POLITECNICO DI TORINO

Master's degree course in Computer engineering

Master's Degree Thesis

Resilient and Low Latency Communications in Smart Grid Environments

Supervisor
prof. Fulvio Risso

Candidate
Claudio Lorina

A.Y. 2020-2021

Abstract

The decarbonization of the power supplies, with the introduction of renewable energies, new kinds of loads, such as smart vehicles, are making the electric power system a dynamic environment. The ancient assets, the legacy software, and protocols used for the control of the power grid are not enough anymore, and the introduction of new ICT technologies is bringing the concept of *smart grid*, allowing the orchestration and the integration of the efforts of the main actors of the power system (generators, carriers, and consumers) [33, 49, 45]. This is translated in the need of increasing the observability of the power grid [49], which can be achieved via a huge network of sensors, interacting among them and with the infrastructure in order to provide information about the physical world. This data can be later stored, processed, analysed in order to control the behaviour of the grid through intelligent actuators [69] or for offline analysis [48]. The smart grid networks should manage a great amount of data, delivered over different physical media, coming from many different types of devices, some of them with limited computational power, and with different requirements in terms of QoS. This thesis work analyses the usage of a publish-subscribe model in the context of a smart grid environment, underlining the aspects of data availability, durability and reliability, keeping latencies under control. Moreover, the design of the ICT of a distribution system as a *data-centric* architecture is provided, a place where data can be asynchronously produced, consumed, processed, aggregated, and stored, without the need of direct interactions between producers and consumers.

Contents

Chapter 1

Introduction

The classical electrical power system architecture, developed over the past 70 years, had a centralized control. There were big power plants (fossil-fuelled, nuclear power, or hydropower), producing up to 1000MW. The production system interacted with the transport system in order to ensure always the same value of frequency and to receive the required amount of energy. This portion of the power system had an automatized control while the distribution system was almost completely passive, with only local real-time monitoring and control for the largest loads, but no additional interactions between the loads and the power system were performed [33].

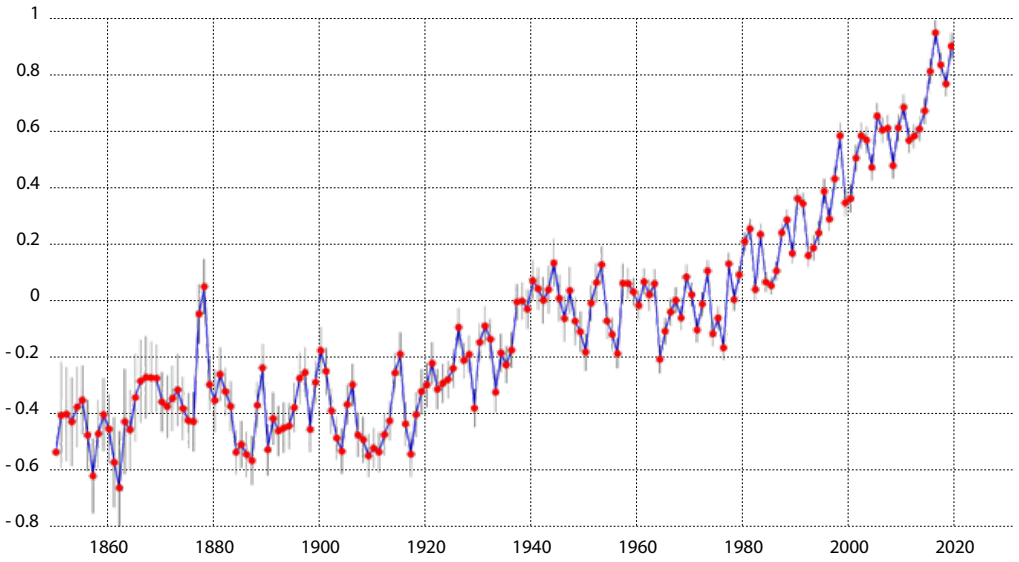


Figure 1.1: Temperature anomaly in C° from 1850 to 2019 to respect the common baseline 1951-1980 mean

From Legambiente climate report 2020 [53] - Berkeley Earth data combined with sea data from UK Hadley center.

The climate changes leading to global warming, driven by the human emissions of greenhouse gases, required the reduction of the produced CO_2. According to NASA, 2020 tie with 2016 was the warmest year on record, with a long-term record of the last seven years, when recorded temperatures were, on average, 1,02 C° higher than the baseline 1951-1980 mean [1]. In order to reduce the carbon footprint, new renewable, green and clean sources of energy were introduced, with some consequences in terms of power grid management. For example, the EU with the *Clean Energy Package* set the target for the 32% for renewable energy sources in the EU's energy mix by 2030, and the goal of carbon neutrality by 2050 [4]. Indeed, the centralized control of the power grid was not enough for a power system where production was not centralized anymore. There was the need to increase the grid observability via a network of sensors providing information about the physical world [49] and allowing the power grid to balance the power supply and the demand. Thanks to the increase of grid observability, new perspectives of automatized control, even in the distribution system, are possible [33]. The usage of the ICT technologies in order to share data from sensors and meters, collect and process it to control the electrical power system is the concept of *smart grid*. However, nowadays, the concept of *smart grid 2.0* [35] 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 [60].

1.1 Power grid resiliency with micro-grids

At this point, the concept of *micro-grid* comes into play, as a portion of the grid with loads, accumulation systems, and production systems, able to work attached to the main grid, or *as an island*, which means autonomously, isolated from the rest of power grid. The concept of micro-grid is not new, and in the past, intended as a way of bringing light to remote communities or as a backup system of the main grid. However, the difference lies in how they are powered. While in the past micro-grids relied on fossil fuels, the introduction of renewable sources of energy not only allowed a reduction of costs, but the energy production at the edge of the power grid, improved the reliability [44]. Coming back to the concept of smart grid 2.0. Each micro-grid can be plugged to the main grid, and it can exchange electricity, supporting the main power grid, injecting power, or requiring electricity, if needed. In any case, if the micro-grid detaches from the rest of the grid, on purpose or because of unintended events, it can survive, go on working even though it is isolated. Resiliency, indeed, is nowadays a crucial aspect for power grids, considering the increase of extreme weather events, due to climate changes. The increase of the average temperatures causes a reduction in rainfall, but a consequent rise of floods, storms, and hydrogeological risk [53, 62]. Figure 1.2 shows the increase

of extreme weather events in Italy for each year. It is evident how the world is changing, extreme climate events are becoming much more frequent, and people are called to get used to this new normality. Human infrastructures need to be redesigned for this new world, to be resistant to the weather pattern of the future. Even the electrical power system, indeed, should be able to predict, react and survive these extreme events, and it is crucial for the design of the smart grid 2.0.

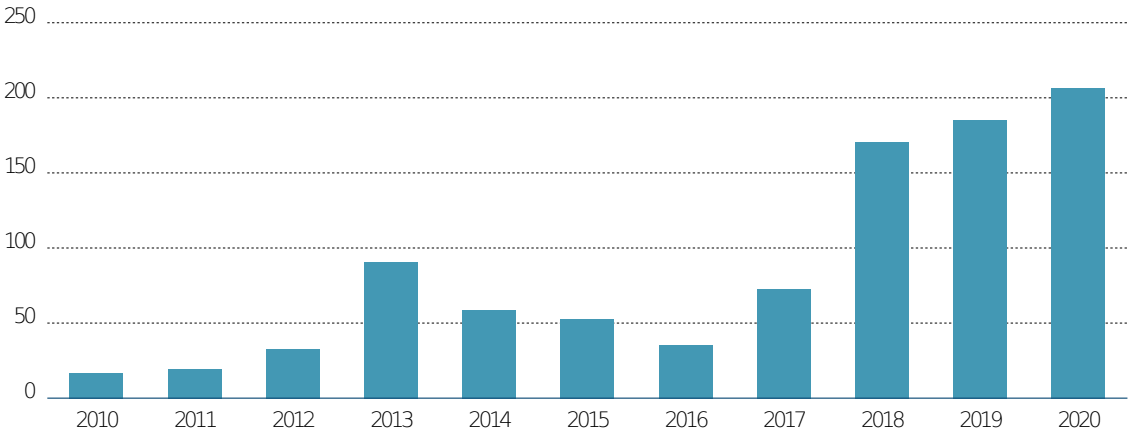


Figure 1.2: Extreme weather events in Italy for each year
From Legambiente climate report 2020 [53] - Osservatorio Cittá Clima, Legambiente 2020

1.2 ICT resiliency in a smart grid 2.0

Smart grid 2.0 requires real-time control for more than 50% of the power demand, requiring the monitoring of great amount of data coming from sensors and devices. The analysis and processing of this huge quantity of data and the control of the grid require the deployment of smart IT technologies and usage of big data analysis techniques [60]. Moreover, the resiliency of the power grid should be provided by a robust ICT infrastructure.

- Services have to be monitored to launch them again, in case of a failure of the application itself or the node where it was running.
- As the power grid should be able to support partitioning, even the ICT must support the partitioning of the infrastructure. Extreme weather events, accidental events, or network failures might isolate one or more sites of the electrical power system. When the site is isolated, its ICT infrastructure should be able to react, go on working, even though the connection with the centralized control has been lost.

The ICT of the power grid should handle the complexity due to the widely geographically distributed infrastructure. The solution scalability is crucial since

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

1.2.1 Overview of communication resiliency

This thesis work will not be about the resiliency of services, but it will focus on the aspect of resilient and low latency communication in smart grid environments. Chapter 2 presents an overview of the ICT of a current power grid, giving an idea of the roles of the three sections of the electrical power system: production system, transmission system, and distribution system. We will analyse, in chapter 3, the potentialities of the publish-subscribe communication pattern in the context of the smart grid environment. A comparison between the publish-subscribe-based IoT protocols has been performed, underlining which, between the presented solutions, seem to be the one that best matches with the communication requirements in a smart grid. Chapter 4 focuses on the synchrophasors exchange between PMUs and PDCs, in order to improve the observability of the power grid. The current standards of synchrophasors exchange protocols have been analysed, in order to understand how this kind of data is currently exchanged. This chapter presents the Hoefling et al. [42] solution allowing to transport the IEEE C37.118 messages over a protocol supporting a publish-subscribe interaction pattern, keeping the compatibility with the current PMU and PDC implementations supporting the C37.118 standard. An implementation of this solution supporting the MQTT protocol is presented in chapter 7, with the aim of evaluating the benefits of the synchrophasors exchange through a publish-subscribe protocol and the additional overhead given by the usage of the MQTT brokers for the interaction and of the adapters, allowing the usage existing PMU and PDC implementations when synchrophasors are exchanged over the MQTT protocol. The usage of publish-subscribe typically requires all messages passing through a broker. If the solution is not correctly designed and configured, the broker might represent a bottleneck or a single point of failure, with some consequences in terms of performance and availability of the services. Chapter 5 is about the resiliency, performance, and scalability of commercial implementations of MQTT broker. These lines report investigations about how

replication of the broker, persistency of the storage, queue mirroring between the broker instances, and autoscaling affect the reliability, availability and scalability of the service and the durability of the exchanged messages. This chapter contains further analysis about benefits and the overhead of the storage replication with Longhorn, varying the workload and message size. A proposal of an ICT architecture in the distribution system of a smart grid is presented in chapter 6, and its implementation in chapter 7.

Chapter 2

ICT architecture in an electrical power grid

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

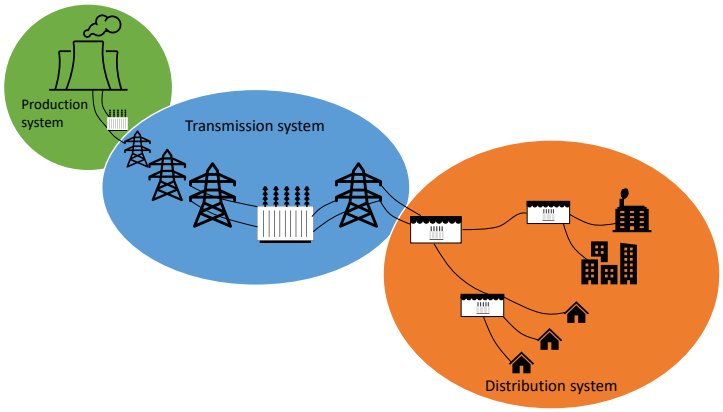


Figure 2.1: Electrical hierarchy overview.

- *Production system*: where the electricity is produced, converted with the right values of current, voltage and frequency and finally introduced in the transmission system. In the past, this was mostly done in huge production plants (e.g., hydroelectric, coal), while in recent years this is being integrated with many small-size production plants (e.g., solar power).

- *Transmission system*: in charge of collecting the electricity from the power plants and transporting it to the distribution systems (i.e. Terna in Italy).
- *Distribution system*: in charge of bringing the electricity to the final users, typically this part of the network is in charge of the energy providers.

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

2.1 Production system

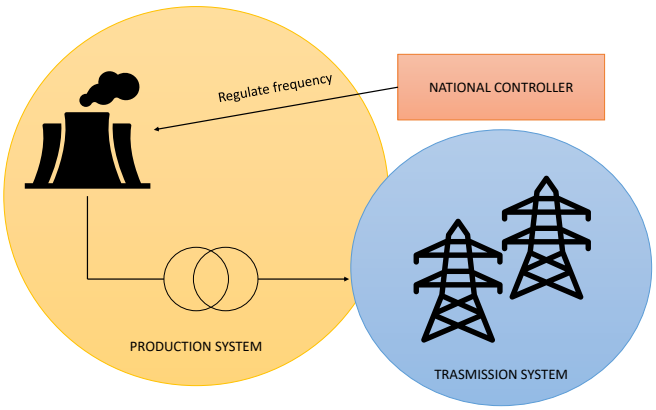


Figure 2.2: Production systems work in synergy with the transmission system.

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

2.2 Transmission system

The transmission system is made of electrical towers of 380kV, 220kV and 130/150kV all connected with the others, forming an unique grid covering the entire national surface. This system is controlled by some stations with a set of transformers

Figure 2.3: Example of a thermoelectric power plant

converting the ultra-high voltage to the high voltage. Electric substations (often abbreviated SSE) are located near a production plant, at the point of delivery to the end user and at the interconnection points between the lines: they therefore constitute the nodes of the electricity transmission network.

Figure 2.4: An example of an electric substation in the transmission system

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). [27]

Even these transformers have some sensors and actuators, the latter are controlled by devices called IED (Intelligent Electronic Device). All the devices running locally, e.g. in a substation, are connected to each other by means of an Ethernet LAN, which also includes a Station controller, e.g. a server with the proper controlling software. Logically, the station controller is connected with the Regional controller, which is further (logically) connected to a National controller.

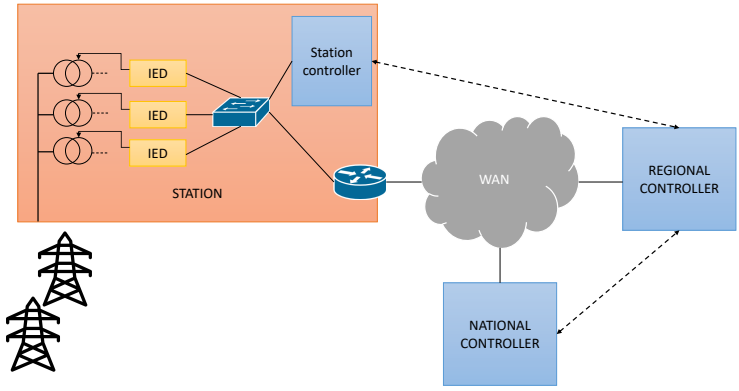


Figure 2.5: ICT network architecture of the transmission system.

The physical network connection between each station and the rest of the ICT network is usually achieved with dedicated links; in the past, this infrastructure was completely under the control of the Electrical company (i.e., ENEL), which was then spinned-out at around 1990-2000 when the Italian telecommunication market was open to competition, leaving to the creation of the Wind telecommunication company. Nowadays, the above physical network connections are in part still under the control of the Electrical company, while others are simply links bought from a telecommunication provider.

Electricity cannot be stored, therefore there is the need to guarantee the balance between the produced energy with the demand. This operation is a real-time control called *dispatching* and it is under the responsibility of the National controller, which acquires data from a large number of players operating both in production and demand, performs forecasts about the national electricity requirements and interacts with producers and remote management centers in order to module the supply and structure of the grid as require. [61]
In this case, the network is based on optical fibers running through the overhead protection cables, but still having a satellite network as backup.

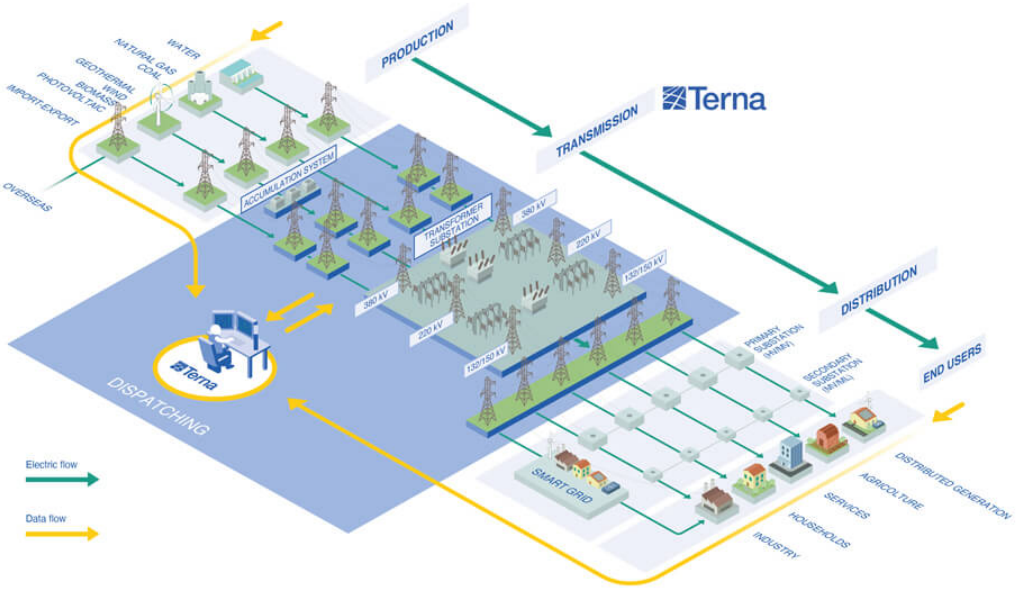


Figure 2.6: Terna's transmission system [61]

2.3 Distribution system

The distribution system starts from the primary substations where the high voltage electricity is converted into medium voltage. Here a set of measurement systems are used in order to track the state of the transformers, and to perform some adjustments opening and closing them, changing the transformation ratio, in order to keep the correct working point of the electrical grid. This is the starting point for the medium voltage lines, each controlled by a switch. These lines arrive at the secondary substations, where the medium voltage is converted to low voltage. These are the starting point for the low voltage lines, connected to user loads,

electricity generation systems, accumulation systems, which could also have some meters and sensors providing information about their working status.

