controller-manager, a component that runs the different controller processes. Example of controllers are: Node controller, deployment controller, Service Account and Token controllers. They are responsible to move the current state towards the desired state [17].

2. Also, there are other processes that we find in both a master and a worker node:

   (a) kubelet, an agent that is responsible of making sure that containers declared in a Pod are running. It checks the liveness of each pod telling all the above information to kube-apiserver. The kubelet takes a set of PodSpecs and ensures that the containers described in those PodSpecs are running and healthy [17];

   (b) 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 [17];

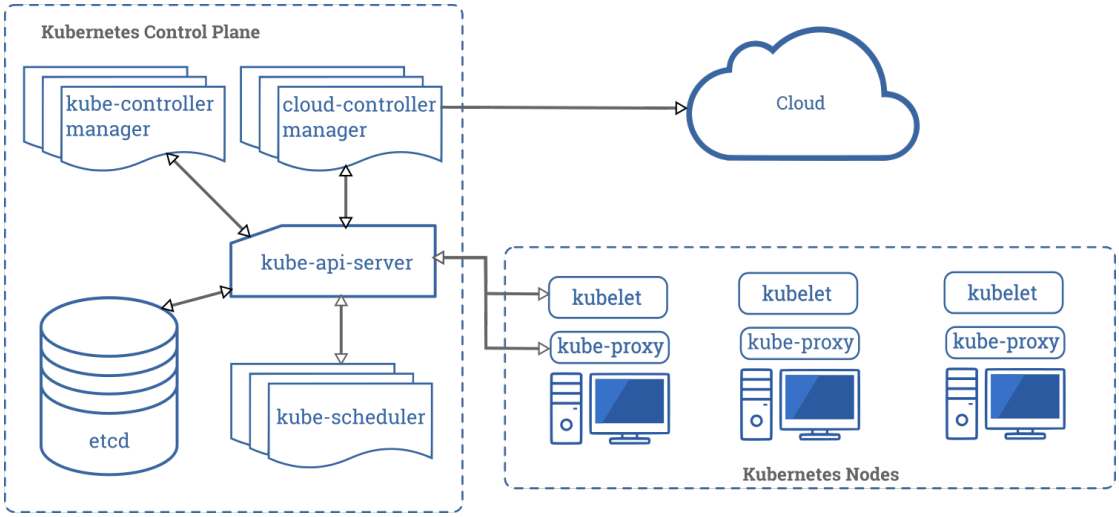


Figure 3.3: Kubernetes Components [17].

(c) CNI (Container Network Interface) is a CNCF project that defines a specification and libraries for configuring network interfaces in Linux containers. If a CNI network plugin is installed in a cluster, it is automatically done in each node. It is responsible for: (*i*) inserting a NIC into the container namespace and making any necessary changes on the host to attach the NIC to the network; (*ii*) perform the IP configuration [17];

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

3.3 K3s

k3s is packaged as a single binary, that makes easier the installation of a cluster simplifying the setup and management. It also aims to reduce the overall load on the host machines 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. For this reason it is called "lightweight Kubernetes", as also demonstrated and analyzed in Bohm and Wirtz [18], in which a comparison with the microk8s and with vanilla Kubernetes is shown. 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 that CPU and memory resource usage were quite similar, with k3s using the lowest amount of disk space in comparison to the other two.

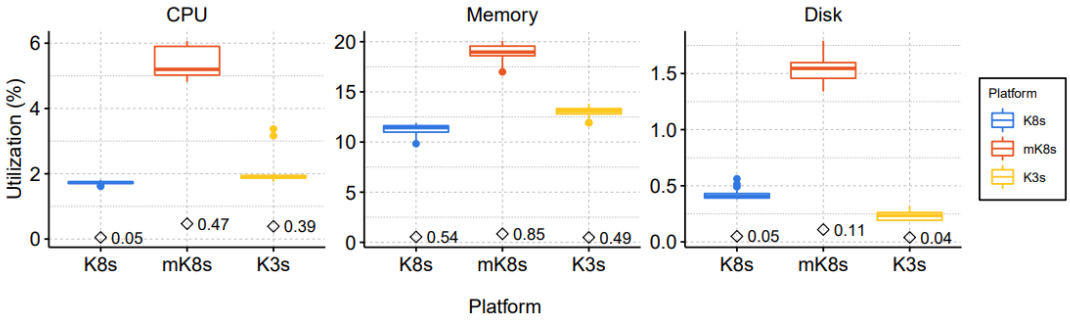


Figure 3.4: Resource utilization for K8s, MicroK8s, and K3s [18].

So, since k3s has low memory/cpu footprint, as shown in Figure 3.4, it works well on bare metal with relatively low computational resources, especially for edge devices or IoT appliances.

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,
- an embedded load balancer,
- an embedded network policy controller,
- a default lightweight CNI Flannel [19].

The user has the option to alter the behaviour of the cluster by disabling any of the above plugins. The installation script provided allows to set all of the upstream Kubernetes parameters as well as a few particular flags that are specific to k3s-only behaviours.

Chapter 4

Orchestrated architecture for the power grid

4.1 Service and infrastructure resiliency

The difficulty of a geographically distributed infrastructure is relevant in a scenario when centralized computations shift to the edge. The infrastructure could include physically insecure sites and hardware from different manufacturers.

We also must consider that resources at the edge are less abundant than in a cloud environment and those network partitioning events that isolate one or more sites are a possibility. Each site must therefore be able to resist network partition and isolation from the cloud, and it is obvious that a fully centralized control plane cannot be the answer. Kubernetes is considered the foundation of the architecture described in this chapter and it can be very useful in orchestrating workloads.

4.1.1 Services

The services considered are PMUs and PDCs:

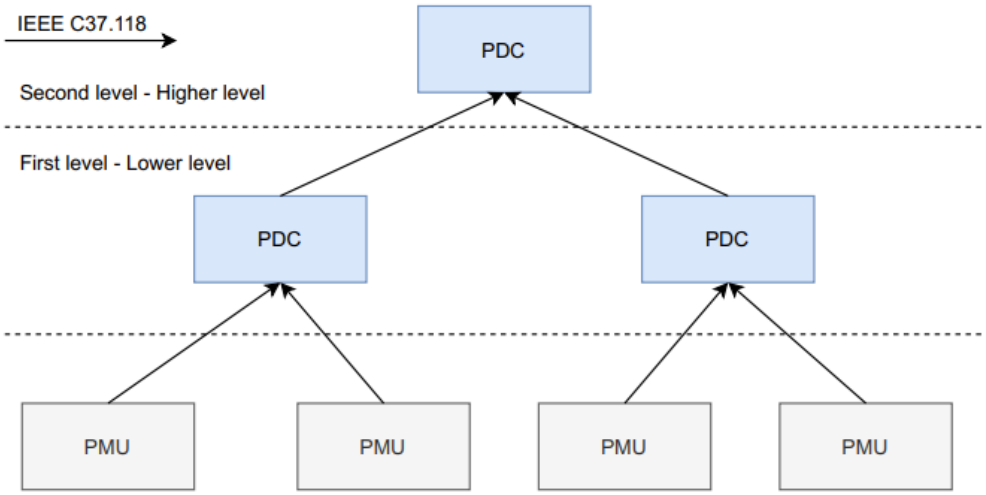
- PMUs (Phasor Measurement Unit) require physical hardware and, being measurements units (data producers), their location is bounded. Therefore their placement is considered fixed and might be in each site (Secondary and Primary stations as well as production sites);
- PDCs (Phasor Data Concentrator)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

4.1.2 Data resiliency

Delivering resilient monitoring and computing services that use real-time data requires strong data resilience. For performing data analysis for statistical meaning or post-incident analysis, historical data durability is definitely essential. In order to withstand hardware and network failure, mechanisms for performing consistent disk or volume backups and data replication are required.

At the cluster level, a first level of data resilience should be achieved so that data is duplicated across multiple nodes rather than being bound to a single node. At a larger scale, another level of data resilience should be attained by performing regular backups and pushing them to the cloud so that analysts may access the data.



4.2 Data flow and communication resiliency

Table 4.1 shows the number of primary and secondary stations across the years reported in the Development plan 2021-2023 of e-distribuzione [20], a company inside the Enel group operating in the electrical distribution sector. The plan shows that over the past six years, the number of stations (secondary and primaries) in

their distribution grid has increased. Furthermore, the number of sites can increase quickly, necessitating a scalable solution that enables the addition of new sites to the group as effortlessly as feasible. Since there are hundreds of thousands of peripheral sites involved, as shown in the table, each one should be independent from centralized control so that it is possible for it to continue operating even if it is isolated.

	2015	2016	2017	2018	2019	2020
Primary substations	2.188	2.195	2.199	2.203	2.200	2.336
Secondary substations	441.056	442.418	443.774	445.159	446.411	447.250

Table 4.1: Number of primary and secondary station over years 2015-2020 [20].

The proposed ICT architecture must take into account the data of the table 4.1 and, therefore, it becomes necessary to have a huge network of sensors, capable of providing information about the physical world and interacting with each other with the aim of increasing the observability of the power system. In addition, the collected data should be later stored, processed, and analyzed in order to control and optimize the behavior of the grid.

Various network problems must be taken into account in the implementation of the ICT architecture. Thus, the proposed architecture should be able to: (*i*) minimize network path lengths, (*ii*) improve network stability, (*iii*) improve resilience and reduce network perturbations (packet loss, jitter) caused by buffer bloat (although the former seems to have limited importance in modern edge networks) and (*iv*) reduce the network latency.

The distribution system typically depends on the general-purpose network connectivity provided by telecommunications companies, frequently through a 3G/4G/5G mobile connection. In contrast to the transmission system, where the IT network topology typically follows the topology of the power grid (often, optical fibres lay together with electrical cables in the same location) and is privately self-managed by the energy provider. As a result, communication between two sites, especially the traffic between PMU and the Low-Level PDC and/or between the Low-Level PDC and the High-Level PDC) in the distribution system of the network requires transit through the telecommunication provider's network since the topology of the power grid differs from the physical structure of the IT network. This means that, as shown in the Figure 4.1, it is likely that the sites of the distribution system are not directly connected by dedicated links but rather that a transit through the network of the telecommunications provider is necessary to get from one site to another. This has a significant impact on the overall IT architecture since the physical topology of the IT network determines where to deploy redundant IT services and helps to improve communications and cut down

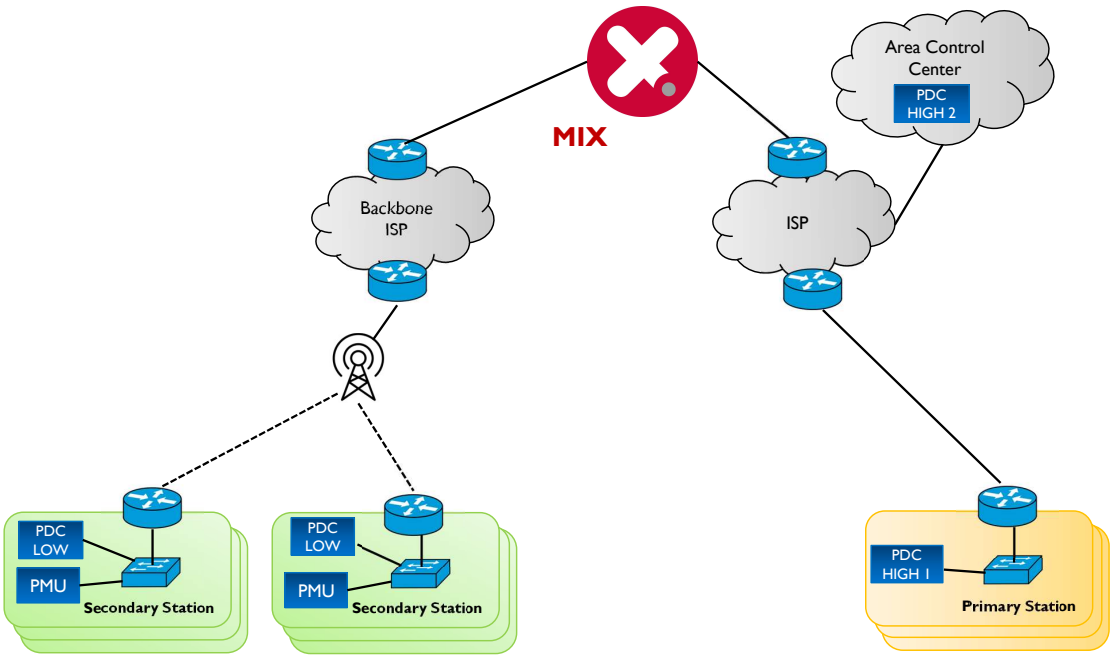


Figure 4.1: Example of ICT network architecture in order to show the path of the packets.

on latency.

