ALMA MATER STUDIORUM

UNIVERSITÀ DI BOLOGNA

Corso di laurea magistrale in

INGEGNERIA INFORMATICA

Sviluppo, Deployment e Validazione Sperimentale di Architetture Distribuite di Machine Learning su Piattaforma fog05

Tesi di laurea in

Sistemi Distribuiti M

Relatore

Chiar.mo Prof. Paolo Bellavista

Presentata da

Davide Di Donato

Sessione Unica

Anno Accademico 2018/2019

Summary

1. Introduction

This master thesis aims to evaluate pros and cons of different distributed architectures for machine learning in IIoT (Industrial Internet of Things), exploiting emerging research concepts such as fog and edge computing (that we will explore in a moment).

In fact, what has become clear, during these years, is the need, in certain use cases more frequently encountered in IIoT (when real-time or near-real-time performances are usually requested), of an architecture that could lower the minimum latency [2] present in classic cloud-centric architectures, in order to achieve more reliability and performance. This happens because, as efficient as clouds and connections can be, they can never be able to compensate the inducted latency they establish once they are deployed, and sometimes, it is more than what we are willing to accept for our IIoT solution. This is for sure the main problem, but it is not the only one we can identify in using cloud-centric solutions. Another non-negligible assumption about using the cloud in a classic way involves connectivity [2]: it is not so trivial to expect a connection to be always active and ready to be used, as it could be expensive to maintain an active link and/or difficult to remain in the covering range of an AP (Access Point) all the time. Also, given that we have constant access to the network, we would need to worry about having sufficient bandwidth to communicate whenever it is needed. And lastly, we have to keep in mind that not every company wants to entrust a public cloud service with its own private data. In fact, the security [2] topic is again having its momentum nowadays, as services and things are becoming more and more accessible and manageable remotely, therefore opening “surfaces” that can be exploited by malicious attackers.

A first attempt at solving these problems has been carried out by using what is called edge-centric architecture [1] (thus employing the concept of edge computing): an architecture that moves the locus of control of cloud computing applications and services to the edges of the network, where an edge may be a mobile device, a wearable device but also a nano-data center or a user-controlled device, that is very close to the end-user. By using this solution, industries have been able to alleviate some of the challenges posed by cloud computing at the cost of introducing some fragmentation in the infrastructure [2], but for sure there are still some necessities, brought to public attention by some IIoT contexts, that companies long to obtain: the capability of moving data where it is needed, supporting local computing while maintaining location transparent access and minimizing resource usage, and the possibility of federate compute, storage, I/O and communication resources regardless of their location [2].

One possible solution, called fogØ5, has been proposed last year with the purpose of addressing the aforementioned concerns: it is a fog computing infrastructure that unifies compute, storage and networking across cloud, edge and things [2], where fog computing stands as a system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum [3]. Another feasible technology could be OpenStack, a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter [4], but it is been already pointed out in [2] how it would be an ill-suited starting point for fog infrastructures.

For this reason, and for the use cases in which fogØ5 already proved to be the right choice (like 5G VNF in heterogeneous environments and the deployment and orchestration of complex services inside a smart factory to enable on-demand manufacturing) [2], we decided to adopt it as our main instrument to create and deploy our solution for this work. In particular, we are going to proceed as follows. Considering as our purpose the provisioning of a prediction service that permits to carry out condition monitoring of a hydraulic system, first we are going to simulate a cloud-centric architecture simply deploying a node resembling the cloud, therefore centralizing all the data and the logic to train a neural network that we are going to use to make predictions. Then, we are going to try to distribute the computations and the data by embracing the edge and fog computing paradigms: we are going to deploy distributed architectures that offers the same service as offered by the cloud-centric architecture, but favoring the improvement of aspects like privacy, control over data and latency. Once defined our architectures, we are going to evaluate different metrics to objectively classify their relative and overall performances, in order to select which is the best suited one based on the environment we are considering. Of course, we are also going to structure a deployment plan of the above architectures using fogØ5, in order to demonstrate how this tool can effectively tackle the problem of provisioning and management of a service like the one we need to provide, offering together resiliency and efficiency in the process.

2. Fog computing

As we just said, classic cloud architectures and pure edge computing approaches fall short in addressing the aforementioned requirements in IIoT. So, fog computing emerged, as the new frontier of distributed computing, born from years of research regarding cloud-centric architecture and, in particular, regarding its drawbacks and weaknesses and how to solve them.

2.1 Cloud-centric architecture

In fact, as we were describing before, what makes cloud-centric architectures so unfashionable for some IIoT solutions is the centralization of the computation and storage: anything that takes place, from a microservice or a container in the cloud to the little sensor collecting data about a machine, all is stored and managed through the cloud itself. In particular, what happens most of the time is that a gateway forwards sensor data to a cloud through a message broker, while additional services (like data storage, analysis, or aggregation) and user-facing applications connect to the broker to get access to the data [11]. While this type of management should be useful, especially regarding the centralization of collected data that can therefore be found all in one place, it can quickly become unpleasant for the motivations explained in the introduction: overall, the combination of problems related to connectivity and latency makes quite clear that a change of paradigm is needed.

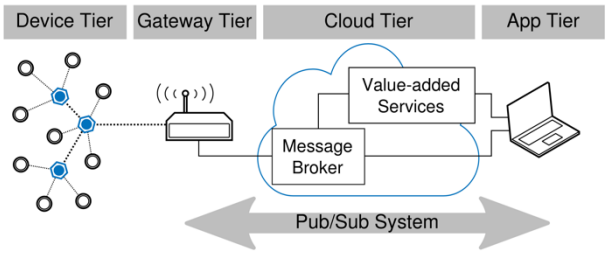

Figure 1. Cloud-centric architecture [11].

2.2 Edge-centric computing

The first evolution brought the industry to gravitate around the concept of edge-centric computing. The main idea behind this approach was to process data where it was produced for the most part, therefore at the edge of the network [12]: this was done in order to reduce both the latency and the connectivity issues, because edge nodes can be any computing and network resource along the path between data sources and cloud data centers, therefore they are always nearer to the user

with respect to the cloud [12]. Following this definition, we refer to edge computing as the enabling technologies allowing computation to be performed at the edge of the network; thus, the focus expands from just requesting services and contents from the cloud [12]. This employment of one or more edge nodes proved to yield different benefits to existing cloud-centric architectures, reducing both the response time and the energy consumption of the system [12].

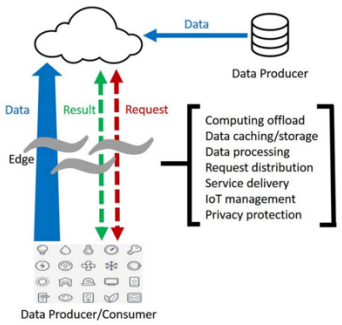

Figure 2. Edge-centric architecture [12].

2.3 Fog computing and OpenFog Reference Architecture (RA)

Still, this was not enough for many motivations, other than what has been presented in the introduction, such as the reliance on an always available cloud, the high cost of data transfer and the still present latency. What was still missing, principally, was the ability of inserting computation where it is needed (not only at the edge of the network), transforming (in the process) the deployment context in a secure, scalable, open, autonomous, agile, hierarchical and programmable way [3].

The adjectives just reported are not written randomly, but they constitute the core principles of the OpenFog Reference Architecture (RA), also named pillars: they need to be embodied by a system if it wants to be compliant with the OpenFog specification. Not only OpenFog defines the pillars, but also an abstract architecture, the RA, that embodies the previous pillars and it can be used as a starting point for an effective compliant implementation [3]. We are mentioning OpenFog RA because it became a standard with the IEEE 1934 [20] in 2018, so it has received also a formal recognition of the quality of the specification.

Therefore, we are going to proceed analyzing the proposed architecture: first we are going to analyze the pillars, in order to make clear what it is really expected from a compliant technology, then we are going to present the RA.

2.3.1 Pillars

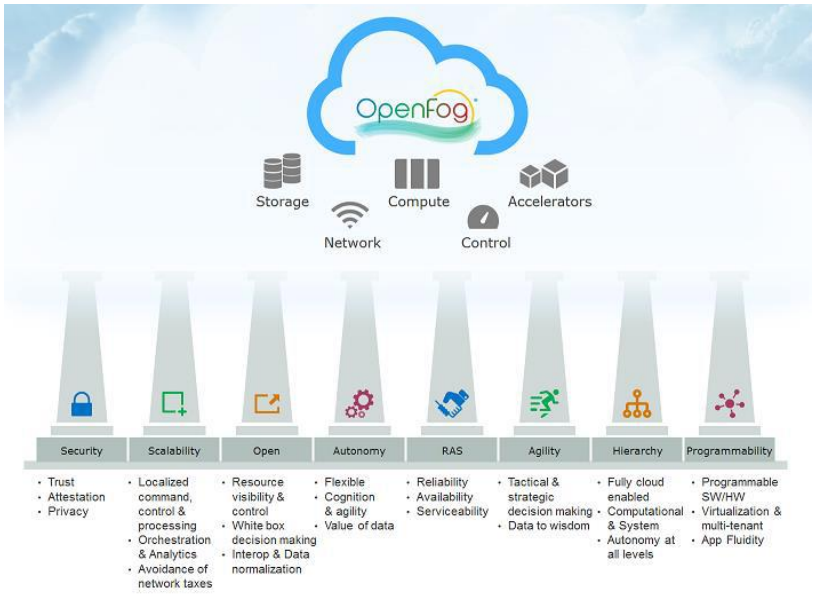

Figure 3. Pillars overview [3].

2.3.1.1 Security

Conformance to the OpenFog RA requirements for security will ensure that an OpenFog deployment will be built on a secure end-to-end compute environment that address a broad spectrum of security concerns spanning from IoT devices to cloud and the fog networks in between: this includes the OpenFog node security, OpenFog network security, and OpenFog management and orchestration security. This will allow architects and designers to focus on the high-value security and privacy problems specific to the types of devices used in their application [3].

Delving into details, it is suggested that, in many applications, particularly for brownfield deployments or for tiny devices and sensors with little to no security capability, an OpenFog node may act as a device's first point of secure entry into an OpenFog compute hierarchy and the cloud. This should be the right way to proceed because all fog nodes must employ a hardware-based immutable root of trust (Hardware Root of Trust): a trusted hardware component which receives control at power-on and then extends the chain of trust to other hardware, firmware, and software components; moreover, the root of trust should then be attestable by software agents running within and throughout the infrastructure. In particular, what should happen is that the OpenFog node should act as the first node of access control and encryption: this means they must provide contextual integrity and isolation, and control aggregation of privacy-sensitive data before these data leave the edge [3].

As more complex topologies are created in FaaS (Fog as a Service) implementations, the attestation continues as a chain of trust from the fog node, to other fog nodes, and to the cloud. It should be noted that since fog nodes may also be dynamically instantiated or torn down, hardware and software resources should be attestable, and components that are not attestable should not be fully allowed to participate in the fog node or may not be deemed to have fully trustworthy data [3].

2.3.1.2 Scalability

This pillar addresses the dynamic, technical, and business needs behind fog deployments. Elastic scalability cuts across all fog computing applications and it is enabled through large mission critical deployments based on demand. It is essential for fog computing implementations to adapt to workload, system cost, performance, and other changing business needs, both in terms of scaling up and down [3]. Scalability may involve many dimensions in fog networks:

- Scalable performance enables growth of fog capabilities in response to application performance demands (for example, reducing latency between sensor reading and resulting actuator responses) [3];
- Scalable capacity allows fog networks to change in size as more applications, endpoints, things, users, or objects are added or removed from the network. You can add capacity to individual fog nodes by adding hardware like processors, storage devices, or network interfaces. You can also add capacity through software and various pay-as-you-grow licensing [3];
- Scalable reliability permits the inclusion of optional redundant fog capabilities to manage faults or overloads [3];
- Scalable security is often achieved through the addition of modules (hardware and software) to a basic fog node as its security needs become more stringent. Capabilities like scalable distribution, rights access, crypto processing capacity, and autonomous security features contribute to scalable security [3];
- Scalable hardware involves the ability to modify the configuration of the internal elements of fog nodes (like processors, network interfaces, storage dimension), as well as the numbers of and relationships between fog nodes in networks [3];
- Scalable software is also important and includes applications, infrastructure, and management [3]:

- The management infrastructure of fog must scale to enable the efficient deployment and ongoing operation of tens of millions of fog nodes in support of billions of smart and connected things [3];
- Orchestration must be scalable to manage the partitioning, balance, and allocation of resources across the fog network [3];
- Analytics as a capability of fog networks has particularly aggressive scalability targets. This is because analytics algorithms undergo several orders of magnitude scaling due to increased capacity demands and several additional orders of magnitude due to ever-increasing sophistication of the analytics algorithms [3];
- Composability and modularity are important aspects of scalability, where individual hardware and software components (perhaps numbering in thousands of types and millions of instantiations) are assembled into a fog network optimized to run the specific applications required [3].

Scalability enables fog nodes to provide basic support to address the business requirements and enable a pay-as-you-grow model for the FaaS, which is essential to the economics of its initial deployment and long-term success [3].

2.3.1.3 Openness

Proprietary or single vendor solutions can result in limited supplier diversity, which can have a negative impact on system cost, quality and innovation. The openness pillar importance is highlighted in order to obtain fully interoperable systems, supported by a vibrant supplier ecosystem. Openness as a foundational principle enables fog nodes to exist anywhere in a network and span networks. This openness enables pooling by discovery, which means that new software-defined fog nodes can be dynamically created to solve a business mission [3]. In particular, direct consequences of this pillar are:

- Composability supports portability and fluidity of apps and services at instantiation [3];
- Interoperability leads to secure discovery of compute, network and storage resources, and enables fluidity and portability during execution [3];
- Open communication enables features like pooling of resources near the edge of the network to collect idle processing power, storage capacity, sensing ability, and wireless connectivity [3];

- Location transparency of any node instance to ensure that nodes can be deployed anywhere in the hierarchy. Each thing or software entity can observe its local conditions and make decisions on which network to join. Each endpoint in a fog network can optimize its path to the computational, networking and storage resources it needs (no matter if those resources are in the hierarchical layers of the fog, or in the cloud) [3].

2.3.1.4 Autonomy

The autonomy pillar enables fog nodes to continue to deliver the designed functionality in the face of the external service failures. Decision making will be made at all levels of a deployment's hierarchy including the layer near the device or higher order layers too, therefore intelligence from local devices and peer data can be used to fulfill the business' mission at the point where it makes the most business sense not only performance-wise, but also cost-wise [3]. Some of the typical areas for autonomy at the edge include:

- Autonomy of discovery to enable resource discovery and registration. For example, an IoT device coming online in the field would typically "phone home" first to let the backend cloud know it is alive and its associated functions are available. But when an uplink network to the cloud is unavailable, it can stop the device from going live. An autonomous fog node can potentially act as a proxy for the device registration, which then allows the device to come online without the backend cloud [3];
- Autonomy of orchestration and management (O&M) automates the process of bringing services online and managing them through the operational lifecycle and decommissioning through programmability and policies. The architecture includes an autonomous and scalable O&M function that is set up to handle any surge of demand for resources, without real-time reliance on the cloud or the need for significant human labor [3];
- Autonomy of security enables devices and services to come online, authenticate themselves against a minimal set of fog security services, and perform their designed functions. In addition, these security services can store records for future audits. With autonomy, these actions can be performed where they are needed, when they are needed, and regardless of connectivity to the cloud [3];
- Autonomy of operation supports localized decision making by IoT systems. Sensors provide data, which is the basis for autonomous actions at the edge. If the cloud or a single place in

the system's hierarchy is the only location where decisions can be made, this violates the ability to ensure reliability and as such, the architecture's operational autonomy [3].