Inside kiosks sensors and actuators are connected via Ethernet LAN, while data from loads, electricity generation systems, accumulation systems coming from the outside could reach the controller in charge of handling it, using GSM, 4G or powerline.

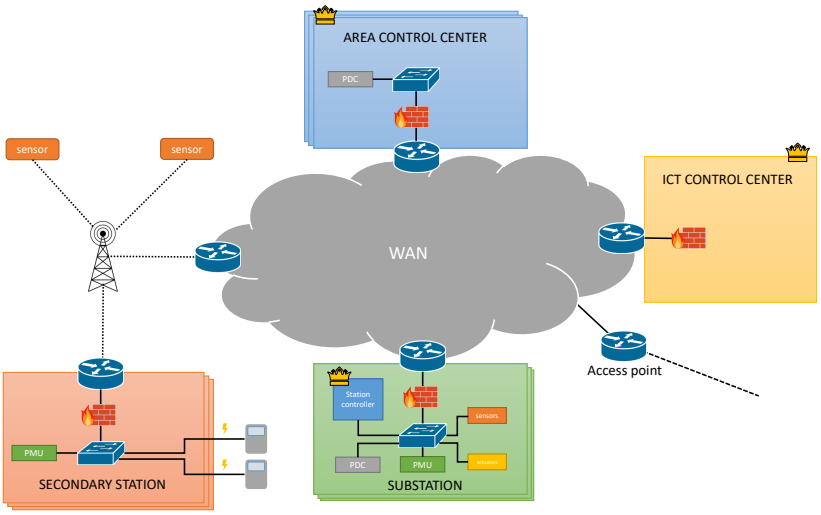


Figure 2.7: ICT network architecture in the distribution system.

The distribution system has three main levels of control:

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

Typically, data flowing between the control centers is carried over a dedicated WAN network, which might be made of fiber or equivalent technology. Each section of the network has a firewall, filtering incoming and outcoming traffic. The overall

Figure 2.8: An example of a primary substation
Available at https://www.deaelettrica.it/inaugurazione-nuova-cabina-primaria-recanati/

ICT system is also protected by an access point which performs some encryption to the incoming and outcoming traffic. That's because different distributors and the ICT of the transmission system, uses different keys, in order to keep them independent. This means that all the outcoming traffic should be decrypted with the internal key, then encrypted with the key shared with the destination, and then decrypted again and encrypted with the key of the destination. The same thing should happen with communications between the transmission systems of two different nations, this is needed because, since Europe runs on a single frequency, a variation of the frequency or a failure of a part of the grid, might affect all the other nations, which should properly react.

Figure 2.9: An example of a secondary substation

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

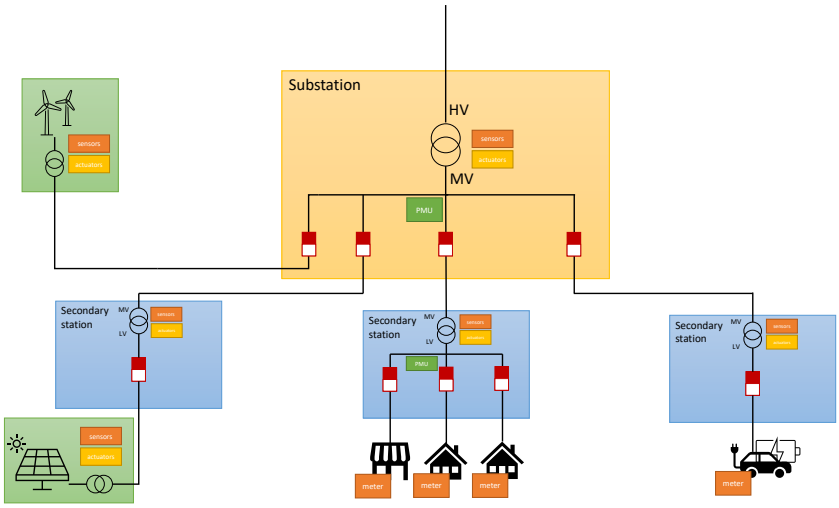


Figure 2.10: Electrical architecture of the distribution system.

Chapter 3

Publish-subscribe in smart grid environment

3.1 A solution for huge distributed systems

A smart grid represents a huge distributed system involving devices and pieces of software of different nature, distributed over a quite large geographical area. In this context, point-to-point synchronous interaction might be a too rigid approach, which leads to applications difficult to write and evolve. *Publish-subscribe*, instead, could be a more flexible communication model, since it allows a dynamic and decoupled interaction between the communicating parts [39]. Subscribers will express to a *broker* their interest in a specific event or pattern of events, and they will be notified as soon as a publisher generates an event of that type.

3.1.1 The three levels of decoupling

It might be possible to identify three levels of decoupling in the interaction between publishers and subscribers:

- **Space decoupling**: this means that publishers and subscribers do not need to know each other. They do not need to know where the others are located, their addresses and how many publishers or subscribers are interacting at the same time since they only have to know where the message broker is [39]. This is an important point, since if a publisher or a subscriber moves to another location or their IP address changes, all the others are not affected. Moreover, an arbitrary number of publishers and subscribers can be deployed in a transparent way, having the possibility to write new services consuming some data, without the need to change all the existing ones.
- **Time decoupling**: publishers and subscribers do not have to be up and

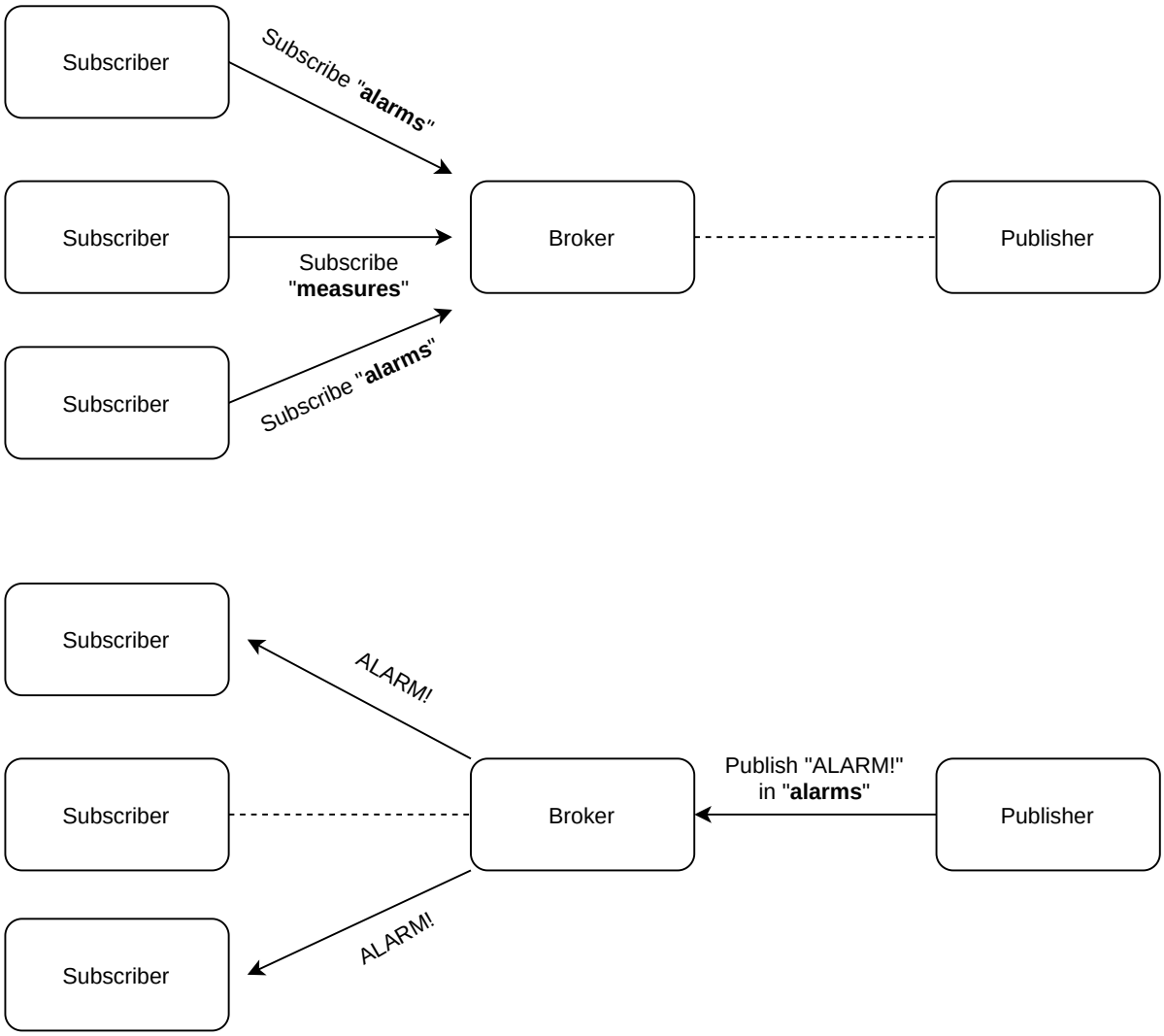


Figure 3.1: Example of interaction between publishers and subsribers with message broker

running at the same time: publishers can produce data, and subscribers will receive it once online again. [39]

- **Synchronization decoupling**: there is no need for synchronization between publishers and subscribers. It means that publishers are able to produce data without blocking, and subscribers can receive it asynchronously [39].

When there are N producers and M consumers, decoupling consumption of data from production might be really interesting since there is no need to configure and synchronize NxM channels, having some advantages in terms of scalability. The result of this communication system is a *data-centric* architecture, where interacting

parties are only interested in data and not in the address of the services providing it. [39].

3.2 Selection of a publish-subsribe-based protocol

A smart grid is characterized by heterogeneity of physical media and devices and it is likely to have low-cost devices with low computational power, sending data over a network with limited bandwidth, such as NB-PLC, UMTS or GPRS. Moreover, requirements in terms of QoS are increasing over time [69], with many functionalities requiring real-time data. That's why this context might need lightweight protocols allowing low latencies. There is a quite large number of protocols used in the IoT environment which might be taken into consideration, among them:

- *AMQP*
- *CoAP*
- *MQTT*
- *XMPP*

3.2.1 Advanced Message Queuing Protocol (AMQP)

AMQP is a broker-based messaging protocol initially designed for the enterprise environment but then extended to multiple application areas [31]. This protocol is based on two main concepts:

- **Exchange**: it is where messages are published, they receive the messages and forward them to zero or more queues, depending on the type of exchange and some rules, associated to the queue, called *bindings* [2, 67]. The *exchange type* represents the logic used by the exchange for selecting the queues where to deliver the message, while the *bindings* defines the matching rule for a specific message queue.
- **Message queue**: this is where messages are stored. Each message queue has some associated binding rules, and according to them, the Exchange deliver all the matching messages to that queue. [2, 67]

Publishers, willing to produce a message, send it to an Exchange, which forwards it to the message queues matching the bindinds. At this point, all the messages inside the queue should be consumed by someone. This job is done by subscribers, which perform a subscription to a specific message queue [2]. This mechanism is a powerful tool allowing to select where messages should be delivered.

In order to provide reliability, AMQP defines acknowledgements, sent by consumers to the broker and by the broker to the publishers, confirming the delivery of the message [2]. This means that only two levels of QoS are available: *at least once* if acknowledgements are enabled and *at most once* if not.

3.2.2 Constrained Application Protocol (CoAP)

CoAP is a protocol designed for constrained nodes with limited resources, sending data over networks with narrow bandwidth and a high level of error rate. That's why, differently from the other presented protocols, it is not based on TCP but UDP. CoAP has been designed to be the HTTP protocol for constrained devices, since it supports an HTTP-like request/response interaction. However, this protocol also supports an *observe-pattern*, meaning that a client can perform a subscription for a specific resource and it can be notified as soon as that resource changes. This pattern might enable the usage of the CoAP protocol with a publish-subscribe interaction, and in 2019 the IETF published a draft, aimed to extend the CoAP protocol, in order to enable a publish-subscribe interaction through a CoAP broker [50].

3.2.3 Message Queue Telemetry Transport (MQTT)

MQTT is an OASIS standard publish-subscribe messaging protocol designed for constrained devices, therefore with minimal requirements in terms of bandwidth and small code footprint [15, 31]. The MQTT protocol runs over TCP and supports three levels of quality of services [29]:

- *QOS-0*: it provides *at most once* warranty. In this case, reliability comes only from TCP, and publishers use a *fire and forget approach*, no acknowledgements exchanged, meaning that there is no confirmation for delivered messages [29].
- *QOS-1*: this level of reliability provides *at least once*warranty, acknowledgements are exchanged confirming the message delivery, but could be delivered multiple times [29].
- *QOS-2*: in this case messages are delivered *exactly once*, each message has an id code allowing the subscribe to discard that messages already received. Unfortunately, this solution has a large overhead due to the high number of acknowledgements to be exchanged [29].

3.2.4 Extensible Messaging and Presence Protocol (XMPP)

XMPP is a collection of open technologies for instant messaging based on the exchange of XML data [28]. Each entity in XMPP is identified by an ID called

JabberID (JID) and it looks like an email address, since it has a name and a domain, while requested resources are expressed at the end of the JID, with format *name@domain/resource* (e.g user1@test.com/measures) [58]. Messages exchange with the XMPP protocol are called *stanzas*, there are three types of stanzas:

- *message*: it contains some information sent by an entity to another, when a response is not expected [5];
- *presence*: it is a message containing information about the status of an entity. Entities can subscribe to the presence messages of a specific entity, receiving a notification when the status changes, for example, when it goes offline, or it is back online [58].
- *IQ*: These kinds of stanzas are sent when an entity requests or modifies a resource, as it happens with the HTTP GET and POST methods. For each iq stanza there should be a corresponding *iq response*, which is associated to the request through its id [5].

One of the main features of the XMPP protocol is its extendability. There is a great number of standardized extensions, introducing additional functionalities to the protocol. For example, the XEP-0060 adds support for the publish/subscribe messaging. This protocol has been adopted in the IoT environment when lightweight solutions were introduced, such as *µXMPP* or *XMPP client for mbed*[31]. At the time of writing, only TCP provides reliability to the XMPP protocol. No additional form of quality of service is provided. However, there are some proposals of extensions aimed to provide it. For example, [59] is a draft of an extension adding message acknowledgements, while [64] is a proposal for an extension introducing the same concept of three levels of QoS of the MQTT protocol.

3.2.5 IoT protocols comparison

	AMQP	**CoAP**	**MQTT**	**XMPP**
Initial target	Enterprise applications	HTTP for costrained devices	Telemetry	Instant messaging
Transport	TCP	UDP	TCP	TCP
Pub/Sub	Depending on exchange configuration	IETF draft	no notes	Provided by XEP-0060 plugin
At most once QoS	Yes	Yes	Yes	No
At least once QoS	Yes	Yes	Yes	No
Exactly once QoS	No	No	Yes	No
Payload type	Binary	Binary	Binary	XML
Fixed header size	8byte	4byte	2byte	-
No. related documents in 2020 (google scholar)	1410	3310	7230	998

Table 3.1: Comparison between IoT protocols.

Since AMQP, MQTT and XMPP are protocols running over TCP, they have a larger data overhead due to the connections establishment and closing [57]. Furthermore, XMPP is not a binary protocol, and an additional overhead is given by XML. Among them, MQTT is the one with the lower impact both in terms of data overhead and band consumption. CoAP, instead, is based on UDP, which is not connection-oriented, and this reduces the transmitted data and the total bandwidth requirements [32, 57]. However, many studies report similar power consumption of MQTT and CoAP when they work in similar scenarios: at most once and at least once transmissions [57]. Considering latency, TCP does not help in this context, due to the slow start approach used to avoid network congestions and the need to open and manage the connection between the interacting parts. This is not true with UDP, and even in this case, CoAP is the protocol guaranteeing the best times in terms of latency, with MQTT registering the best times in the group of TCP-based protocols [31, 57]. Moreover, MQTT is the protocol of the group, that, with QoS-2, guarantees the higher level of reliability, thanks to *exactly once* delivery of the messages. Unfortunately, this requires a four-way handshake, causing a large increase of the transmitted data, which is double than the CoAP one [57].

3.2.6 Event streaming platform (Kafka) vs messaging systems

The publish-subscribe pattern for message delivery is a common feature of two different technologies *event-streaming platforms* and *messagging systems*. The main difference between them depends on how messages are stored; while messaging systems store messages for the needed time for all the interested subscribers to receive them, with the event-streaming platform, how much time a message is stored, does not depend on the subscribers, but on an explicit configuration. This means that, with an event-streaming platform, it could be possible to configure the broker in order to delete a message after a certain amount of time or even not to delete messages at all. This difference in how messages are stored, also involve the way in which subscribers consume messages with the two technologies. While, in the case of the event-streaming platforms, subscribers are able to access to the previously sent messages, this is not possible with a messaging system, which can only receive messages sent after their first subscription. Moreover, with a messaging system, subscribers are typically able to perform a subscription to more fine-grained topics, receiving only the messages they are interested in. This is not always true with an event-streaming platform. Kafka, for example, uses partitions for scaling, since they allow to split the load of messages consumption to different subscribers. Unfortunately, creating a partition requires the allocation of some resources, and since for each topic there is the need of at least one partition, the number of topics should be limited, resulting in topics with data which is not always strictly related, and subscribers which need to filter messages. [3, 34]

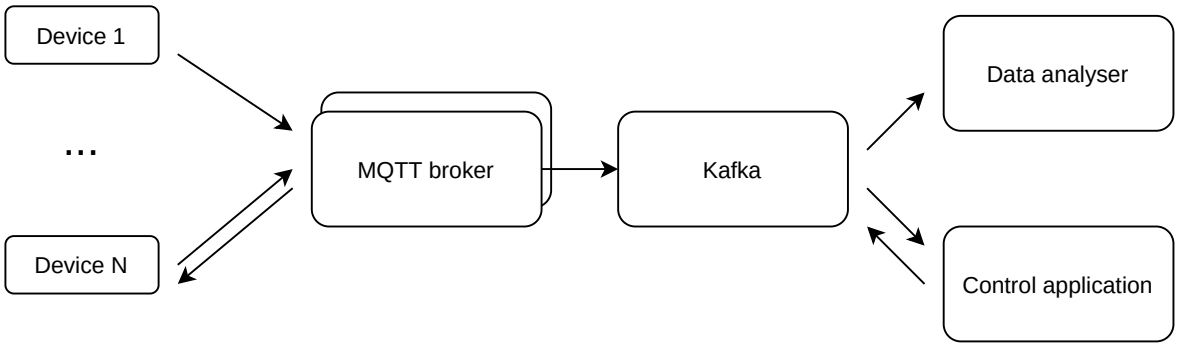


Figure 3.2: Example of integration of MQTT and Kafka

It might be clear that the two technologies offer completely different opportunities, and one not necessarily exclude the other. For example, as previously seen, MQTT is a protocol designed for constrained devices, sending data over unstable and narrow-bandwidth networks. Kafka, instead, as event-streaming platform, allows to persistently store messages and to retrieve historical data, which is a suitable solution for applications analysing bunch of data, such as, for example, machine learning applications for pattern recognition or analytics. Then, these applications could send back a command to some actuators, according to the result of the computation. This means that it could be possible to collect data from IoT devices through an MQTT broker, and then forward the messages to Kafka, where data could be processed. [14, 43, 47]

Chapter 4

Synchrophasors exchange over publish-subscribe

A way to improve the observability of the network might be the usage of *PMUs (Phase Measurement Units)*, providing an accurate picture of the state of the grid [49]. PMUs monitor the value of the intensity of the current, voltage and frequency, and marks the phasors with a timestamp, obtaining a *synchrophasor*. The PMUs located over the grid produce and send the synchrophasors to the *PDC (Phasor data concentrators)* of their area, whose role is performing a sort of multiplexing. They align incoming synchrophasors according to their timestamp and remove the unnecessary data. The output produced by PDCs is then sent as input of applications or of a higher level PDC, performing a higher level of aggregation [42]. Protocols currently used for synchrophasors exchange are all based on point-to-point interaction between PMUs and PDCs [36], this might represent a problem for multiple reasons:

- **PDC and PMUs need to know each other**: PDCs need to directly reach PMUs in order to start receiving data from them. They need to know the IP addresses of all the PMUs or PDCs from which they want to receive data. If, for example, a service is moved and its IP address changes, additional methods are needed in order to make it still reachable.