Knowing the path of traffic is extremely important for optimizing it and reducing latency. For example, we can assume to have at a high level an architecture like the one in Figure 4.1: a lower level PDC for each site of the power grid, collecting the data of the local PMUs. It redirects the input stream to one of the many higher level PDCs, located inside the primary station. Then the latter in turn redirects its input data stream to a third-level PDC, located in the ISP, where the output can be sent as input of the local state estimator or other applications performing data processing or storage of historical data.

However from the point of view of IT architecture, looking at Figure 4.1 and following the traffic, it might be clear how the exchanged data across many times the public network:

1. PMU data exits the electricity provider network and, through a 5G antena, enters the ISP network;

2. the data stream must cross the Milan Internet eXchange (assuming that we are considering the network of Milan), to allow the exchange of data between different Internet Service Providers (ISP);

3. from the MIX, it is routed again to another ISP, the one directly connected to the GSE;
4. from the latter network, it is routed to the power provider network in order to be delivered to the PDC;
5. the output stream of the PDC needs to go back to the GSE's ISP network;
6. finally, the stream reaches the state estimator.

It is important to notice the ICT network of the distribution system is completely different from the architecture of the electricity network (shown in Figure 2.5) . The ICT network uses one or more telecommunication provider network and in this way the data traffic of the distribution system follows the path mentioned above.

Chapter 5

Redundancy Database

A fault-tolerant system strongly requires the redundancy of the components, in this way a system works even in the face of a fault. In particular, with the replicated databases it is no longer enough to maintain and control a single server, but now it is necessary to manage a much greater number. In addition, since the servers collaborate with each other, there is the need to solve several problems, such as network partitioning or split-brain scenarios.

As a result, the ultimate difficulty is to combine database and data replication logic with the logic of having several servers coordinated in a consistent and simple manner. So, having many servers agree on the status of the system and the data on each change that the system undergoes. The final goal would be to have a distributed database system capable of behaving as a single database or capable of converging all components to the same state [21].

From the current architecture, an interesting fact has arisen: MySQL represents a Single Point of Failure because it is the Pod that allows PDCs to read their configuration, but it is also where the PDC stores the data streams they receive from PMUs. So its failure results in a lot of lost data for the entire duration of the downtime. Therefore, it is necessary to lower, if not reset, the downtime in order to ensure greater resilience of the database. In order to realize this in the next section, various solutions that guarantee the redundancy of MySQL have been analyzed and they come shown the various tried solutions.

5.1 Single-Master vs Multi-Master

There are several solutions that provide different methods of data replication. These mechanisms come into play when you want to have a live backup of information or when you want to adopt a solution that guarantees us greater reliability and availability of data. In the next section the two main replication methods are

presented: (*i*) Master-Slave replication; (*ii*) Master-Master replication.

5.1.1 Single-Master

In a single-master solution, there is only a master node capable of accepting both writing and reading operations and, also, more replicas (slaves) that accept only reading operations. The data storage process is first carried out in the master node and then the data will be copied to the slave nodes, asynchronously.

The main advantages of this solution are:

1. the possibility of decoupling the reading operations from the writing operations, in such a way as to reduce the computational load in the master node;
2. it is very fast and doesn't impose any restrictions on performance [22].

The disadvantages, instead, are:

1. the asynchronous replication approach reduces reliability. Asynchronous replication works in the same way as store-and-forward. Each change is made in the master node and then propagated and applied to the other nodes in the cluster at regular intervals. Using this type of replication, a certain time interval is required before all nodes reach the convergence of data. As a result, failure of the master node, before replication operation, causes data loss;
2. a significant increase in writing operations can cause congestion in the master node, and the only method to scale the writing operations is to expand the master node's processing capacity [22];
3. when a master node failure, an implementation of a mechanism capable of electing a new master node is required [23].

From the implementation point of view there is also another limitation: in the PDC yaml file you can only specify a database URL to contact. This requires a mechanism to expose the master Pod through a Service K3s NodePort and also implements a logic capable of attaching a new Pod as an endpoint in case the master fails, as shown in Figure 5.1.

5.1.2 Multi-Master

In a multi-master solution, the hierarchy between databases is broken down and it is assumed that you have two or more master nodes where both read and write operations are allowed. In addition, this solution requires the implementation of a synchronization mechanism and management of concurrent writes.

The main advantages of this solution are:

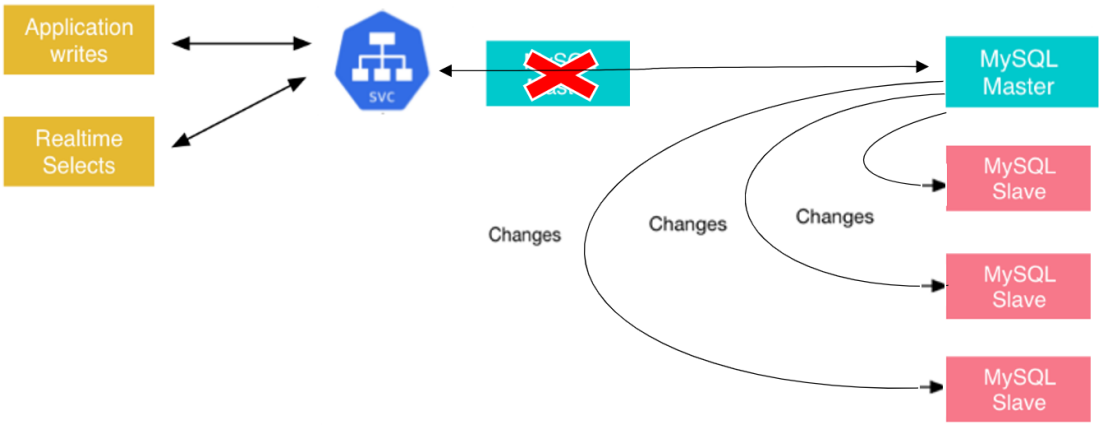


Figure 5.1: What should be done for proper operation on Kubernetes when a Master Node fails.

1. the ability to scale the infrastructure not only by increasing the computational capacity of the nodes but also by adding other master nodes [22];

2. greater reliability and resilience, since the probability that all nodes will fail at the same time, is very low and, moreover, even if a node fails there would be other nodes capable of handling requests;

3. there is no need for any mechanism of the election of a new master;

4. since it is based on synchronous replication, you can be confident that you will not lose committed data in case one of the Master node fails;

5. regarding Kubernetes, another great advantage is that there is the possibility to expose the entire cluster through a single Service K3s, which will act as a Load-Balancing among the nodes available.

The disadvantages, instead, are:

1. it uses more bandwidth because it is based on synchronous replication;

2. the cluster performance is limited by the worst node, the so-called short board effect (bucket law);

3. adding a new node is expensive because when you add a new node, you must copy the complete data set from one of the existing nodes. You must copy 100GB of data if the existing database contains 100GB of data.

5.2 Redundancy for MySQL

In this section a description of several used solutions will be provided. In addition, the reason why one solution was chosen over another will be stated. The several used solutions are:

- MySQL Operator Oracle – Single Master,
- MySQL Group Replication – Single Master,
- Bitnami Chart MYSQL - Single-master,
- MySQL Group Replication – Multi Master,
- Percona XtraDB Cluster - Multi Master.

5.2.1 MySQL Operator Oracle – Single Master

The MySQL Operator for Kubernetes was created by the MySQL team at Oracle and it is an operator responsible for the setup and management, including periodic updates and backups, of a MySQL Innodb Cluster within a Kubernetes cluster [24]. The main key features of this solution are:

1. Self-healing solution: it provides a (*i*) complete high availability solution for MySQL running on Kubernetes and (*ii*) it is built on InnoDB storage, using group replication;
2. Backup and restore databases, which permits to create a backup on-demand or schedule a time period;
3. Monitoring: the operator has a built-in Prometheus metrics entry point [25].

The figure 5.2 shows an overview of the infrastructure:
Currently, the main limitations are:

1. the database backup is implemented using mysqldump,
2. it doesn't support bootstrapping a new database from SQL script or from backup.[25]

Another important limitation regarding Kubernetes is that:

- the operator keeps the cluster healthy. It changes the role of the master to another MySQL instance and starts a new MySQL instance. This means that, before the problem was the failure of the node where the single instance of the MySQL pod was running, now the problem is the failure of the node where the operator is running.

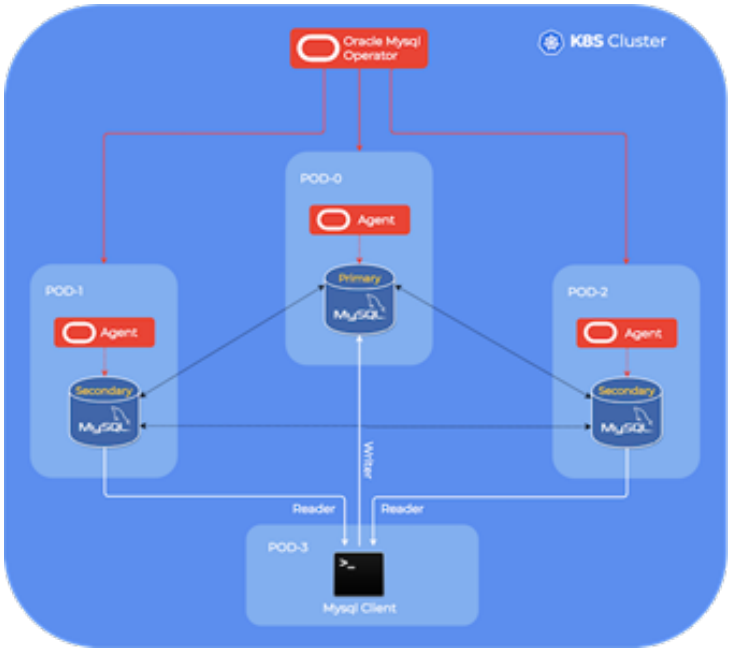


Figure 5.2: Infrastructure overview [25].

5.2.2 Bitnami Chart MySQL - Single-master

This chart bootstraps a MySQL replication cluster deployment on a Kubernetes cluster using the Helm package manager [26]. Compared to the previous solution, this solution does not have an "operator", but it does not implement a Master election process. So, in case of a failure of the master node, it is necessary to wait for its re-creation.

5.2.3 MySQL Group Replication – Single and Multi Master

MySQL Group Replication ensures a strong correlation between servers belonging to the same group. Also here, there are two modes of operation: (*i*) single-master with the automatic master election, in which only one server at a time accepts updates; (*ii*) multi-master mode, in which all servers can accept updates, even if released at the same time.

There are two important automatisms: (*i*) the view of the group is consistent and available for all servers at any given point in time. Both servers can exit and join the group and servers can leave the group unexpectedly, in which case the fault detection mechanism detects it and notifies the group that the group has changed. (*ii*) automatic protection mechanism from the split-brain event: for a transaction to commit, the majority of the group has to agree on the order of a

given transaction in the global sequence of transactions. Each server independently decides whether to commit or cancel a transaction, but every server must come to the same conclusion. The system cannot advance if there is a network split caused by members who are unable to come to an agreement, so long as this problem is not fixed [21].

Group Replication operates either in single-master mode or in multi-master mode. The group's mode is a group-wide configuration setting, specified by the *group_replication_single_master mode* system variable, which must be the same on all members. ON means single-master mode, which is the default mode, and OFF means multi-master mode [21].

Single-Master

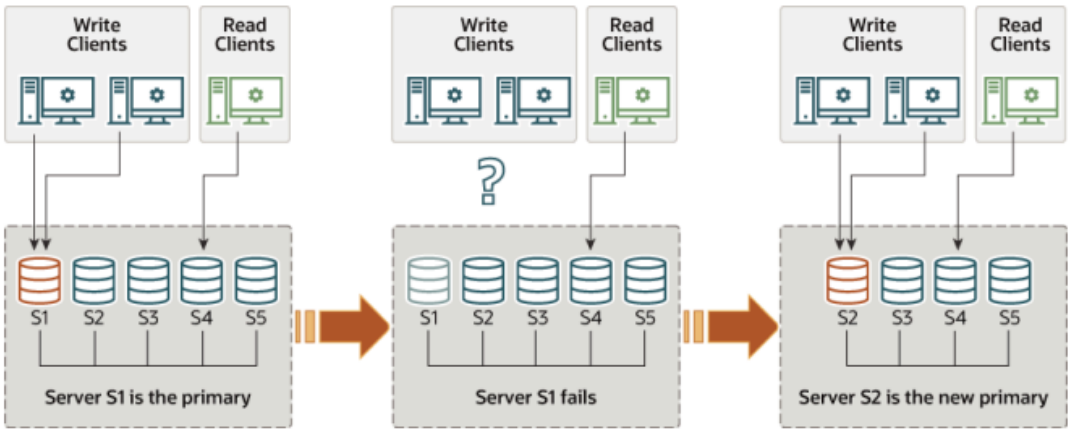


Figure 5.3: New master Election [21].