2.3.1.5 Programmability

This pillar enables highly adaptive deployments including support for programming at the software and hardware layers. This means that re-tasking a fog node or cluster of fog nodes for accommodating operational dynamics can be completely automated. The re-tasking can be done with the help of the fog nodes inherent programmability interfaces which we describe using general purpose compute or accelerator interfaces [3]. Programmability of a fog node includes the following benefits:

- Adaptive infrastructure for diverse IoT deployment scenarios and support for changing business needs [3];
- Resource efficient deployments minimizing used resources by using a multitude of features including containerization [3];
- Multi-tenancy to accommodate multiple tenants in a logically isolated runtime environment [3];
- Economical operations that results adaptive infrastructure to changing requirements [3];
- Enhanced security to automatically apply patches and respond more quickly to evolving threats [3].

2.3.1.6 Reliability, Availability, and Serviceability (RAS)

A reliable deployment will continue to deliver designed functionality under normal and adverse operating conditions, therefore also in harsh environmental conditions and remote locations [3]. The reliability of the RAS pillar includes but is not limited to the following properties:

- Ensuring reliable operation of the underlying hardware upon which the software is operating, enabling reliable and resilient software and a reliable fog network, which is generally measured in uptime [3];
- Safeguarding the availability and integrity of data and compute on edge gateways using enhanced hardware, software, and network designs [3];
- Autonomous predictive and adaptive self-managing capabilities when required by the health of the system to initiate self-healing routines for hardware and software and update their firmware/application and apply security patches [3];

- Testing and validation of system components, including device drivers and diagnostic tools under a variety of environmental conditions [3];
- Validation of system platforms and architectures through interoperability certification test suites [3].

Availability ensures continuous management and orchestration, which is usually measured in uptime [3]. The availability of the RAS pillar includes but it is not limited by the following properties:

- Secure access at all levels of a fog hierarchy for orchestration, manageability, and control, which includes upgradeability, diagnostics and secure firmware modification [3];
- Fault isolation, fault syndrome detection, and machine learning to help improve Mean Time To Repair (MTTR) of a failed system to achieve higher availability [3];
- Concept of cloud based back-end support with availability of interfaces throughout the system:
    - Secure remote access from a plurality of devices (not just a single console) [3];
    - Mesh access capabilities of end-point sensor/peering [3];
    - Remote boot capabilities of the platform. Moreover, modification and control from the lowest level firmware (BIOS) through to the highest software in the hierarchy (cloud) [3];
    - Support for redundant configurations for persistent productivity [3].

Servicing a fog deployment ensures correct operation [3]. Serviceability of the RAS pillar includes but is not limited by the following properties:

- Highly automated installation, upgrade, and repair (hardware/software) to efficiently deploy fog computing at scale [3];
- Hardware or software can either autonomously heal or be serviced by the various manufacturers [3];
- Ease of use to accommodate maintenance [3];
- Serviceability of the system:
    - Ease of access/swap-out of the hardware (component interoperability) [3];
    - Ease of secure upgradeability of software, BIOS, and applications locally or remotely and in real time [3];
    - Replication of system configuration over cloud on replaced/swap-out systems [3].
- Support for redundant configurations for persistent productivity [3].

2.3.1.7 Agility

The agility pillar addresses business operational decisions for an OpenFog RA deployment. It is not possible for humans alone to analyze the data generated by IoT as the basis for rapid, sound business and operational decisions, so, this pillar focuses on transforming this volume of data into actionable insights. In fact, data generation by sensors and systems in an OpenFog RA deployment are turbulent, bursty, and are often created in huge volumes. Most importantly, at first encounter, data may not have context, which is created only when the data is collated, aggregated, and analyzed. Our first thought could be to analyze data at the cloud level, but this subjects the data to increasing levels of latency. The ideal approach is to make operational decisions as soon as data can be turned into a meaningful context, therefore the architecture needed should enable the creation of context close to the data generation where it makes the most sense for a given scenario. More strategic, system-wide decisions and policy management can be made further up the layers in the fog hierarchy. OpenFog architectural approaches allow IoT system developers to optimize the placement of their applications as decision making components [3].

To conclude, agility also deals with the highly dynamic nature of fog deployments and the need to respond quickly to change [3].

2.3.1.8 Hierarchy

Computational and system hierarchy is not required for all OpenFog architectures, but it is still expressed in most deployments. OpenFog RA computing resources can be seen as a logical hierarchy based on the functional requirements of an end-to-end IoT system. Depending on the scale and nature of the scenario being addressed, the hierarchy may be a network of smart and connected partitioned systems arranged in physical or logical layers, or it may collapse into a single physical system (scalability pillar) [3].

2.3.2 Reference architecture

Having presented the pillars, now we can delve into the RA. In particular, this architecture is defined by three main concepts, that are presented below.

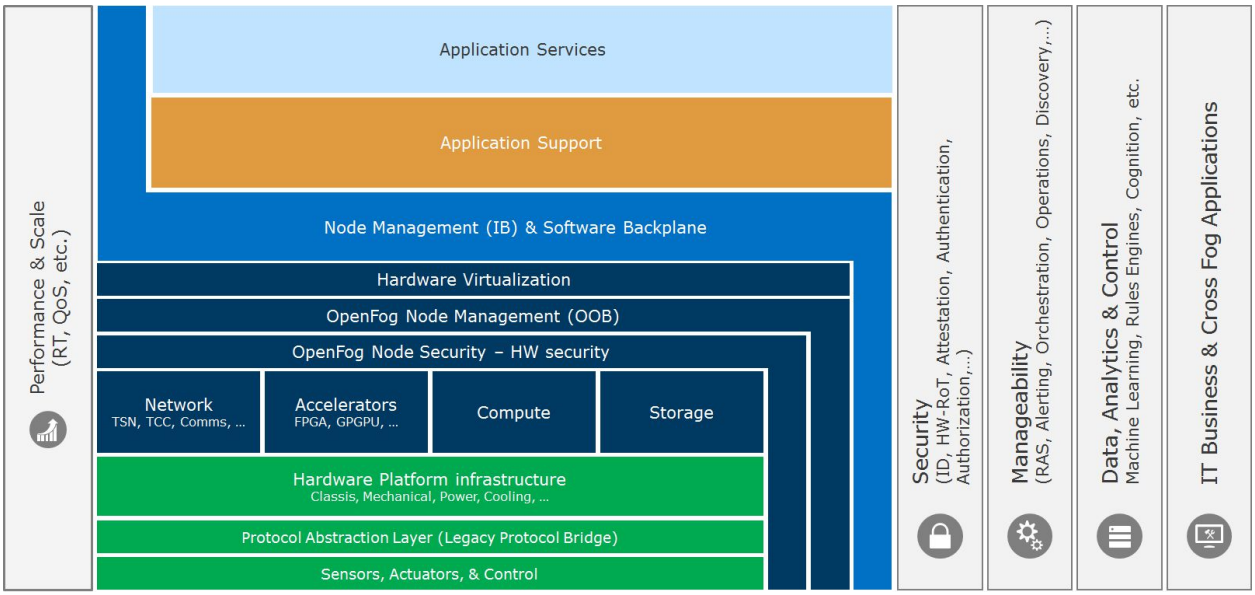

Figure 4. OpenFog RA overview [3].

2.3.2.1 Number of tiers

In most fog deployments, there are usually several tiers (N-tiers) of nodes:

- Those at the edge are typically focused on sensor data acquisition/collection, data normalization, and command/control of sensors and actuators. However, sometimes we can also find some kind of intelligence at this level, because it is not so unusual (as we were saying) that network pipes are not large enough to effectively transport raw sensor data to higher layer of fog nodes; given also that computation capabilities are growing, we can say that the intelligence of fog deployments is rapidly increasing over time [3];
- Those in the first higher tier are focused on data filtering, compression, and transformation. They may also provide some edge analytics required for critical real time or near real time processing. We can see how, as we move away from the true network edge, we see higher level machine and system learning (analytics) capabilities [3];
- Nodes at the higher tiers or nearest the backend cloud are typically focused on aggregating data and turning the data into knowledge. It is important to note that the farther from the true edge, the greater the insights that can be realized [3].

Therefore, scaling horizontally (adding new nodes) means adding performance and operational capabilities to the system, while scaling vertically (adding new layers) means adding intelligence to the system [3].

The problem remains defining the number of tiers to be used in a fog deployment. This will be dictated by the scenario requirements, including:

- Amount and type of work required by each tier [3];
- Number of sensors [3];
- Capabilities of the nodes at each tier [3];
- Latency between nodes and latency between sensors and actuation [3];
- Reliability/availability of nodes [3];

In general, each level of the N-tier environment would be sifting and extracting meaningful data to create more intelligence at each level. Tiers are created in order to deal efficiently with the amount of data that needs to be processed and provide better operational and system intelligence.

2.3.2.2 Fog node capabilities

The architectural elements of a node will vary based on its role and position within an N-tier fog deployment. As described previously, nodes at the edge may be architected with less processing, communications, and storage than nodes at higher levels [3].

Fog nodes may be linked to form a mesh to provide load balancing, resilience, fault tolerance, data sharing, and minimization of cloud communication. Architecturally, this requires that fog nodes have the ability to communicate laterally (peer to peer or east to west) as well as up and down (north to south) within the fog hierarchy. The node must also be able to discover, trust, and utilize the services of another node in order to sustain RAS [3].

2.3.2.3 Layers and perspectives

The reference architecture is a very complex and articulated one, involving ten layers and five so-called perspectives (capabilities that cut across architectural layers) [3].

The layers can be grouped in three views:

- Software view: composed by a software running on fog platforms used to satisfy a given deployment scenario. A robust fog deployment requires that the relationship between a fog node, fog platform, and fog software are seamless [3]. It is composed by these layers:
    - Application Services: services that are dependent on infrastructure provided by the other two layers, fulfill specific end use case requirements, and solve domain specific needs [3];
    - Application Support: infrastructure software that does not fulfill any use case on its own, but helps to support and facilitate multiple application services [3];

- Node Management and Software Backplane: general operation and management of the node and its communications with other nodes and systems. IB (in figure 4) refers to In Band management. This is generally how software interacts with the management subsystem [3].

- System view: it includes Hardware Virtualization down through the Hardware Platform Infrastructure. The latter primarily ensures that fog platforms provide robust mechanical support and protection for their internal components, because in many deployments, fog platforms must survive in harsh environmental conditions, but it also enables hardware-based virtualization mechanisms in almost all processor hardware that would be used to implement fog platforms. It may also play an important role in system security. Hardware virtualization for I/O and compute enables multiple entities to share the same physical system. Virtualization is also very useful in ensuring that virtual machines (VMs) may not utilize instructions or system components that they are not (by design) supposed to utilize [3];
- Node view: the lowest level view in the architectural description, that defines how a node can be introduced in a fog computing network [3]. It is composed by:
    - Node security: it is essential for the overall security of the system. This includes protection for interfaces and compute (for example). In many cases, a node will act as a gateway for legacy sensors and actuators to higher-level fog functions and therefore can act as a security gateway [3];
    - Node management: a node should support management interfaces. Management interfaces enable higher level system management agents to see and control the lowest level node silicon. The same management protocol can be used across many different physical interfaces [3];
    - Network: every fog node must be able to communicate through the network. Since many fog applications are time sensitive and time aware, some fog computing networks may need to support Time Sensitive Networking (TSN) [3];
    - Accelerators: many fog applications utilize accelerators to satisfy both latency and power constraints as it relates to a given scenario [3];
    - Compute: a node should have general purpose compute capabilities. It is also important that standard software (e.g., Commercial off the Shelf or open source) is

able to run on this node. This enables a higher level of interoperability between fog nodes [3];

- Storage: an autonomous node must be able to learn. Before any learning is possible, it must have the ability to store data. Storage devices attached or embedded to a node need to meet the exptected performance, reliability, and data integrity requirements of the system and scenario. In addition, storage devices should also provide information and early warnings about the health of the media, support self-healing properties, and support ID-based performance allocation. Some kind of local, standalone storage will be required for local context data, logging, code images, and to service applications that run on the node. There will often be more than one kind of storage required – e.g., local hard disk, SSD, and secure storage for keys and other secret material [3];
- Sensors, Actuators, and Control: these hardware or software-based devices are considered the lowest level elements in IoT. There could be several hundred or more of these ones associated with a single fog node. Some of these are dumb devices without any significant processing capability, while others may have some basic fog functions. These elements generally have some amount of connectivity, and include wired or wireless protocols, such as I2C, GPIO, SPI, BTLE, ZigBee, USB, and Ethernet, etc. [3];
- Protocol Abstraction Layer: many of the sensors and actuators on the market today are not capable of interfacing directly with a fog node. The protocol abstraction layer makes it logically possible to bring these elements under the supervision of a fog node so that their data can be utilized for analytics and higher level system and software functions [3].

Then, we have the perspectives:

- Performance: low latency is one of the driving reasons to adopt fog architectures. There are multiple requirements and design considerations involving multiple stakeholders to ensure this aspect is satisfied. This includes time critical computing, time sensitive networking and network time protocols (for example). It is a cross cutting concern because it has system and deployment scenario impacts [3];
- Security: end-to-end security is critical to the success of all fog computing deployment scenarios. If the underlying silicon is secure, but the upper layer software has security issues

(and vice versa), the solution is not secure. Data integrity is a special aspect of security for devices that currently lack adequate security. This includes intentional and unintentional corruption [3];

- Manageability: managing all aspects of fog deployments, which include RAS, DevOps, etc., is a critical aspect across all layers of a fog computing hierarchy [3];
- Data Analytics and Control: the ability for fog nodes to be autonomous requires localized data analytics coupled with control. The actuation/control needs to occur at the correct tier or location in the hierarchy as dictated by the given scenario. It is not always at the physical edge, but may be at a higher tier [3];
- IT Business and Cross Fog Applications: in a multi-vendor ecosystem, applications need the ability to migrate and properly operate at any level of a fog deployment's hierarchy. Applications should also have the ability to span all levels of a deployment to maximize their value [3].

3. fogØ5

As we were saying before, fogØ5 aims at providing a decentralized infrastructure for provisioning and managing compute, storage, communication and I/O resources available anywhere across the network, sticking to the pillars of OpenFog RA (to which is, in fact, compliant). So, it addresses highly heterogeneous systems, even those with extremely resource-constrained nodes.

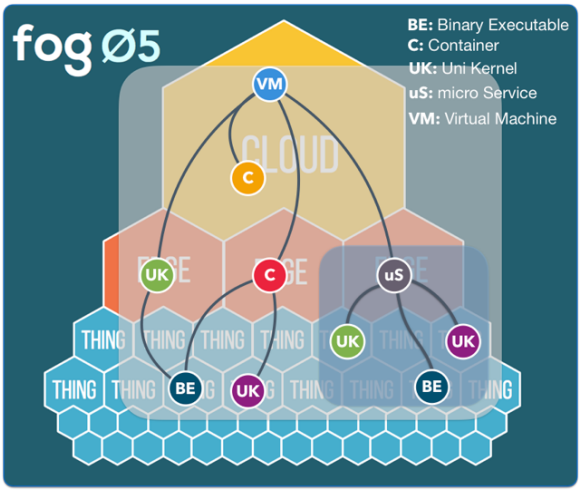

Figure 5. fogØ5 overview [3].

3.1 Stack

In order to do so, it employs a software stack that it is built on-top of level 4 of the ISO/OSI stack, composed by the following layers (proceeding bottom-up).

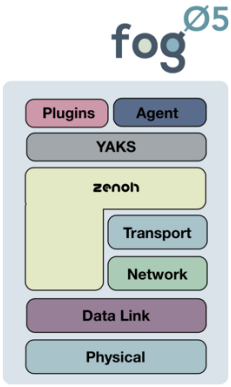

Figure 6. fogØ5 stack [25].