- **PDC and PMUs have to directly synchronize**: since the interaction is point-to-point, a single channel for each connection is needed.

- **Data produced during the downtime is lost**: when a PDC loses connection with a PMU, the data produced while it tries to reconnect is lost. However, this is a problem, since even though data cannot be used for real-time control, since it is delayed, the historical data has a crucial role for post-mortem analysis. For example, MacIver at al. [55] performed some investigations about

an incident in Great Britain in 2019, which interrupted the electricity supply to around 1.1 million of customers. They showed how the PMU data can be used in order to reconstruct the incident with a high level of precision. This highlights the importance of the PMU data when the goal is to improve the reliability of the power grid.

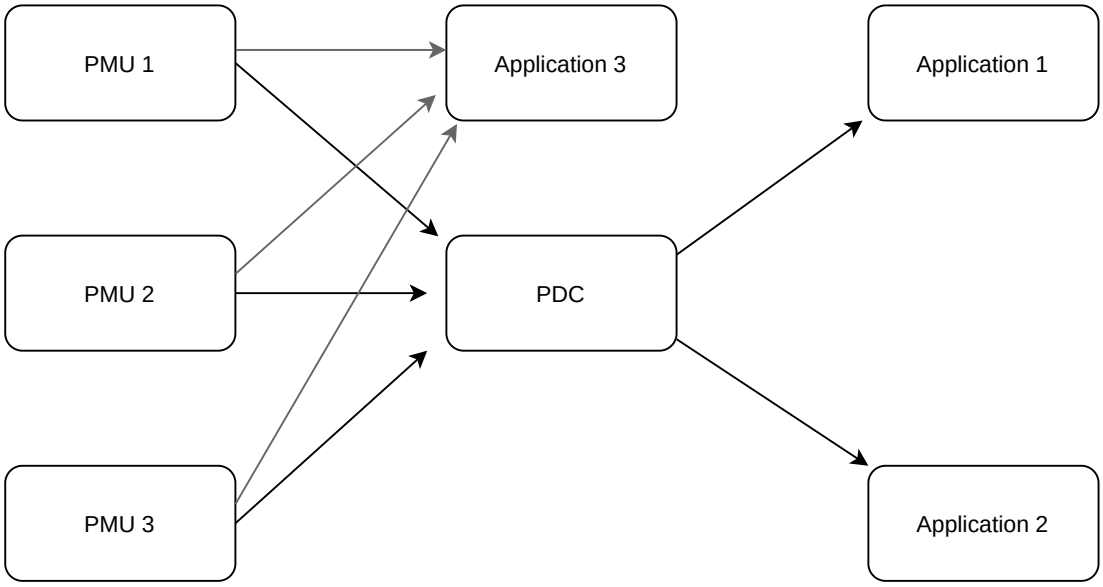


Figure 4.1: Point-to-point interaction between PMUs, PDC and applications

Figure 4.1 and 4.2 show an example of point-to-point interaction and with a intermediary broker. The two examples have the same actors exchanging data, the only difference is how data is transferred. While the actors of the first example have a direct interaction with the source of the data they are interested in, in the example with the broker, the interacting parts do not know each other. It means that:

- **PMUs, PDC and applications do not know the IP address of the others**, they only subscribe to the topic of the data they are interested in, without concerns about who is producing it and what its state is.

- **Each of the actors produces and consumes independently**: if, for example, the PDC disconnects from the broker, PMUs won't be affected and can go on producing data, which is received by the PDC once back online. If instead, a PMU disconnects, the PDC or the applications, listening for the data coming from it, don't have to restore the connection with it, but they will come back receiving data once the PMU is online again.

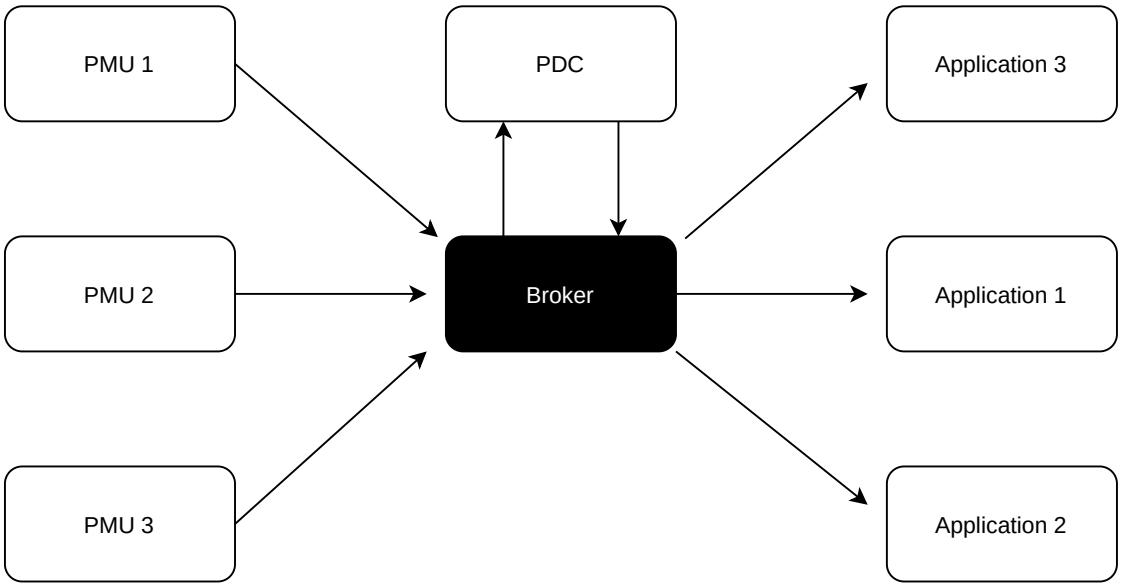


Figure 4.2: Interaction between PMUs, PDC and applications through publish-subscribe

- **Seamless support for multiple subscribers for the same data**: this allows not to have to synchronize a channel with each of the applications requiring the same data. Moreover, it is possible to introduce new applications, for example implementing new data processing algorithms, without the need to perform any change to what already exists.

4.1 Current phasors communication protocols

4.1.1 IEEE C37.118

In 2005 the definition of the *IEEE C37.118* protocol for the synchrophasor exchange has been completed. The idea was combining the *IEEE 1344*, the first protocol defined for exchanging synchrophasors, and the *PDCstream* defining the way in which PDC collects data and produces the output stream [36]. With version 2011 the protocol has been splitted into two parts:

- *Part 1*: defining measures of synchrophasors, frequency and rate both in stready state and in dynamic conditions;
- *Part 2*: defining the messaging part: how messages should be exchanged and the messaging format.

This new definition does not introduce any improvement to respect the 2005 version, that's because the idea was facilitating the integration with the *IEC 61850*,

another protocol for transporting synchrophasors, and keeping the backward compatibility with the 2005 version. Moreover, many of the functionalities to be introduced, were already provided by the IEC 61850 [56].

The IEEE C37.118.2-2011 standard, thanks to its simplicity and efficiency, is the most widely adopted protocol for synchrophasors exchange. It only defines how data should be represented, but it is independent of what is used for transporting it. Each of the interacting parts is identified with a 16-bit id code, placed in the common header of all the messages. Data is sent over binary frames, of 4 different types:

- **Command frame**: a frame used to control the behaviour of the node sending the synchrophasors. Different types of command are available:
  - *CMD-1*: Stop the transmission of the synchrophasors,
  - *CMD-2*: Start the transmission of the synchrophasors,
  - *CMD-3*: Send a header frame.
  - *CMD-4*: Send a configuration frame CFG-1,
  - *CMD-5*: Send a configuration frame CFG-2,
  - *CMD-6*: Send a configuration frame CFG-3.
- **Configuration frame**: this frame tells the receiver the format of the data frame, therefore it is needed before start parsing the data.
- **Data frame**: containing the synchrophasor measures.
- **Header frame**: a message containing human-readable information about the sender.

The protocol does not provide any security option; security should be provided by the underlining levels. About the bandwidth usage, even though no data compression is performed, the IEEE C37.118 protocol is the solution sending the smallest amount of data [36].

4.1.2 IEC 61850-90-5

The aim of the *IEC 61850-90-5* was to standardize the communication protocols used for synchrophasors, even in those environments where the IEC 61850 protocol was deployed, such as *substations*. The idea was to reuse the IEEE C37.118 protocol for the data measurement and representation, and transport it over the IEC 81650 stack. In this way, it was possible to exploit the native security options already provided by this protocol. Therefore, the role of the standard is mapping the IEEE C37.118 elements with the concepts and models of the IEC 61850. In terms of

bandwidth usage, the IEC standard is the one adding the largest overhead, with an additional header of 45 bytes and a prefix of at least 45 bytes, before the actual synchrophasor data, in data frames. Moreover, additional data is transmitted if the repetition of past measures is enabled. [36]

4.1.3 STTP: a new standard for phasor communication

In 2004 a new protocol called *Gateway Exchange Protocol* was introduced, with the aim of satisfying the requirements of a secure exchange of real-time data in smart grids. However, the GEP protocol has never become a standard. In 2017 a new project started, the ideas was defining a new protocol, called *Advanced Synchrophaso Protocol (ASP)*, whose aim was improving the GEP protocol and making it a standard. The IEEE working group in charge of defining the new standard was set up in 2018, and the new protocol candidate for becoming a new standard was called *Streaming Telemetry Protocol*. The name of the protocol emphasizes its capacity to transport any data representable longitudinally, not only synchrophasors. The STTP protocol is based on a *broker-less publish-subscribe pattern*. This means that the client directly performs the subscription to the producer and it is able to require only the data it is interested in, with full or down-sampled resolution. Moreover, it is possible to receive historical data, queried through a *STTP filter expression*, which is a string, written with a SQL-like languages, allowing the subscriber to specify which data it would like to receive.

The interaction happens through two different channels:

- *command channel*: used for negotiating a session, performing a subscription, authenticating, sending commands and so on.
- *data channel*: used for sending the actual data.

The traffic from both channels could be transported over a single TCP session or with a TCP session for the command channel and UDP datagrams for the traffic of the data channel. Since the command channel messages are sent over TCP, it is possible to secure them through TLS. Moreover, it allows to perform access control through X.509 certificates. When data are sent over TCP, security could be easily provided by TLS, while if UDP is used, data can be encrypted through a shared key exchanged in the command channel. [20, 36]

For each measure transported over the STTP protocol, there is a related timestamp. That is the reason why raw binary STTP messages are typically larger than the ones produced by the previously seen protocols, that have a single timestamp for a collection of measures. However, compression allows to overcome this problem. All data sent through the STTP protocol is compressed, resulting in messages smaller than the IEEE C37.118, when using TCP, and stateful compression.

With UDP, instead, the amount of sent data is comparable to the IEC 61850-90-5 protocol. [36]

Even though the definition of the STTP protocol as IEEE standard (*IEEE-2664*) is still working in progress, some implementations and tools implementing the protocol are already available [21, 22, 23, 24].

4.2 STTP as a solution for synchrophasors exchange

The STTP protocol thanks to the publish-subscribe pattern, security and compression of the data, might be the solution for synchrophasor exchange in modern smart grids. It, indeed, might allow to address the previously underlined problems:

- **PMUs, PDC and applications do not know the IP address of the others**: according to the STTP protocol specifications [20], the *forward connection* is the standard one, this means that starting the interaction is the role of subscribers. That is, each PDC should know IP addresses and port of each PMU it has to connect to. However, this is not mandatory, thanks to the *reverse connection*. This allows the producers (PMUs) to directly reach the subscribers (PDC), so that only the address of the PDC should be known. Moreover, this approach allows to set up a connection even when PMUs are behind a NAT, and cannot be reached by the PDC from the outside.
- **Each of the actors produces and consumes independently**: the STTP protocol does not directly address this problem, since the interaction between producers and consumers is point-to-point, in the specification, there is no intermediary broker [20]. However, a solution might be looking at the architecture from another point of view. It might possible to see at the PDC as sort of broker, receiving the data from sensors and forwarding it to all the subscribers that perform a subscription to that data. Moreover, the STTP protocol has some instruments allowing subscribers to query the historical data, sending a request with a *STTP filter expression*. However, sensors are typically devices with low computational power and small amount of memory, therefore, storing the data and deliver historical data is not the role of a sensor. That is why thinking the PDC as an intermediary service, which receives data from PMUs, allows subscriptions from applications, which can receive real-time data and historical data, could be reasonable and aligned to what already exists. At this point, the PDC is not a real consumer, but an intermediary. Therefore, producers and consumers are free to work independently.
- **Seamless support for multiple subscribers for the same data**: once the PDC becomes a sort of broker, anyone could perform a subscription to the data the PDC receives.

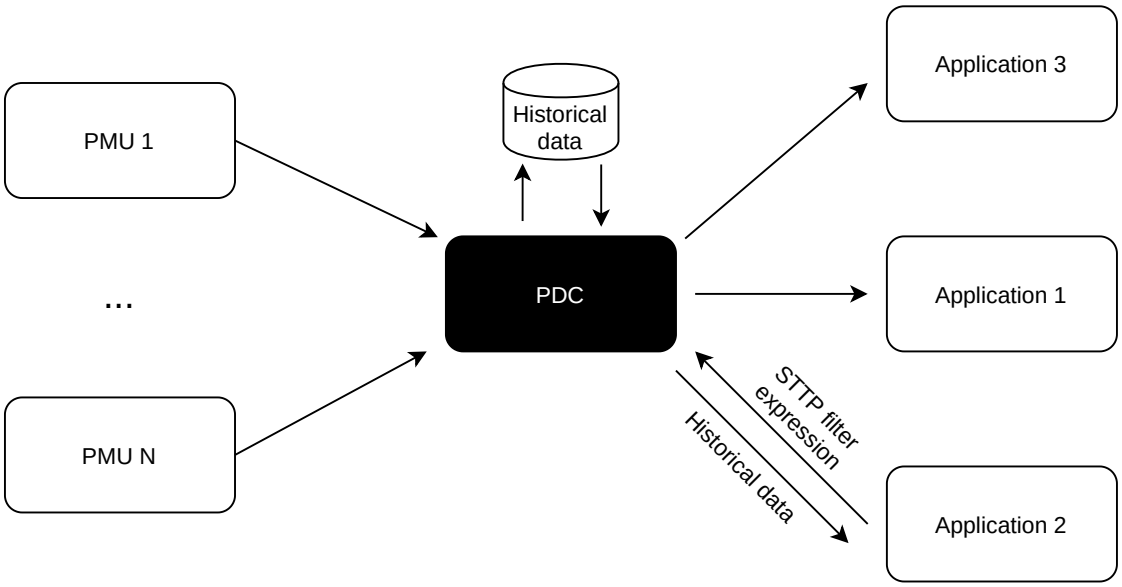


Figure 4.3: Example of architecture using the STTP protocol and the PDC as a broker

The validation of this architecture will not be the aim of this work. What will be presented in the following paragraphs is a solution addressing these problems with existing protocols and technologies.

4.3 IEEE C37.118 messages over publish-subscribe

As previously seen, the existing standards provide a point-to-point interaction between PMUs and PDCs, but this pattern has a set of limitations. A possible idea could be transporting the standard IEEE C37.118 messages over a protocol supporting the publish-subscribe pattern. This is perfectly possible since the IEEE C37.118 is independent from what is used for transporting it [42].

4.3.1 An intermediate layer between PMUs and PDCs

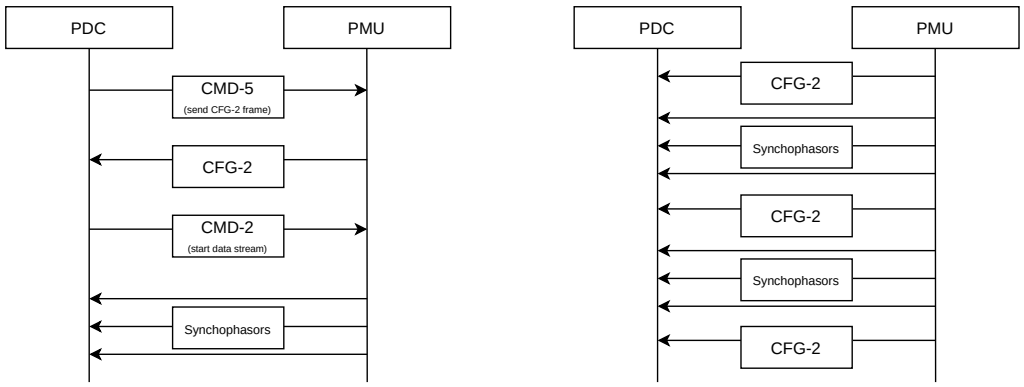
Unfortunately, existing PMUs and PDCs implementations do not support a publish-subscribe protocol. Hoefling et al. [42] already addressed this problem, providing a solution allowing to transport data over a publish-subscribe-based protocol, still keeping the existing implementations of PMUs and PDCs, which support the IEEE C37.118 protocol. The idea is introducing an additional layer, a sort of adapter, allowing both PMU and PDC to talk with the broker. There should be two different adapters:

- *PMU-adapter*: its role is pretending to be a PDC, obtaining the data from the PMU, and publishing it into a topic in the broker.
- *PDC-adapter*: this component, instead, pretends to be a PMU. It accepts connections from PDCs, performs the subscription to the topic related to the PMU it pretends to be, and forwards this data to the real PDC.