In single-master mode there is only one server configured in read-write mode and it is typically the first server to bootstrap the group. All the other members in the group are set to read-only mode (with *super_read_only variable* set ON). The member that is designated as the master server can change in the following ways:

- if the existing master leaves the group, whether voluntarily or unexpectedly, a new master is elected automatically, as shown in the figure 5.3;
- you can appoint a specific member as the new master using the *group_replication_set_as_master()* function [21];

This solution guarantees that the database service is continuously available. However, the usage of a connector, router, or load balancer, is fundamental to redirect the clients toward a different server in the group, when the group member

connected to the client becomes unavailable. This solution does not have an inbuilt method to do this.

In addition, for Kubernetes, the main limitation is that in case of failure of the Master node it is useful to hook the new master to the Service K3s.

Multi-Master

In multi-master mode no member has a special role. Any member can process write transactions, even if they are issued concurrently. Also in this case, if an unexpected server exit happens and a member stops accepting write transactions, clients connected to it can be redirected to any other member that is in read-write mode. Since this solution does not handle client-side failover, as before, you need an external framework, proxy or connector to solve this problem.

Another important implementing limitation is that multi-master mode groups do not support tables with multi-level foreign key dependencies. This is because foreign key constraints that result in cascading operations executed by a multi-master mode group can result in undetected conflicts and lead to inconsistent data across the members of the group [21]. The solution is changing the openPDC.sql.template architecture in the openpdc-init container.

5.3 Percona XtraDB Cluster

Percona XtraDB Cluster (PXC) is a fully open-source high-availability solution for MySQL. It integrates Percona Server for MySQL and Percona XtraBackup with the Galera library to enable synchronous multi-master replication [27].

The most prominent feature of PXC is to solve the replication delay problem, belonging to a master-slave architecture, and it is basically achieved by using real-time synchronization. Because synchronous replication applies any modifications to all sites involved in the replication environment as part of a single transaction, each node of the cluster will have the same set of synchronized data.

The main benefits of this solution are:

- prevents downtime and data loss;
- there is no need for remote access because when you execute a query, it is executed locally on the node;
- there is no central management. You can lose any node at any point in time, and the cluster will continue to function without any data loss;
- high availability;

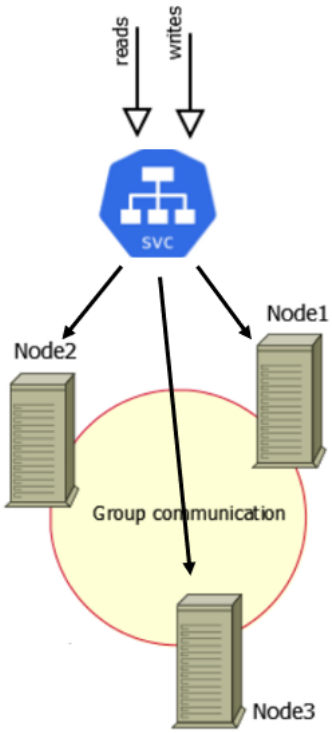


Figure 5.4: Percona XtraDB Cluster Architecture

- good solution for scaling both a write and a read workload. You can put read and write queries to any of the nodes [27];
- regarding Kubernetes, there is no longer the problem of the previous solutions. Since each node is equal, as shown in figure 5.4, it is enough to have a single K3s service to expose the entire cluster (specifically, the StatefulSet resource of the Cluster is exposed by the K3s Service). However, in this case, each client (PMUs in our case) establishes a TCP connection with one of the database replica, so when this node goes down we have to force the establishment of a new TCP connection towards another node (this code was realized into the liveness probe of the PDC pod).

Instead, the main drawbacks are:

- overhead of provisioning new node. A new node must replicate the entire data set from an existing node when you add it. If it is 100 GB, it copies 100 GB [27];
- any updated transaction needs to pass the global verification before it can be executed on other nodes;

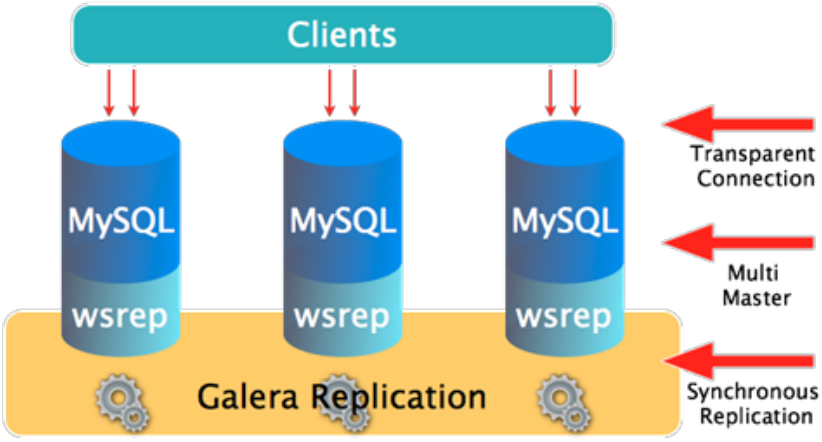


Figure 5.5: Galera with MySQL Architecture [28].

- because it relies on synchronous replication, additional bandwidth is needed.;
- you have several duplicates of data: for 3 nodes you have 3 duplicates [27];
- the cluster performance is limited by the worst node, the so-called short board effect.

Percona XtraDB Cluster is based on Percona Server for MySQL running with the XtraDB storage engine. Basically, Percona Server is a fork of MySQL e XtraDB is a fork of storage engine InnoDB.

This solution uses the Galera library, which is an implementation of the write set replication (wsrep) API. As shown in the figure 5.5, the wsrep API is an interface between the database server and its the replication provider (Galera in our case). Galera library provides a synchronous multi-master replication plug-in for InnoDB. An application can write to any node in a Galera cluster, and transaction commits are then applied on all servers, via a certification-based replication.

Certification-based replication executes transactions in a single node and, at commit time, runs a coordinated certification process to enforce global consistency. A broadcast service is used to establish global coordination. The fundamental principle is that up until the commit point, a transaction is carried out ordinarily. The primary keys of the altered rows and all database changes performed by the transaction are collected into a writeset at this time (but before the actual commit has taken place). After that, the remaining nodes receive a replication of this writeset. Then, all nodes execute a deterministic certification test, wherein they check whether the content of the writeset is applicable, using the collected primary keys. Naturally, if the certification test fails, the writeset is dropped and the original transaction is rolled back. If the test succeeds, the transaction is committed and the writeset is applied on each node [29].

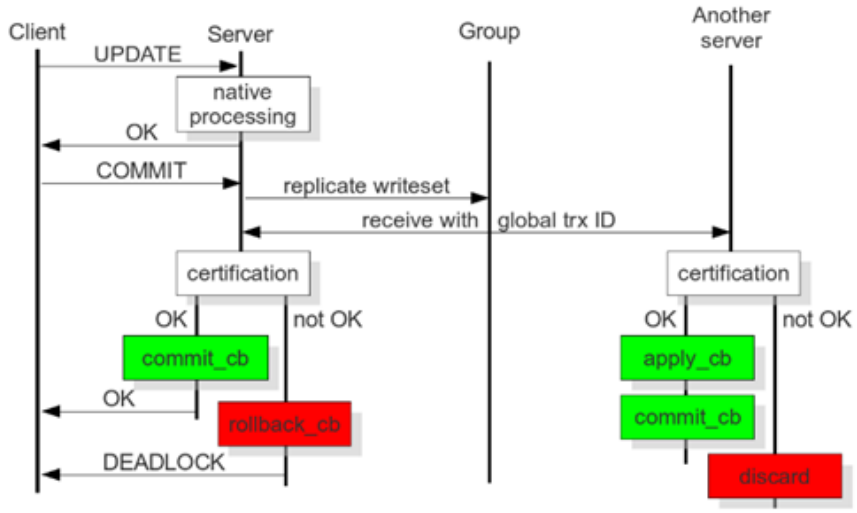


Figure 5.6: What is Certification based Replication? [29]

To sum up:

- Galera replication happens at transaction commit time, by broadcasting transaction write set to the cluster for applying and by executing a certification-based process;
- client connects directly to the DBMS and experiences close to native DBMS behavior;
- wsrep API (write set replication API), defines the interface between Galera replication and the DBMS.

Another main feature is the High Availability: if we have a setup with 2-3 nodes, PXC will continue to work, without any data loss, even if any of the nodes take down (due to a node crashing or if it becomes unreachable over the network) or if we shut down any node to perform maintenance.

When a new node joins a cluster or when the failed node restarts working, it will be automatically synchronized with the other nodes thanks to State Snapshot Transfer (SST). It refers to a full data copy from one cluster node (i.e., a donor) to the joining node (i.e., a joiner) [30]. Therefore, Percona XtraDB Cluster uses Percona XtraBackup for State Snapshot Transfer and the *wsrep_sst_method* variable is always set to xtrabackup-v2.

Additionally, Percona XtraBackup is based on InnoDB's crash-recovery functionality. This works because InnoDB maintains a file called transaction log. To recover from an unexpected MySQL server exit, InnoDB automatically checks the logs file and performs a roll-forward of the database to the present [21].

5.4 OpenEBS for resilient data persistency

Since Kubernetes relies on plugins and volume abstractions to isolate storage hardware from applications and services, it is by nature infrastructure agnostic. Containers, on the other hand, are transient and instantly delete all data when they end. Using volumes and persistent volumes, Kubernetes allows containerized applications to store and persist data on physical storage devices [31].

Because the goal is to have resilient and persistent data, OpenEBS is the solution adopted. This solution is highly suitable and compatible with Percona and is very simple to setup.

OpenEBS allows mapping the virtual disk of the MySQL Pod with the volume of the node where it is running with a 1:1 ratio.

The Figure 5.7 shows a high-level view of the current implementation of MySQL replicas within the Kubernetes cluster.

To connect applications directly with storage from a single node, OpenEBS leverages LocalPV provisioners. This storage object, known as LocalPV, is subject to the availability of the node on which it is mounted [32].

Local volumes seem not appropriate for all applications because they can only be accessed from a single node and are dependent on that node's availability. A local volume used by a pod won't be accessible if a node becomes unhealthy, making it impossible for the pod to run. Depending on the durability properties of the underlying disk, applications employing local volumes must be able to withstand this reduced availability as well as potential data loss [33]. Despite this, the main feature of OpenEBS is its High Availability. This means that the loss of any node results in the loss of only those volume replicas present on that node and that availability is provided at the application level, thanks to the presence of three replicated pods. When a node fails, the volume data can still be accessed at the same performance levels because it can be synchronously replicated at other nodes.

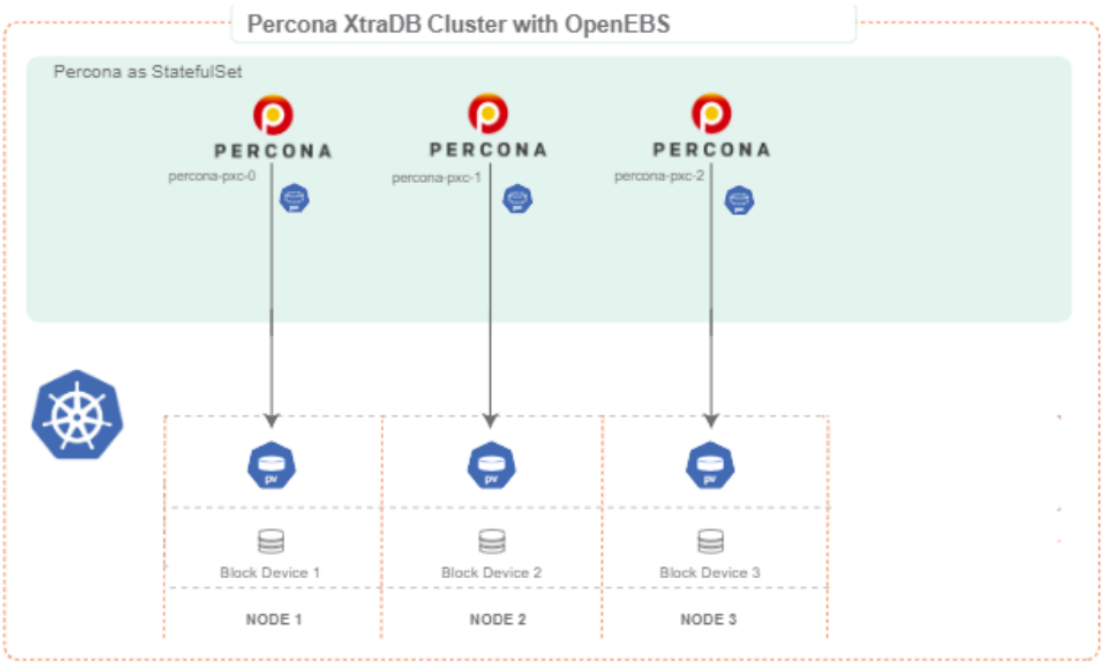


Figure 5.7: Deployment model [34].