3.1.1 Data-centric protocol

For historical reasons, Internet has been built around a host-centric communication model, that means that communication takes place between named hosts [7]. But the increasing demand for highly scalable and efficient distribution of content has motivated the development of modern architectures (called data-centric) based on named data objects [7], instead of hosts. Being the domain to which fogØ5 refers to strictly coupled with the concept previously presented, it has been decided to support a data-centric approach providing a compatible technology and modeling objects like named data objects, precisely.

For what concerns the technology, fogØ5 supports usage of DDS (Data Distribution Service), a standard for data-centric messaging [6] and zenoh (zero network overhead), a protocol similar to DDS whose purpose is to bring data-centric abstractions and connectivity also to devices that are constrained with respect to the node resources [5] [2]. While the former is very famous and used in many domains, the latter is an emerging and recent protocol, for which it is necessary to spend some words of introduction. It is important to say that it permits a transparent deployment of devices constrained by local resources and/or networks using peer-to-peer and/or client-to-broker kinds of interaction, offering extremely low footprint (addressing constrained targets), extremely efficient wire protocol (inducing a protocol overhead of just a few bytes over the user data) and support for devices that undergo aggressive sleep cycles [5]. Zenoh applications coordinate themselves by autonomously, anonymously and asynchronously writing and reading in a data space while being decoupled in time and space, fundamental to support applications that undergo sleep cycles, because it decouples the availability of data from the availability of the applications that wrote it (as the latter may be sleeping now) [5]. In order to do what has been just described, zenoh can be used stand-alone, bridged to DDS through a gateway and run as a replacement of the DDSI-RTPS protocol for better scalability and performance [5]. Its functioning is based on two main abstractions:

- Resource: identifies the information to be exchanged between readers and writers. They are composed by properties, used to specify the characteristics of exchanged data. Depending on the context, a resource may represent a topic or an instance; for example, a resource for a topic can be `zenoh://myhouse/floor/1/musicroom/LightStatus`. There are also system defined read-only resources (called meta resources) such as `zenoh://myhouse/floor/2/bedroom\$ls`, that returns the set of child resources of bedroom,

or `$id`, that arbitrary returns an ID from a list of IDs currently associated with the resources [5];

- Selection: a conjunction of a resource, a list of hashes, and a predicate over the resource content; in other words, a query. The resource and the list of hashes are separated by a "#" while the hashes and the predicate are separated by a "?". For example, the query "`zenoh://myhouse/floor/*/bedroom/*/LightStatus#{transient}?{luminosity>0}`" [5] is used to find all the transient resources with luminosity greater than 0.

These two abstractions are then used in zenoh interactions: a typical one is based on a scouting phase, where a node interested in finding out other nodes sends a special kind of message, then if someone answers, a session can be established and communications can be carried out (for example, using selectors to reference resources). It is important to note that zenoh messages used in these interactions are based on a small and precise DSL, to efficiently and unambiguously describe the message format [5]. Ultimately, for what regards the APIs, they are not addressed in order to ensure that the right level of variability can be achieved to address the constraints imposed by different targets, but they are conceptually the same as those for DDS: operations that are associated with one or more resources identified by a URI [5].

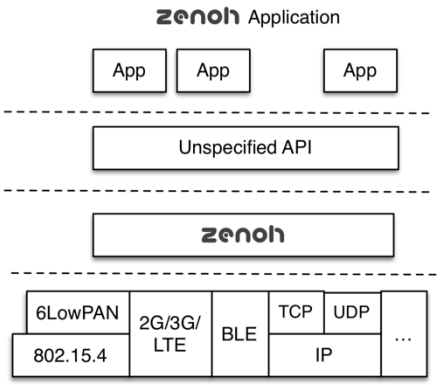

Figure 7. zenoh stack overview [25].

For what concerns modeling the domain through named data objects, they used the concept of resource as in zenoh: in fogØ5, in particular, a resource is an abstraction used to represent nearly everything in the system; it can be a node, a FPGA, or a specific GPIO on a node, for example [2]. Each resource in a system has conceptually two values, the actual and the desired value, that are maintained on two separate distributed key value stores: the former can only be changed by the owner of the resource, while the latter can be changed by anyone and it is used to trigger transformation of the actual value; this technique allows us to have concurrency only on the desired value of a resource, but never on its actual value [2]. Moreover, they can be called "named" because

they are expressed as URI, organized as a tree whose root indicates the administrative domain [2]. The syntax employed to communicate with resources is "`<prefix>/<system id>/<node id>/<type of resource>/<id of resource>/<type of subresource>/<id of subresource>/...['?'query]['#'fragment]`" [2], where we can highlight the following important syntax aspects:

- Prefix: needed to reference the store in which the resource is contained (we will see further details later);
- Wildcard: we can use `*`, to indicate an arbitrary sub-path of length 1, and `**`, to indicate a sub-path of arbitrary length. Therefore, we can easily substitute one or more tag of the previous path using one of the above wildcards [2];
- Query: used to retrieve resources matching a given predicate [2];
- Fragments: represent delta-updates to the value of a resource [2].

3.1.2 Distributed key/value store

Everything in fogØ5 is a resource, as we were saying, and following the aforementioned data-centric concept all resources are stored in a distributed key/value store, that provides an eventual consistency semantics and has built-in versioning. The store has a root, a home and a cache capacity. The root and the home are URI indicating respectively the scope of resolution for the store and the resources to be maintained in main memory: a store is going to keep in main memory any resource that has home as a prefix, and is going to store on a fixed capacity cache resources whose prefix is the root but not the home. Resources that are not found in the store are resolved using a distributed cache miss protocol, and a distributed cache coherency protocol is used to ensure eventual consistency of cached resources. In this way, a global actual store (the correspondent prefix for a resource URI contained in this store is "`agfos`") and a global desired store (with associated prefix "`dgfos`") are created, maintaining all the information about the existing resources and permitting changes on them, respectively. In the end, this key/value store, along to being used at the core of fogØ5, is provided to the user as a way for virtualizing memory and storage end-to-end, thus allowing its usage as a single and scalable abstraction across high-end and extremely constrained nodes [2].

The current implementation of this distributed key/value store is YAKS [8]: a distributed service that implements an eventually consistent, scalable, location transparent, high performance and distributed key/value store with pluggable back-ends (DBMS and main memory, at the moment)

and front-ends (REST, Socket and WebSocket, at the moment). It is equipped with dynamic discovery and supports extremely well dynamic environments. YAKS data is globally accessible without requiring local replication as in traditional key/value stores. In a way, YAKS can be thought as a distributed fabric to integrate data at rest and data in movement and access it in a location transparent manner [9]. It is defined by various abstractions:

- Path: a set of strings separated by a "/" as in a filesystem path, such as "`/demo/ac/one`", "`/building/1/home/2/floor/3`" [9], representing the key of the key/value pair;
- Selector: an expression that contains wildcards, query and projections, such as "`/building/1/home//floor/`", "`/building/1/**`" or "`/building/1/home/1/floor/*/lamp?luminosity<10`" [9];
- Value: a user-provided data item along with its encoding. YAKS supports natively several encodings, such as raw, string, JSON, and DB. For natively supported encodings YAKS will automatically try to perform conversions when necessary [9];
- Backend: a storage technology, such as DBMS, Main Memory, NoSQL stores, etc. [9];
- Frontend: a connectivity technology, such as REST, TCP/IP, etc. [9];
- Storage: an entity storing tuples on a specific backend. Storages can be created by applications and take responsibility for storing all tuples whose path matches the storage selector [9];
- Subscriber: an entity registering interest for being notified whenever a tuple with a path matching the subscription's selector is put on YAKS [9];
- Eval: a computation registered at a specific path. These computations can be triggered by evaluating those matching a selector [9];
- Workspace: the abstraction that give you access to YAKS primitives [9].

It provides various operations to manage `<key, value>` pairs:

- `Put`: stores the tuple `<path, value>` (passed as argument) on all storages in YAKS whose selector matches the path parameter [9];
- `Get`: gets the set of tuples `<path, value>` available in YAKS for which the path matches the selector (passed as argument) [9];
- `Remove`: removes from all YAKS storages the tuples having the given path (passed as argument) [9];
- `Subscribe`: registers a subscription to tuples whose path matches the selector (passed as argument). It returns a subscription identifier [9];

- `Unsubscribe`: unregisters an active subscription with the subscription identifier (passed as argument) [9];
- `Register_eval`: registers an evaluation function under the provided path (both passed as arguments) [9];
- `Unregister_eval`: unregisters a previously registered evaluation function under the given path (passed as arguments) [9];
- `Eval`: requests the evaluation of registered functions whose registration path matches the given selector (passed as arguments) [9].

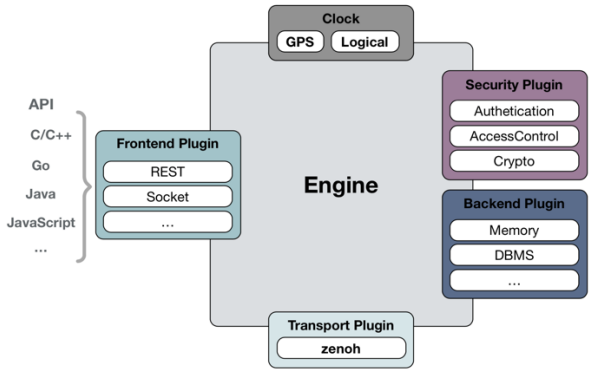

Figure 8. YAKS instance overview [25].

3.1.3 Agent

The core component of fogØ5. It represents a manageable resource on the system, which from now on we will call a fogØ5 node [2]. Each agent has four stores in YAKS, each pair called node-local and node-local constraint respectively: the former represents the actual state of the node's resources (with prefix "`alfos`") and the desired state of the node's resources (with prefix "`dlfos`"), the latter the actual state of the node's constrained resources (with prefix "`aclfos`") and the desired state of the node's constrained resources (with prefix "`dclfos`"). Also here, the actual state can only be written by the agent running on that node, while the desired state can be written by anybody to cause state transitions, such as provisioning a VM or a binary executable. The agents dynamically discover each other by leveraging the dynamic discovery provided by the distributed store, which in turn leverage the dynamic discovery provided by the data sharing layer. The set of functionalities supported by an agent is controlled by plug-ins (that we will see in a moment): what the agent does is orchestrate the plug-ins state transitions [2]. It is important to notice how, usually, developers employing this technology directly interact with this layer, ignoring the actual implementation of layer 1 and 2, therefore resulting completely transparent to them.

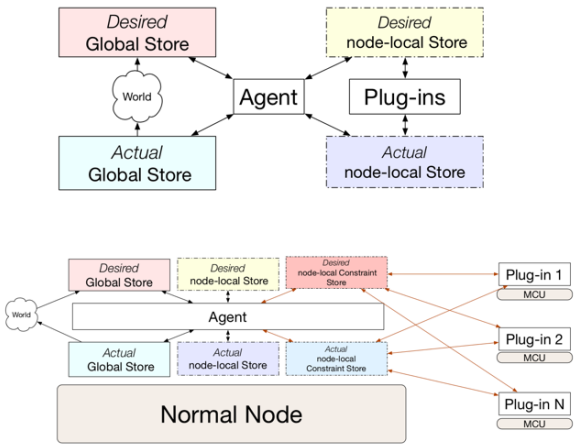

Figure 9. Complete store view [25].

3.1.4 Plugins

Middlewares translating agent's abstract maneuvers to specific actions based on the target entity. Each plugin can be of two types: standalone, meaning that it does not need other running plugins to work, or dependent, meaning that it needs other running plugins in order to work correctly. fogØ5 is shipped with many interfaces, each one referring to a specific plugin type, that can be used as basis for our implementations:

- Atomic entities (runtimes): for example, they have to implement the FSM for the kind of atomic entity they will be managing;
- Networks: for example, they have to offer the possibility of creating a virtual interface, a virtual bridge and a virtual network;
- OSes: for example, they have to communicate with the OS to read a file, copy a file and remove a file;
- Monitoring: they need to start and stop monitoring of an entity;
- Resource orchestration: they need to onboard and offload application;
- Resource management: to configure an application and fetch the configuration of an already configured application.

3.1.5 Entities / FDUs (Fog Deployment Units)

Abstractions used by fogØ5 to provision, manage and orchestrate applications or network functions [2]. An entity is either an atomic entity, such as a Virtual Machine (VM), a container, a Unikernel, a binary executable or a Directed Acyclic Graph (DAG) of entities [2]. Both entity and atomic entity

have an FSM (Finite State Machine) that defines their current state and associated legal state transitions; these FSM have been defined to be sufficiently generic to encompass essentially any kind of atomic entity ranging from a VM to a binary executable [2]. The life cycle can be described as follows (it is nearly the same for both atomic entity and entity), and can be managed through REST APIs or CLI commands (as we will see in each state):

1. UNDEFINED: just a bogus state to represent the fact that the entity is not yet defined. It is in this state that we make the onboarding of a FDU descriptor, that means to load the FDU descriptor (a JSON file containing information for each component of that entity) of an entity to a YAKS instance. What we need to do is just make sure to correctly write it, and in order to do so, we must follow the manifests in [8] describing the various properties we can set, depending on the type of entity and atomic entity we want to create: in particular, each entity should have a human readable name and an UUID to be identifiable. Once we have the FDU descriptor ready, we can onboard it using (as for any functionality) the CLI interface, through the command "`fos fim fdu onboard -d <path to fdu descriptor>`", or the REST APIs, through the function "`onboard(self, descriptor, wait=True)`". When we finish working with the entity just onboarded, we can delete its descriptor from the system by calling "`fos fim fdu offload -f <fdu uuid>`" or "`offload(self, uuid, wait=True)`";
2. DEFINED: when it is proven that a specific node can satisfy the constraints for the considered entity. We can reach this state using "`fos fim fdu define -n <nodeuuid> -f <fdu uuid>`" (from the CLI interface), or "`define(self, fduid, node_uuid, wait=True)`" (from the REST APIs), passing as arguments the identifier of an onboarded FDU and an UUID of the target node. In this way, we obtain the instance identifier, that we will use for all the future operations (except where stated otherwise). At this point, we can undefine the entity by calling "`fos fim fdu undefine -i <instance id>`" or "`undefine(self, instanceid, wait=True)`", or proceed to the next state;
3. CONFIGURED: now there is an instance associated to the entity with all the parameters, settings and data it needs. For example, for a microservice this means that all the configuration is loaded, while for a VM/Container it could mean that the appropriate resources have been created and configured (such as disk files and network bridge). We can reach this state using "`fos fim fdu configure -i <instance id>`" or "`configure(self, instanceid, wait=True)`". Then, we can come back to defined by calling "`clean(self,`

`instanceid, wait=True)` or "`fos fim fdu clean -i <instance id>`", or proceed to the next state;

4. RUNNING: the entity is actually running. For example, if the entity is a microservice it should implement an onStart() callback to initiate properly, if it is a VM/Container it should simply start (maybe using libvirt or LXD APIs). We can reach this state using "`fos fim fdu start -i <instance id>`" or "`start(self, instanceid, wait=True)`". Then, we can come back to configured, using "`fos fim fdu stop -i <instance id>`" or "`stop(self, instanceid, wait=True)`", or directly to undefined, using "`fos fim fdu undefine -i <instance id>`" or "`undefine(self,instanceid,wait=True)`", or proceed to one of the next reported states;
5. PAUSED: the entity interrupted the execution without being killed. We can reach this state using "`fos fim fdu pause -i <instance id>`" or "`pause(self, instanceid, wait=True)`". It can then resume the execution from the current state using "`resume(self, instanceid)`" or "`fos fim fdu resume -i <instance id>`";
6. SCALING: the entity is being replicated on one or more nodes. When the scaling is done, the entity comes back to the running state, and all instances replicated are therefore running. Currently, scaling is still not supported but it will be added soon to the working functionalities.