The behaviour of these components varies depending on the operational mode the PMUS and PDCs use. The standard defines two modes:

- *Commanded mode*: in this case, the PDC sends commands to the PMU in order to require the configuration frame and to start and stop the stream of synchrophasors.
- *Spontaneous mode*: when this operational mode is used, the PMU sends unsolicited messages via UDP. In order to allow the PDC to parse the received messages, the PMU periodically sends a CFG-2 frame. This solution is the one that best matches with the implementation of an adapter, since no interaction between PDC and PMU is required. However, this mode is typically not supported by off-the-shelf implementations.

In section 7.2 we will see an example of implementation of this solution integrating the IEEE C37.118 protocol with MQTT.



(a) Commanded mode (b) Spontaneous mode

Figure 4.4: Exchanged messages in commanded and spontaneous modes. Time from up to down.

Chapter 5

Performance and resiliency of a distributed MQTT broker

The centralization of communication due to the usage of the publish-subscribe pattern might be source of some concerns. All the interactions pass through the broker, which might become a bottleneck and a single point of failure:

- If the broker is not scalable enough, it might cause many problems in terms of performance. Messages might be delayed and, in some cases, if the broker is not able to accept other connections from the clients, communication is not possible at all. Scalability should not only be a property of the used broker, but of all the layers the messaging system relies on [39].

- The introduction of an additional layer requires more interactions since publishers and subscribers have point-to-point channels with the intermediary broker. It might negatively affect the performance and the scalability of the system.

- Messages are sent by publishers to the broker, stored, and then forwarded to the publishers. Having the warranty that messages have correctly reached the message broker is not enough. Messages should be persisted and eventually delivered to the destination even when the message broker fails [39].

This chapter is aimed to provide an overview of the reliability, scalability and performance of some of the MQTT brokers in the market, analysing the investigations already done in literature [41, 51].

5.1 MQTT broker resiliency

5.1.1 Message loss and latencies varying QoS

An interesting feature of the MQTT protocol is the possibility to select three different levels of QoS, depending on the grade of reliability needed for the transmission:

- *QoS-0*: messages are delivered most once,
- *QoS-1*: a message is received at least once,
- *QoS-2*: each message is delivered exactly once.

Grüner et al. [41] analysed the behaviour of each of these levels of QoS on different conditions of packet loss. They connected to an instance of *Mosquitto* MQTT broker a publisher, publishing 1000 messages, with a rate of 100msg/s, and two subscribers. They simulated different network conditions, in each test, increasing the value of message loss, thanks to Chaos Mesh.

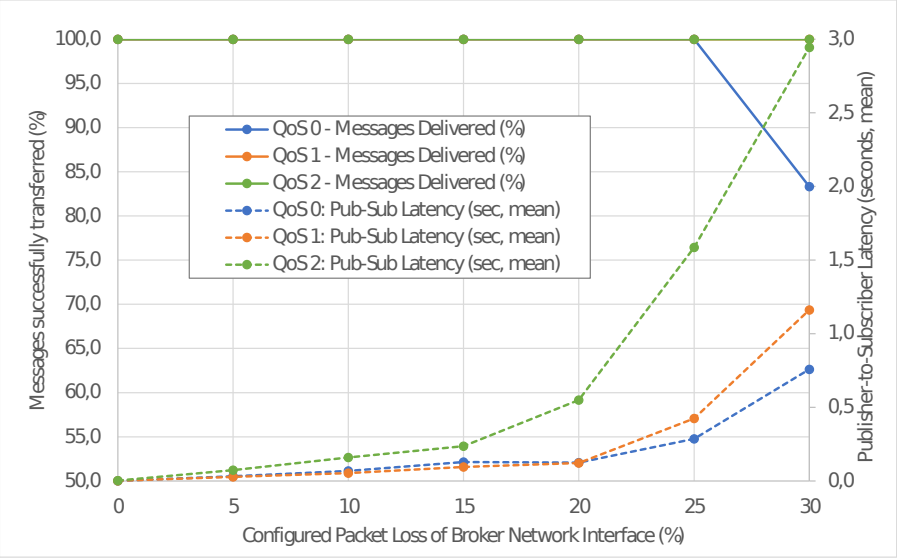


Figure 5.1: © 2021 IEEE - Message loss and latency for each QoS level, increasing the network loss rate [41]

The result they obtained (Figure 5.1) shows how under a network loss rate of 25% no message loss is registered, this shows how until this value, the reliability provided by TCP is enough. By further increasing the rate to 30% the value of delivered messages for QoS-0 decreases to 84%, still having the 100% for QoS-1 and QoS-2. Looking, instead, to latencies, the overhead given by QoS-1 is irrelevant to respect QoS-0. However, from the 25% on of loss rate, they start diverging, reaching

a difference of about 0,5s at 30% of message rate. The *four-way handshake* of QoS-2 becomes a problem with loss rate higher than 15%, from which the curve quickly grows.

On the whole, QoS-1 is a good solution whenever we need not to lose messages, due to the limited overhead under the 25% of packet loss. QoS-2 allows to avoid messages duplication, but the value of latency becomes problematic with higher value of newtwork loss rate.

5.1.2 Persistency of the storage

The reliability of message delivery can be provided by QoS 1 and 2. However, once the message is delivered to the broker, it must be able to forward the data to the subscribers, without any loss, even in case of failures. A way to provide the durability of the messages is the usage of persistent storage, allowing message recovery even when the broker crashes and it is restarted.

Grüner et al. [41] investigated on how message persistency allows preventing message loss in case of a broker failure. They used a K8S deployment with a publisher and two subscribers connected to the message broker, storing the message in a PVC with persistency provided by *Ceph*. Different commercial and open-source implementations of MQTT brokers have been tested. The idea was to publish with QoS-1 5000 messages. At message 2000, one of the subscribers *s2* was disconnected, in order to store some messages inside the message queue, then at message 2500 the broker was killed, and the subscriber reconnected.

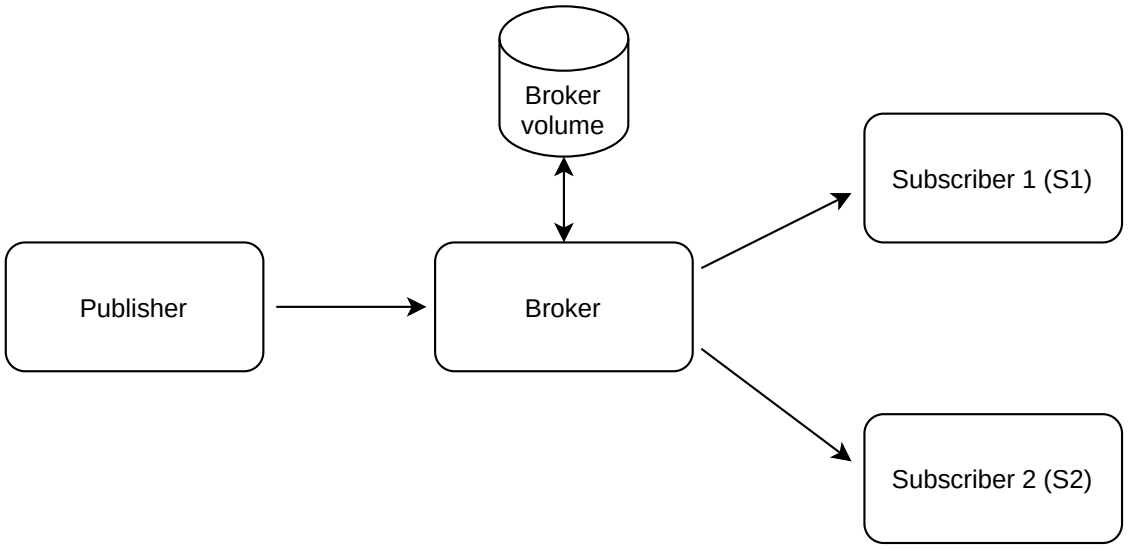


Figure 5.2: Schema of the scenario of the Grüner et al. data persistency experiment

Due to the storage persistency and the usage of QoS-1, the broker, once back online, should be able to restore the messages, delivering the ones not received by the

subscriber *s2*, after the disconnection, and going on forwarding the messages produced by the publisher, once it reconnects. Four different brokers have been tested: *EMQ X*, *HiveMQ*, *Mosquitto* and *VerneMQ*, and the experiment was repeated 50 times for each of these brokers. The result of the experiment was unexpected; while Mosquitto and HiveMQ, actually presented no message loss, EMQ X and VerneMQ showed respectively an overall loss rate of 1,07% and 5,34%. However, the authors supposed two different reasons why each of these brokers showed these losses.

- VerneMQ showed message loss in the 19% of runs, when both the subscribers were unable to reconnect to the restarted instance of the broker. Therefore, all the messages to be delivered after the broker restart went lost. The authors supposed it might be related to a VerneMQ bug.
- EMQ X showed message loss in the 33% of the experiments. In most cases the number of lost messages was around 500, the number of messages queued. However, the Redis database, used for storing the messages, always contained all the messages. As pointed out by the authors, this problem might be related to a bug, which, in some cases, prevents the messages to be transferred from Redis to the broker instance.

What it is possible to learn from this experiment is that persistency of the storage is not the single condition allowing to prevent the message loss. While the persistency allows reducing loss of messages, there are some cases in which it is not enough, and messages go lost in any case. Moreover, messages stored by the broker but not yet delivered, due to a broker failure, can be recovered only after the broker restart, which according to the authors, might be up to 30 seconds. This means that these messages are delayed by the broker restart time, and, depending on the application, this data might not be useful anymore, as old.

5.1.3 Queue mirroring

From the previous chapter, we learned that storage persistency allows to prevent message loss, but still, messages are delayed by the broker restart time. However, if we could be able to make available this data to multiple instances of the broker, after a broker failure, subscribers would connect to another instance of the broker, and they would be able to recover the previous session, receiving all the messages not yet delivered, without waiting for the failed broker instance restart.

This solution requires all the QoS-1 and QoS-2 messages to be replicated over multiple instances of the broker, so performing a synchronization between the broker instances before sending back a PUBACK to the publisher. This increases the complexity of the solution and might reduce the message throughput for messages with higher level of QoS. [41] HiveMQ MQTT broker supports the queue mirroring, thanks to its message duplication functionality, which allows to configure the

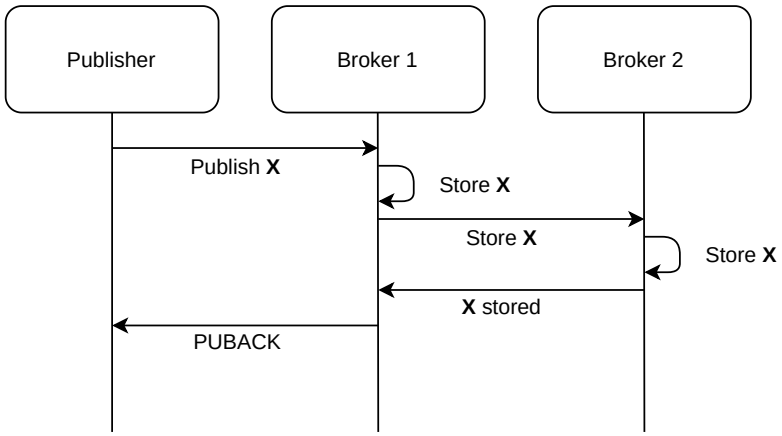


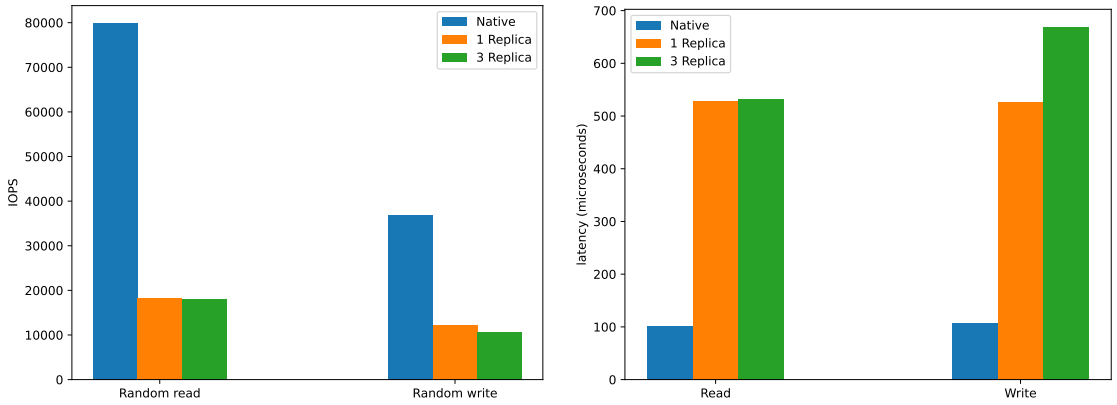
Figure 5.3: Interaction between publishers and broker instances with queue mirroring enabled. Time from up to down.

broker cluster to replicate the messages over multiple instances of the broker [10]. Grüner et al. [41] tested this functionality by deploying a cluster of two instances of the HiveMQ broker, without any persistent storage. They connected a publisher to the first instance of the brokers, publishing 50000 messages, and two subscribers, one for each broker instances. At message 2500, the second instance of the broker was ungracefully terminated. As expected, even without persistency of the storage, once the subscriber was able to reconnect, it received almost all the previous messages (only two messages has been lost in one single run). In conclusion, queue mirroring is a solution allowing to improve the availability of the messages, if used in conjunction with storage persistency, might provide an high level of durability of the data, even without the need to wait for the broker restart for recovering messages. Unfortunately, this is not for free, since the previous experiment showed an average difference in latency of about 2ms, probably due to the broker instances data synchronization.

5.1.4 Data replication overhead

As seen in the previous chapters, the persistency of the storage allows to avoid data loss when an instance of the broker goes down. However, even the storage should be resilient, since if there is a hardware failure or a partitioning of the network which makes the single copy of the data unreachable, data might be not be accessed or, in the worst case, can go lost. When we work with Kubernetes, there are storage providers allowing to have multiple replicas of the data in different physical locations. This increases the availability of the data, since if a replica is not reachable, there will be other available replicas. Moreover, in case of hardware failures, other physical replicas of the data are available, minimizing the probability

of data loss.



(a) IOPS (b) Latency

Figure 5.4: IOPS and latency of read/write memory access with Longhorn and the Kubernetes native storage [68]

Storage provides, such as Rook Ceph or Longhorn, allow to have multiple copies of the data, presenting one single logical volume to the applications. This allows to replicate the data over the cluster in a completely transparent way. However, whenever a writing operation is performed on a logical volume, the same operation should be performed in each single replica of the data, requiring synchronization between replicas, providing an additional overhead to the memory access. Figure 5.4 shows a coparison between the native Kubernetes storage and Longhorn with 1 or 3 replicas. It is possible to notice how the amount of reading and writing operation can be performed with Longhorn is lower, due to the need of synchronization between replicas. Moreover, Longhorn introduces an additional overhead in terms of latency, which in figure 5.4b goes between 100 to 500 microseonds per each IO operation depending the number of replicas and the kind of operation. Once data is replicated over multiple locations, it is possible to perform parallel reading access to the data, in order to increase the workload. As showed in figure 5.5, a solution using multiple replicas of the data presentes higher values of bandwidth in reading operations, since it is able to handle the workload request by sharing the load over the multiple physical volumes located in the nodes [68].

We wanted to measure the performance overhead provided by the storage replication in the context of a MQTT broker. In order to do that, we performed an experiment similar to the one carried out by Koziolek et al. [51], for testing the performance of different implementations of MQTT brokers. We constructed a Kubernetes cluster with two VMs:

- Intel Xeon (Cascadelake) 2.2 GHz

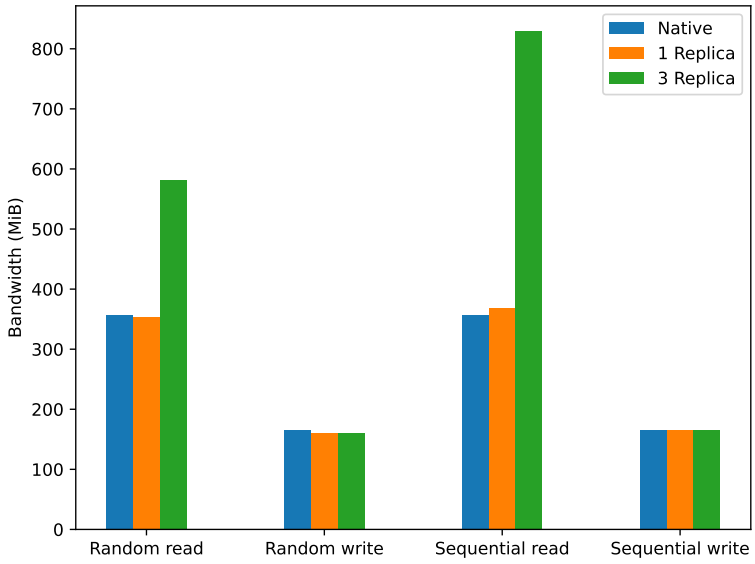
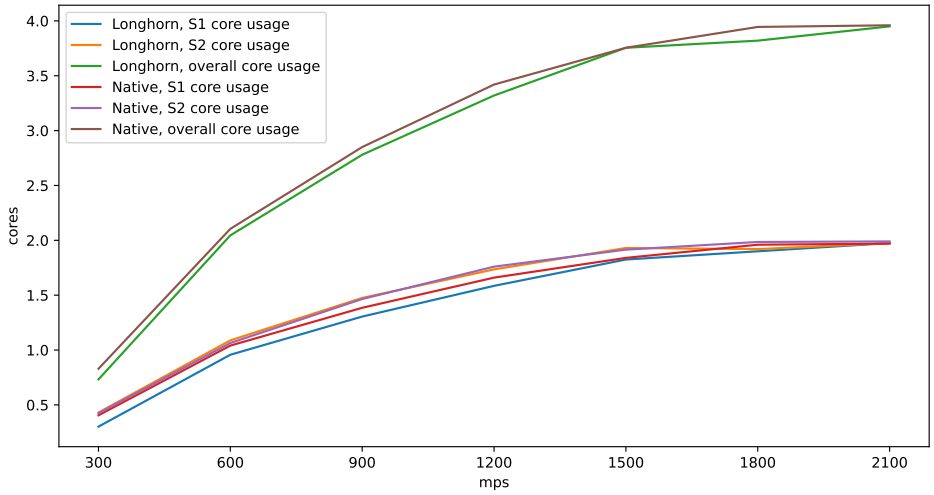


Figure 5.5: Bandwidth of read/write memory access with Longhorn and the Kubernetes native storage [68]

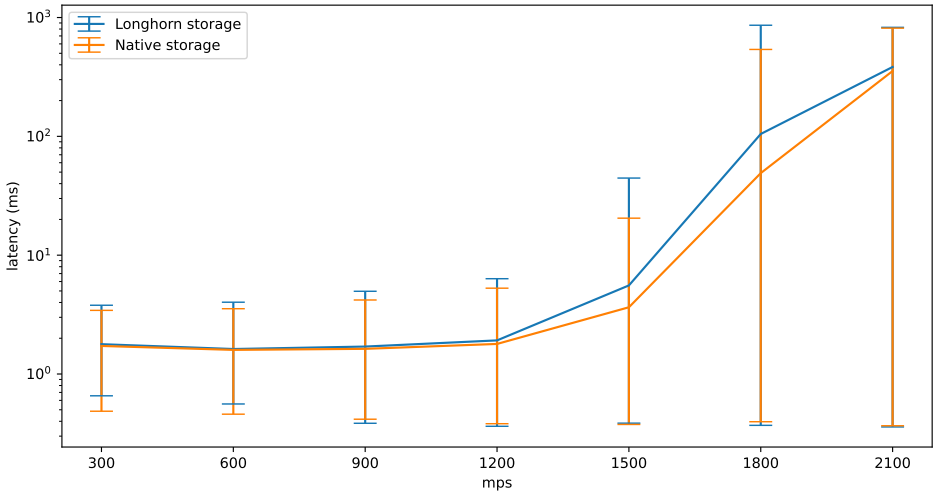
- 4 vcores (50% of the physical core)
- 8GB ram
- Ubuntu 20.04.3 LTS
- Kernel: 5.4.0-48-generic

We deployed Longhorn 1.1.2 as storage provider and two replicas of VerneMQ MQTT broker (v. 1.12.3). We reserved two cores for each instance of the broker, in order to avoid any interference given by other services running in the cluster. We connected to the broker 10 subscribers, consuming the data inside a topic, and every two minutes we added a publisher, producing 300msg/s, until we reached the cores saturation. We repeated the experiment two times, one with the Kubernetes native ephemeral storage and the other with Longhorn, providing 2 replicas of the data. We generated the requests with a third VM having 8 vcores and 16GB of RAM in a physical machine with the same specifications.