Chapter 6

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 containers, if necessary.

6.1 Infrastructure

6.1.1 Orchestrator

The architecture is heavily based on Kubernetes, which includes implementations targeting also low-resource devices. It has a large software ecosystem that can give well-tested solutions for many common problems, such as data redundancy, in addition to its native features (e.g., automatic service restart/re-spawn in case of failure, multi-master capabilities, etc.). The Kubernetes distribution chosen for edge sites is K3s, which has a very low resource consumption (CPU, RAM, disk) and a very straightforward configuration. Orchestrator redundancy has been achieved with multiple masters (some active and others in standby). The idea is to put a K3s cluster in each station.

K3s has been configured to have master redundancy using the embedded etcd option. This allowed having 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. Focusing on a single station the realization of the cluster is based on four nodes – three masters and one slave – in order to guarantee the correct functioning of the cluster even in the face of a failure in any of the nodes with the consequent transition of its instances in the nodes properly running.

In addition, the default configuration has been modified to minimize the time between detecting a node failure and re-spawning the services that were previously operating on the failed node from 5 minutes (the default setting) to 40 seconds.

This was accomplished by changing the API server options:

- *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 a node failure, to around a minute after the failure.

6.2 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. Configuring the PDC connection to PMUs is done by the OpenPDC manager helper, which gives this possibility through a user interface although available only on Windows systems. The ability to visualize real-time graphs of measurements obtained by the PMUs connected to the openPDC Manager is a significant benefit.
- PMUsim is an open-source, C-based, IEEE C37.118-complaint PMU simulator tool, allowing to generate random synchrophasors. When you launch the PMUsim application, the main process forks and the newly created process starts the PMU-server, which is in charge of connecting to the PDC and generating random synchrophasors. The main process, on the other hand, is responsible for drawing the GUI and connecting with the PMU-server process, via signals, in order to apply the configurations made by the user via the GUI.
- MySQL is a well known DBMS, which is needed since OpenPDC can store its configuration in DBs. In this case it has been used to store data (measurements) and it is needed by OpenPDC software to read and store its configuration. In our scenario three instances of MySQL database, configured in multi-master mode, are run in every cluster so that PDCs can easily reach their configuration and so that we have a database cluster capable of tolerate node fault.

6.2.1 Percona XtraDB Cluster and OpenEBS

The data persistence problem is addressed by Kubernetes with built-in abstractions such as PersistentVolumes and PersistentVolumeClaims; however, the default driver provides basic data persistence but does not provide data replication. Consequently, the proposed solution introduces an additional layer of abstraction for data persistence leveraging the enhanced storage features provided by Percona and OpenEBS.

The Percona XtraDB Cluster (PXC) is a MySQL high-availability solution that is fully open source. It combines Percona Server for MySQL and Percona XtraBackup with the Galera library to enable synchronous multi-master replication, which ensures data synchronization between multiple replicated MySQL nodes. A cluster is made up of nodes, each of which contains the same set of data that is synced across nodes.

Percona XtraDB Cluster can be configured with OpenEBS volumes via the OpenEBS Local PV storage engine. OpenEBS is the leading Open Source implementation of the Container Attached Storage(CAS) pattern [35]. OpenEBS helps to deploy Kubernetes Stateful Workloads that require fast and highly durable, reliable, and scalable Container Attached Storage [34]. In addition, it permits attaching each instance volume to a distinct physical data volume created on the node itself and the OpenEBS instances are coordinated to guarantee that any data is replicated on different physical volumes (hence nodes).

6.3 Results from the current implementation

This paragraph will offer an evaluation of the proposed solution.

Only Linux-based operating systems have been taken into consideration because the workloads are designed to run in containerized environments. Every measurement has been conducted using Ubuntu 20.04 as the base OS to maintain consistency and allow a fair comparison. The x86 architecture is being taken into consideration, and table 6.1 contains more details about the used machine.

A sophisticated container orchestration solution like Kubernetes is not just a tiny piece of software; orchestration requires constant communication with the container runtime backend, monitoring of active resources, and either executing or interacting with the control plane. The sysstat tool, which automatically collects data using Linux primitives, has been used to collect metrics for CPU and memory usage. Every test scenario has been set up in a setting that is similar to reality, with PDCs linked to remote PMUs providing output streams and also storing received streams into the databases.

Values for CPU use indicate the amount of time the CPU is not idle. The values taken into account are the average values across all CPUs at a specific

Architecture	x86 (64-bit)
Machine	VM
Linux kernel	5.4.0-122-generic
CPU Model	Intel Core i7-6700
CPU Cores	4
CPU Frequency	3.4 GHz
Memory size	6 GB
Disk size	40 GB (SSD)
OS	Ubuntu 20.04.4 LTS

Table 6.1: Specifications of the machine used to carry out the tests

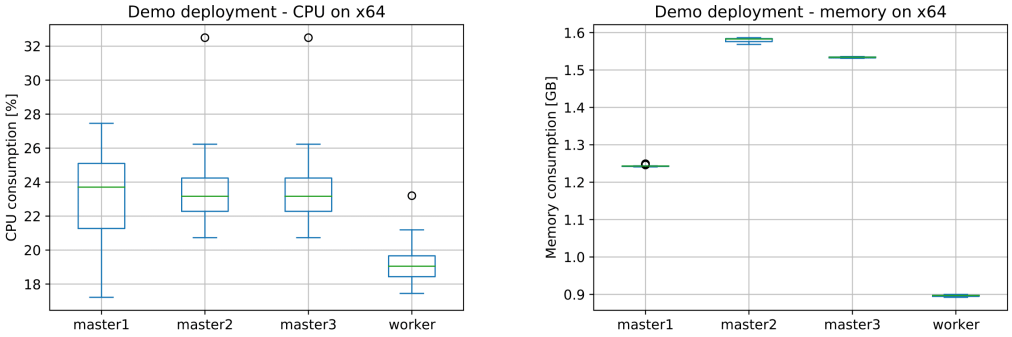


Figure 6.1: CPU and Memory Usage of the demo deployed in a 4 node cluster (x64)

time, and they are displayed as box plots to highlight the values with the highest frequency and separate them from outliers. Memory usage values simply represent the non-free memory at a given time.

A general evaluation of the presented demo is shown in the Figure 6.1. In all the nodes we have the system pods both the ones related to OpenEBS and to the Kubernetes system. Resource usage clearly shows a higher usage in master node 1 (approximately 17-27%), where the low-level PDC is running. This is because, as mentioned above, the PDC is responsible for receiving, processing, analyzing, and saving the received data stream. Therefore, in this case, an important notice is that the node running the low-level PDC shows a consumption similar to the other nodes, in which, instead, we find the OpenEBS pods needed for the High-Availability of the database (in this case the analysis was conducted using a virtual machine with Intel i7-6700 CPU model).

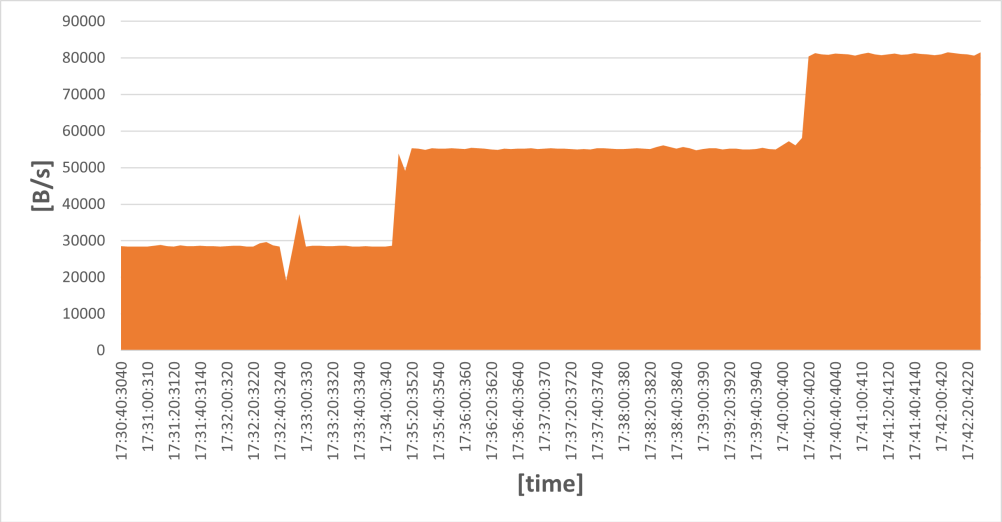


Figure 6.2: Increasing the number of connected PMUs, the number of bytes written increases

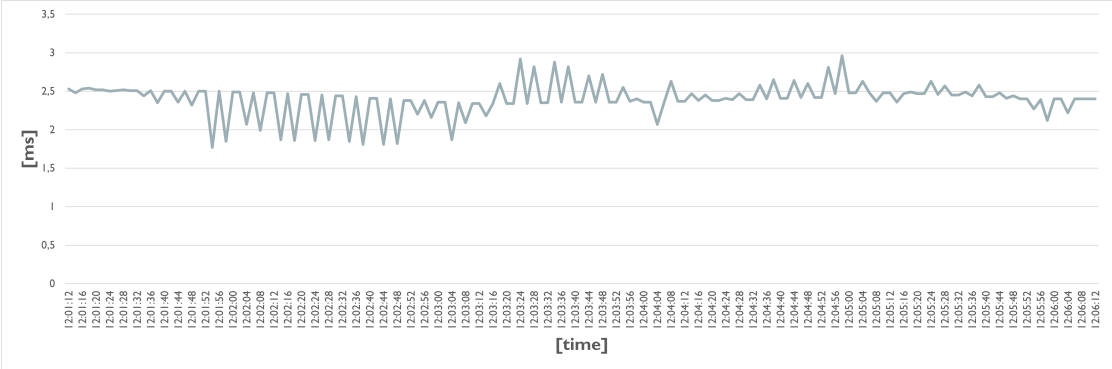


Figure 6.3: Galera replication latency.

Another important thing to note is that replication of MySQL has been scheduled in the worker node and as a result, this shows high CPU consumption (approximately 18%). Finally, the scheduling of MySQL pods in master2 and master3 nodes also results in increased memory consumption.

6.3.1 Results about PXC Implementation

This section shows the results about the High-Availability of the database. They regard both the number of bytes written on MySQL pods and the latency needed for the synchronization of nodes, considering also that the analysis was carried out on a real K3s cluster. In addition, there is an analysis of how the system reacts to a

failure both on the node in which the system is writing, and on a replication node.

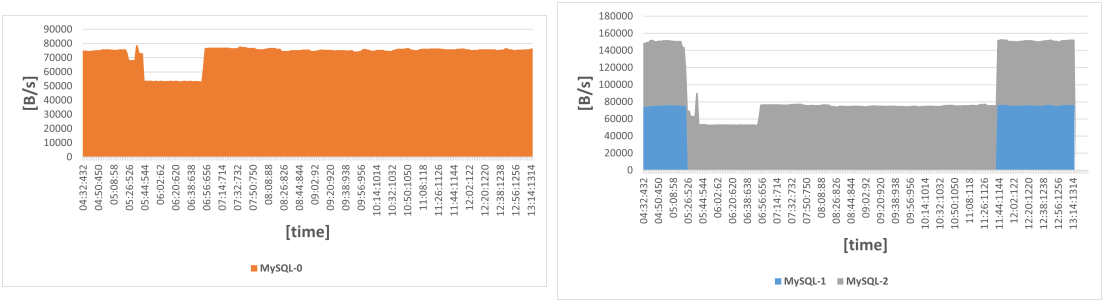
Percona Monitoring and Management was used to take measurements (PMM). Oracle MySQL Community Edition, Oracle MySQL Enterprise Edition, and MariaDB are just a few of the MySQL varieties that Percona Monitoring and Management delivers actionable performance statistics for. The InnoDB and XtraDB storage engines are tracked by PMM, which also offers specialized dashboards for specific engine information. [36].

The figure 6.2 shows the number of bytes written per second on a MySQL node as the number of PMUs connected to the single PDC increases. The results show that one PMU writes about 30kB/s of data; while three PMUs connected to the same PDC generate about 80kB/s of data. It is important to note that the trend is not perfectly linear because the PDC performs data aggregation operations trying to optimize the writing of the data stream received. However, increasing the number of PMUs connected, also the incoming traffic increases and consequently the replicated traffic into other nodes.

The figure 6.3 shows the replication latency on group communication. It measures latency from the time point when a writeset is sent out to the time point when am ACK is received. Since replication is a group operation, latency essentially is given by the slowest ACK and longest RTT in the cluster [37]. As can be seen from the graph, the average latency for data replication is about 2.5 ms.