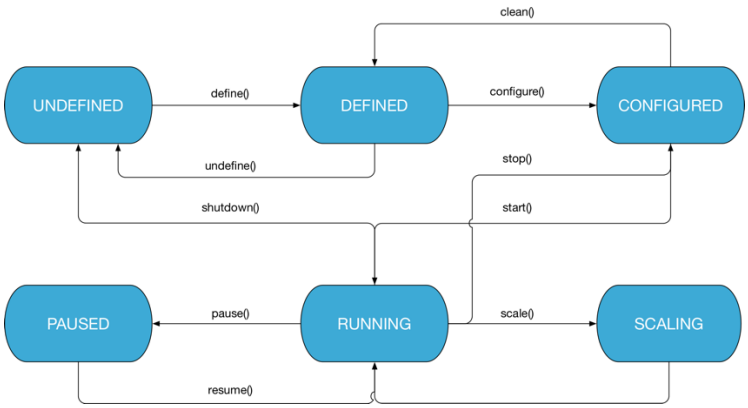

Figure 10. Entity life cycle [3].

The only difference between an entity and an atomic entity is the MIGRATING state, that can be reached only by the latter: it happens when we want to move an atomic entity from a node to another. In particular, the agent should check if the destination system is capable to handle this new atomic entity (in terms of constraints like memory, disk, network, accelerators and so on), then, if it is the case, there are going to be actually two representations of the same atomic entity instance running on two different systems; each representation has a different state: the source one is in TAKING_OFF substate, and after migration is going to be destroyed so that

instance is not going to be available anymore, the destination one is in LANDING substate, and after the migration is going to be in RUNNING state. For example, for VMs and Containers the actual migration is handled by the underlying hypervisor (therefore by KVM or LXD, respectively), for microservices through callbacks (like "`before_migration()`", in which the microservice saves its state and give it to the framework, and "`after_migration()`", called by the framework to restore the state).

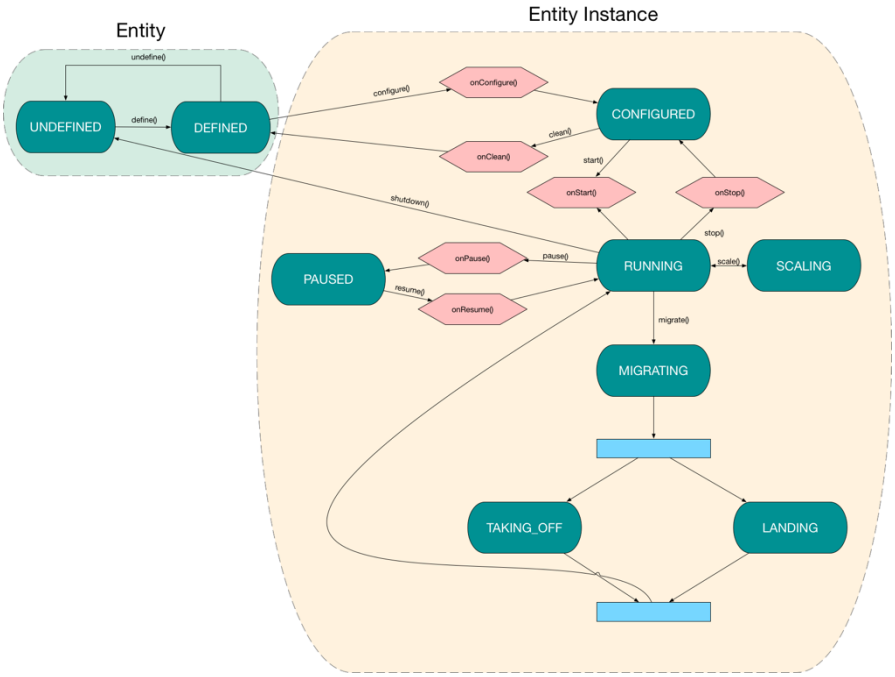

Figure 11. Atomic entity life cycle [3].

3.2 Execution model

This architecture is thought to be executed as either a process on a traditional OS or as a trusted service inside a Trusted Execution Environment (TEE), where the TEE could be running on a hypervisor or on bare metal (ideally on a trust zone such as those supported by ARM processors) [2]. This last option, in particular, is the foundation for the recommended deployment infrastructure, because with TEE, we ensure that we are going to launch our service in an isolated execution environment that runs alongside the OS but at the same time protected from it through the protection profile [10]. Therefore, we could think of running, on the same hardware, two fogØ5 nodes: one effectively managing entities through an internal VPN, the other (running on the TEE) only managing network and hypervisors plugins, and that can therefore be used to be the only communication channel with external networks (also for the other node) [2]. This deployment would ensure that all communications with the external network are done through a run-time that

has a very small surface of attack as opposed to regular OS like Windows or Linux, that are extremely hard to secure [2].

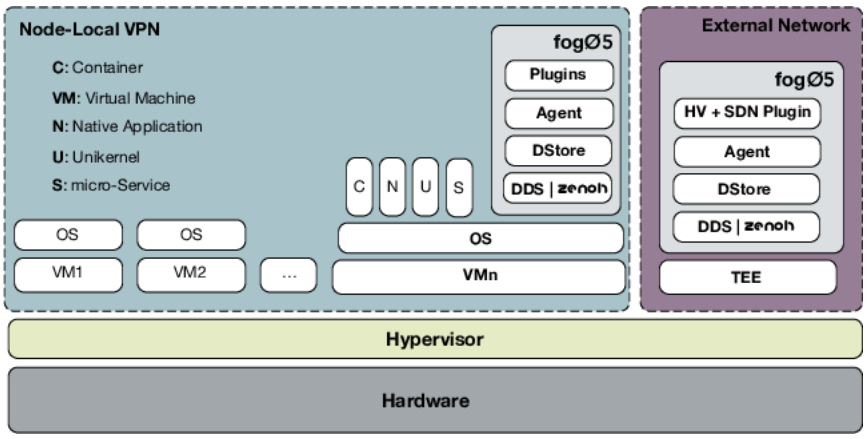

Figure 10. Execution model [3].

3.3 Real-world use case

fogØ5 is currently being used in R&D laboratories around the world, such as ITRI, UC3M, III and NCTU, for applications ranging from 5G VNF in heterogeneous environments to deployment and orchestration of complex services inside a smart factory to enable on-demand manufacturing. In particular, for this last scenario, one notable use case regards general purpose robots, that will be used in factories to accomplish a wide range of tasks through reconfiguration achieved by simply deploying a different control logic. Flexible provisioning of robot tasks, within a swarm of robots, requires dynamic discovery of available robots along with the discovery of their computational, I/O and manipulation capabilities, therefore it is the perfect case study for fog computing infrastructures (also for requirements like computational harvesting, minimal communication latency and jitter) [2].

Diving deep inside the managed application, fogØ5 is used to provision, deploy and manage components of a complex robotic control application, and dynamically decide the most suitable target among those spanning from the cloud to robots. Specifically, fogØ5 may decide to deploy analytics on the cloud or edge nodes, while the control logic may be deployed and migrated to remain close to the robot or perhaps directly on the robot. Moreover, in this case fogØ5 is used also for provisioning, deploying and managing network. In particular, in our business case, we have a robot equipped with a stereoscopic camera, that has to follow another robot; the logic for this robot is deployed on edge servers and live migrated as the robot moves, to minimize network latency and jitter [2]. The hardware platform is composed by:

- Two edge servers, one of which has a GPU accelerator [2];
- One Raspberry Pi 3, with an extra 802.11ac NIC [2];
- One robot using Robotic Operating System v2 (ROS2), ADLINK Neuron board, a stereoscopic camera and a set of motors to move around [2];
- One robot using ROS2, ADLINK Neuron board, a set of motors to move around [2].

Moreover, all robots have some computational power, multiple NICs and I/O interfaces. Each device runs a fogØ5 agent and has a NIC connected to the management [2].

The application is composed by a complex robot control application, an analytics application, a Virtual Access Point VNF, an image recognition client, a simple robot control logic, a simple robot client and a monitoring application. Each entity has different requirements in terms of I/O, compute, accelerators and networking requirements. The application components are extremely heterogeneous: analytics, monitoring and robot controlling applications are packaged in VMs, the gateway and the Access Point VNF are LXD containers, the others are ROS2 nodelets [2].

fogØ5 greatly facilitates the provisioning, deployment and management tasks as it dynamically discovers available nodes and deploys the components of this complex application in the right place, where the right place depends on resources and affinity. To maintain affinity between some entities it also executes live migration of some components. Additionally, fogØ5 is also used to create an overlay network used by all components on the application [2].

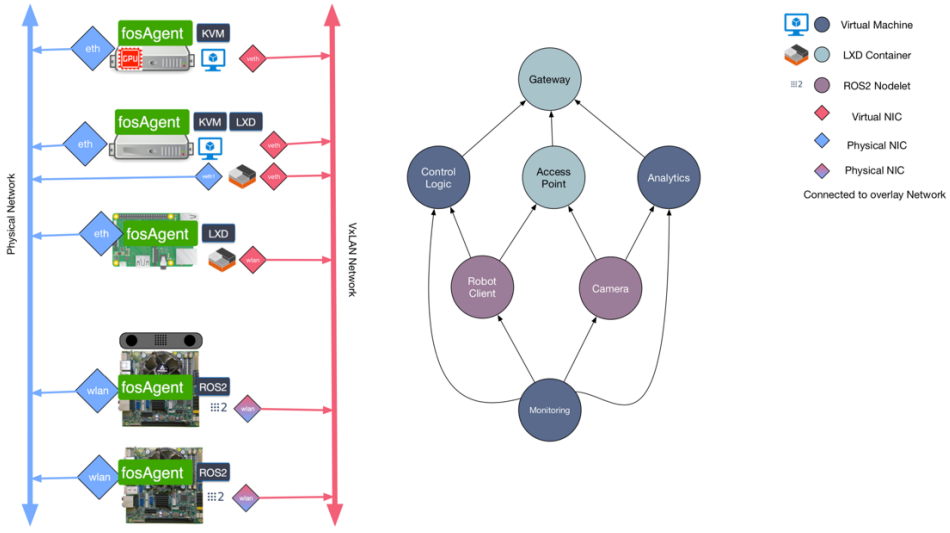

Figure 11. Final use case architecture [3].

3.4 Usage

Bringing the pieces together, we can schematize a simple fogØ5 usage with the following steps:

1. Launch a YAKS instance;
2. On each node that we want to control remotely, we need to have at least one fogØ5 agent instance running (if it is possible, two instances following the previously described architecture);
3. Write a FDU descriptor for each entity that we want to define;
4. Create the entities through YAKS instance providing the descriptors;
5. Manage the entities' lifecycle as needed.

4. Neural networks and related software

As we were anticipating in the introduction, the aim for this master thesis, along with the deployment of various infrastructures, is the provisioning of a prediction service through them. This service is going to be based on a neural network (as we will see in the following chapters), therefore, in the next paragraphs, we are going to explore what they are and which are the widely used software instruments to employ them in an effective and efficient way.

4.1 Neural networks

Neural networks (NN) are computing systems made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs [21]. They are particularly useful for many machine learning tasks such as computer vision, speech recognition, recommender systems [22] and so on. As shown in [22], such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules. For example, in speech recognition, the purpose is to map an acoustic signal containing a spoken natural language utterance into the corresponding sequence of words intended by the speaker [22]. In order to do that, neural networks, as understandable from [22], automatically generate identifying characteristics from the learning material (spoken phrases) that they process.

4.1.1 Structure

These networks are called neural because they are loosely inspired by neuroscience. Each element of these networks may be interpreted as playing a role analogous to a neuron. These neurons are arranged in layers: we can think of each one of them as consisting of many units that act in parallel, each representing a vector-to-scalar function, in the sense that it receives input from many other units and computes its own activation value [22] (a real number). It is important to note that, as stated in [22], the input given to each neuron of the NN is weighted prior to pass it to the activation function, therefore its value is modified in order to achieve the expected prediction behavior from the NN.

We can employ various functions to compute the activation value:

1. Sigmoid: defined by the function $\frac{1}{1+e^{-x}}$. Therefore, sigmoid returns values restricted to the open interval (0, 1) [22]: it is a non-binary activation, since we can have any value inside that

interval. It is similar to a step function, but it is nonlinear, therefore combinations of this function are also nonlinear, meaning that we can employ more than one layer in order to obtain complex output functions. The only problem regards saturation: sigmoid sticks to 0 when x becomes very negative, while if x becomes very positive, to 1 [22]. The gradient can become too small to be useful for learning when this happens (gradient vanishing problem), whether the model has the correct answer or the incorrect answer [22];

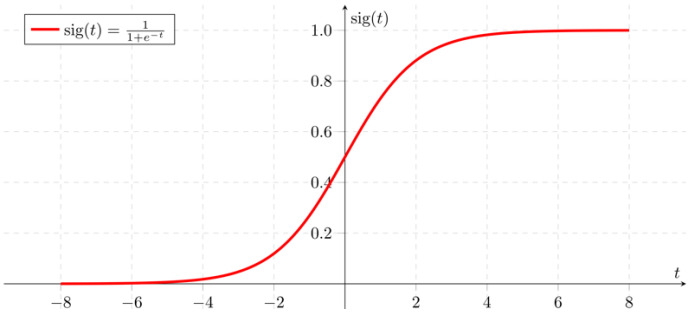

Figure 12. Sigmoid function.

2. Tanh: defined by the function $\frac{2}{1+e^{-2x}} - 1$. It is very similar to the sigmoid, also for the gradient vanishing problem. In particular, when a sigmoidal activation function must be used, the usage of this activation function instead typically performs better than the sigmoid [22]. The only difference is that, as we can see from the formula, the gradient is stronger for tanh, therefore the descent will be steeper;

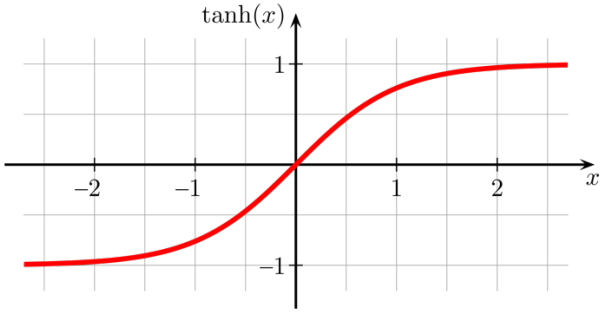

Figure 13. Tanh function.

3. ReLU (Rectified Linear Unit): defined by the function $\max(0, x)$ [22]. In fact, it gives as output the input, if it is greater than 0, otherwise it just outputs 0. These units are easy to be optimized because they are so similar to linear units: the only difference that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active, and provides a consistent gradient. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient

direction is far more useful for learning than it would be with activation functions that introduce second order effects. One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero [22], because ReLu associate simply 0 to negative examples.

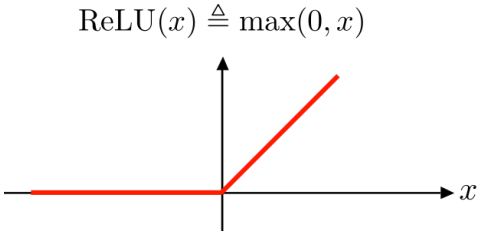

Figure 14. ReLu function.

ReLu is the default activation function recommended for use with most neural networks [22], but it is not uncommon to see also sigmoid or tanh used.

Through activation functions, signals travel from the first layer (the input layer) to the last layer (the output layer), after traversing the intermediate (hidden) layers. The output coming from the output layer is the one that determines the prediction of the neural network.