We sent 150-bytes messages, a dimension close to the typical size of a C37.118 data frame [49]. The results (Figure 5.6) show almost no difference between the two cases. Looking at the received message rate at figure 5.7, the two curves are overllaped and both the solutions reach saturation at 1800m/s. In terms of latency, it is possible to notice the slight difference given by Longhorn for the memory access, since in all the cases, the solution using Longhorn presented higher latencies.



(a) Broker cores usage



(b) Message latency

Figure 5.6: 150 bytes messages -performance comparison between native storage and Longhorn replicated storage (2 replicas)

On the whole, with the condition we tested, we were not able to observe a real difference in performance between native and Longhorn replicated storage. However, further investigations can be carried out, by performing some tests at larger scale. In this case, the CPU was the bottleneck the difference in IOPS between the two solutions could have an impact with higher message rates with

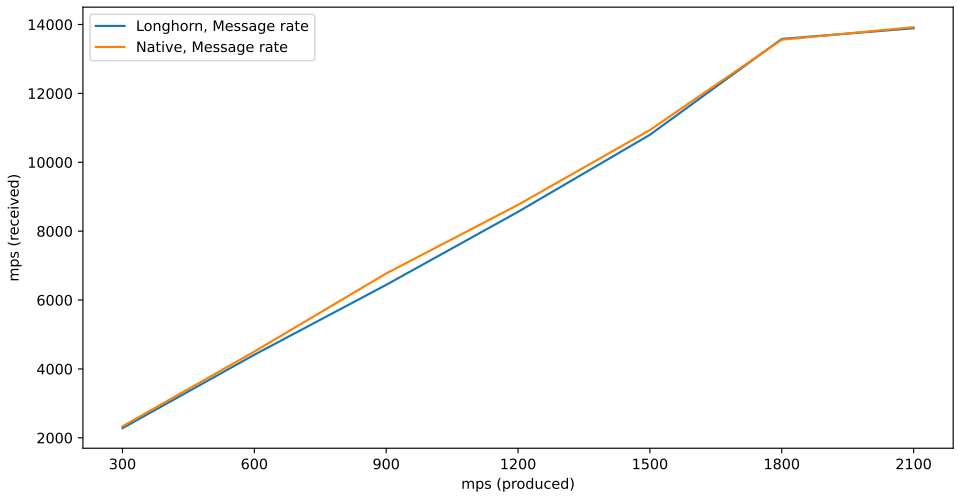


Figure 5.7: 150 bytes messages - received message rate comparison increasing producers message rate, with native storage and Longhorn replicated storage

brokers with more resources available.

It is possible to notice how in figure 5.7 the workload does not match with the message rate at the receiving side, that is because in some cases messages went lost even when the broker did not reach the saturation point. Looking at the VerneMQ logs, we noticed that occasionally the broker instances randomly disconnected and took some minutes before reconnecting. During the downtime, the replicas started discarding the messages directed to the other. This means that all the messages produced in one of the replicas but directed to a subscriber, connected to the other broker instance, went lost. This behaviour is not expected and might be related to a VerneMQ bug. Moreover, with QoS > 0, once the two replicas reconnect, the messages not yet delivered to the subscriber of the other broker instances should be eventually sent, but they were dropped.

5.2 Scalability and performance of an MQTT broker

Performance and scalability are two key features for an MQTT broker, since they are supposed to work in IoT environments where there might be a great amount of devices producing data at large scale, sometimes even with some requirements in terms of latency. Koziolek et al. [51] carried out a comparison between commercial and open-source distributed MQTT brokers, well analysing even the aspect of performance and scalability. They carried out an experiment deploying in a K8S

cluster, two instances of the analysed broker, an instance for each node (CentOS 8.1, Intel Xeon CPU E5-2660 v4 @ 2.00 GHz, 16 cores (32 threads) and 8 GB of RAM), with four core reserved to each broker instances in order to avoid any interference. They connected ten publishers and ten subscribers to the brokers, so having a one-to-one configuration for testing. At the beginning, publishers started publishing with QoS1 at a given publish rate, which was incremented by a constant value every two minutes. This allowed the brokers to reach stability before increasing the publishing rate. The results shows values of latencies under 10ms with until 5000msg/s as workload for VerneMQ and EMQX, and higher latencies for HiveMQ, which remain constant to a value around 100ms. Once reached the CPU saturation latency increases really quickly. In any case, message latency was under the 150ms for all the brokers, before reaching the CPU bottleneck.

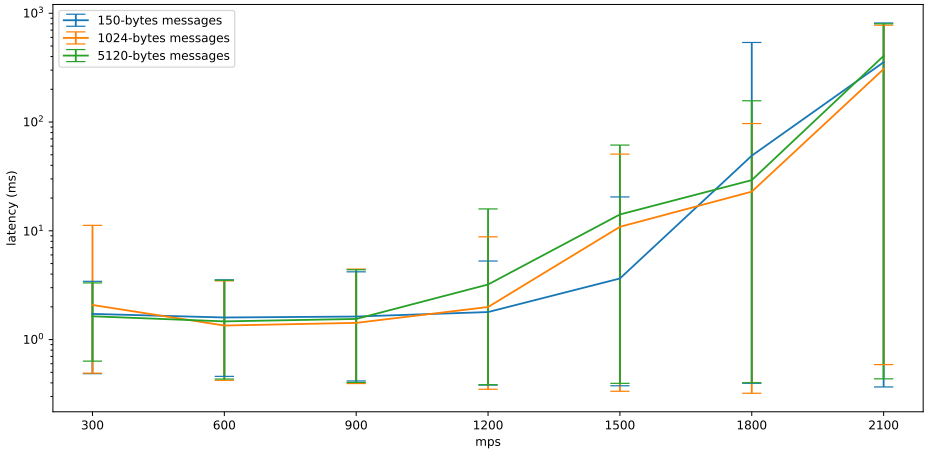


Figure 5.8: Message latencies varying the dimension of the payload with QoS-1

Koziolek et al. did not take into consideration the different payload dimensions, that is why we carried out some additional experiments with VerneMQ and the configuration reported in section 5.1.4. Figure 5.8 shows the results we obtained. Until 900 mps all the messages, with any dimension, was sent with latency lower than 1ms, this value start increasing for the 5120-bytes messages at 1200mps and for the others at 1500mps. Even though the delivery latency for 150-bytes messages grows slowly to respect the others, at the end, with the tested conditions, we did not notice great difference in latency, changing the dimension of the payload.

5.2.1 Benefits of the autoscaling

Multithread MQTT broker implementations seem to exploit all the available cores of the machines, being able to vertically scale [51]. Horizontal scalability is another

aspect to take into consideration. Many studies point out an incredibly high horizontal scalability of distributed MQTT brokers, being able to connect millions of clients [37, 30]. In the previous lines, we talked about CPU as a bottleneck, but it is not the only limitation, as the scalability of the broker is the only aspect to take into consideration. During the North America KubeCon 2018, Doyle et al. [37] presented the way in which they connected five millions of clients to a VerneMQ Kubernetes deployment. What comes out from this presentation is how considering all the dependencies is important when we want to go at scale. The MQTT brokers rely on many different layers, such as routing, load balancing, storage, and each of them could have an impact on the scalability and performance of the entire system. Moreover, performing troubleshooting of these layered architectures and understanding the source of problems is never easy. That is a reason why setting up monitoring tools is so much important.

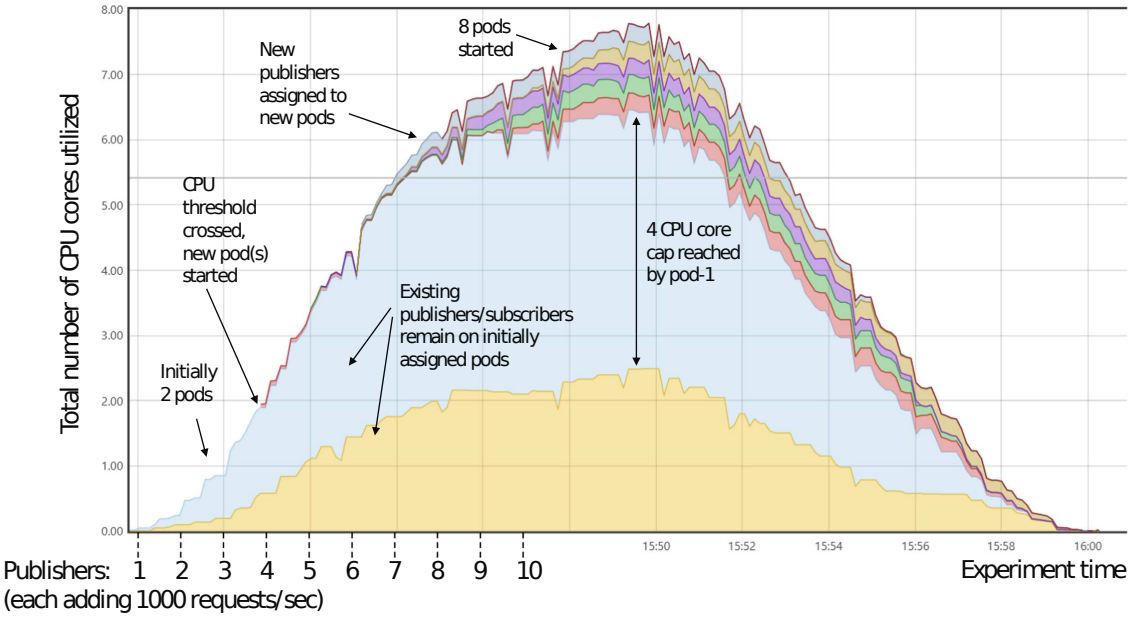


Figure 5.9: CPU usage of VerneMQ instances with Kubernetes autoscaling

Koziolek et al. [51] analysed in which terms the Kubernetes autoscaling allows to improve the scalability of a distributed MQTT broker. They carried out some tests with the same configuration of the latency experiment we saw at section 5.2, therefore with two instances of the broker running on top of two nodes and ten subscribers connected to the broker instances. However, this time one single publisher at a time has been connected, adding 1000msg/s to the overall rate, reaching a rate of 10000msg/s at the end of the experiment. Moreover, a CPU threshold has been configured for the Kubernetes autoscaling, so that, once the instances of the broker reach the threshold, a new one can be automatically spawned

by Kubernetes, from a minimum of 2 instances to a maximum of 8.

The figure 5.9 shows the results they obtained in terms of core used by each of the instances spawned by the Kubernetes autoscaling. The blue and yellow areas show the CPU core consumption of the initial two instances of the broker, which are much larger to respect the ones of the other instances. That is because, as pointed out by the authors, once a new instance was spawned, the new connections were redirected to it. However, all the existing connections (10 subscribers and 1 publisher spread between the two initial brokers) were not shifted to the new instances. Therefore, the broker cluster can benefit from the Kubernetes autoscaling when there is the need to support a larger number of publishers and subscribers, but, since connections are not shifted, if the higher load is due to an increase of the produced messages by publishers, the autoscaling will not help.

Chapter 6

Proposal of an architecture for a distribution system

6.1 Service and infrastructure resiliency

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

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

6.1.1 Geographically distributed clusters

Table 6.1 shows the number of primary and secondary stations across the years reported in the *Development plan 2020-2022* of *e-distribuzione* [38], a company inside the Enel group operating in the electrical distribution sector. The plan presents

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

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

Table 6.1: Number of primary and secondary station over years 2011-2019 [38].

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

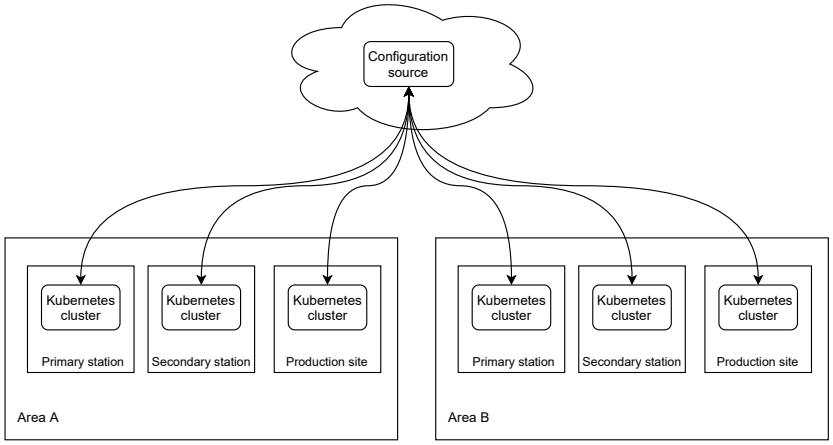


Figure 6.1: Clusters organization with single source of truth

Managing such a high number of clusters is not a trivial and multi-cluster approaches, originally designed for multi-cloud clusters, might not be the best choice. Even though a centralized control plane is not convenient, the services to be deployed are known a priori, since they depend on the kind of site and area. Scheduling

of services inside clusters is in charge of its local control plane. However, since the services to be deployed and configurations are known a priori, they can be retrieved from a single source of truth in cloud, via a push or pull fashion (Fig. 6.1).

6.1.2 Services

As previously stated, the services considered are PMUs and PDCs:

- PMUs 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.

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

6.1.3 Data resiliency

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

6.2 Data flow and communication resiliency

The introduction of renewable energy and new kind of loads, such as electric vehicles, make the power system a dynamic environment. [49] Its orchestration can be

facilitated via smart grids, allowing the integration of the efforts of the main actors of the power system (generators, carriers and consumers) [45]. This is translated in the need of increasing the observability of the power system [49], which can be achieved via a huge network of sensors, interacting among them and with the infrastructure in order to provide information about the physical world. This data can be later stored, processed, analysed in order to control the behaviour of the grid through intelligent actuators [69] or for offline analysis [48]. It is clear that smart grid networks should manage a great amount of data, delivered over different physical media, coming from many different types of devices, some of them with limited computational power, and with different requirements in terms of QoS [69].

6.2.1 Reducing distances with the Point of Presence

Some applications, especially the ones related to the control and stability of the power system, have strict requirements in terms of data delivery latency [48]. Reducing the distance between the interacting parts could be a way for reducing latencies. Differently from the transmission systems, where the network follows the power grid topology, is private, owned and self-managed by the transmission grid operator, this is not always true with the distribution system, where the electrical companies might need to rely on telecommunication providers. This means that, it is likely that the sites of the distribution system are not directly connected through dedicated links, but reaching a site from another requires a transit in the network of the telecommunication provider.

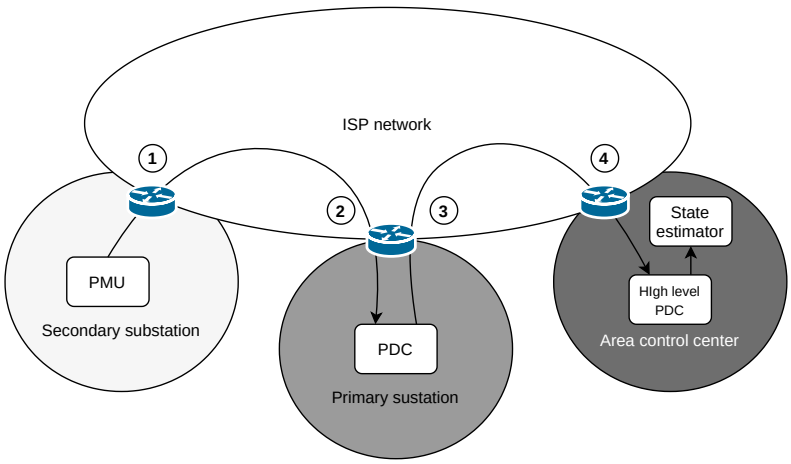


Figure 6.2: Data enters and exits from the power provider network four times

Knowing the path of traffic is extremely important for optimizing it and reducing latency. For example, let's imagine having a set of PMUs producing data and sending it to the PDC located in the closest primary station. The output stream is then sent to the *state estimator* inside the area control center. Looking at figure 6.2

and following the traffic, it might be clear how the exchanged data has to enter and exit from the power provider network to the ISP network and vice versa multiple times:

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

Figure 6.3: PoPs in Italy of Tiscali network
Available at https://business.tiscali.it/aziende/

This configuration is clearly not convenient since we are, at least, doubling the round-trip time. We would like to have a geographical area, in the middle between all these services allowing to terminate the traffic before, without entering and exiting the electricity provider network. A *Point Of Presence* might be the perfect place where to locate the critical services: it stays in the ISP network and there is a great number of PoPs spread in a geographical area. Therefore, it would be

the closest hop from any site of a distribution system area. For example, figure 6.3 shows a representation of the network of Tiscali, an Italian telecommunication operator. As it is possible to notice, there is a large number of PoPs located all over the national surface.