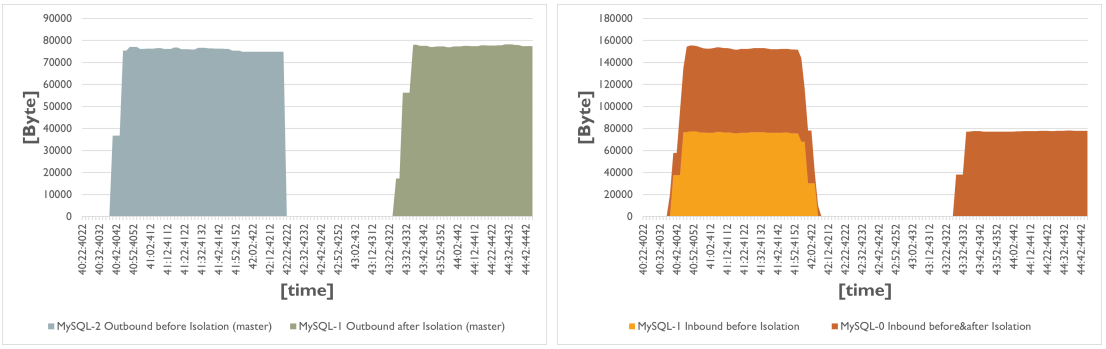
The figures 6.4a and 6.4b show what happens when a "replica pod" fails. In the beginning, the application establishes a stateful TCP connection with one of the three nodes and will write always and only on this, consequently "replica pod" stands for one of the other replicas where no write operations take place. So, in this case, the TCP connection was established with the pod MySQL-0 (fig. 6.4a) and the other two pods receiving the replicated data from it. When a replica pod fails (in this case the replica MySQL-1 fails), the data are still written into the remaining nodes (fig. 6.4b). One important thing to note is that in the failed node there was also a PMU scheduled and for this reason, we can note a slight deflection.

Instead, in the scenario represented in fig. 6.5, the node running the MySQL pod, with which a TCP connection was previously opened, is analyzed. As mentioned above, the application establishes a stateful TCP connection with one of the three nodes and will write always and only on this. So “writing pod" stands for the pod where writing operations take place. Therefore, in this scenario, the "writing pod" is isolated. As shown in figure 6.5a, there is a time interval when there is no traffic because as soon as the node where you are writing fails, a «pending» connection remains open to this pod and the only way to force the application to write to another pod is to launch the command Initialize in the OpenPDC GUI or to write a script that periodically launches this command. In the current implementation, instead, there is a liveness probe in the PDC code that periodically checks how many nodes are running and if one of the nodes fails, as a result, the liveness probe



(a) Incoming Traffic **(b)** Replicated Traffic

Figure 6.4: What happens when a replica pod fails?



(a) Incoming Traffic **(b)** Replicated Traffic

Figure 6.5: What happens when a writing pod fails?

fails and the pod is restarted. If the liveness probe fails, it means that the PDC can reach a lower number of nodes and, therefore, as a result, that network partitioning has occurred.

According to figure 6.5a, We can deduce that the write operations are first carried out in the node called MySQL-2 and then, after performing the liveness probe, a TCP connection with the node called MySQL-1 is restored. From the figure 6.5b, instead, we can deduce that, after the failure, the operations of replication happen only in the node properly in execution.

Chapter 7

Orchestrator reaction times

In distributed systems and in an electrical context the resilience and self-healing of the adopted solution are key features. In addition, in order to ensure system availability and reliability, another important aspect is the ability to react quickly in the face of a fault.

In order to create resilient and self-healing applications, Kubernetes provides, through the ReplicaSet resource, the possibility to create N replicas of the same service. In this way, we would be able to have a resilient system able to tolerate n<N failures. It might be interesting to guarantee the above-mentioned reliability, simply by creating different replicas of the same service; but doing that, we will have implementation problems because:

1. you can not use multiple instances of PMU since they should share the same device and, therefore, compete for the same hardware;

2. Kubernetes is not suitable to replicate the PMU data stream in different PDCs (a PMU sends to one and only one PDC);

3. having replicated PDC would mean that the data stream received by the different PMUs, must then be aggregated before further processing.

Therefore, in our case the possibility of creating replicas does not provide benefits, and as a result, scenarios are evaluated in which we only have a single replicated service.

Further analysis are evaluated against two possible failures:

1. container restart after an unexpected failure. Specifically, the fault was simulated by forcibly sending a "kill" signal inside the container, thus killing the process delegated to the exchange of synchrophasor. In addition, a liveness probe inserted into the PMU pods guarantees to periodically verify its

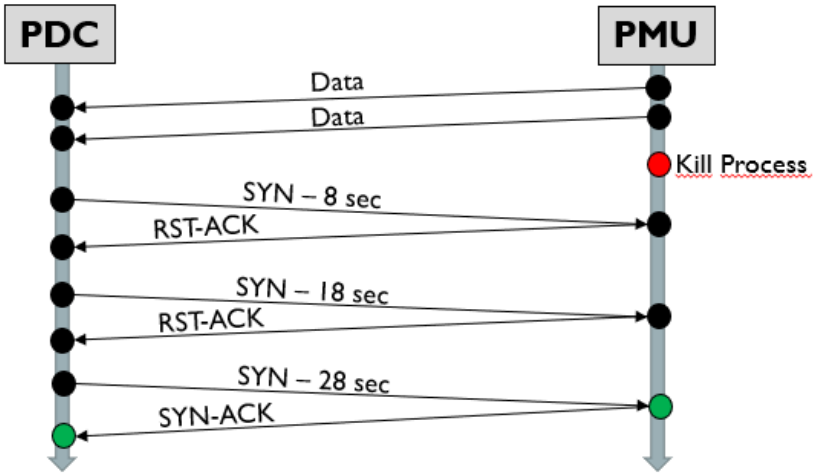


Figure 7.1: PDC Behavior when PMU no longer sends data.

operability and, in the event of failure, the Orchestrator becomes responsible for rescheduling the container;

2. inaccessibility of a node, due to network partitioning, and consequent rewriting of pods into a healthy node.

Three shell were mainly used:

1. the first runs a script for the command `tcpdump`. It is useful for measuring the time required by PMU and PDC to restore their communication, and monitoring the actual exchange of network packets between components.

2. the second executes the kill of the process, thus simulating the failure of the Pod.

3. in the third there is a script to analyze at any moment of time the current state of the Pods, according to Kubernetes.

tcpdump output showed that:

1. even if in Running state for Kubernetes, the PMU waits for a request to open a TCP connection from the PDC and starts sending data after the three-way-handshake;

2. after the connection fails, simulated through the `kill` process, the PDC attempts to establish a TCP connection with the PMUs, sending SYN packets at regular 10 seconds intervals, as demonstrated in figure 7.1, and as soon as the PMU returns up, a new TCP connection will be established..

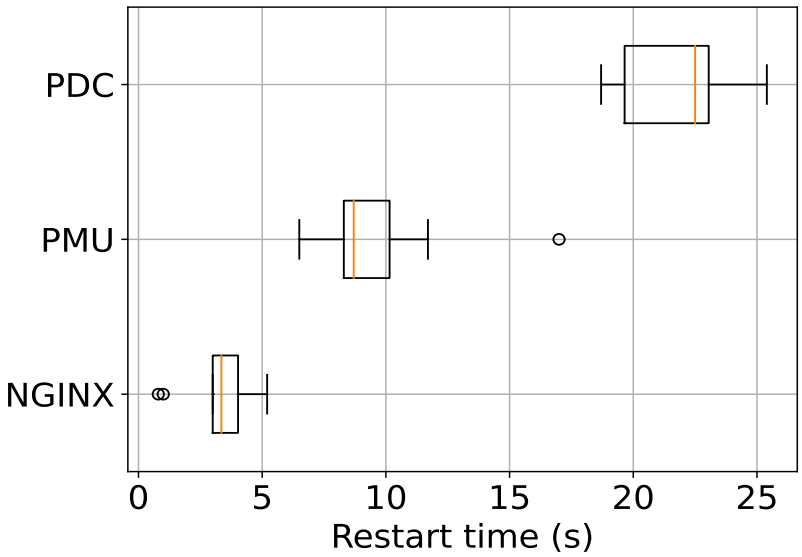


Figure 7.2: Data flow restart time interval in case of Nginx, PMUs, and PDCs.

In the Figure 7.2 there is an analysis of the orchestrator reaction time comparing the timing of recreation of a `nginx` pod (a cloud-native application) with the timing of the services of `PMU` and `PDC` (non cloud-native applications).

From this figure, we can see that Nginx, cloud-native application, has a restart time that hardly exceeds 5s. Instead, PMU and PDC, non-cloud-native applications, have a reboot time that is between $6-12$ seconds for the PMU and $18-25$ seconds for the PDC.

Some considerations on the PMU are: (*i*) as long as the PMU are in a state of "Terminating" according to Kubernetes, they continue to send data; (*ii*) if the PMUs return to a state of "Running", again according to Kubernetes, it does not mean that they are sending data. They start sending data after establishing a TCP connection with the low-level PDC, as soon as it receives a SYN package from it.

The proposed results raise some additional considerations: (*i*) in our setup, the Kubernetes control plane checks the status of the specific service every $5s$ (e.g., healthy, unhealthy). The orchestrator's contribution to the overall restart time can therefore never, in the worst case, exceed $5s$ and may even be further decreased by configuration. The remaining time is thus related to the service control logic to re-instantiate the communication and can be reduced only with proper code refactoring. (*ii*) currently, there is no automated recovery process in place in the event of a monitoring service failure; instead, manual intervention is still frequently needed. This implies that monitoring services have different resiliency requirements, compared to control services, and can withstand longer service disruption (e.g., minutes), without compromising the smart-grid operability.

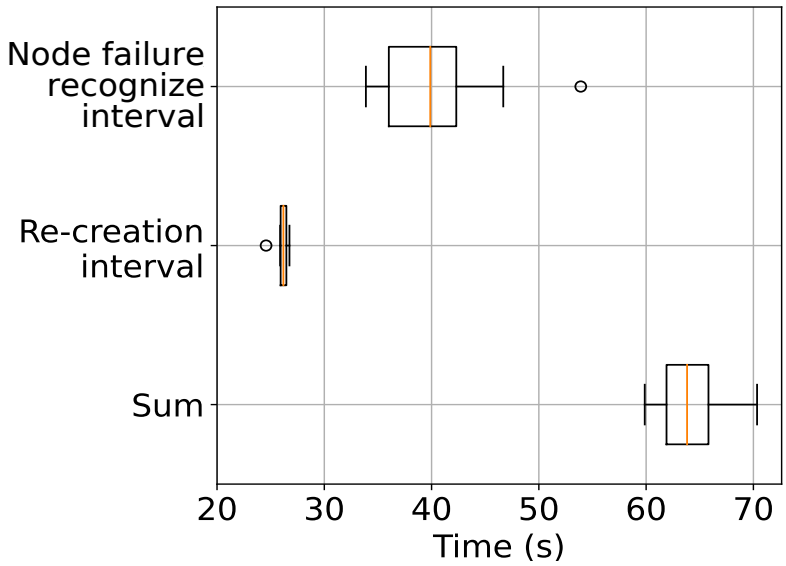


Figure 7.3: Time required to recover services on a disconnected node.

The second scenario is about the simulation of the unreachability of one node. A case of network partitioning has been induced, pushing firewall rules in `iptables` to isolate the node from the rest of the infrastructure. In this context, it has been reduced to 20 seconds (rather than the default 5 minutes) the maximum time that a pod can remain scheduled in a node in a NotReady state; and reduced to 20 sec the maximum time that a pod can remain in a node in an Unreachable state. This means that nodes are periodically interrogated to determine their reachability and possibly mark them as Unreachable.

So, the time between the detection of a failed node and the rescheduling of the services present in this node was measured. In the Figure 7.3 we can see the reaction times that come into play in this scenario:

1. how long the master node takes to recognize a failed node and set its status as NotReady/Unreachable;

2. time to re-create all the containers hosted in the failed node and to restore the application data flow;

3. total time to restore service on running nodes.

Specifically, the time required to identify a node failure strongly depends on Kubernetes control logic and experiences substantial variability, depending on the moment of the failure and the next node health check. It is also important to note that the calculation of the time needed to restore the actual operation of the cluster depends strictly on what was previously scheduled in the failed node. Indeed, the

overall re-creation interval is strictly constrained to the slowest service (i.e., PDC). Still, even in the worst case, the proposed infrastructure can recover to node failure event within $70s$, well below the requirements.

Chapter 8

Scalability and Resiliency Evaluation

This chapter provides an explanation of the simulator used and the results obtained. There are several reasons for using a simulator, such as making tasks easier and saving significant amount of development time. The simulator has been used mainly to deepen the analysis of two scenarios: (*i*) show CPU consumption of the low-level PDC; (*ii*) analyze the resilience of the infrastructure, injecting failure at run-time.