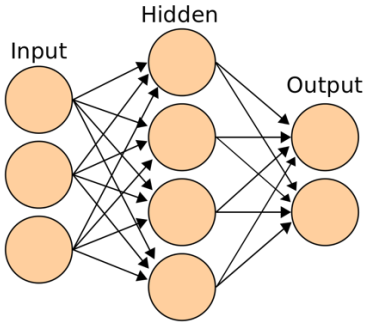

Figure 15. Simple NN architecture.

4.1.2 Weights' change

When employing a NN, the first important process through which it needs to undergo it is the learning process, by which it adapts to the problem at hand and "translate" the correlations in data into its parameters to reflect the observed domain. In order to carry out this process, we need to change the weights modifying their values.

One of the most famous methods is the gradient descent, that is a technique for which we can trace its origins back to Louis Augustin Cauchy, that invented this method to compute the orbit of a heavenly body because he wanted to avoid solving differential equations: instead, he wanted to use the algebraic equations representing the motion of this body. In order to do that, he did not want

to solve them as it is normally done by reducing them to one by successive eliminations, because he could eventually get stuck, for the eliminations are not always applicable or the resulting equation can be too complicated [13]. Therefore, he decided to go down on a different path, that eventually leaded to the definition of the gradient descent technique as we know it today.

Suppose we have a function $y = f(x)$, where both x and y are real numbers. The derivative of this function is denoted as $f'(x)$ or as $\frac{dy}{dx}$ and gives the slope of $f(x)$ at the point x. In other words, it specifies how to scale a small change in the input in order to obtain the corresponding change in the output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$. The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y. Given that for most of the time, in this domain, we minimize multi-input functions, we must make use of the concept of partial derivatives. The partial derivative $\frac{\partial}{\partial x_i} f(x)$ measures how f changes as only the variable x_i increases at point x. The gradient generalizes the notion of derivative to the case where the derivative is with respect to a vector: the gradient of f is the vector containing all of the partial derivatives, denoted $\nabla_x f(x)$. Element i of the gradient is the partial derivative of f with respect to x_i. Then, if we add a direction u to the derivative computation, we find out that f is minimized as fast as possible when u points in the opposite direction of the gradient. Thus, we can decrease f by moving in the direction of the negative gradient. This is known as the method of steepest descent or gradient descent. Steepest descent proposes a new point through the equation $x' = x - \epsilon \nabla_x f(x)$ where ϵ is the learning rate. Ultimately, it converges when every element of the gradient is zero (or, in practice, very close to zero) [22].

This short mathematical discussion quickly becomes very practical, if we imagine having to manage a loss or cost function like in the case of machine learning, where we usually want to minimize an error function like the total squared error $\frac{1}{N}\sum_{i=1}^{N}(Y' - Y)^2$, where N is the number of examples on which the error is calculated, Y' is the predicted value and Y is the expected value.

The only doubt regards how to employ the minimization of a loss function to make the NN learn, therefore how to change weights. In order to do that, we need to consider that in a NN we have many layers, and in each layer we have a different vector of weights. Starting from the weights of the output layer, we need to update each layer's weights until the input layer. As explained in [22], the most used method is for sure backpropagation: it consists on exploit the differentiable functions used throughout the NN, from a layer to another, to get the error for each weight. In particular, we know that a neuron is composed by an activation function and a set of connections (which are

weighted) to other neurons, that fuel the function above. We only need to use as activation functions the ones that are differentiable and then, calculate the derivative of the error function for each weight. This means that, for each weight, we need to calculate three derivative (following the chain rule of calculus):

1. The derivative of the error function with respect to the output of the activation function of the neuron to which the weight "arrives"
2. The derivative of the output of the activation function above with respect to the input of the neuron to which the weight "arrives"
3. The derivative of the input of the neuron above with respect to the weight considered

This is straightforward when calculating the weights' errors for the output layer, but when considering the hidden layers, we need to remember that when calculating the first derivative, we need to take into account as error the summation of the errors of the neurons of the next layer to which the neuron considered is connected.

Having obtained weights' errors, we now need to choose a method to define how to change weights. One of the oldest methods is the stochastic gradient descent (SGD), that can be traced back to 1951 by Herbert Robbins and Sutton Monro: the fundamental idea revolved around evaluating the gradient on a single example, instead that on the entire dataset, as it can be deduced from [24]. Through this approach, we can avoid compute the true gradient on the entire dataset, that can be very computationally expensive, obtaining anyway a decent precision for most of the classic machine learning purposes. We can define SGD as $\theta = \theta - \eta \cdot \nabla_\theta J(\theta;\, x^{(i)};\, y^{(i)})$, where J is an objective function, θ is the vector of weights, η is a learning rate, $x^{(i)}$ is a training example and $y^{(i)}$ the corresponding label [24]. As we said, it solves the problem of computationally expensive calculations, but it complicates convergence, as SGD introduces large fluctuations in the learning process [24]. In order to solve this, the best of both worlds have been merged to define a new approach, the mini-batch gradient descent: it simply updates the weights after a certain number of evaluated examples, with the number being less than the total number of records in the dataset, and not after any single evaluation [24]. This way, it reduces the wide fluctuations of pure SGD while keeping computation complexity under a reasonable limit [24].

However, many problems remain:

1. Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge [24];

2. Learning rate schedules try to adjust the learning rate during training by for example reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to the dataset's characteristics [24];
3. The same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features [24];
4. Minimizing highly non-convex error functions for neural networks can lead to getting trapped in their numerous suboptimal local minima [24].

One recent proposed extension to solve the aforementioned problems is Adam (Adaptive Moment Estimation), that computes adaptive learning rates for each parameter [24]. In addition to storing an exponentially decaying average of past squared gradients v_t, Adam also keeps an exponentially decaying average of past gradients m_t:

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\theta_t} J(\theta_t)$$
$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)(\nabla_{\theta_t} J(\theta_t))^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method; t indexes the current training iteration (starting at 0) and βs are forgetting factors for gradients and second moments of gradients. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small ($\beta_1$and $\beta_2$are close to 1). They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

These values are then combined as follows to update the weights:

$$\theta_{t+1} \leftarrow \theta_t - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}$$

Where ϵ is a small scalar used to prevent division by 0. The authors propose default values of 0.9 for β1, 0.999 for β2, and 10−8 for ε. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms [24].

4.1.3 Convolutional Neural Networks (CNNs)

Among the various types of neural networks defined over the years, for sure the CNN deserves an honorable mention, as it has been tremendously successful in practical applications. It is a type of neural network specialized for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels [22].

The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution in place of general matrix multiplication in at least one of its layers. The convolution is a specialized kind of linear operation particularly useful in case of noisy source of data. Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output x(t): the position of the spaceship at time t. Both x and t are real valued, that is, we can get a different reading from the laser sensor at any instant in time. Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship: $s(t) = \int x(a)w(t-a)da$; it is usually represented as $s(t) = (x * w)(t)$. The first argument to the convolution (in this example, x) is referred to as the input, while the second as the kernel (or filter); the output is sometimes referred to as the feature map [22], whose size depends on depth, that is the number of kernels applied, on stride, that determines how the filter convolves around the input data (by default, the kernel convolves around the input data by shifting one unit at a time) and on zero-padding, that permits to temporarily enlarge input size in order to convolve around the "border" of the input data too. The primary purpose of this operation is to extract features from the input data: what we do is to reduce the input size while preserving important characteristics of the input itself; like before, where we want to smooth the effective measurements received in order to compute a more precise position of the spaceship.

Once we end executing convolutions on input data, usually a detector layer is used, where the set of linear activation produced through convolutions are passed through a nonlinear activation

function (most of the time, a ReLU). The rationale behind adding this layer is the introduction of nonlinearity that helps the CNN learn [22].

Then, we have a pooling layer, that brings us to modify the output of the layer further. In particular, it replaces the output of the net with a summary statistic of the nearby outputs, like the max pooling, which reports the maximum within a rectangular neighborhood. It is used in order to make the representation invariant to translation: this helps in recognizing the presence of features, rather in where they are, because with pooling if we translate the input by a small amount, the values of most pooled outputs do not change [22].

At the end, we can also find dropout layers, that help a CNN to not overfit on the training data by setting a random number of activations to 0 [22].

This complex structure permits to obtain three important properties with respect to a traditional NN:

1. Sparse interactions: traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights): this is accomplished by making the kernel smaller than the input (as we saw before). Ultimately, it means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency, and computing the output requires fewer operations [22];
2. Parameter sharing: it refers to using the same parameter for more than one function in a model. In a traditional neural network, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural network, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency [22];

3. Equivariant representations: as we were saying, this CNN is invariant to translation. To generalize, when we say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function f(x) is equivariant to a function g if f(g(x)) = g(f(x)). In the case of convolution, if we let g be any function that translates the input, then the convolution function is equivariant to g [22].

Nowadays, CNNs are employed for various tasks, such as:

1. Face recognition: in [26], the first facial expression recognition system with subject independence has been presented, offering robustness with regard to variability in face images in terms of appearance and location. It has been proved that this system has robust face detection with 1% false rejection rate and 6% false acceptance rate with quite good generalization ability and reliable detection of smiles with the recognition rate of 97.6% for 5600 still images of more than 10 subjects [26];
2. Recommender system: in [29], a music recommender system has been created to extract the latent factors from music audio when they cannot be obtained from usage data. It has been shown how convolutional neural networks helps in detecting latent factors from new and unpopular music with respect to other classic methods like multi-layer perceptron and linear regression;
3. Text recognition: in [27], large, multi-layer CNNs are employed to train powerful and robust text detection and recognition modules with performances that outclassed methods based on elaborate models incorporating carefully hand-engineered features or large amounts of prior knowledge.

One last important use case of CNNs is the medical field. In particular, [28] showed how CNNs can help in detecting outliers of human body; in this case, lung abnormalities like lung nodules and diffuse lung diseases. Radiologists use computer-aided diagnosis (CAD) systems to carry out their identification work; these systems come in two types: computer-aided detection (CADe), in this case used to detect abnormal lesion, and computer-aided diagnosis (CADx), in this case used to detect if an abnormal lesion is benign or malignant. The problem is that these systems need image features to work properly. In order to circumvent this problem, in [28] has been showed how we can think of using a CNN that is properly trained through lung images in order to easily detect anomalies. The model developed in order to solve the CADe issue makes use of a CNN to extract features from images, then these features are used to train a support vector machine (SVM). In CADx, it has been showed also how a R-CNN (an object detection framework based on a CNN) can be used for

bounding abnormalities in an image; once obtained these bounding boxes, again a SVM has been used to classify them into a lung nodule or a region with diffuse lung disease pattern. CADx results proved very efficient, yielding a 99.4% as the mean accuracy for benign and malignant lung nodules and an 84.7% for five patterns of diffuse lung diseases. CADe results also showed a good functioning, correctly detecting lung nodules attached to the chest wall and mediastinum and various kinds of nodules such as nodule with air bronchogram and nodule with ground-glass opacity; it also well detected diffuse lung disease patterns.

4.2 Related software

Throughout the years, many tools have been created to ease the development of machine learning models in order to automatize both most wanted features and expected common behaviors and convey them into user-friendly software. Given that machine learning is such a broad domain, these instruments sometimes target a large spectrum of stakeholders, therefore for most of the time we will find ourselves only marginally using them, as the much-needed functionalities are spread among various libraries and frameworks. In the following paragraphs, we are going to describe tools strictly related to Python, as it has been decided (in order to give continuity to fogØ5's programming model) to use it as the main programming language for this thesis.

4.2.1 Data-managing libraries

NumPy builds on (and is a successor to) the successful Numeric array object. Its goal is to create the cornerstone for a useful environment for scientific computing [14]. The fundamental objects it provides are:

1. ndarray: an n-dimensional array object. An n-dimensional array is a homogeneous collection of "items" indexed using n integers. There are two essential pieces of information that define an n-dimensional array: the shape of the array and the kind of item the array is composed of. The shape of the array is a tuple of n integers (one for each dimension) that provides information on how far the index can vary along that dimension. The other important information describing an array is the kind of item the array is composed of. Because every ndarray is a homogeneous collection of exactly the same data-type (dtype), every item takes up the same size block of memory, and each block of memory in the array is interpreted in exactly the same way [14];

2. ufunc: a universal function object. NumPy provides a wealth of mathematical functions that operate on the ndarray object. From algebraic functions such as addition and multiplication to trigonometric functions such as sin and cos. Each universal function (ufunc) is an instance of a general class so that function behavior is the same. All ufuncs perform element-by-element operations over an array or a set of arrays (for multi-input functions) [14].

Moreover, NumPy is the foundational library for many other useful tools, like Scikit-learn and Pandas.

Scikit-learn is a toolbox built around SciPy (another useful Python library, part of the NumPy stack) that provides various utility methods and algorithms like classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN [15].

Pandas, otherwise, is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive [16]. Pandas is well suited for many different kinds of data:

1. Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet [16];
2. Ordered and unordered (not necessarily fixed-frequency) time series data [16];
3. Arbitrary matrix data (homogeneously-typed or heterogeneous) with row and column labels [16];
4. Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a Pandas data structure [16].

The two primary data structures of Pandas, Series (1-dimensional) and DataFrame (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering [16].

4.2.2 Machine learning tools

Nowadays, many platforms have been created to support developers in their machine learning related works. We can mention:

1. TensorFlow: an interface for expressing machine learning algorithms, and an implementation for executing such algorithms. A computation expressed using TensorFlow can be executed with little or no change on a wide variety of heterogeneous systems, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of

hundreds of machines and thousands of computational devices such as GPU cards. The system is flexible and can be used to express a wide variety of algorithms, including training and inference algorithms for deep neural network models, and it has been used for conducting research and for deploying machine learning systems into production across more than a dozen areas of computer science and other fields, including speech recognition, computer vision, robotics, and so on [18];

2. PyTorch: a library designed to enable rapid research on machine learning models. It builds upon a few projects, most notably Lua Torch, Chainer, and HIPS Autograd, and provides a high-performance environment with easy access to automatic differentiation of models executed on different devices (CPU and GPU). To make prototyping easier, PyTorch does not follow the symbolic approach used in many other deep learning frameworks, but focuses on differentiation of purely imperative programs, with a focus on extensibility and low overhead [19];
3. Microsoft Cognitive Toolkit (CNTK): a unified deep learning toolkit that describes neural networks as a series of computational steps via a directed graph. In this directed graph, leaf nodes represent input values or network parameters, while other nodes represent matrix operations upon their inputs. CNTK allows users to easily realize and combine popular model types such as feed-forward DNNs, convolutional networks (CNNs), and recurrent networks (RNNs/LSTMs). It implements stochastic gradient descent (SGD, error backpropagation) learning with automatic differentiation and parallelization across multiple GPUs and servers [17].

All these platforms are valid tools that can be employed if we need to solve a machine learning related task. In particular, for this thesis, we decided to use TensorFlow as it is for sure the most popular one, with one of the biggest communities of the domain.

But we did not use directly TensorFlow: instead, we decided to use it through Keras, to lower even more the needed time for prototyping. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano (but this last one is no longer supported by the community). It allows for easy and fast prototyping (through user friendliness, modularity, and extensibility), supports both convolutional networks and recurrent networks, as well as combinations of the two and runs seamlessly on CPU and GPU [23].

5. Problem analysis and derived solution components

For this thesis, we decided to use all the aforementioned tools in order to solve a simple condition monitoring problem. What we aim to do, in fact, is not to discover the best solution for a complex machine learning problem, but to show how easily we can think of defining and maintaining a prediction service by using principally fogØ5 and modern distributed machine learning architectures. Therefore, our work plan is going to consist of seven different phases:

1. Analyze the problem at hand on the basis of the associated data;
2. Create the solution components we are going to exploit to build our machine learning architectures;
3. Choose the best suited distributed approaches for our purpose;
4. Encode them exploiting the aforementioned components;
5. Define a deployment program to effectively and efficiently manage the architectures we are going to employ;
6. Map each node of the architectures to a fogØ5 atomic entity;
7. Deploy the above architectures using the deployment program.