At this point it might be possible to consider a lower level PDC for each site of the power grid, collecting the data of the local PMUs. Their output stream, is then sent to the higher level PDC, located inside the ISP, where the ouput can be sent as input of the local state estimator or other applications performing data processing or storage of historical data. Figure 6.4 shows the resulting topology. In this case the output stream of the PDC exists from the private network of the site, reaching the Point of Presence where the PDC and the local estimator are placed. This allows to reduce the path followed by the traffic, and consequently even the transmission latency of the data. That is why we decided to physically move the services of the *area control center* to the PoP.

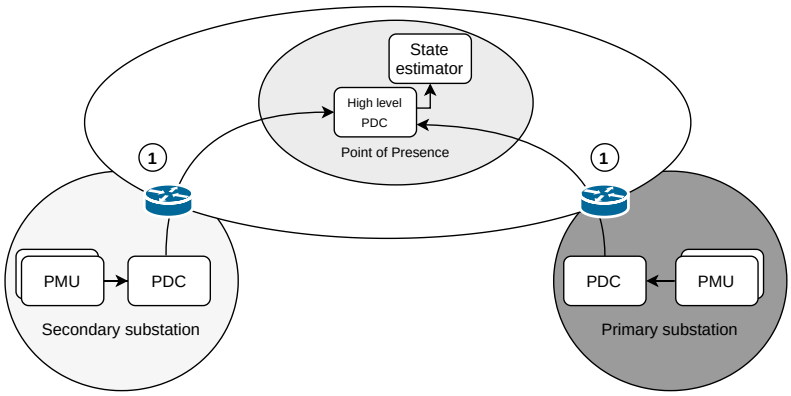


Figure 6.4: Data exits a single time from the electricity provider network reaching the PoP

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

6.2.2 A data-centric architecture

A smart grid might be seen as a huge distributed system, with devices and applications of different natures, producing, consuming, processing data and interacting in order to provide a cohrerent service, which is a *resilient power grid*. In this context a point-to-point interaction represents a too rigid approach, applications are difficult to be written, the interactions between the components is fixed and introducing new components or changing the existing ones could represent a problem [39]. The

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

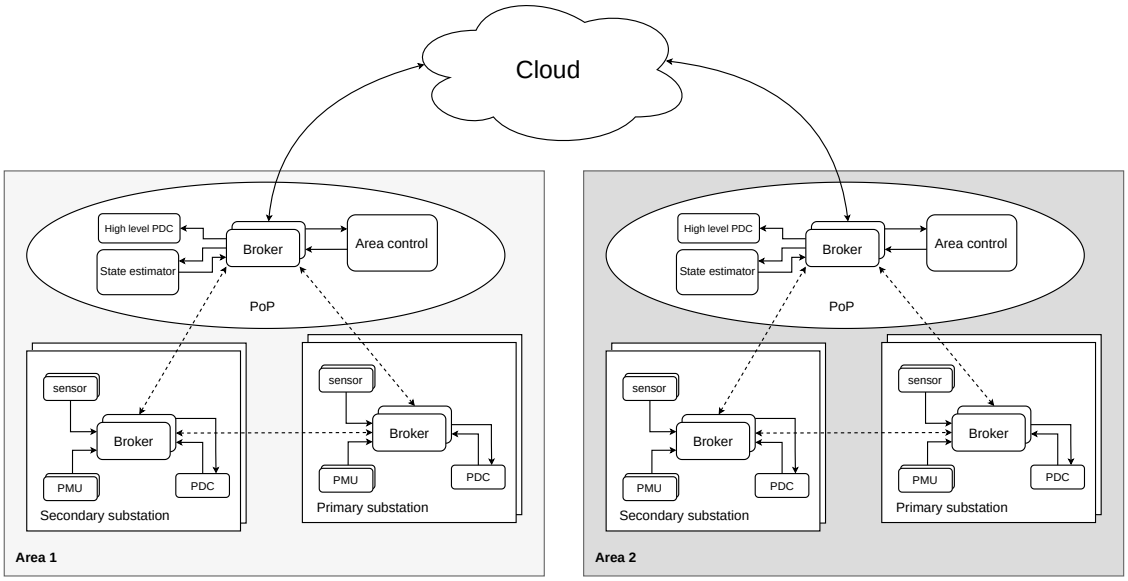


Figure 6.5: Distribution system as a data-centric architecture

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

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

Any measure, processed or aggregated data published in a broker inside a cluster, should be reachable from any broker belonging to the same cluster in a completely transparent way. This enables the services belonging to the same area, to freely exchange data, and to deploy new applications or data processing algorithms without changing what already exists. In some cases, different areas of the distribution

system might need to interact, for example, that's the case of the *inter control center* communication between the adjacent areas [48] or other data useful for offline analysis. Typically this kind of data does not have strict requirements in terms of latency. The idea is using services aimed to export a part of the data published in the local cluster of brokers, in some cases performing some aggregation or local processing, in order to reduce the dimension. Finally this data is sent to the cloud. From here, it is possible to perform further processing or aggregation, data can be stored, given to wide-area controllers, or delivered to other areas of the distribution system or to any other component of the power grid which might require it.

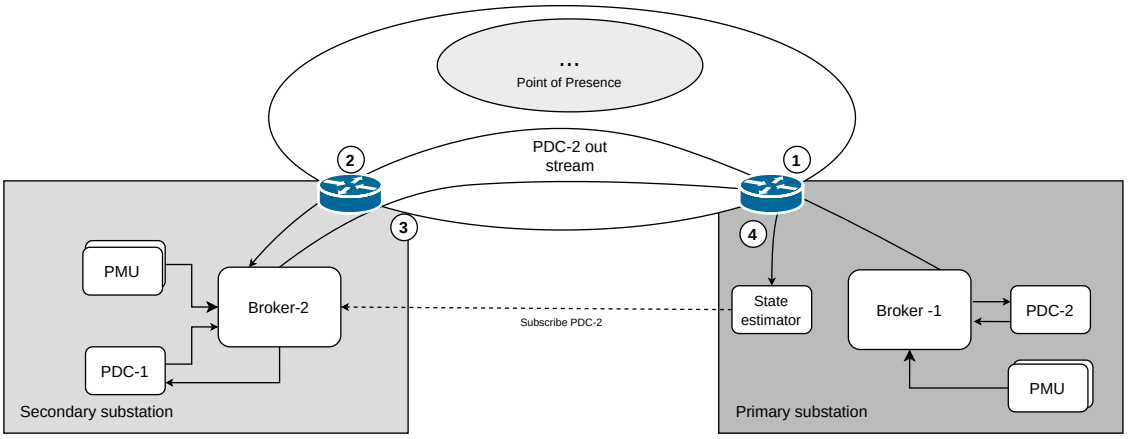


Figure 6.6: Unoptimized traffic caused by subscription to a wrong broker instance

Another aspect to take into consideration is how the load is balanced over the different instances of the brokers located over the area. We would like to be able to connect the client to the closest broker, this has great importance in terms of optimization of performance. For example, let's consider the case of a PDC placed inside a primary substation, this PDC produces the output stream which is sent to the an instance of the broker located in that primary substation. In the same location, the state estimator connects to a broker and subscribes to the output stream of the PDC. However, we are not managing where the clients should connect, therefore it connects to an instance of a broker in a secondary substation. Following the path of the data in figure 6.6, the broker located in the secondary substation should first retrieve the data from the instance of the broker in the primary substation, where the PDC is sending the data, and then it can forward it to the state estimator. This shows how selecting the right instance of the broker is crucial when latency matters, otherwise we might have completely unoptimized paths, with a negative impact on performance.

Chapter 7

Implementation

7.1 Implementation of the data-centric architecture

As we have already seen in chapter 6 the main idea is having a data-centric architecture where sensors and services are able to transparently produce and consume data without even knowing who is producing and consuming and what is their state. However, a smart grid consists of a huge amount of devices, many of them low-cost, with limited resource and computational power, sending data over networks with narrow bandwidth, such as NB-PLC, UMTS or GPRS [69]. This is a fundamental aspect to take into consideration when the infrastructure is designed, and technologies are chosen.

7.1.1 A cluster of MQTT brokers

The closer to the edge we are, the more is likely to find constrained devices. At this level, the aim is collecting the produced data and forward it to the destination, there is no need to persistently store data after it has been received by all the consumers which subscribed. MQTT is the protocol guaranteeing the best trade-off between lightness and reliability of the data, and among the TCP-based IoT protocols we took into consideration, it is the one with the lower data overhead, latencies and power consumption. That is why we decided to go to an MQTT-based solution. We analysed different implementations of MQTT brokers, among them:

- *Eclipse Mosquitto*: it is a really popular MQTT broker, counting more than 5K stars in GitHub. It is a C-based implementation compliant with versions 5, 3.1 and 3. Unfortunately, Mosquitto does not support clusters, which means that it is not possible to connect multiple Mosquitto broker instances so that when a client publishes on one of the instances, the subscribers connected to

all the others are able to receive that data. That is why we couldn't take into consideration this solution. [6]

- *EMQ X*: presented as really scalable solution, EMQ X is a distributed Erlang-based broker, not only supporting the MQTT protocol from version 3 to 5, but also the MQTT-SN, CoAP, LwM2M, WebSocket and STOMP protocols. While the EMQ X core is open-source, some of its functionalities are enterprise only, among them the possibility to persistently save messages, so that after a broker instance failure, messages do not go lost. [8]

- *Hive MQ*: it is a Java-based MQTT broker implementation. As the previous solution, the Hive MQ core is open-source, leaving many other functionalities to the enterprise version. That is the open-source version of Hive MQ does not support the cluster mode. However, among the solutions we considered, HiveMQ is the only one supporting the *queue mirroring*, the message replication over multiple instances over the broker, in order to improve the data availability. [11]

- *VerneMQ*: it is a completely open-source distributed Erlang-based MQTT compliant broker. VerneMQ supports a cluster of brokers and the persistent storage of messages so that it is possible to recover not delivered messages after a broker failure. [25]

	Mosquitto	**EMQ X**	**HiveMQ**	**VerneMQ**
Stars	5.1K	8.6K	649	2.5K
Contributors	111	72	20	39
Forks	1.7K	1.6K	181	303
Language	C	Erlang	Java	Erlang
Cluster mode	No	Yes	Yes (Enterprise)	Yes
Message persistency	No	Yes (Enterprise)	Yes	Yes

Table 7.1: Comparison between MQTT broker implementations.

We decided to select VerneMQ since it is the most complete open-source solution among the group we took into consideration.

The aim is being able to connect to a broker and access to all the data sent to any of the broker instances located in the same area. This can be achieved by deploying a wide-area VerneMQ cluster. This means that all the VerneMQ brokers of the area belong to the same cluster. However, it is important to notice that the concept of Kubernetes cluster and VerneMQ cluster are totally different, we are not obliged to have a VerneMQ cluster for each Kubernetes cluster, but we can have a VerneMQ cluster across many different Kubernetes clusters, what is important, is that each instance of the broker should be able to reach all the

others in order to synchronize. Each of the site belonging to the area can separatly deploy the VerneMQ instances, choosing the initial number of replicas and being able to independently scale them. All these instances will eventually belong to the VerneMQ cluster of their area. Once a VerneMQ broker starts, it can send a join to any of the brokers of the cluster, and it will be informed about all the other brokers belonging to that cluster [12].

A single MQTT broker cluster for each area allows to subscribe to any of the data produced inside the area. However, network partitionings should be taken into consideration, for example, a failure making a site not reachable from the rest of the area. The default VerneMQ behaviour is not suitable in this case, since when it detects the network partition, it will try to keep consistent all the replicas of the broker, preventing all the clients performing any action (connect, publish, subscribe, unsubscribe). That is during the time in which the cluster is splitted, no one in the area is able to exchange data, with obvious catastrophic outcomes. However, the default behaviour can be changed, it is possible to configure VerneMQ in order to keep the eventual consistency by enabling the following parameters: *allow_register_during_netsplit*, *allow_publish_during_netsplit*, *allow_subscribe_during_netsplit*, and *allow_unsubscribe_during_netsplit*. This means that when the VerneMQ cluster splits, all the partitions are able to go on performing all the actions, finally synchronizing with all the others once the connection is restored.

7.1.2 Brokers clustering and load balancing

A single cluster of brokers for each area provides a set of advantages in terms of data availability. However, there are some problems to challenge in order to enable this mechanism. They involve two aspects:

Inter-broker instances communication

With distributed brokers, each instance needs to directly interact with the others, in order to synchronize, exchange information about the subscription, and exchange the data required by the subscribers. However, each Kubernetes cluster independently deploys its set of broker instances, and, by default, pods running in different clusters are not able to talk directly. We found two possible solutions for this problem:

- **Exposing each broker instance with a service**: in this case, each broker instance is exposed to the outside cluster in order to make it reachable from all the other clusters. This can be achieved by creating and exposing a service for each pod of the StatefulSet. In order to support the autoscaling, it is possible to write a controller in charge of assigning a new service to the newly created pod of a StatefulSet set, and destroying it, when it is removed.

- **Liqo**: as a tool for sharing resources between different Kubernetes clusters, it allows network flattening, automatically setting up all the needed rules and the VPN tunnels, making reachable pods and services from different clusters. Since each pod is able to reach all the services of the federated cluster, it might be possible to use headless services, allowing to discover the existing broker instances without the need to expose each single instance of the broker to the outside world. However, something to take into consideration is that the traffic between two different clusters always passes through a WireGuard tunnel connecting the ones where the active *Liqo gateway* runs. This could represent a bottleneck if there is a great amount of traffic exchanged between the clusters.

  Unfortunately, we were not able to test this solution due to a bug of the virtual Kuberlet of Liqo. VerneMQ uses StatefulSet in order to launch the broker instances. Whenever Liqo forwards the request of pod creation to the child clusters, it adds a random number to the StatefulSet pod name, in order to avoid conflicts. However, when using the downward APIs in order to obtain the Pod name, the original name is returned, causing some problems with the DNS name resolution. At the moment of writing the Liqo team is working on this problem resolution. Therefore, the bug is going to be fixed in the following releases.

Exposing the broker to the outside world

As reported in chapter 6, each publisher and subscriber accessing the MQTT broker, should be redirected to the closest instance, in order to avoid an inefficient path of the traffic. This could be achieved by exposing a separate entrypoint for each site. Kubernetes allows the services running over the cluster to access the MQTT broker, providing load balancing between the running instances. However, since communication happens through the broker, sensors and devices outside the clusters need to access it, still keeping the same load over the running instances. Using a physical load balancer is not the solution due to the high cost of the device and the need to have a load balancer for each site. Therefore, the broker can be exposed via virtualized load balancers. Open-source solutions such as *Metallb* or *Purelb*, allow to expose a service with an external IP address, and balancing the load between the running instances of an application using the service load balancer. Both the solutions provide two working modes:

- **L2 load-balancing**: in this case, whenever a service of type load balancer is created, it receives an external IP address from the configured pool. Then, one of the physical nodes of the cluster is elected as the leader for that service. The leader is the node in charge of responding to the ARP requests related to the associated IP address. This allows service to be reachable from the LAN.

In any case, independently from the working mode, once the traffic reaches one of the node of the cluster, the kube-proxy will be in charge of redirecting the request to one of the running instances in the Kubernetes cluster. However, this solution has some limitations related to the fact that all the traffic directed to the exposed service always passes through its leader node. Since, in this case, the major part of the traffic of the network pass through the MQTT broker, this might becomes a serious bottleneck.

- **L3 load-balancing**: this solution exploits a routing protocol in order to announce the service IP addresses to the routers of the network, so making reachable all the exposed services. While Metallb supports BGP only, Purelb allows using any routing protocols for the distribution of the routes. However, this means that the routers of the network should be configured in order to perform *equal cost multipath routing*. In this case, the traffic for a service is not directed to a single physical node, but it is spread over the different nodes of the Kubernetes cluster, since the routers of the network perform the load balancing. However, this solution does not gracefully react to the changes of the nodes serving the requests. That's because there is the need to redirect the traffic belonging to the same session to the same node, and this is typically done by performing the hash of the 5-tuple (protocol, source ip, source port, dst ip and dst port) and, through it, selecting one of the nodes serving the requests. However, if one of the nodes goes down or a new node is introduced, even the way in which the hash is computed changes. Therefore, the traffic will be redirected to different nodes. This means that a great part of the active connections are interrupted, because if a node receives traffic from a TCP session, which is not in its TCP session table, a TCP RST message is sent back, closing the connection. In any case, this problem can be mitigated by using a resilient ECMP hashing algorithm, allowing to keep stable the assignments of the traffic, even after a change of the members serving the requests. [13, 18]

7.1.3 Organization of topics in the MQTT cluster

The MQTT protocol follows a *topic-based publish-subscribe pattern*, which means that data is labelled with topics, that can be used by subscribers in order to select the data they are interested in. The MQTT protocol uses hierarchical topics, allowing to perform fine-grained or large-grained subscriptions, thanks to the wildcards. For example, it is possible to perform a subscription to a sensor with id *sensor-id* by subscribing to `sensors/sensord-id`, if the application is interested in all the data produced by sensors, the wildcard allows to do that `sensors/#`.

Two types of wildcards can be used:

- **Single-level wildcard**: expressed with the `+` symbol, allows to replace an element from the topic name.

If, for example, the topic patten is *area/location/data-type*, a subscription to

`area-1/+/temperature`

allows to receive all the data about temperature from all the existing locations in area-1.

- **Multi-level wildcard**: while the single-level wildcard can be used multiple times, the multi-level one can be used only at the end of the topic, this allows the subscriber to receive all the data associated with a topic starting with all there is before the wildcard. A subscription to:

  `area-1/location-1/#`

  allows to receive all the data produced in *location-1* inside the *area-1*. [16]

A good design of the hierarchical structure of the topics allows the clients to receive only what is needed, reducing the used bandwidth and allowing the subscribers to receive only the data they need.