8.1 What is CloudSim?

CloudSim Plus is a Java 8 simulation framework that allows modelling and simulation of different cloud computing services [38]. The framework allows developers to specify the characteristics of different entities of a cloud provider such as: (*i*) physical resources like datacenters, physical machines (hosts or servers) and network assets; (*ii*) logical resources like storage area networks (SANs), network topologies and applications; (*iii*) the virtualization layer that provides elements such as virtual machines (VMs) to enable virtualizing physical and logical resources; (*iv*) requirements and behaviour of applications and workloads.

It automates the management of all these resources that are provided as a service, working as virtual machine monitors (hypervisors or simply VMMs) that perform low level administration tasks such as: (*i*) VM lifecycle management (like creation, start, stop, destruction, placement and migration); (*ii*) management of active physical machines for energy saving; (*iii*) scheduling of VMs execution inside PMs and applications execution inside VMs; (*iv*) allocation of VMs for application and management of application lifecycle inside VMs [38].

The main entities to define on are Cloudsim: Host, VM/Container and Cloudlet. Hosts emulate a physical machine and for each of them it is necessary to establish:

1. CPU's cores;
2. MIPS CPU (Million Instruction per Second), that is the CPU clock speed;
3. RAM;
4. Storage;
5. Availabe Bandwidth;

The VM represents the virtual machine (or we can assume the container) running inside a Host and has the main task of running applications (called Cloudlets). Also for the VM it is necessary to establish the same features of the Host.

Finally, Cloudlets represent applications running within the VM. Each of them is defined in terms of computational resources occupied within the VM in which it is running. As a result, the cloudlet also needs to define the above. In our scenario, Cloudlets are the PMU and PDC processes that exchange data. Specifically, in the exchange is also implemented a double level of aggregation between a low level PDC and a high level PDC.

Other inputs depend, instead, from the analyzed scenario:

1. in a scenario where we want to analyze CPU consumption as the number of PMUs connected to the PDC increases, it is necessary to establish:
   (a) the number of PMUs connected to each low level PDC,
   (b) the number of Clusters. A CLUSTER consists of an aggregation of several secondary station (called Host in CloudSim),
   (c) the number of cores to be assigned to each Host.
2. when we want to analyze the failure of a Host (the physical machine) it is necessary to establish
   (a) the number of expected failures at each instant of time;
   (b) the instants of time when failures occur.

Instead for the outputs we could have:

1. an output that shows the impact that the scheduling of a PDC has on a Host, in terms of CPU consumption and bandwidth;
2. in the scenario of host failure, an output showing availability. "Availability" means the number of hosts available to run the PDC at the time of migration.

In summary, both outputs and inputs depend strictly on the scenario analyzed. The only fixed values are given by the architecture skeleton and therefore by the computational resources available in each entity.

8.2 Results

8.2.1 CPU and Bandwidth Analysis

The goal of this analysis is to show the trend of CPU consumption on the station where the low-level PDC is scheduled, as the number of connected PMUs increases. This analysis allows us to understand how the computational resources of the PDC code are able to saturate the resources available in the host machine, so it permits us to quantify the degree of saturation of computing resources.

The code includes the placement of a PMU in each station, a PDC in only one of the other secondary stations and, in addition, allows you to change the number of PMUs connected to each PDC. As a first model, the distribution system stations are represented aggregating four Raspberry Pi4B with the following computational capabilities: (*i*) CPU with 4 cores of 1.5GHz (Cortex A72), (*ii*) 4GB of RAM, (*iii*) 256GB of disk.

The code also allows to define more CLUSTERS – a CLUSTER is the set of more *secondary stations* (called Host in CloudSim) characterized by the same computational capacity – in order to analyze several different scenarios at the same time. The code allows to define multiple CLUSTERS because the computational capabilities of the Raspberry, in which the PDC is scheduled, will always be lower than the four total cores. This happens because likely in the same Raspberry are scheduled other system Pods (Kubernetes control plane, OpenEBS, MySQL and so on).

In the Figure 8.1, which shows the infrastructure designed with CloudSim, we can see an example with two clusters, in which the Raspberry are characterized by a different number of available cores, respectively two and three cores. Additionally, the router is the aggregation of all the possible network components placed between several secondary stations.

The study started with the evaluation of the resource consumption requested by PDC and PMU in a real K3s cluster and then it was expanded by using the simulator.

Figure 8.2 estimates the trend of CPU consumption (y-axis) as the number of PMUs connected to the PDC increases (x-axis), based on the number of cores that the PDC code can use, respectively one, two or three cores. This analysis allows us to quantify the degree of saturation of computing resources.

In addition, the study was conducted in two phases: in the first one, we considered the PDC code monolithic and not parallelizable; instead, in the second phase we assumed that the code will use multiple core, so as to carry out a preliminary analysis in the event that in the future the PDC code becomes parallelizable.

From the figure 8.3 we can deduce that when the CPU reaches the maximum of 100%, bandwidth consumption begins to degrade, due to the degradation of CPU

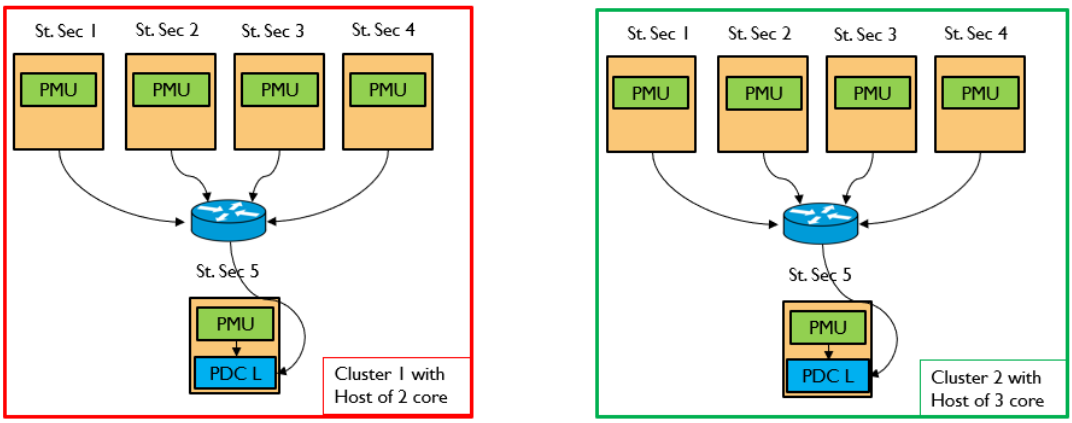


Figure 8.1: Infrastructure designed with CloudSim

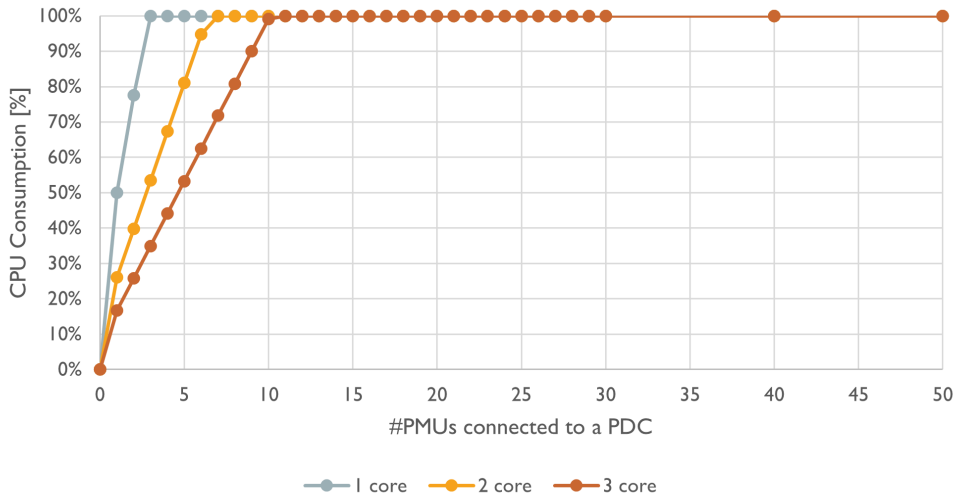


Figure 8.2: CPU Consumption

performance and, therefore, to a smaller number of packets analyzed. Specifically, it would seem that the low utilization of the bandwidth is an advantage; however, it is observed that the ideal trend is considerably different from the real one because of a degradation of the performances of the CPU, and consequently the used bandwidth decreases.

However, the bandwidth does not represent a limit since the traffic exchanged between a PMU and a low-level PDC is 25 packets of 36 Bytes and, moreover, the traffic exchanged between the low-level and high-level PDC is always a few hundred bytes. In the simulation bandwidth consumption is given by the following

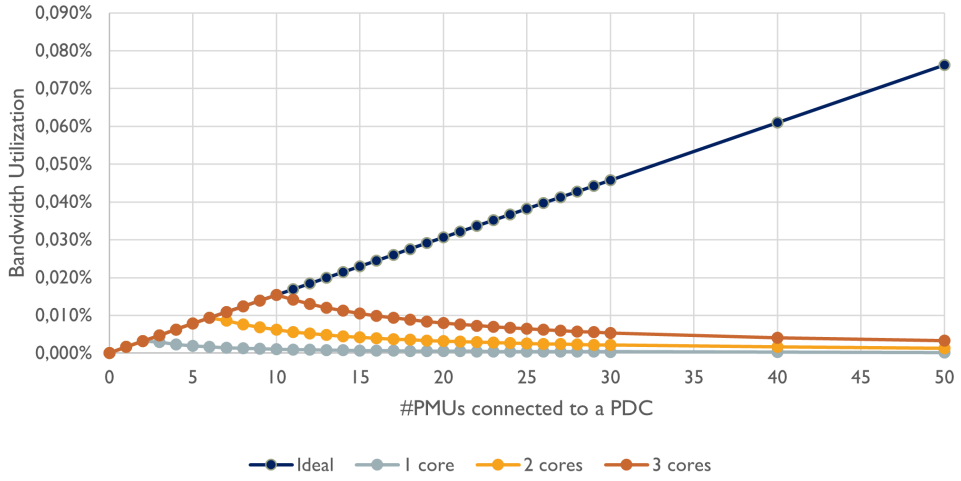


Figure 8.3: Bandwidth Consumption

formula: $(x-1)\cdot y + x \cdot delta$, where x is equal to the number of PMUs connected to the PDC; y is the traffic exchanged between a PMU and a low level PDC, which is equal to $(25 packets \cdot 36 Bytes)$; finally, *delta* is given by the increased traffic exchanged between the low-level PDC and the high-level PDC.

From these results we can deduce, for example, that, supposing that the code of the PDC is parallelizable and that it uses 2 cores of the Raspberry, we can connect up to a maximum of 6 PMU and in this case the consumption of the two cores would be about 95%. Another example could be that using only one core we can connect up to a maximum of 2 PMUs.

Latency due to Computing

Figure 8.4 shows how the latency, required for processing a packet, depends on the number of cores used. Also in this graph, as in the previous one, the number of PMUs connected to the PDC is represented on the x-axis, while the y-axis shows the latency due to the computational resources. It is important to note that if we had enough resources available each packet should be processed without delays due to computing; since the available resources are lower than the required, we have a higher processing time for each packet. For this reason, once the limit value is exceeded, there is a processing delay due to CPU overhead. In this case, the latency is identified as the difference between the time actually taken to process a packet (given by the output of the simulator) and the time theoretically necessary. This figure, also, shows several lines that refer to cases with different core numbers available. As is evident from the figure, by increasing the number of cores, the processing latency decreases at the same PMUs connected.

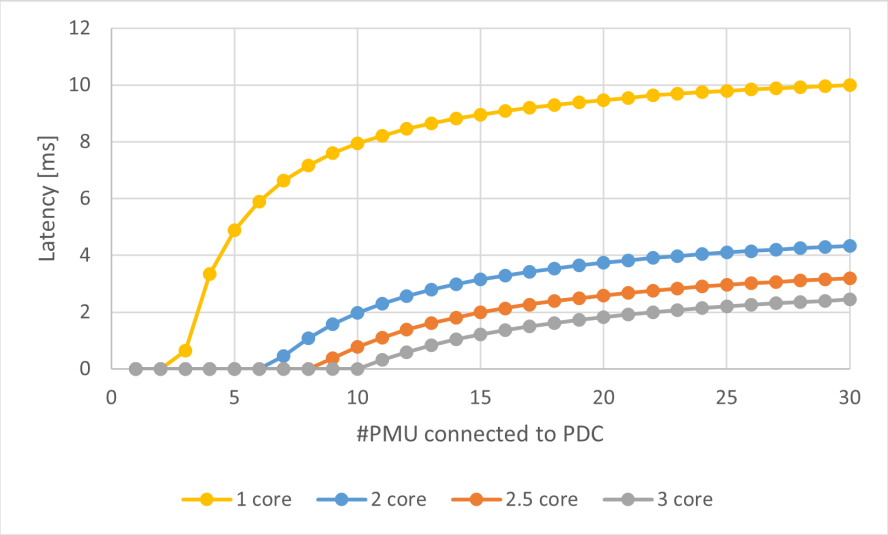


Figure 8.4: Latency due to Computing - Single Core vs Multi Core

What CPU do we need?