In this chapter, we are going to address the first two phases.

5.1 Domain analysis

The problem at hand consists of monitoring a hydraulic test rig. This test rig consists of a primary working and a secondary cooling-filtration circuit which are connected via the oil tank. The system cyclically repeats constant load cycles (lasting 60 seconds) and measures process values such as pressures, volume flows and temperatures while the condition of four hydraulic components (cooler, valve, pump and accumulator) is quantitatively varied [30].

The associated dataset (composed by 2205 records near-perfectly balanced across the available classes, without missing attribute values) addresses the condition assessment of this hydraulic test rig based on multi-sensor data [30]. In particular, it offers:

1. Raw process sensor data (without feature extraction): continuous numeric values which are structured as matrices (tab-delimited) with the rows representing the cycles and the columns the data points within a cycle. Pressure, motor power, volume flow, temperature, vibration, efficiency, virtual cooling and virtual cooling power sensors are provided [30];

2. Hydraulic components states: for each of the aforementioned components, this dataset offers the associated status with respect to the raw process sensor data. Therefore, for each second of sampling (corresponding to one row of the dataset), we have the corresponding status for each component. The cooler, valve, pump and accumulator conditions are provided, together with a so-called stable flag which is an overall indicator of the stability of the hydraulic test rig. All of them, except for this last one, describe degradation processes over time and, thus, their values do not represent distinct categories, but continuous values [30].

Our purpose, for this thesis work, is to predict the cooler condition using the temperature sensors. The cooler condition, in this dataset, is represented by three different states: close to total failure (represented by the number 3), reduced efficiency (number 20) and full efficiency (number 100); as stated before, they are not categories but continuous values, but for the purpose of this thesis we can safely consider them as labels. The temperature sensors are four, sequentially named from TS1 to TS4, that have a sampling rate of 1 Hz, leading to have 60 attributes per row for each sensor; these attributes are temperatures in degrees Celsius [30].

5.2 Solution components

Here we are going to define the solution components that we are going to reuse in the following chapters, on the basis of the analysis just finished. In particular, we are going to need a neural network and an algorithm to properly transform data to feed the neural network.

5.2.1 Neural network topology

In searching for the best model to use in order to tackle as effectively as possible the hydraulic test rig monitoring problem, the CNNs (presented in the previous chapter) immediately appear as one of the best choices, in particular for what concerns their “one-dimensional form”, as it can be found stated in [31], especially for anomaly detection. Therefore, we decide to adopt them by employing the following neural network, obtained modifying the model suggested by Keras for sequence classification [23]:

```
model = Sequential()
model.add(Conv1D(100, 6, activation='relu', input_shape=(4, 60)))
model.add(Conv1D(100, 6, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(160, 6, activation='relu'))
```

```
model.add(Conv1D(160, 6, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

The Sequential object creates an empty neural network that can be composed by a linear stack of layers. The first one we add is a 1D convolutional layer, that creates a convolution kernel that is convolved with the input layer over a single dimension to produce a tensor of outputs. In our case, we specifiedthe number of filters (100, the first argument), the kernel size (6, the second argument) and the activation function (ReLu, the third argument), and we obtaine an output matrix consisting of features extracted from the input layer: in our case, a 55x100 matrix. This happens also because of the input layer, that is implicitly created by specifying the fourth argument of the above layer where we feed a tuple specifying the dimensions of the input data: in our case, we have a 60x4 matrix in order to consider, for each input row, a temperature reading for each available sensor. Then, we have another convolutional layer, identical to the one just described. As a third layer, we insert a maximum pooling layer, taking 3 rows at a time (as pointed out from the first and only argument) and substituting them with the maximum value obtained from them. Then, we use another two convolutional layers extracting 160 features (always using a kernel size of 6 and ReLu as the activation function) that combined produce a 6x160 matrix that is fed to a pooling layer that computes the global average returning a vector of 160 data points. Next, a dropout layer is added, deactivating half of the activation functions (as the argument is set to 0.5). At the end, our output layer is a dense layer (a regular-densely connected NN layer [23]) composed by 3 nodes (as stated in the first argument); these nodes have as activation function softmax, a function similar to a sigmoid that assigns for each node in the layer a value that, added to the others already assigned, equals 1: this function is extremely useful in our situation because the cooler condition can have at most 3 different values, therefore each one of our nodes returns the probability of the input data to belong to each of the different cooler conditions and the highest returned probability has to be the predicted condition.

Once created the topology of our neural network, we finish its definition by using the method compile, which allows us to specify (following the order of the arguments) the error function to minimize, the update function through which change the weights and the metrics to keep track of while training or testing. First, as for the loss function, we adopt the categorical crossentropy instead of the binary crossentropy as our error function, as we have more than two labels (therefore it

would not be correct to use binary crossentropy) and each example can be associated with one and only one label (our problem is a multi-class classification problem); this loss function can be defined as $-\sum_{i=0}^{N} \log(y_{pred_i})$, where y_{pred_i} is the probability computed for the example to belong to the true class, therefore the loss is as high as the uncertainty (to be intended as 1 minus the predicted probability of labeling a specific example with a certain class) of belonging to the true class. Second, as for the update function, we use Adam instead of RMSProp, as the former can be seen as a more structured approach at updating weights with respect to the latter [24]. Finally, as for the metrics, we pick accuracy, as it is for sure the most immediate and useful indicator for our problem at hand: in particular, Keras is implicitly going to use the categorical accuracy (as we are using the categorical crossentropy as our loss function), that is the mean number of times that the highest predicted probability was associated to the true class.

5.2.2 Data transformation

Now that our machine learning model is ready, we need to transform our input data in order to conform to the input shape requested by the input layer and enable the training, evaluation and predict processes. The resulting transformation algorithm is as follows:

```
label = pd.read_csv(os.path.join(fileDir, 'profile.txt'), sep='\t', header=None)
label = np_utils.to_categorical(label[0].factorize()[0])

df = pd.DataFrame()
for txt in data:
        read_df = pd.read_csv(txt, sep='\t', header=None)
        df = df.append(read_df)
df = df.apply(self.__train_scale)
df = df.sort_index(kind='mergesort').values.reshape(-1,4,60).transpose(0,2,1)
```

In the first section, we transform the numbers associated to the cooler conditions to categorical labels. In particular, we use the method read_csv from Pandas to create a DataFrame where each row is the cycle number and each column a component condition: therefore, in our case, we obtain 5 columns. Then, we select the first column of the DataFrame, corresponding to the cooler condition, and we transform it using factorize, obtaining an array consisting of the same values of the column selected, but with each original value mapped to different unique sequential numbers from 1 to infinite (the association between the input value and the transformed one is memorized, so that if the algorithm re-encounters the same value in input it is going to use the same mapping as before). This last output, at the end, is then fed to method to_categorical from NumPy, that

applies a one-hot encoding to the input data: in our case, it results in having 3 <-> 100, 20 <-> 010, 100 <-> 001; this step is fundamental as the categorical crossentropy in Keras needs the labels to be one-hot encoded to work properly.

In the second section, we transform the temperature data. We have four different files, one for each temperature sensor, therefore we read each one of them (always through read_csv) and we concatenate them together through the method append. At the end of this process, we have a DataFrame with a row for each cycle and a column for each temperature reading executed in one minute (as we said before, we have 60 columns). Once we have all the data together, we first need to normalize them, as we have that each temperature sensor is located in a different position on the hydraulic test rig, therefore each one of them manage a different value range from the others: if we do not apply the normalization we could have that one or more sensors could mislead our training and predict processes with their "out-of-bound" readings; the normalization applied in our case is a simple standardization where each data point of each column of the DataFrame is subtracted by the mean and then divided by the standard deviation (both calculated on the considered column). Once we have our dataset normalized, we need to reshape it in order to properly feed the neural network defined before. This is done by first using the method sort_index, that orders each row in ascending order: by applying this function, we expect to pair all the readings associated to the same cycle near each other; for example, after this transformation, we are going to have four rows (one for each temperature sensor) with index 0 (cycle 0), then other four rows with index 1, and so on. Once they are coupled, we can obtain a NumPy array out of it and feed it to the method reshape to change the form of our array: in our case, we ask reshape to take our n * 60 2D matrix into a n * 4 * 60 3D matrix (through the arguments -1, 4 and 60, where the first one means no preference, that is "translated" in an automatic generation of the number of rows resulting from the transformation of the actual matrix in the preferred shape specified by the other two positive arguments), therefore obtaining for each 1D member of the array one sensor reading for each temperature sensor. But still, this is not the shape we need to obtain, as the target one needs to get the last two dimensions swapped: in order to get to that form, we can apply transpose and swap the second dimension with the third one, obtaining the final result of a n * 60 * 4 3D matrix, where each 2D sub-array contains the set of sensors' readings for the specified cycle, each 1D sub-array contains a measurement for each sensor for the specified cycle (so, four measurements), and each atomic element is a measurement of the sensor requested for the specified cycle.

6. Analysis of distributed machine learning approaches

Now, it is time to carry out phase three. As we were saying in the previous chapters, we are forced to provide various architectures because the cloud-centric solution, that represents (we could say) the standard and most used approach nowadays for machine learning problems, in spite of offering high accuracy, can create problems as we cannot always rely on an always-available cloud. In light of this, it is important to provide alternative solutions that should (if possible) be deployed efficiently and automatically when the cloud is not available. Among the many distributed machine learning approaches available in papers, two of them that acquired a certain popularity (and therefore the one we choose to implement and deploy for our problem at hand) are the gossip learning and the federated learning solutions. The concept at the foundation of these two architectures is the collaboration: we can easily imagine how many different factories of the same company can be put in communication, by placing a node in each one of them (we could call them edge nodes) and then make them exchange information related to the production of their factories using one of the architectures above; for sure each one of these nodes, taken individually, will not have the computational and storage capacity of the cloud, but together they can somehow overcome this limitation and acquire (as we are about to see) cloud-like performances, while minimizing latency and preserving privacy and control over the produced data.

6.1 Cloud-centric solution

As we were saying, as a starting point, we cannot think of not implementing a classic cloud-centric prediction service, that should offer us the best performances in terms of accuracy and time needed to properly train the neural network. In doing so, we imagine having an always-available cloud ready to host the entire dataset of 2205 records (for training and testing) and the prediction service.

6.1.1 Training and testing processes

Supposing to already have the neural network defined in the previous chapter, along with the data transformation algorithm, we can just write the following instructions to issue a training and testing process on the cloud:

```
X_train, X_test, y_train, y_test = data
self.model.fit(X_train, y_train, batch_size=32, epochs=10, validation_split=0.2,
verbose=1)
```

```
self.model.evaluate(X_test, y_test, verbose=1)
```

In the block of code above, the variable data represents the variables label and df (shown in 5.2.2) that were divided in two datasets: 80% of the original dataset for the training dataset, where X_train contains the labeled examples and y_train the corresponding labels, and the 20% remaining data for the test set, with X_test and y_test equivalent to X_train and y_train.

Then, first we use X_train and y_train to train the model by calling the method fit: we specify to update the weights only after 32 evaluated samples following the principles of mini-batch gradient descent (through the argument batch_size), to make 10 epochs of learning (argument epochs), and to split a 20% of the training set to use it as the validation set (argument validation_split).

Finally, we estimate the test accuracy through the method evaluate, using X_test and y_test. From 20 tests carried out with the above settings, it has been showed how the mean accuracy can be measured between 0.99 and 0.985. Further increasing the number of epochs and/or reducing the batch size usually results in an overfitted model, uncapable to correctly classify edge examples.

6.1.2 Prediction service

Finally, we have our neural network ready to make predictions and therefore calculate which is the current condition of our hydraulic test rig. In order to offer the aforementioned service, we are going to offer a REST API using Flask [34] as follows:

```
@app.route("/predict", methods=['POST'])
def predict():
        r_json = request.get_json()
        data = r_json['TS1'], r_json['TS2'], r_json['TS3'], r_json['TS4']
        max_prob_index = np.argmax(model.predict(dl.transform(data)))
        if max_prob_index == 0:
                prediction = '3'
        elif max_prob_index == 1:
                prediction = '20'
        else:
                prediction = '100'
        print("Current predicted status: " + prediction)
        return prediction
```

As we can see, it is pretty straightforward: we get the data from the arrived request and we combine together the data (originally divided by the name of the sensor producing them) to obtain the format expected by the method transform (that acts on the data as we saw in 5.2.2). Then, we feed the transformed data to the method predict, that generates output predictions for the input

samples [23]: for each possible target label (3, 20 and 100), we obtain a probability for the input samples to belong to that label. But we do not directly store this array of prediction, as we pass it to the function argmax that returns the index of the highest value inside that array. Remembering the mapping obtained before between classes and one-hot encodings (3 <-> 100, 20 <-> 010, 100 <-> 001), we have that if the maximum probability index corresponds to the first element of the array, then we have 3, otherwise if it corresponds to the second element, we have 20, and in the remaining case, 100. Having obtained the predicted class, we print it out and then return it to the client.

6.2 Distributed machine learning approaches

Now that we presented the cloud-centric solution, we need to think how we can adapt that implementation to exploit the distributed approaches we choose to implement.

6.2.1 Gossip learning

One of the first papers to expose the gossip learning to the academic world with relevant results was for sure [32]. In this paper, different approaches implementing the gossip learning idea are presented, but we are not going to exactly replicate any of them, as we are only going to take away the key concepts and apply them using the aforementioned tools.

In this type of solution, there is only one type of node to implement: we are going to call this node the edge node, as in our case is the node inside the factory. Each one of these edge nodes can be thought as simple servers that have a fixed timeout (it is the same for everyone) that when expires triggers the sending of information to other edge nodes (or peers). The information sent are two: a model and a subset of peers.

For what concerns the former, we need to say that each node, when it starts, prepares the same neural network as presented in the previous section. Then, when the timeout expires, the node sends the above model to a peer. When that peer receives the model, it combines that one (we will see in a moment how this operation can be carried out) with the most recent neural network it has in store (that is always the last received model) to obtain a new improved model. Each model loaded this way is added to a cache of models that are kept in memory to implement an ensemble classification method, where each neural network provides its prediction that are then added up and the most predicted class is the one that gets to label the considered example. Of course, we cannot think of memorizing a large number of models, since as we were saying before, the

computational capacity is limited: we solve this issue by memorizing a constant number of models and discard the oldest ones, if the cache is already full.

Instead, for what concerns the latter, it is a common concept in large gossip architectures: nodes do not have the entire "vision" of the network they belong to, but instead they start operating only knowing a small subset of the existing peers. Knowing all the nodes would be very useful, both for having many alternatives in case one peer does not respond and for reaching many different peers (that is fundamental in our case, as we need to train our models on as many examples as we can, and since now these nodes cannot contain the same number of examples as the cloud, these examples are scattered all across these edge nodes). But as it is already explained in [33], it is unrealistic to give to all the nodes the entire view of the network. Therefore, we are forced to load each peer with only a subset of the entire set of existing nodes, and then use a decentralized approach (a gossiping algorithm that we are going to analyze) to refresh the local subset removing the oldest (and maybe fault) peers and adding new working peers. One of the main aspects of this algorithm is to send a subset of the local known peers to another peer, when the timeout expires: this procedure ensures that both new participating nodes can be discovered and old (probably not functioning) nodes can be deleted.

Finally, it is immediate to map the edge node we presented in this approach to the cloud of the cloud-centric solution, as it is obvious that the edge node is going to host the neural network (and its related operations, along with the data transformation algorithm), but also the prediction service to monitor our hydraulic test rig.

6.2.2 Federated averaging

While the gossip learning has come a long way since its first presentation to the academic world, a fairly new machine learning approach is for sure the federated learning, presented for the first time in 2016 and exploited by Google researchers in [35]. It differs from the traditional distributed machine learning with parameter server because in federated learning the node who produces data is the data owner and choose when and how to help to carry out machine learning tasks, as opposed to parameter server that manages data as needed and impose jobs to be executed [36]. Focusing now on the proposed approach, even if it has been specifically defined for mobile devices, we can understand how it offers many interesting concepts that can be easily reused in various use cases, and also for our problem at hand.

Its main idea is not to aggregate data into the cloud, but directly perform machine learning tasks where the data are and then combine their results together: through this approach, we preserve privacy and control over data, as we do not need to move them to the cloud, and also, we do not need to have a cloud as computationally capable as intended in the local machine learning solution. In order to reach this goal, the proposed solution is composed by many different types of node. Of course, as in the architecture before, we are not going to map each proposed concept to an actual implementation, but only the types of nodes that are relevant for our use case are going to be discussed in depth. Here are the ones that we are going to implement:

1. Mobile device: in our case, it is the edge node present in each factory. It contains data produced by the factory it manages and has the computational capabilities to perform machine learning tasks. Once these tasks are carried out, their results are then sent to the aggregator to which they refer to;
2. Aggregator: in our situation, it can be a fog node present in the network "between" the coordinator and the edge node, but "nearer" to the edge node. It combines together two or more outcomes coming from the edge nodes and send the aggregate to the master aggregator. This is done in order to distribute the combination operations instead of leaving everything up to the coordinator;
3. Master aggregator: in our domain, it can be a fog node as the one before, but "nearer" to the coordinator with respect to the aggregator. It merges together the results coming from the aggregators and send the final value to the coordinator. As before, this is done in order to ease the workload on the coordinator;
4. Coordinator: it represents the cloud. In particular, it maintains the neural network in memory and manages the operations to be carried out in the network by notifying them to the edge nodes that effectively execute the tasks. Once the tasks are terminated, results come back to the coordinator following the chain we explained in the previous points.

For the sake of completeness, it is to be noted that in [35] another entity is proposed, the selector, that chooses which mobile devices need to execute the task requested and report to aggregators; in our use case, each one of the edge nodes is always ready to execute a task and therefore there is no need, like in [35], to choose the best edge nodes based on their availability.

As it can be deduced from the description above, the presented architecture can offer, we could say, the execution of machine learning related tasks on-demand (as opposed to the gossip learning architecture, where the training is implicitly continuous), just by notifying the coordinator about the

type of task to be carried out and how we want it to be performed. This permits, moreover, to define, if it is detectable with respect to the problem at hand, a workflow of machine learning tasks (to be executed one after the other) that should bring the neural network to converge to a good grade of accuracy in the shortest time possible; this is particularly useful if we think we could experience a change in the correlations between data that negatively impact on the prediction quality, as with a ready workflow we can react, learn (also from scratch, if needed) the new correlations and come back fully operational as fast as possible.

For what regards mapping, here we have the same situation of the gossip learning approach: the only difference is that the edge node receives the neural network from the coordinator and then provide the prediction service.

Finally, for what concerns overall requirements, this approach seems to be halfway between the cloud-centric solution and the gossip learning approach, but the types of nodes involved here are many more than both of the previous solutions, therefore the workload can be much more distributed: if we have a widespread network ready to host the entities just described, we can deploy each one of these nodes to the most appropriate VM available in order to carry out various machine learning tasks while minimizing latency.

7. Implementation and validation of chosen distributed machine learning approaches

We can now execute phase four and delve into the implementation and the validation of the machine learning approaches chosen in the previous chapters.

7.1 Gossip learning architecture

7.1.1 Implementation

As described in the previous chapter, we only need to describe the implementation of the edge node, but being that it is very complex, we are going to divide the presentation of the implementation in four sections, to better highlight the crucial aspects of the functioning: server-side implementation, client-side implementation, the ModelRepository class (the manager of the models) and the PeerRepository class (the manager of the subset of peers).

7.1.1.1 Server-side implementation

First, we can examine how the server-side of the edge node has been created:

```
@app.route("/model", methods=['POST'])
def model():
        data = json.loads(request.form['json'])

        file_name = 'my_model' + str(datetime.timestamp(datetime.now())) + '.h5'
        file_path = os.path.join(os.path.dirname(os.path.abspath(__file__)),
file_name)
        exported_model = request.files['file']
        exported_model.save(file_path)
        mr.load(file_path, data['age'], train_data)

        peers = []
        for peer in json.loads(data['peers']):
                peers.append(AgedPeer.from_json(peer))
        pr.load(peers)
```

Using Flask [34], we create a REST API that is executable through a POST invocation. In the request, the server expects to find a binary file and a JSON. In the first block of code, after having obtained

the JSON through the first instruction, we take care of the file, that is no other than the model we are receiving from another peer: with the first two instructions, we define a path to which we are going to save the file, then we access the file through the instruction request.files['file'] and we save it to the path defined before. Then, we memorize it by passing its file path to the method load of the ModelRepository, which is a class managing the models received (we will examine it later on), together with the age of the model (useful to find out the oldest one and then remove it when needed) and the training data needed to train the model just obtained. In the second block of code, instead, we manage the received peers from the calling node by loading those peers from the JSON and converting them to instances of the class AgedPeer, which consists in a class storing an address (used as identifier and to contact that peer) and an age, and finally memorizing them through the method load in the PeerRepository, which is a class managing the subset of peers that the receiver node holds (we will analyze it shortly).

Apart from the REST API just presented, the server also offers an endpoint to provide the prediction service: it is not different from the REST API defined for the cloud-centric solution, therefore we are not going to analyze it again.

7.1.1.2 Client-side implementation

Now, it is time to dive into the client-side of the node:

```
recipient, others = pr.select()
if others:
        model = mr.get_freshest_model()
        exported_model, file_path = model.export()
        self.__send_model(exported_model, model.age, recipient, others)
time.sleep(pause)
```

In this block of code, we have the loop we were describing before, where each peer waits the same pause and then sends a subset of its known peers, along with the most recent combined model, to a random peer. In fact, with the first instruction, a random peer and a subset of the locally available peers are selected through the method select of the class PeerRepository, and then, if the selection successfully executes, the node sends its most recent received model and its age (we will see in a moment through which strategy it is chosen), together with the chosen subset of its locally available peers to the target node through the method send_model, that simply sends a POST request to the chosen peer with the information described above. Then, the node waits a constant number of seconds specified in the variable pause prior to repeat the above operations.

7.1.1.3 ModelRepository

We have seen what happens from a high-level point of view on the server and client-side of these edge nodes, now it is time to analyze in depth how the various strategies presented before really works. Starting from the ModelRepository, we first have the constructor:

```
def __init__(self,train_data):
        self.max_size = 10
        self.aged_models = []
        self.aged_models.append(AgedModel())
        self.aged_models[0].train(train_data)
        self.last_model = self.aged_models[0]
```

In this constructor, we define the maximum size of the cache, assigning it to 10 with the first instruction. Then, with the next instructions, we create the actual cache and we append a newly created AgedModel (a class containing a model completely analogous to the neural network defined in the previous section and a timestamp of the current time representing the age of the model, needed to discriminate which model needs to be eliminated when the cache is full) and we train it on the local available records. At the end, we only initialize as the last received model the one we just trained: we are going to need it when the first peer will contact this node, as we can see in the next block of code.

```
def load(self,exported_model,age,train_data):
        aged_model = AgedModel(exported_model, age)
        self.aged_models.append(self.create_model_mu(aged_model,
self.last_model, train_data))
        self.last_model = aged_model
        self.aged_models.sort(reverse=True, key=lambda x: x.age)
        if self.max_size + 1 == len(self.aged_models):
                self.aged_models.pop()
```

The method load, as we saw before, is the core logic that describes how models are combined and the cache is maintained. In particular, we can see in the first instruction how the model just received (still under the form of a simple file) is used (along with the age, provided as an argument) to instantiate an AgedModel that is then combined to the last received model and appended to the cache (for this purpose, we use the method create_method_mu, that identifies a precise approach of combining models, but it is not the only one; we will delve into the strategies in a moment). Then, we change the last received model to the actual received one and finally we remove the oldest model in the cache if its size is greater than the one specified in the constructor.

The last important concept to analyze regarding ModelRepository is about the strategies to combine models together:

```
def create_model_um(self,aged_model1,aged_model2,data):
        return
self.merge(self.update(aged_model1,data),self.update(aged_model2,data))

def create_model_mu(self,aged_model1,aged_model2,data):
        return self.update(self.merge(aged_model1,aged_model2),data)
```

As reported in the block of code above, we have essentially two methods for combining models, and it is easily understandable how one is the contrary of the other, in terms of order of operations. The former, create_model_um, first update both models then merge them together, while the latter, create_model_mu, does exactly the opposite. With update, we train the specified model with the local available records, while with merge, we compute the average of the weights between the two models and then creates a new AgedModel using those weights. As demonstrated in [32], these two techniques are not equivalent, as the create_model_mu preserves a greater independence between the models, therefore we are going to use it as it could be highly beneficial with regards to our ensemble method for classification.

7.1.1.4 PeerRepository

The last component remained to examine is the PeerRepository:

```
def __init__(self, max_size, healing_factor, swap_factor, my_info, peers_info):
        self.max_size = max_size
        self.separation_factor = math.ceil(self.max_size / 2) - 1
        self.healing_factor = healing_factor
        self.swap_factor = swap_factor
        self.my_info = AgedPeer(my_info)
        self.aged_peers = []
        for p in peers_info:
                self.aged_peers.append(AgedPeer(p))
```

This class is instantiated through three different parameters (the three initial ones) that configure the functioning of the select and load methods (as we are going to see shortly), and two additional parameters (my_info and peers_info) that are respectively the information to contact the actual node and the subset of existing peers (taken randomly from the entire network). Going in depth on the parameters we have max_size, that defines the maximum number of memorizable peers, separation_factor, that indicates how many peers are going to be suggested to the other peers, healing_factor, that permits to ignore and progressively eliminate old peers, and swap_factor, that

allows to not entirely load the peers received from another node in order to preserve the number of unique addresses existing between this node and the calling one. These constants rapidly find a practical usage in the select and load methods of this class. First, we can start by examining the select method:

```
def select(self):
        result = []
        self.aged_peers.sort(key=lambda x: x.age)
        if self.healing_factor == 0:
                newest_peers = list(self.aged_peers)
                oldest_peers = []
        else:
                oldest_peers = self.aged_peers[-self.healing_factor:]
                newest_peers = list(self.aged_peers[:-self.healing_factor])
        random.shuffle(newest_peers)
        temp_peers = newest_peers + oldest_peers
        result = copy.deepcopy(temp_peers[:self.separation_factor])
        result.append(self.my_info)
        self.__increase_age()
        result.sort(key=lambda x: x.age)
        return result[-1], result[:-1]
```

The selection starts by sorting the peers by their age. Once the ordered list is obtained, we divide it in two parts, the first representing the oldest peers and the second the newest ones (if the healing factor is different from 0, otherwise everyone is considered new). Then, the newest ones are shuffled randomly, and the resulting list is concatenated to the oldest ones. Now, it is time to use the separation factor to select the subset of peers we are going to send to the destination node but not before appending the information of the current node; in doing this operation we execute a deepcopy, in order to carry out the increase_age operation without increasing the age of the selected peers before sending them. At the end, we define our strategy for selecting which peer is to contact and which peers are to be suggested: we communicate with the oldest one and we suggest the others; another strategy could be randomly choosing a peer from the selected ones to be the one to contact, but it would not maximize the possibility of knowing new peers as we should usually have contacting the oldest known peers, because they obviously represent the least recently contacted peers and therefore the ones with the highest probability to have the most different subset of peers with respect to the local one.

The last algorithm remained to examine is the method load, which locally saves the received peers:

```
def load(self, aged_peers):
```

```
        for p in aged_peers:
                self.aged_peers.append(p)
        temp = []
        for i in self.aged_peers:
                if i not in temp:
                        temp.append(i)
                else:
                        for j in temp:
                                if i == j:
                                        if i.age < j.age:
                                                temp.remove(j)
                                                temp.append(i)

        temp.sort(key=lambda x: x.age)
        if len(temp) > self.max_size:
                if self.healing_factor > 0:
                        temp = temp[:-min(self.healing_factor, len(temp) -
self.max_size)]
                temp = temp[min(self.swap_factor, len(temp) - self.max_size):]
                random_delete_number = len(temp) - self.max_size
                for i in range(0, random_delete_number):
                        temp.pop(random.randrange(len(temp)))