The architecture of the distribution system consists of many different sites, in each, different sensors and services produce some data. This information can be reported in the topic, in order to be able to select the location of the area from which the subscribers wants to extract data. The pattern used by producers for topics in this context will be:

`{area}/{location-type}/{location-id}/{data-type}/{producer-id}`

For example, let's consider a PMU and a PDC exchanging synchrophasors with the IEEE C37.118 protocol over MQTT. The PMU is located in a secondary station with id `1156` and it has ID `765`. The PDC will perform a subscription to the topic: `south-italy-113/secondary-station/1156/PMU/765`. However, the PDC might require all the data produced by PMU in that secondary station subscribing to `south-italy-113/secondary-station/1156/PMU/#`.
Let's now consider a high-level PDC located in the PoP of the same area, it is able to subscribe to the output data stream of the PDC located over the area, by performing a subscription to the topic `south-italy-113/+/+/PDC/#`.

7.1.4 Data processing with Kafka

The distributed MQTT broker allows to collect data from producers at the edge of the power system and provides it to all the services requiring real-time data, which are close to the data, for example, inside one of the sites of the power system. However, some of this data require a higher level of processing, data-aggregation and, then to be stored in operational databases. This can still be achieved using

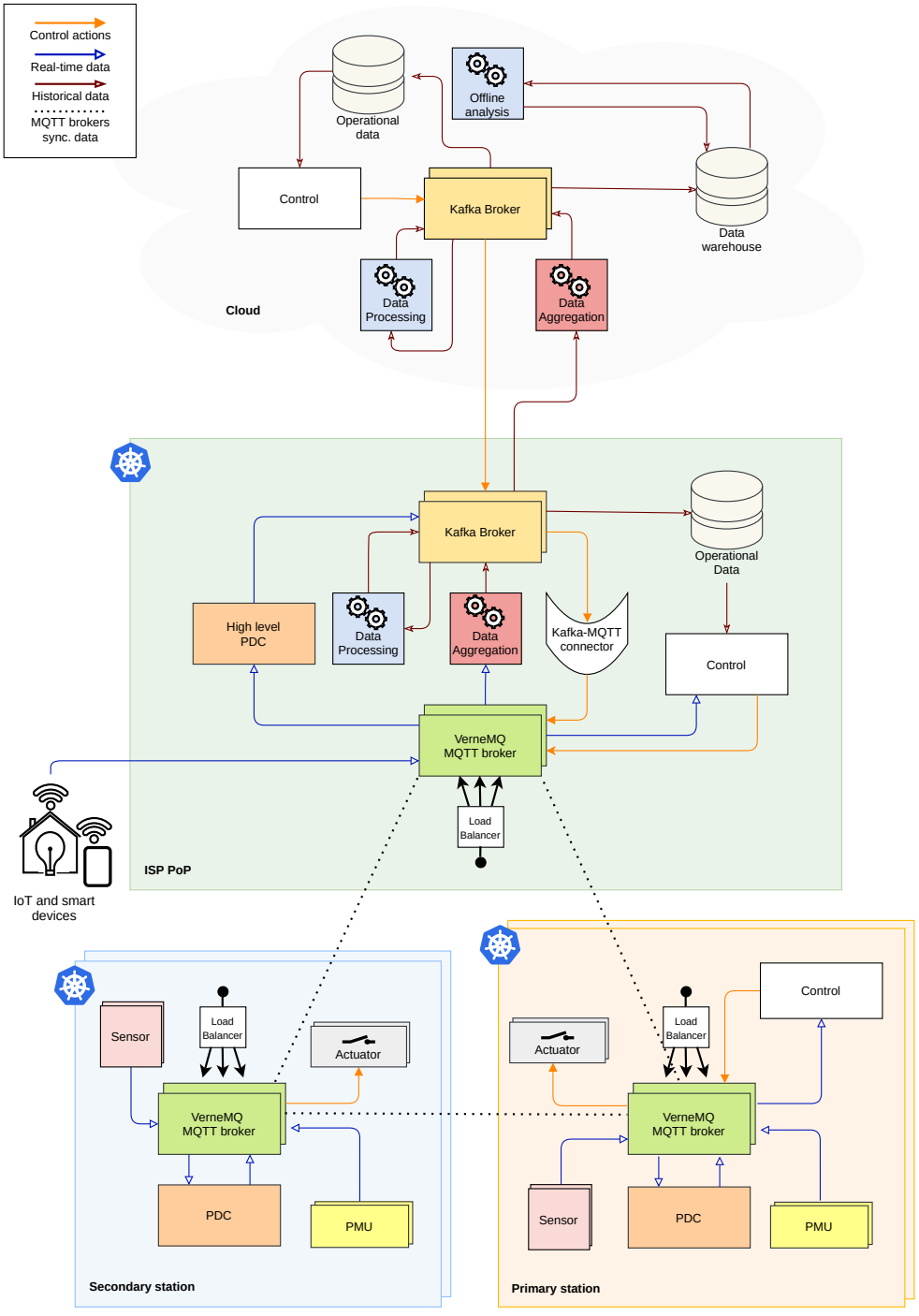


Figure 7.1: Imeplemented architecture: the distributed broker collect data from the edge, which is then processed in Kafka

the MQTT broker. However, a more natural way to do so is by using an *event streaming platform*. Kafka is a high scalable solution allowing to ingest the great amount of data coming from the different sites of the area, and it can be integrated with a *Stream data processing systems* such as *Storm*, *Spark Streaming*, *Samnza*, *Flink*, *Kafka streams* [66]. These solutions allow to perform data processing on the fly, by constructing data processing pipelines, whose output can be reinjected to Kafka, from which applications can consume the processed data, showing it in dashboards, performing control operation, or it possible to store the result in a database. This kind of processing should be performed where there is available computational power and the closest place from all the sites of the area, where the data is produced. This place is the PoP of the ISP, as seen in section 6.2.1, it is the closet place from all the data of the area, PoPs indeed are widespread in the territory, and it is a place where it is possible to have some computational power available, given by general-purpose servers.

Another Kafka cluster can be placed in Cloud, where it is possible to import the data coming from all the areas, and perform further processing and data aggregation. Here data can be stored for offline analysis, or, for example, it can be consumed by applications providing a higher level of control.

The deployment of a Kafka cluster in Kubernetes is simplified by *Strimzi*, a set of operators and CRDs allowing the control of Kafka clusters by means of the Kubernetes APIs, but also automatizing some not trivial procedures (such as security configuration) [26, 19]. Strimizi allows to explicit the Kafka configuration with a declarative approach, improving the maintainability and the portability of the configuration, since replicating the same deployment or performing some changes to the existing ones can be done by applying the yaml files containing the configurations. The Strimzi operators will be in charge of applying all the required changes.

7.2 Implementation of a solution integrating IEEE C37.118 and MQTT

As seen in chapter 4, current implementations of PDCs and PMUs do not support any publish subscribe protocol. The idea is to implement a version of the adapters described by Hoefling et all. [42], supporting MQTT and the commanded mode, since it is the most widely supported one. Adapters have been implemented in python, using a python implementation of the IEEE C37.118 protocol [63] and *Eclipse Paho* [7] as MQTT client library .

We developed two different components in order to allow the interaction with the MQTT broker:

- **PMU-adapter**: This component stays between PMUs and broker, it acts as PDC, asking the PMU for its data. Once the PMU adapter is able to

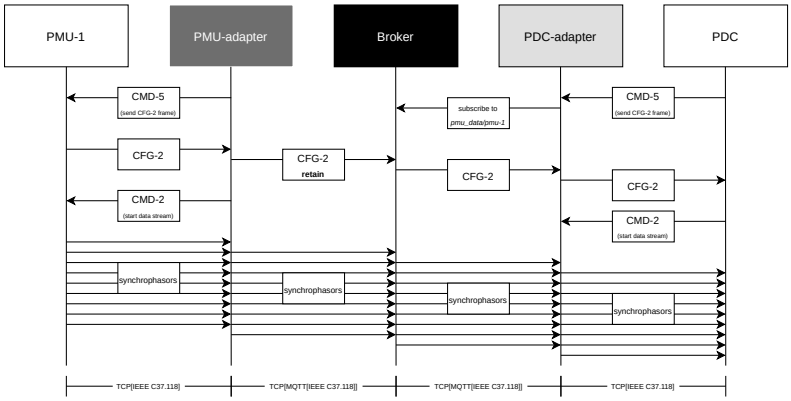


Figure 7.2: Schema of the interaction between adapters and broker

receive the data from the PMU, it can forward it to a topic in the MQTT broker (by default the topic is `pmu_data/pmu-N` where N is the PMU ID). At the moment each PMU adapter can be configured in order to set up a TCP connection with a single PMU instance, therefore a PMU-adapter per PMU is needed. Whenever the PDC requires to start the synchrophasors data stream, it needs to receive a CGF-2 frame, so that it can be able to parse the received synchrophasors. This can be achieved by instructing the PMU-adapter to publish a CFG-2 with the *retain* flag every time it receives a new one or it connects to the MQTT broker, so that the PDC is able to receive the CFG-2 frame as soon as it performs the subscription to the broker.

- **PDC-adapter**: Once synchrophasors are published on the topic of the MQTT broker, a PDC needs to consume this data. It leverages a set of PDC adapters in order to consume the data from PMUs. The PDC adapter, indeed, acts as a PMU, so that the PDC can be configured in order to create a TCP connection with the adapter, instead of the real PMU, and it will provide all the synchrophasors published by the PMU in the related topic. This implementation requires an instance of the PDC-adapter for each PMU we want to connect.

We checked the validity of the implementation using *PMU connection tester* [9], a tool allowing the validation of the compliance of phasor measurement devices. We connected the PDC-adapter to the PMU connection tester, receiving the data forwarded to the broker by a PMU-adapter. It was also possible to interact with the adapter as if it was a PMU, controlling the data stream, by sending the *CMD-1* and *CMD-2* command frames, and requiring the header frame or a specific configuration frames.

After the validation of the correctness of the implementation, we tested the capability of this architecture to restore the messages when the connection between the PDC and the PMU is interrupted. In order to test this scenario we used

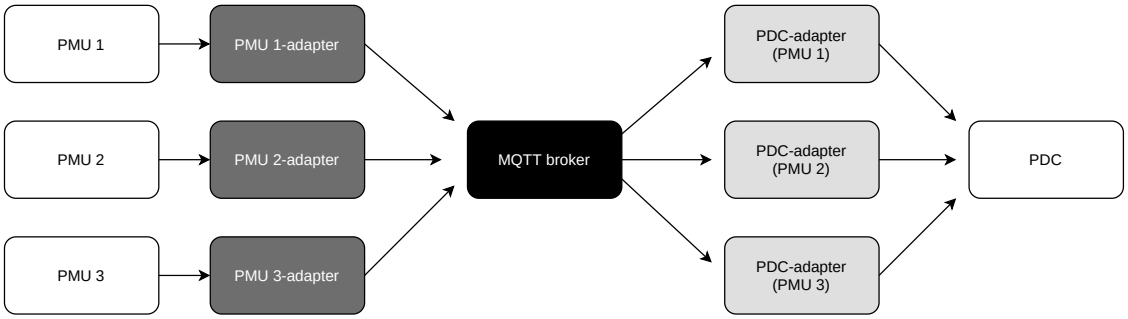


Figure 7.3: Example of a deployment of the adapters

a K3s deployment, represented in figure 7.4, with an *OpenPDC* instance and a PMU simulator exchanging data through the *VerneMQ* mqtt broker and the proper adapters. We simulated a network partition via *Chaos Mesh* [17], making the broker unreachable from the PDC-adapter. After a minute, the connection was restored, and the PDC-adapter was able again to connect to the broker. As soon as it reconnected, the broker delivered all the messages produced by the PMU during the downtime.

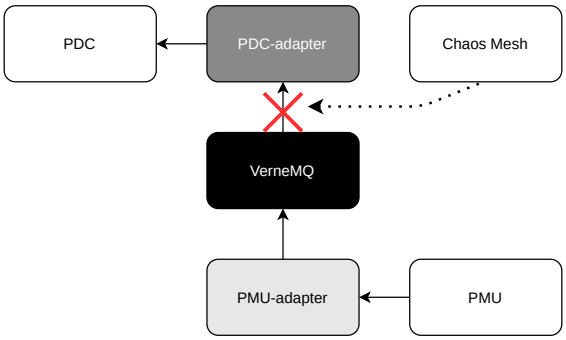


Figure 7.4: Message recovery experiment scenario

The figure 7.5 shows the message rate from the moment of the reconnection. It is possible to notice the high message rate when the adapter reconnects to the broker, going down to the 25 msg/s (2.5 msg/100ms) after about 500ms. This means that before being able to receive all the previous messages and start receiving only real-time data, 500 ms were needed. This might be the evidence that restoring the messages is not enough, because:

1. When the connection with the broker is restored, the PDC might be overwhelmed by the messages received during the downtime.

2. Previous messages delay the real-time data.

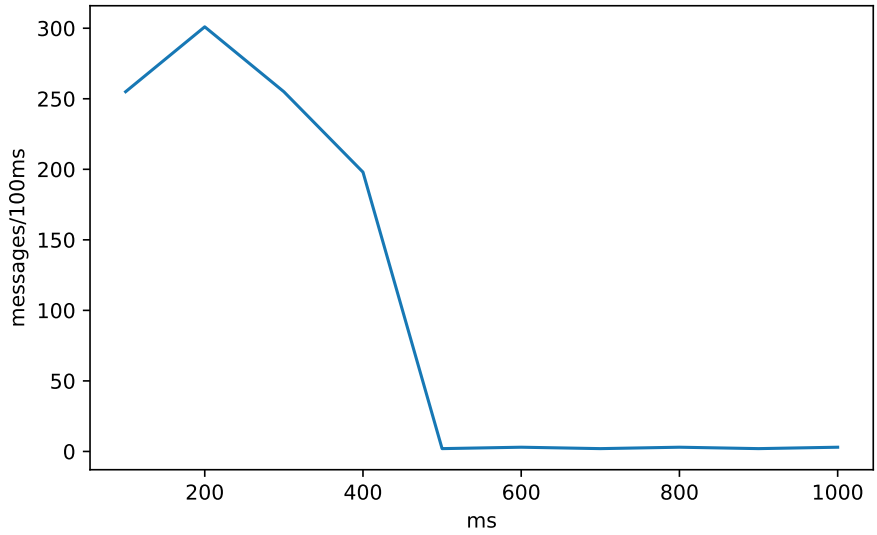


Figure 7.5: Message rate from connection restore after 1 minute of disconnection from the broker

Therefore, smarter strategies should be used in order to be able to restore all the messages received during the downtime and to handle the real-time data with the right timing.

7.2.1 Latency overhead to respect TCP

The introduction of an intermediary broker, as previously seen, provides a set of benefits, but also the introduction of an additional overhead due to the data processing time of the broker, and the fact that, in the case of MQTT, data traverses two different TCP connections, one from the PMU to the broker, and the other from the broker to the PDC. Moreover, if we also consider the adapters, the TCP connections traversed by the data become four.

Figure 7.6, shows the message latency of IEEE C37.118 data considering three different cases, represented in figure 7.7:

1. Direct connection between PMU and PDC through a TCP channel,
2. Connection of a PMU and a PDC supporting the MQTT protocol, therefore **without any adapters in between**. We measured this latency as the enlapsed time between the message sent by the PMU-adapter and its delivery to the PDC-adapter, using QoS 1.
3. Connection of a PMU and a PDC connected to the broker through the adapters we implemented, even in this case we used QoS 1.

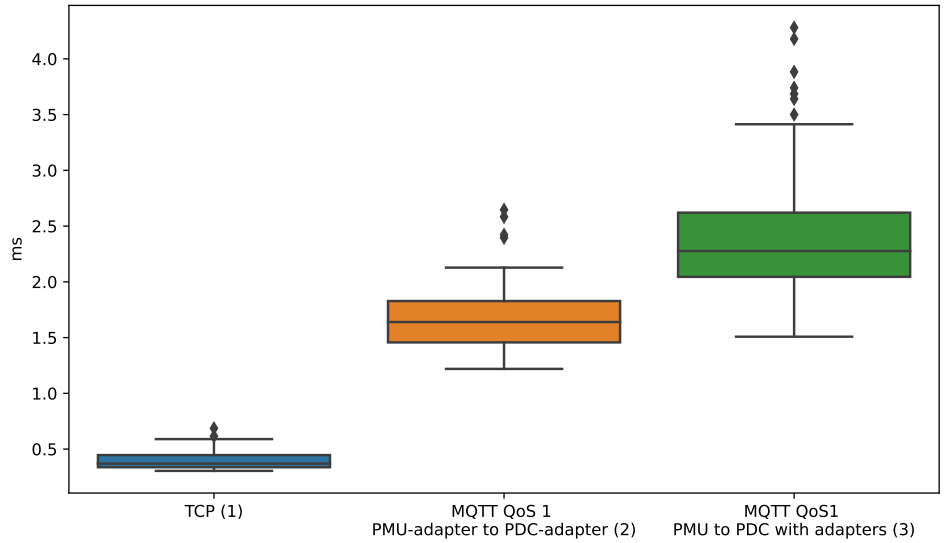


Figure 7.6: Message latency of IEEE C37.118 data transported over TCP and MQTT QoS 1

The figure shows a difference of 1/1,5ms between the C37.118 over TCP-only (*case 1*) and the case without any adapter in between (*case 2*). This difference increases of less than 1ms in most cases, when we consider the additional overhead given by the introduction of the adapters for connecting the PMU and the PDC to the broker (*case-3*). This additional time represents the time needed for a message to reach the PMU-adapter from the PMU, and to reach the PDC from the PDC adapter. On the whole, the latency introduced by the MQTT protocol and the adapters does not seem to be relevant in most cases.