So far, it has been identified the limit value of PMUs connected to PDC to allow the system to function properly, by using a Raspberry. Then, instead, it was analyzed what type of CPU (specifically how many Cores and MIPS) is necessary for the proper functioning of the system for a value higher than the limit value. As mentioned above, not all available cores are used by the PDC, but we must assume that in the Raspberry in which the PDC is scheduled there are also other system pods.

The figures 8.5, 8.6, and 8.7 show the number of PMUs connected to the PDC on the x-axis, while the y-axis shows the MIPS value necessary to avoid a degradation of communications. Figure 8.5 shows some examples of CPUs that could be used as the number of PMUs connected to the PDC varies, in the single-core case. It is clear, from the figure, how, using a Raspberry, we can connect up to a maximum of two PMUs; while if we want to connect more than 9 PMUs it is necessary to have a CPU that has a single core clock speed higher than 4.10GHz. The figure 8.6, instead, shows that, by using the same CPUs, whether the PDC code could exploit two cores, we could connect up to a maximum of 19 PMU. Finally, the figure 8.7 shows that if the PDC code is capable of leveraging three cores, we can link up to a maximum of 30 PMUs for each PDC.

So, as is evident from the previous figures, increasing the number of MIPS required by the PDC, it is necessary to have more and more powerful CPUs or, whether the PDC's code is parallelizable, to use more cores of the CPU in parallel.

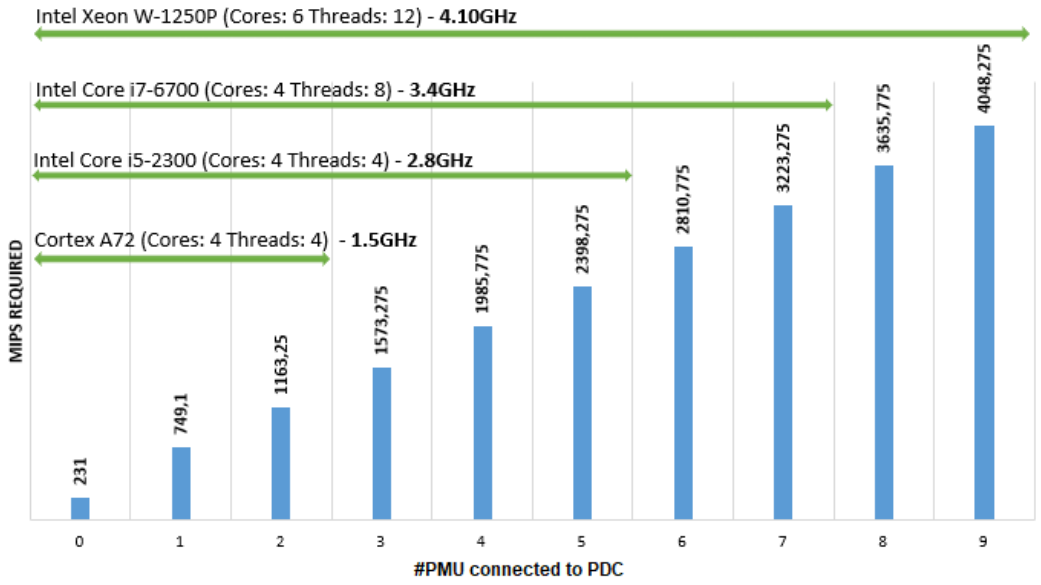


Figure 8.5: MIPS required by PDC vs MIPS available on CPU - Single core

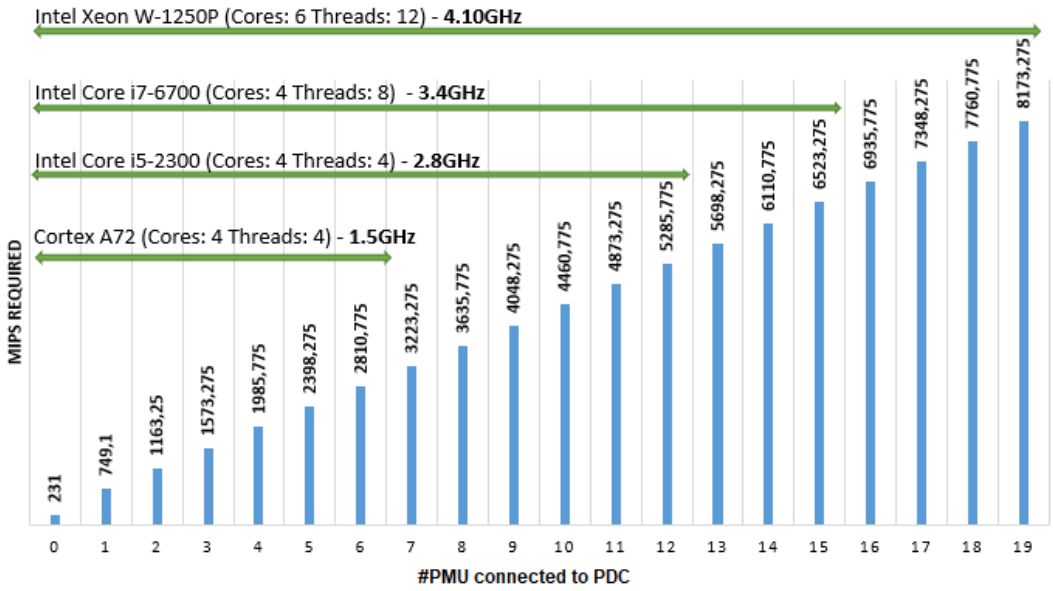


Figure 8.6: MIPS required by PDC vs MIPS available on CPU - Dual core

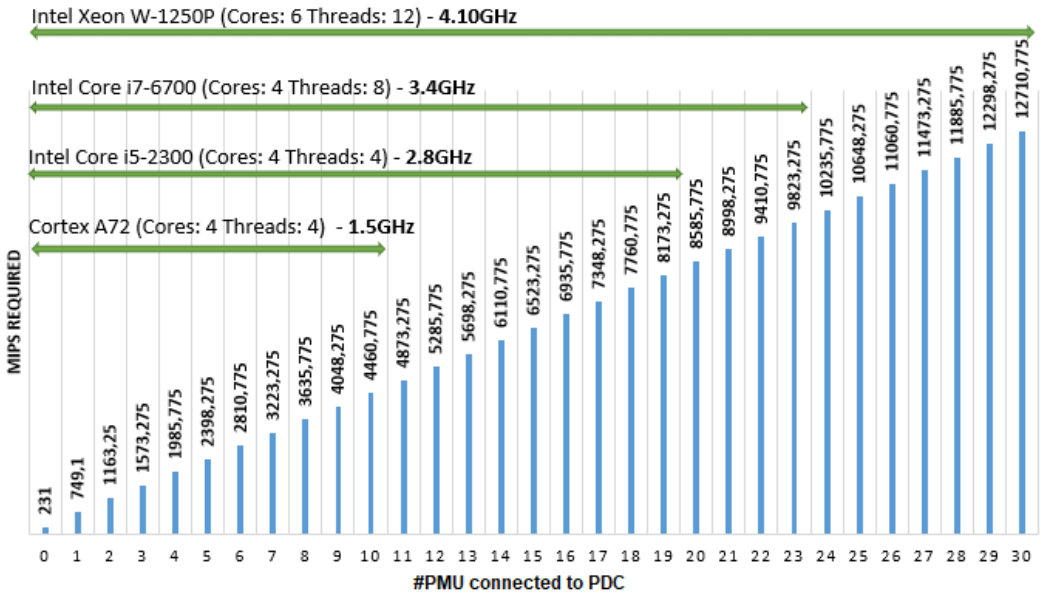


Figure 8.7: MIPS required by PDC vs MIPS available on CPU - Three cores

8.2.2 Host Failure

This second scenario was implemented by simulating the failure of Raspberry at run-time in order to analyze the resilience of the infrastructure. Failures were injected following the modeling in figure 8.8. In the analysis, this modeling seemed the most realistic in terms of possible failures over time. For this reason, it has an exponential trend and shows cumulative failures (y-axis) as the months (x-axis) increase. It is also possible to note that, going forward in time, failures become more and more frequent.

It is important to highlight that failures are injected randomly. This means that, for example, the first 4 failures can affect the 4 Raspberry of the same station or can be distributed between Raspberry belonging to different stations. Based on this randomness the migration of the PDC can take place at different times.

Specifically, the focus has been placed on the station where the PDC is running (the PDC is the only element in the simulator that can be migrated). So far we have seen that a higher number of PMUs connected to the PDC causes a high computation load on the PDC itself. Regarding failures, however, a larger number of connected PMUs ensures a greater availability of Station, where possibly migrate the PDC. So, from a consumption point of view, it would be better to have fewer PMUs connected to the PDC; from an availability point of view it would be better to have more PMUs connected to the PDC (since in this way we would have more Stations available, which use fewer resources, where to migrate the PDC).

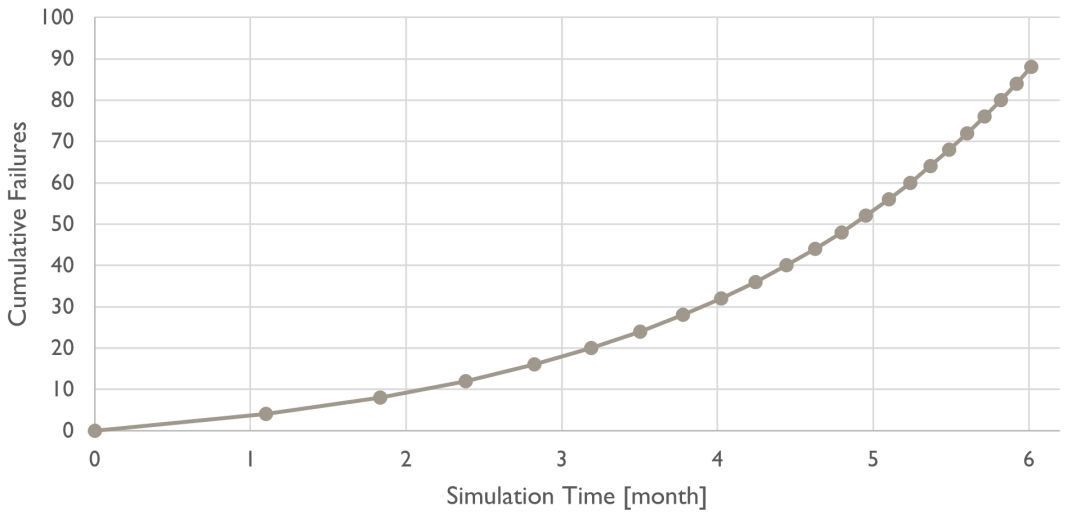


Figure 8.8: Cumulative Failures over Time

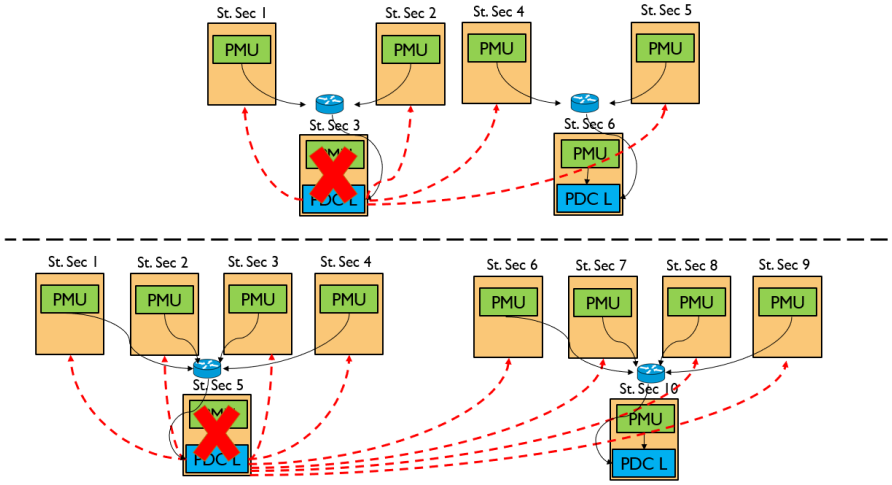


Figure 8.9: Station Available on CloudSim

For example, as shown in the fig. 8.9, supposing you have two clusters with a PMU:PDC ratio of 3:1, in case of failure of the PDC, we would have four more Stations where to migrate the PDC (red arrows); if instead, we assume to have a PMU ratio 5:1 PDC, we would have 8 Stations available where to migrate the PDC.

When the resources available in the station are less than those necessary for

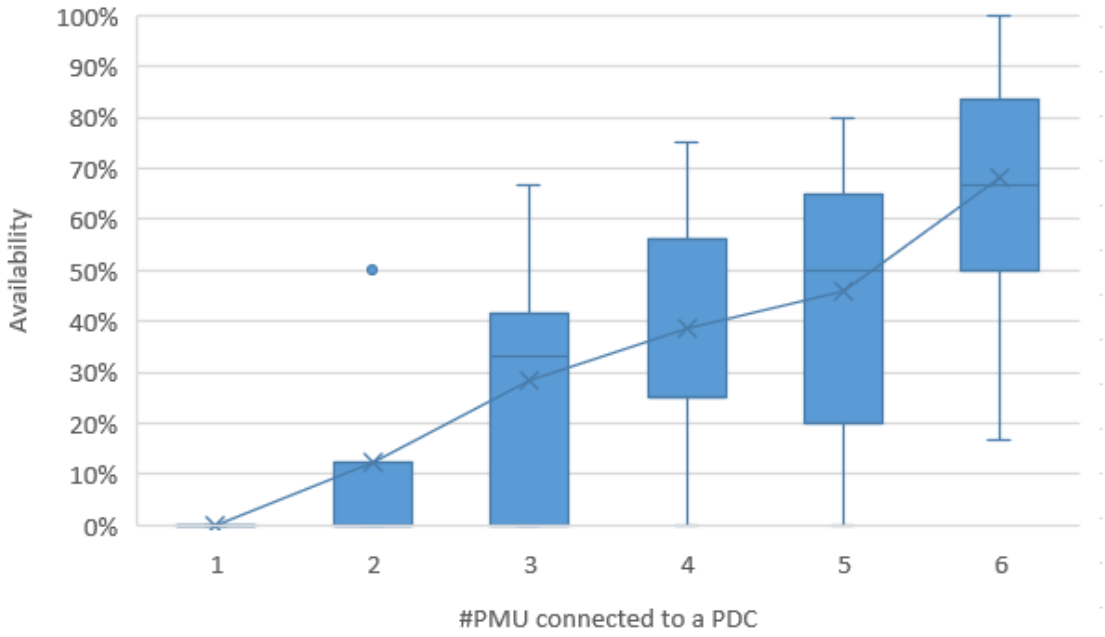


Figure 8.10: Availability of Stations

the proper functioning of the system, the PDC must migrate and the simulator output shows *Host availability*, which defines how many Stations still have enough resources to run the PDC. Availability (y-axis), in percentage, is calculated using the below formula:

$$\frac{(number\_hosts\_available\_to\_execute\_pdc)}{(number\_hosts\_total)} * 100$$

The Figure 8.10 shows how the availability increases, in the case of a possible migration, as the number of PMUs connected to the PDC increases.

For example, looking at the Figure 8.10, with 3 PMU connected to the PDC having an average availability of 30% means that, when a failure of the station where the PDC is running occurs, there are still on average 30% of stations available where the PDC can migrate (because they still have sufficient resources available). Since failures are injected randomly at run-time, in order to obtain almost reliable results the code has been executed 20 times for each number of PMU connected and then the results are shown through box-plots.

In any case, the goal is to show that connecting multiple PMUs to a PDC increases availability in the event of a failure (as opposed to increased resource consumption).

Chapter 9

Conclusions and future work

The work of this thesis analyzed the resiliency needed in the aforementioned infrastructure in the electric scenario. Initially, the analysis focused on the resilience of single clusters but can be extended to each of them. Storage resiliency has been taken into account and considered a critical part of the architecture. As a result, several configurations have been analyzed to ensure the high availability of the database and, in the end, Percona XtraDB Cluster was chosen. Its main strengths are: (*i*) high availability, (*ii*) prevents downtime and data loss, (*iii*) and provides linear scalability for a growing environment [27]. The evaluation focused on the overhead brought by the orchestrator, as well as storage resiliency and the time taken by Kubernetes to react to simulated faults. Then the analysis was extended using the Cloudsim simulator in order to evaluate both the overhead brought by an increasing number of PMU connected to one PDC and the availability of Hosts upon the occurrence of a failure.
The results show relevant information necessary to design edge clusters both in hardware resources and number of nodes. The analysis of the consumption of the hardware resources gives a general view on the minimal amount of resources to use in order to guarantee the correct operation of the infrastructure. On the other hand, the analysis of reaction times provides a general overview of Kubernetes behavior and shows areas that need to be tuned to enhance response times in the event of failures. Moreover, analysis using CloudSim shows that a Host's hardware resources consumption is closely related to the services they run, in particular, the PDC service requires a considerable computational load. In addition, another result shows that a growing number of Hosts within a cluster ensures greater availability and reliability of the entire infrastructure.

9.1 Future work

This thesis analyzed the infrastructure built with K3s and the other services already mentioned, with the aim of finding its critical points and analyzing its feasibility in an edge computing context. Several interesting conclusions have emerged both on the timing at stake and on possible single points of failure of the infrastructure. In this work, OpenEBS together with Percona XtraDB Cluster, were used to ensure the high availability of the database within a single cluster. Some of the other solutions listed above weren't designed with this use case in mind, but they might be modified in the future to fully satisfy the requirements. Several considerations regarding database management remained open: (*i*) the use of a technology other than MySQL (for example Microsoft SQL Server, MariaDB, PostgreSQL etc.); (*ii*) the analysis of replication latency and the implementation of the current solution in a geo-distributed environment; (*iii*) using OpenEBS to provide backup of the database.

As presented in Chapter 7, the contribution of the orchestrator on the final restart time cannot exceed $5s$, so this means that the remaining time is thus related to the service control logic to re-instantiate the communication and can be reduced only with proper code refactoring. Another problem that might suggest code refactoring is the one presented in section 6.3.1. In this case, the PDC code does not provide a solution for the management of a TCP connection in a "pending" state and consequently causes an almost "manual" management in case of network partitioning.

From the analysis carried out with CloudSim it is clear that the main problem is to have the PDC code monolithic and not parallelizable. Breaking this code down into microservices, or at least manually changing the code so that it can run on multiple cores, would increase the scalability of the infrastructure. Also from the analysis extracted from the simulator was born a question about the stations where the PDC is scheduled: these are stations that require a high computational capacity and therefore you could think of creating stations of different capacities to better support the scheduling of the PDC itself.

Other possible continuations concerns: (*i*) the analysis of latency in packet transmission with the infrastructure proposed in section 4.2, considering the path that packets take from when they are generated to the area control center; (*ii*) analysis of database capacity and the number of past information you need to store. Of course, with a continuous write to the database, it is necessary to establish a target value, beyond which it is necessary to empty the database in order to continue with the subsequent writes.

The resource requirements for each of the above prospective directions should then be assessed in order to determine the costs associated with adding these more features to the systems.

Bibliography

[1] Ramazan Bayindir, Ilhami Colak, Gianluca Fulli, and Kenan Demirtas. «Smart grid technologies and applications». In: *Renewable and sustainable energy reviews* 66 (2016), pp. 499–516 (cit. on pp. 1, 2).

[2] Legambiente. *Il clima è già cambiato*. 2021 (cit. on pp. 2, 4).

[3] Zhu Zhongming, Lu Linong, Yao Xiaona, Zhang Wangqiang, Liu Wei, et al. «2020 Tied for Warmest Year on Record, NASA Analysis Shows». In: (2021). URL: `https://www.nasa.gov/press-release/2020-tied-for-warmest-yearon-record-nasa-analysis-shows` (cit. on p. 1).

[4] «Clean energy for all Europeans package / Energy». In: (). URL: `https://energy.ec.europa.eu/topics/energy-strategy/clean-energy-all-europeans-package_en` (cit. on p. 1).

[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)*. IEEE. 2014, pp. 392–397 (cit. on p. 1).

[6] Junwei Cao and Mingbo Yang. «Energy internet–towards smart grid 2.0». In: *2013 Fourth international conference on networking and distributed computing*. IEEE. 2013, pp. 105–110 (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)*. IEEE. 2019, pp. 22–30 (cit. on pp. 2, 3).

[8] IRENA. «Renewable mini-grids». In: *International Renewable Energy Agency* (2016) (cit. on p. 3).

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

[11] *Wikipedia - Sottostazione Elettrica.* URL: https://it.wikipedia.org/wiki/Sottostazione_elettrica (cit. on p. 8).

[12] Terna. *Italian National Grid.* URL: https://www.terna.it/en/aboutus/business/italian-national-grid (cit. on pp. 9, 10).

[13] Fulvio RISSO. «Delivering Resilient Virtualized Services in Smart Grid Environments». In: (2021) (cit. on pp. 11, 12).

[14] Kubernetes. *What is Kubernetes?* URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/ (cit. on p. 14).

[15] *Kubernetes Architecture.* URL: https://github.com/kubernetes/community/blob/master/icons/examples/schemas/std-app.png (cit. on p. 15).

[16] *Cluster Architecture.* URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (cit. on p. 16).

[17] *Kubernetes Components.* URL: https://kubernetes.io/it/docs/concepts/overview/components/ (cit. on pp. 16, 17).

[18] Sebastian Böhm and Guido Wirtz. «Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes.» In: *ZEUS*. 2021, pp. 65–73 (cit. on pp. 17, 18).

[19] *K3s - Lightweight Kubernetes.* URL: https://rancher.com/docs/k3s/latest/en/ (cit. on p. 18).

[20] *E-distribuzione. Piano di Sviluppo 2020-2022 (Development plan 2020-2022).* URL: https://www.e-distribuzione.it/content/dam/e-distribuzione/documenti/e-distribuzione/Piano_di_Sviluppo_delle_infrastrutture_di_EDistribuzione_2021_2023.pdf (cit. on pp. 20, 21).

[21] Charles Bell. «MySQL Group Replication». In: *Group Replication*. Springer, 2022, pp. 3694–3702 (cit. on pp. 24, 29, 30, 33).

[22] *Advantages and disadvantages of MySQL replication types.* URL: https://hidora.io/en/ressources/avantages-et-inconvenients-des-types-de-replication-mysql-et-comment-les-executer-dans-la-cloud/ (cit. on pp. 25, 26).

[23] *Pros and Cons of MySQL Replication.* URL: https://dzone.com/articles/pros-and-cons-of-mysql-replication-types (cit. on p. 25).

[24] *MySQL Operator for Kubernetes.* URL: https://github.com/mysql/mysql-operator (cit. on p. 27).

[25] *Running MySQL on Kubernetes using an operator*. URL: https://banzaicloud.com/blog/mysql-on-kubernetes/ (cit. on pp. 27, 28).

[26] *MySQL packaged by Bitnami*. URL: https://github.com/bitnami/charts/tree/master/bitnami/mysql (cit. on p. 28).

[27] *About Percona XtraDB Cluster*. URL: https://docs.percona.com/percona-xtradb-cluster/8.0/intro.html (cit. on pp. 30–32, 58).

[28] *MySQL for Galera Cluster*. URL: https://severalnines.com/sites/default/files/blog/node_5100/image1.png (cit. on p. 32).

[29] *Certification-Based Replication*. URL: https://galeracluster.com/library/documentation/certification-based-replication.html (cit. on pp. 32, 33).

[30] *State Snapshot Transfers*. URL: https://galeracluster.com/library/documentation/sst.html (cit. on p. 33).

[31] URL: https://dzone.com/articles/container-attached-storage-cas-vs-software-defined (cit. on p. 34).

[32] *OpenEBS for Kubernetes*. URL: https://sudip-says-hi.medium.com/why-openebs-3-0-for-kubernetes-and-storage-64ca65f921c5 (cit. on p. 34).

[33] *OpenEBS Local PV*. URL: https://openebs.io/docs/concepts/localpv (cit. on p. 34).

[34] *OpenEBS for Percona*. URL: https://openebs.io/docs/stateful-applications/percona (cit. on pp. 35, 38).

[35] *OpenEBS Architecture*. URL: https://openebs.io/docs/concepts/architecture (cit. on p. 38).

[36] *Percona Monitoring and Management*. URL: https://www.percona.com/software/database-tools/percona-monitoring-and-management (cit. on p. 41).

[37] *Galera Replication Latency PMM*. URL: https://docs.percona.com/percona-monitoring-and-management/details/dashboards/dashboard-pxc-galera-node-summary.html#galera-replication-latency (cit. on p. 41).

[38] Manoel C Silva Filho, Raysa L Oliveira, Claudio C Monteiro, Pedro RM Inácio, and Mário M Freire. «CloudSim Plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness». In: *2017 IFIP/IEEE symposium on integrated network and service management (IM)*. IEEE. 2017, pp. 400–406 (cit. on p. 48).