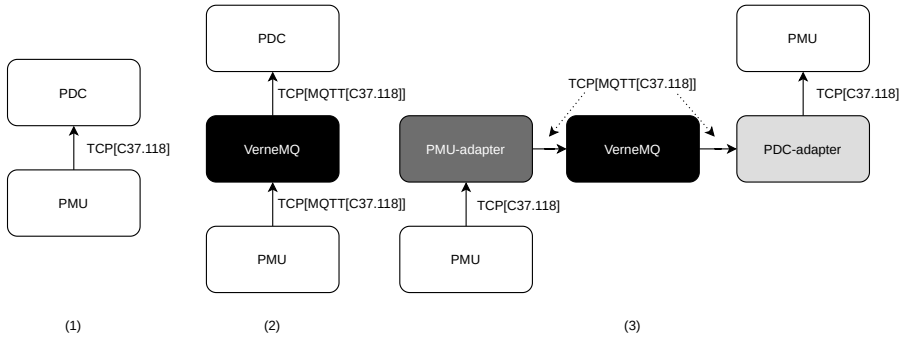


Figure 7.7: Latency experiment scenarios

Chapter 8

Conclusions

The world is moving forward renewable sources of energy, more resilient and smart grids are needed, and this can be achieved only being able to collect the data from the physical world, processing it for real-time control or offline analysis. Data is a crucial aspect, since not only provides the observability of the power grid but it enables technologies, such as artificial intelligence, requiring a huge amount of data. Thanks to the self-learning and the capability to evaluate great collection of data, the AI allows to forecast the power load, analyse the consumers' behaviour, forecast the renewable energy production, power system fault diagnosis and forecasting [46]. However, these features can only be provided if all the data produced by sensors, devices, services across the smart grid are able to safely reach all the consumers, in compliance with their requirements in terms of QoS. A smart grid can be seen as a great distributed system located in a huge geographical area, where devices of different nature, some of them with limited computational power, exchange huge quantity of data over different physical media. In this context, synchronous point-to-point interaction is a too strict approach, since it makes really difficult to handle the interaction between producers and consumers, and the solution should cope with services and devices moving across the grid, for example, changing the IP address. The publish-subscribe pattern seem to simplify the interaction between producers and consumers, since they do not have to directly know each other and the interaction is not synchronous. Therefore, the interacting parts are able to independently produce and consume data. This lead to a *data-centric* architecture, where producers and consumers do not have to concern about IP addresses, but they deal only with data, simplifying the way in which applications are written, improving the maintainability of the applications and the availability of the data. Since anyone is able to listen to the data produced by consumers, integrating new applications or processing algorithms is simpler and might not require any modification to the rest of the system. The MQTT protocol is the best compromise between the current IoT protocols, in terms of data overhead, reliability, performance and power consumption. However, MQTT relies on TCP, and it is

widely known that, in lossy networks, the congestion control have a heavy impact on performance and end-to-end latency. The solution might be using MQTT-SN, in these contexts, a version UDP-based version of MQTT designed for lossy networks. However with UDP it is not possible to provide reliability, unless it is done at the application level, and the absence of congestion control could lead to packet loss and unnecessary power consumption [65]. The performance overhead given by congestion control and connection establishment could be limited by adopting *QUIC*, an application-based alternative of TCP, using UDP under the hood [40]. There is a good number of researches about the deployment of QUIC in IoT environments. Liri et al. [54] showed how QUIC performance in lossy environments can be compared to MQTT-SN. Kumar and Behnam [52] showed how MQTT over QUIC performs much better than the solution using TPC. It reduces the amount of exchanged packets, the connection overhead, the CPU and memory usage, solves the problem of the head-of-line blocking and reduces the dropped packets in case of connection migration. Further investigations could be carried out in order to understand the benefits and the shortcomings of QUIC in the power grid environments, where both the reliability given by TCP and the better performance and lower resource consumption of UDP are needed.

We showed how the publish-subscribe pattern allows to decouple PMUs and PDCs, to improve the availability of the data, since it is possible to write applications consuming the data pushed by PMUs and PDCs into the broker. Moreover, the broker prevents data loss, allowing the subscribers to recover the data that has been produced while it was down. Even though that data might not be used for real-time applications, since they could be old, this data can be used for offline analsys, as showed by MacIver et al. [55] with the Great Britain incident of 2019. The python implementation of the Hoefling et al. [42] adapters integrating the IEEE C37.118 standard and MQTT, shows how it is possible to recover messages after a PDC downtime, but cleaver strategies should be deployed in order to restore the previous messages, handling the real-time data with the right timing, without any delay. The performance overhead given by the usage of the MQTT brokers and the adapters remain still acceptable, with an increase of latency of about 1.5/2ms, going down to 1/1.5ms considering PMUs and PDCs supporting the MQTT protocol. The presented adapters are not a solutiton to be deployed in production, since the protocol should be directly supported by devices, but it is a way to understand the benefits and possible shortcoming given by the adoption of the MQTT protocol and a publish-subscribe pattern in this context. However, with the proper optimization, it might be a good instrument supporting the transition to the new communication pattern.

The broker represents the center of communication in our architecture, this means that it could represent a single point of failure, as all the messages pass through it. The reliability of the data, when are in travel, is guaranteed by TCP

and the MQTT protocol. However, even the durability of the data should be provided, so that when the broker receives the messages, they are safely delivered to all the subscribers requiring it. We saw that the persistency of the storage is a way to reduce message loss, but in order to recover them, a full broker restart is needed, requiring tens of seconds. Moreover, it does not completely solve the problem of loss, since bugs or misconfiguration of the broker could be a cause of messages drop. Another crucial aspect is scalability. The broker should be able to support the load given by all the exchanged data. Using a distributed solution, such as VerneMQ, HiveMQ or EMQX, allows to support larger amount of messages, since the load is splitted over the different replicas of the broker, but still giving the possibility to publish and receive the same message performing a subscription to any other instance of the broker. Many investigations underline both the vertical and horizontal scalability of these solutions [30, 37, 51], allowing even millions of clients sending and receiving messages. The Kubernetes autoscaling is another instrument allowing to handle some load variation on the distributed broker, but it is effective only if the increase of loads depends on an increase of the number of clients, since connections cannot be moved to the newly launched instances. Another aspect to be taken into consideration is a proper broker configuration. Without the correct tuning of the parameters, there might be unwanted messages drop or an increase of latencies. For example, the VerneMQ distributed broker allows to set the value of `MAX_OFFLINE_MESSAGES` or `MAX_ONLINE_MESSAGES`, which is respectively the maximum number of messages in the queue, waiting to be delivered to offline clients or to online clients. These values allow to keep under control the memory consumption, the pressure on the broker and, in the case of offline messages, to discard messages that will never be delivered to an offline client, since it will never reconnect. Unfortunately, if these parameters are too low, we risk too many useful messages discarding. The value of `MAX_INFLIGHT_MESSAGES`, instead, controls the maximum number of messages, with QoS > 0, which can be *inflight*, which means all that messages sent but not yet confirmed. This value avoids subscribers overwhelming when configured in the broker, and to protect the broker when this value is configured in the MQTT client. A too low value of MAX_INFLIGHT_MESSAGES might cause some unwanted delays, since the message publishing is continuously interrupted, while, with higher values, it is possible to increase the throughput but with the risk of receivers saturation.

The presented architecture has been designed in order to achieve the goal of data-centric architecture, where services, sensors and devices should only concern about data, but keeping into consideration the huge scale of the power system. That is the reason why we decided to have a VerneMQ cluster for each area of the power system. Even though each Kubernetes cluster separately deploys the brokers, they can join the same VerneMQ cluster, and they can exchange subscriptions and messages. Even though all the brokers of the area belongs to the same VerneMQ cluster, the brokers of the same site are separately exposed, so that each client is

able to connect to the closest instance of the broker. The communication between the single broker instances can be transparently enabled by Liqo, this unfortunately do not work at the moment of writing, due to a Liqo bug preventing the VerneMQ autoclustering. This bug is going to be fixed, but a workaround could be creating and exposing a service for each broker to the outside of the cluster, or making the services accessible via Liqo from the inside cluster. We already analysed the scalability of the distributed MQTT brokers. Further analysis could be performed in terms of broker parameters tuning, with the aim of optimizing performance and efficiency in this large scale deployment. Moreover, this thesis work focused on the application of synchrophasors exchange, in order to improve the observability of the power system. Further investigations could be performed focusing on how, in the context of power system and smart grid 2.0, other use cases fit with the presented architecture.

Bibliography

[1] 2020 tied for warmest year on record, nasa analysis shows | nasa. https://www.nasa.gov/press-release/2020-tied-for-warmest-year-on-record-nasa-analysis-shows. (Accessed on 09/28/2021).

[2] AMQP 0-9-1 Model Explained — RabbitMQ. https://www.rabbitmq.com/tutorials/amqp-concepts.html. (Accessed on 08/19/2021).

[3] Apache kafka. https://kafka.apache.org/intro. (Accessed on 28 Aug. 2021).

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

[5] Core stanzas - slixmpp. https://slixmpp.readthedocs.io/en/latest/api/stanza/. (Accessed on 20 Aug. 2021).

[6] eclipse/mosquitto: Eclipse mosquitto - an open source mqtt broker. https://github.com/eclipse/mosquitto.

[7] eclipse/paho.mqtt.python: paho.mqtt.python. https://github.com/eclipse/paho.mqtt.python.

[8] emqx/emqx: An open-source, cloud-native, distributed mqtt message broker for iot. https://github.com/emqx/emqx.

[9] Gridprotectionalliance/pmuconnectiontester: Verifies data streams from synchrophasor measurement devices. https://github.com/GridProtectionAlliance/PMUConnectionTester.

[10] Hivemq cluster :: Hivemq documentation. https://www.hivemq.com/docs/hivemq/4.6/user-guide/cluster.html#replication. (Accessed on 08 Sep. 2021).

[11] hivemq/hivemq-community-edition: Hivemq ce is a java-based open source mqtt broker that fully supports mqtt 3.x and mqtt 5. it is the foundation of the hivemq enterprise connectivity and messaging platform. https://github.com/hivemq/hivemq-community-edition.

[12] Introduction - vernemq. https://docs.vernemq.com/clustering/introduction. (Accessed on 09/19/2021).

[13] Metallb, bare metal load-balancer for kubernetes. https://metallb.universe.tf/concepts/. (Accessed on 09/21/2021).

[14] Mqtt and kafka. how to combine two complementary. https://medium.com/python-point/mqtt-and-kafka-8e470eff606b. (Accessed on 28 Aug. 2021).

[15] Mqtt: The standard for iot messaging. https://mqtt.org/. (Accessed on 08/19/2021).

[16] Mqtt topics & best practices - mqtt essentials: Part 5. https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/. (Accessed on 09/19/2021).

[17] A powerful chaos engineering platform for kubernetes | chaos mesh®. https://chaos-mesh.org/.

[18] Purelb documentation. https://purelb.gitlab.io/docs/how_it_works/. (Accessed on 09/22/2021).

[19] Strimzi overview guide (0.25.0). https://strimzi.io/docs/operators/latest/overview.html. (Accessed on 09/23/2021).

[20] Sttp draft specification. https://github.com/sttp/Specification/raw/master/Output/sttp-specification.pdf. (Accessed on 08/29/2021).

[21] sttp/connection-tester: Sttp connection tester. https://github.com/sttp/connection-tester. (Accessed on 08/29/2021).

[22] sttp/cppapi: Native c++ api for sttp. https://github.com/sttp/cppapi. (Accessed on 08/29/2021).

[23] sttp.net 1.0.11. https://www.nuget.org/packages/sttp.net/. (Accessed on 08/29/2021).

[24] sttp/net-cppapi: Wrapped .net target apis for sttp. https://github.com/sttp/net-cppapi. (Accessed on 08/29/2021).

[25] vernemq/vernemq: A distributed mqtt message broker based on erlang/otp. built for high quality & industrial use cases. https://github.com/vernemq/vernemq.

[26] Why run apache kafka on kubernetes? https://www.redhat.com/en/topics/integration/why-run-apache-kafka-on-kubernetes. (Accessed on 09/23/2021).

[27] Wikipedia - sottostazione elettrica. https://it.wikipedia.org/wiki/Sottostazione_elettrica. (Accessed on 10 Sep. 2021).

[28] Xmpp | an overview of xmpp. https://xmpp.org/about/technology-overview.html. (Accessed on 20 Aug. 2021).

[29] Mqtt version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, 12 2015. (Accessed on 20 Aug. 2021).

[30] 10,000,000 mqtt clients, hivemq cluster benchmark paper, 2017.

[31] Zoran B. Babovic, Jelica Protic, and Veljko Milutinovic. Web Performance Evaluation for Internet of Things Applications. *IEEE Access*, 4:6974–6992, 2016.

[32] Soma Bandyopadhyay and Abhijan Bhattacharyya. Lightweight internet protocols for web enablement of sensors using constrained gateway devices. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 334–340, 2013.

[33] R. Bayindir, I. Colak, G. Fulli, and K. Demirtas. Smart grid technologies and applications. *Renewable and Sustainable Energy Reviews*, 66:499–516, 2016.

[34] Dale Lane Callum Jackson, Kate Stanley. Why enterprise messaging and event streaming are different – ibm developer. `https://developer.ibm.com/articles/difference-between-events-and-messages/`, 2020. (Accessed on 27 Aug. /2021).

[35] Junwei Cao and Mingbo Yang. Energy internet – towards smart grid 2.0. In *2013 Fourth International Conference on Networking and Distributed Computing*, pages 105–110, 2013.

[36] J. Ritchie Carroll and F. Russell Robertson. A COMPARISON OF PHASOR COMMUNICATIONS PROTOCOLS. Technical Report PNNL-28499, 1504742, March 2019.

[37] Dylan O'Mahony Dave Doyle. Kubecon + cloudnativecon north america 2018: Our journey to service 5 million messaging connections on kubernetes. `https://kccna18.sched.com/event/GrRy`. (Accessed on 10 Sep. 2021).

[38] E-distribuzione. Piano di sviluppo 2020-2022 (development plan 2020-2022). `https://www.e-distribuzione.it/content/dam/e-distribuzione/documenti/e-distribuzione/Piano_di_Sviluppo_2020_22_30giu2020.pdf`, 2020.

[39] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[40] Fátima Fernández, Mihail Zverev, Pablo Garrido, José R. Juárez, Josu Bilbao, and Ramón Agüero. And quic meets iot: performance assessment of mqtt over quic. In *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–6, 2020.

[41] Sten Gruener, Heiko Koziolek, and Julius Rückert. Towards resilient iot messaging: An experience report analyzing mqtt brokers. In *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, pages 69–79, 2021.

[42] Michael Hoefling, Florian Heimgaertner, Daniel Fuchs, Michael Menth, Paolo Romano, Teklemariam Tesfay, Mario Paolone, Jimmie Adolph, and Vidar Gronas. Integration of IEEE C37.118 and publish/subscribe communication. In *2015 IEEE International Conference on Communications (ICC)*, pages 764–769, London, June 2015. IEEE.

[43] Åsmund Hugo, Brice Morin, and Karl Svantorp. Bridging mqtt and kafka to support c-its: a feasibility study. In *2020 21st IEEE International Conference on Mobile Data Management (MDM)*, pages 371–376, 2020.

[44] IRENA. Innovation outlook: Renewable mini-grids, 2016.

[45] Nick Jenkins, Janaka Ekanayake, and Goran Strbac. *Distributed Generation*. Institution of Engineering and Technology, January 2010.

[46] Jian Jiao. Application and prospect of artificial intelligence in smart grid. *IOP Conference Series: Earth and Environmental Science*, 510:022012, jul 2020.

[47] Paul Lysak Kai Waehner. Confluent, mqtt, and apache kafka power real-time iot use cases. `https://www.confluent.io/blog/iot-streaming-use-cases-with-kafka-mqtt-confluent-and-waterstream/`, 2020. (Accessed on 28 Aug. 2021).

[48] Prashant Kansal and Anjan Bose. Bandwidth and latency requirements for smart transmission grid applications. *IEEE Transactions on Smart Grid*, 3(3):1344–1352, 2012.

[49] 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)*, pages 392–397, Venice, Italy, November 2014. IEEE.

[50] Michael Koster, Ari Keränen, and Jaime Jimenez. Publish-Subscribe Broker for the Constrained Application Protocol (CoAP). Internet-Draft draft-ietf-core-CoAP-pubsub-09, Internet Engineering Task Force, September 2019. Work in Progress.

[51] Heiko Koziolek, Sten Grüner, and Julius Rückert. A comparison of mqtt brokers for distributed iot edge computing. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, pages 352–368, Cham, 2020. Springer International Publishing.

[52] Puneet Kumar and Behnam Dezfouli. Implementation and analysis of QUIC for MQTT. *CoRR*, abs/1810.07730, 2018.

[53] Legambiente. Il clima é giá cambiato. Technical report, 11 2020.

[54] Elizabeth Liri, Prateek Kumar Singh, Abdulrahman BIN Rabiah, Koushik Kar, Kiran Makhijani, and K.K. Ramakrishnan. Robustness of iot application protocols to network impairments. In *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 97–103, 2018.

[55] Callum MacIver, Keith Bell, and Marcel Nedd. An analysis of the august 9th 2019 gb transmission system frequency incident. *Electric Power Systems Research*, 199:107444, 2021.

[56] K. E. Martin, G. Brunello, M. G. Adamiak, G. Antonova, M. Begovic, G. Benmouyal, P. D. Bui, H. Falk, V. Gharpure, A. Goldstein, Y. Hu, C. Huntley, T. Kase, M. Kezunovic, A. Kulshrestha, Y. Lu, R. Midence, J. Murphy, M. Patel, F. Rahmatian, V. Skendzic, B. Vandiver, and A. Zahid. An overview of the ieee standard c37.118.2—synchrophasor data transfer for power systems. *IEEE Transactions on Smart Grid*, 5(4):1980–1984, 2014.

[57] Nitin Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In *2017 IEEE International Systems Engineering Symposium*

(ISSE), pages 1–7, 2017.

[58] Ozgur Ozturk. Introduction to xmpp protocol and developing online collaboration applications using open source software and libraries. In *2010 International Symposium on Collaborative Technologies and Systems*. IEEE, 2010.

[59] Peter Saint-Andre and Joe Hildebrand. Message delivery receipts. `https://xmpp.org/extensions/xep-0184.html`.

[60] 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)*, pages 22–30, 2019.

[61] Terna. Italian national grid. `https://www.terna.it/en/about-us/business/italian-national-grid`. (Accessed on 10 Sep. 2021).

[62] Terna. Piano di sviluppo 2021 (development plan 2021). `https://download.terna.it/terna/Piano_Sviluppo_2021_8d94126f94dc233.pdf`, 2021.

[63] Stevan Sandi Tomo Popovic, Bozo Krstajic. pypmu - python implementation of the ieee c37.118 synchrophasor standard. `https://github.com/iicsys/pypmu`.

[64] Peter Waher. Quality of service. `https://xmpp.org/extensions/inbox/qos.html`.

[65] Chonggang Wang, K. Sohraby, Bo Li, M. Daneshmand, and Yueming Hu. A survey of transport protocols for wireless sensor networks. *IEEE Network*, 20(3):34–40, 2006.

[66] Han Wu. *Performance and Reliability Evaluation of Apache Kafka Messaging System*. PhD thesis, Freien Universität Berlin, 2020.

[67] Xuandong Xiong and Jiandan Fu. Active status certificate publish and subscribe based on amqp. In *2011 International Conference on Computational and Information Sciences*. IEEE, Oct 2011.

[68] Sheng Yang. Performance and scalability report for longhorn v1.0 | the longhorn blog. `https://longhorn.io/blog/performance-scalability-report-aug-2020/`, 2020. (Accessed on 10/10/2021).

[69] Agustin Zaballos, Alex Vallejo, and Josep Selga. Heterogeneous communication architecture for the smart grid. *IEEE Network*, 25(5):30–37, September 2011.