        self.aged_peers = temp
        self.__increase_age()
```

As it can be easily noticeable, in the first section of the block of code, with the first two for loops we save the received peers in our local subset (first loop), removing the duplicates while preserving the memorization of the most recent ones (second loop). Then, we start removing surplus peers until the subset comes back to the maximum size set. The strategy for removing peers starts with reordering peers from the most recent to the oldest one: this is essential, as the following removals are strictly dependent on the age of the peers. In fact, we have three different type of removals: the first is about removing the oldest peers using the healing factor, in order to delete potentially fault peers, then the second deletes the most recent ones using the swap factor, in order to not lose completely the uniqueness of the subsets of the actual node and the calling node, and finally, the third and last removes random element from the local subset. Once completed all these deletions, in the last section of code we store the resulting subset and increase its peers' age.

7.1.2 Validation

Now that the edge node is implemented, we can start to prepare a test architecture and test it in order to evaluate different metrics such as the maximum reachable accuracy, the contacted peers, the suggested peers and the known peers.

7.1.2.1 Test settings

Having prepared as explained the current architecture, it is time now to evaluate the best possible configuration for the many parameters existing in the coded algorithms, to find out the best combination of them with respect to the problem at hand. As a first parameter to set, we can of course think of the pause time between consequent sends of the most recent model in a node: higher the wait, higher the needed time to reach convergence for the nodes, but we also need to think that we need sufficient time to receive and merge models from other peers too, otherwise we could find ourselves in the situation where a node pushes the same model over and over again to other peers (we want to avoid this situation, as we aim to have as many different models as possible); concluding, we decide to set the pause to 10 seconds, as our test environment is not capable to keep up the pace with shorter pause time. Then, we set other two of them, being the cache size and the combination technique for two models, as suggested in [32], therefore we have a maximum number of 10 models memorized simultaneously per node (to better compare our results with the ones obtained on [32]), while we choose the combination technique to be merge and then update (as said before, to preserve as much as possible the independence between models, as we are going to use an ensemble method for classification). Also, for what regards the training process of the model in each node, we decide to follow the same scenario laid out in [32], where we have fully distributed data (for the lack of a centralized entity and/or privacy issues): in our case, we have 3 records for each node, where each record is an example of a different class to predict (therefore, having 3 different class to recognize, we have 1 example for each class); moreover, the intersection between all the distributed subsets of 3 records is empty, therefore a model always train on fresh information when it reaches a new node. Having 3 records per dataset, we also need to use a batch size of 3 records, as we do not want to apply corrections on the weights based only on the error in predicting one class because it could introduce major fluctuations on the learning process. Strictly related to the batch size, we have the number of epochs: as we have only 3 records, and we update the weights only after the total error is computed, we need for sure to make more than 1 epoch of training, but we cannot just calculate too many corrections based on

the same resulting error, therefore we set the number of epochs to 5; it can for sure be increased, but we decide to set such a low number both to demonstrate as clear as possible the quality of the presented architecture (as with lower epochs the contribution to reach the highest possible accuracy is dependent more on the exchange of models than on the local training itself) and to avoid overfitting to local data, given the low number of examples we have for each node. Now, the only remaining parameters are the ones controlling the peers' selection and loading. Starting from the maximum size of the subset of known peers, we set it to 1/5 – 1 (as one position in the subset is reserved for the information for contacting the node we are considering) of the total existing peers on the network, as it seems a good starting point to better observe how the algorithm would affect the contacted, known and suggested peers. Then, we set the healing factor to half of the maximum size (considering it comprehensive of the information of the node we are considering): as specified in [33], it should be the best choice if we want to keep the freshest entries in the subset of peers and therefore maximize the probability of having a successful contact when a peer contacts another to send its model and selected peers. Simultaneously, we decide to set the swap factor to 0, implicitly deciding not to care if subsets of peers of different nodes become more similar, as our aim is to send our models to as many peers as possible as fast as possible, to quickly reach convergence in the considered network and provide as effectively as possible our prediction service. Of course, this choice weights in the capacity of discovering new nodes by the peers participating in the network, therefore, we decide to balance this problem setting as the peer to contact the oldest among the ones selected (as shown in the code before) and that should have the most different subset of peers with respect to the local one.

7.1.2.2 Test results

Now that we have defined all our parameters to be used, we can test our architecture. In particular, we create and deploy a test architecture composed by 40 nodes, configured as explained before: apart from the constant parameters, we have 7 (as the eighth element is the information of the current node) as maximum size for the subset of known peers and 4 as the healing factor. With this configuration, we manage to achieve the following results after about 100 iterations, intended as the number of times a node has been contacted by a peer:

1. Mean number of unique peers contacted by a node: 14;
2. Mean number of unique peers suggested by a node: 18;
3. Mean number of unique peers discovered by a node: 26;

4. Mean iteration index of maximum accuracy achievement by a single combined model: 15;
5. Mean maximum accuracy obtained by a single combined model: 96.18 %;
6. Mean final accuracy: 94.97 %.

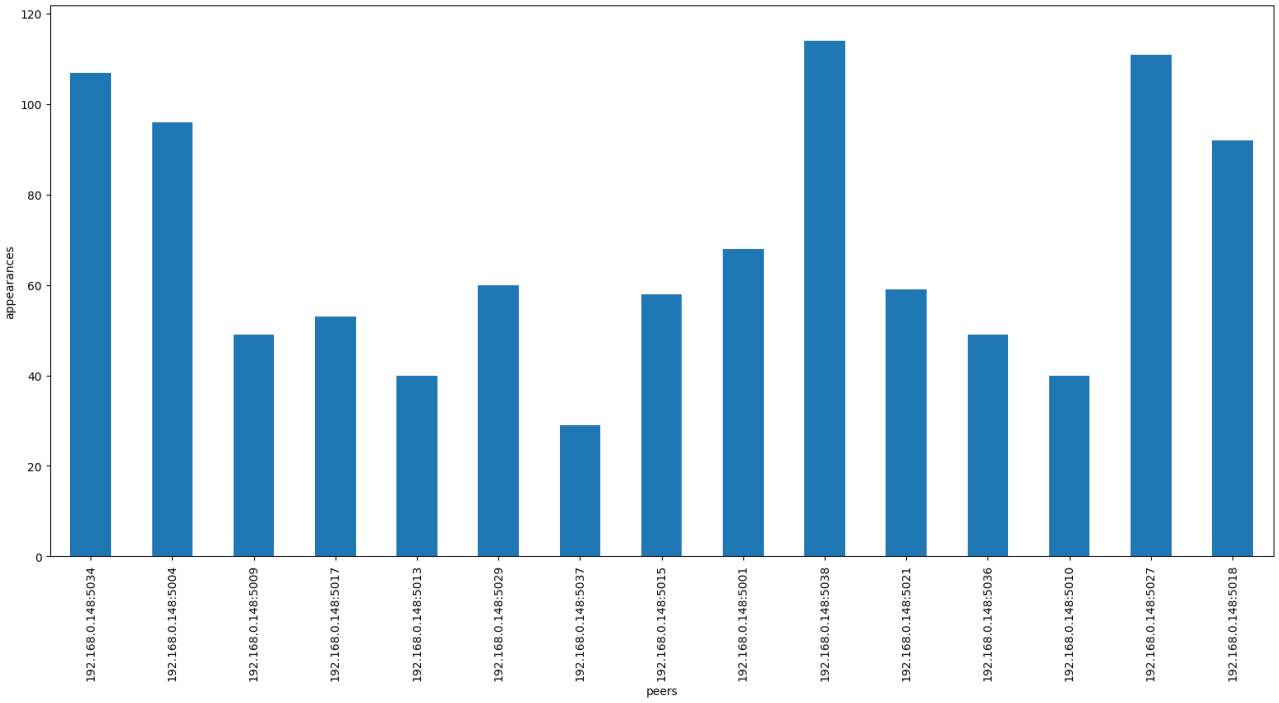

Figure 16. Distribution of peers contacted by a single node.

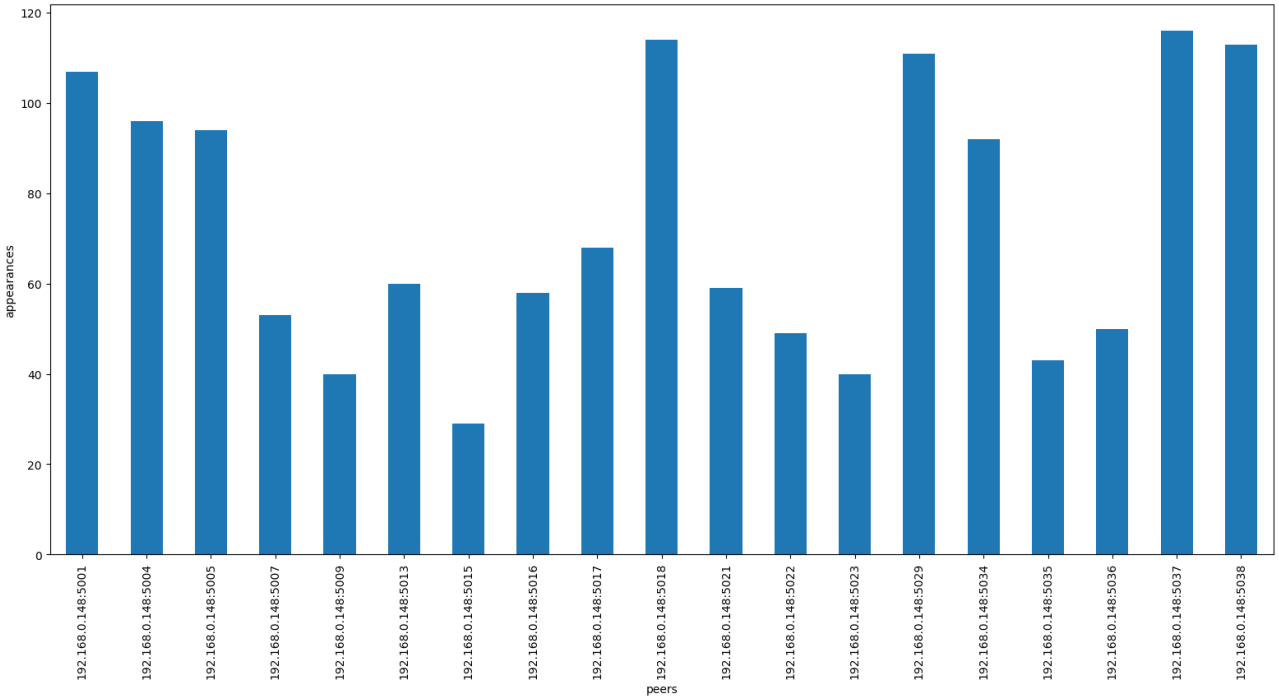

Figure 17. Distribution of peers suggested by a single node to other peers.

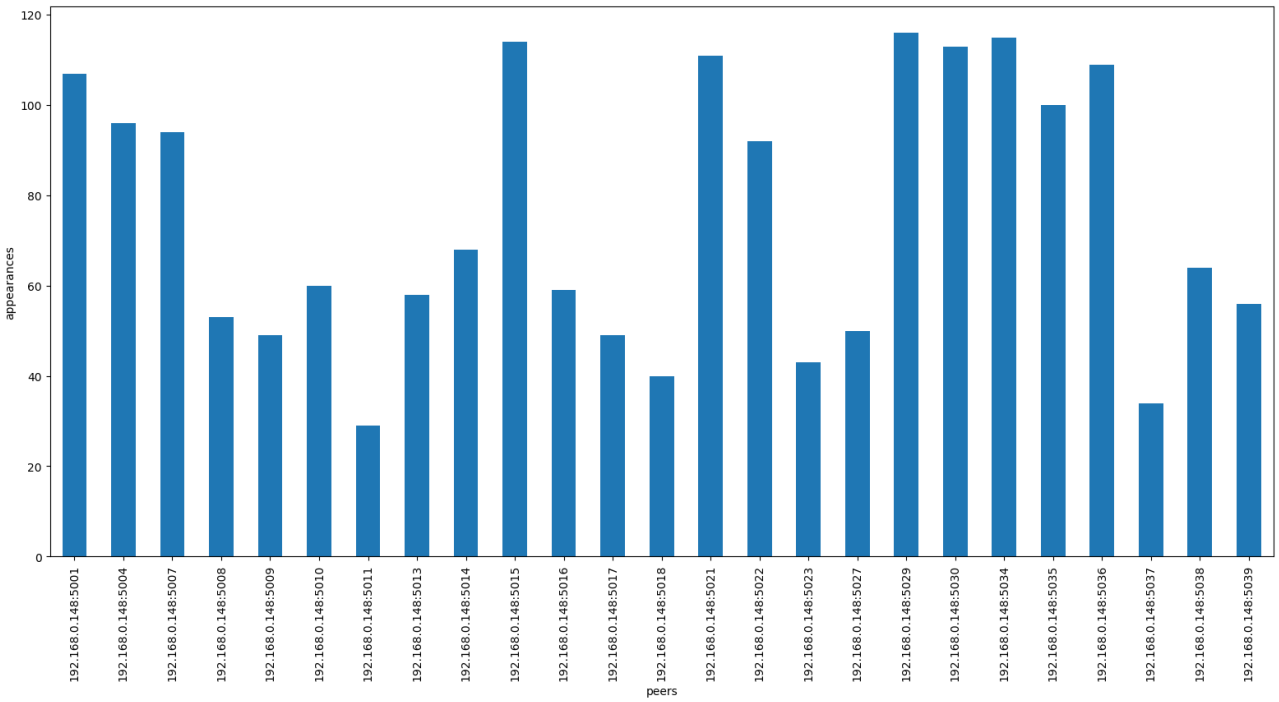

Figure 18. Distribution of peers suggested to a single node (known peers of a single node).

Starting from the peer related data, it is important to note how the number of peers contacted have doubled with respect to the initial subset of peers provided (only 7): this means that each node, on average, have had the possibility of sending its own most recent model to 14 different nodes (35% of the entire network), therefore to 14 different datasets (excluding its local one) for a total number of records reached of 45, including the local dataset (not intended for the training on the same model, of course). Also, it is important to say that the data just presented, together with the number of unique peers suggested (45% of the entire network) and the number of discovered peers (65% of the entire network) are very good results considering the strategy used, that, as we said, is more

oriented towards successful communications among peers rather than towards discovering new nodes.

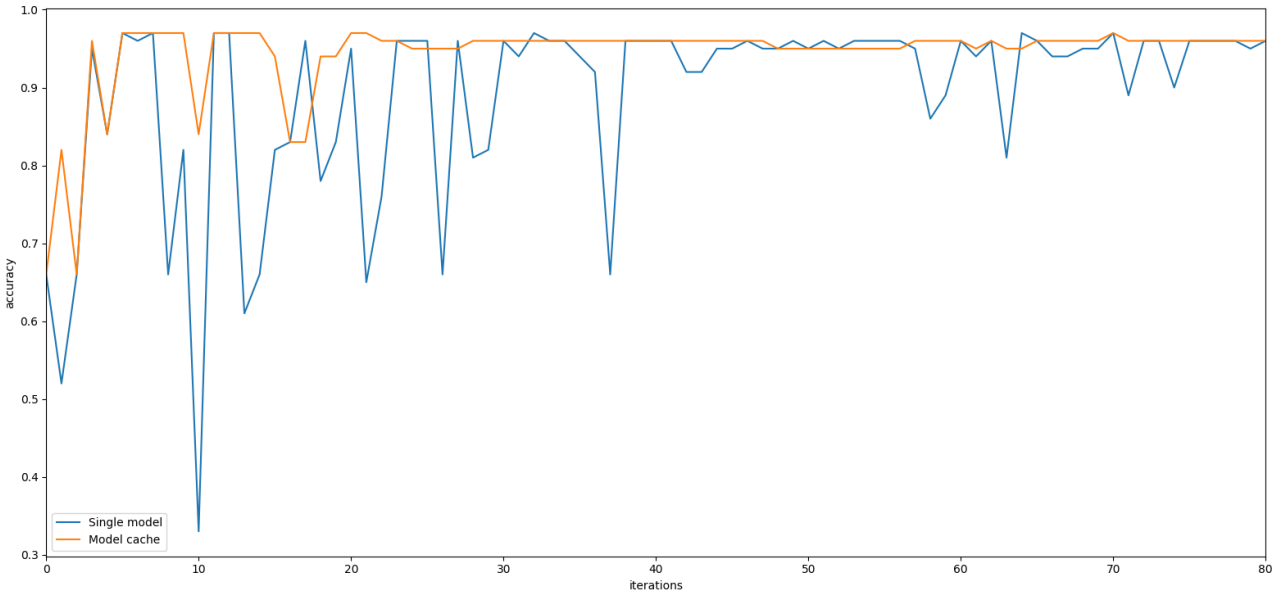

Figure 19. Comparison of the accuracy progress for a single node, considering both the last model combined (blue line) and the ensemble method classification (orange line).

Going on to analyze the accuracy related data, we find that the maximum accuracy, calculated using the last combined model through the method evaluate (presented before) on an independent (its intersection with any node's dataset is empty) test set made of 100 records, is reached in just 15 iterations and it achieves an accuracy of 96.18%, on average (with peak accuracies being equal to 97%): this is indicative both of the low complexity of the dataset at hand and of the good performances of the defined algorithm. Simultaneously, the mean maximum accuracy of our ensemble method for classification settles around 96% too, therefore we lose very little from the confrontation with the last single combined model, but we gain, on the other hand, a great stability in the quality of the prediction service provided (as it is possible to see from the figure 21), as the ensemble method succeeds in smoothing the wide fluctuations introduced when the model is obtained from a merge process where one of the two models is greatly under-performing with respect to the other. Unfortunately, we are not able to maintain this level of accuracy until the end of our test, because overfitting chimes in to lower the performances, resulting in a mean final accuracy (calculated through the ensemble method) of 94.97%, therefore losing about a point with respect to the peak performance, but this can be easily prevented by stopping the gossip learning algorithm to work after a certain number of iterations. The accuracy just reported does not take into consideration some faulty nodes (regarding the poor achieved accuracy of their ensemble methods) that it is possible to find across the network when employing an algorithm like this (in our

case, we found four nodes with an accuracy of, respectively, 66%, 87%, 81% and 80%): this can happen both with small datasets, as some of them could have as examples the anomalies rather than the common ones and therefore find themselves with poor training outcomes with respect to the other nodes, and/or with low-performing received models, that do not contribute at all to reach a satisfying accuracy. Even if this happens in real use cases, we can manage it by identifying these faulty nodes and substitute the models of those nodes with another set of models that instead is performing well: we can think, for example, of using a clustering algorithm running in parallel to classify the on-going performances of the nodes and identify the faulty ones as the farthest from the clusters' centers, and then impose, for the faulty ones, to exchange their current models with others fetched from the best node available in the network.

7.2 Federated learning architecture

7.2.1 Implementation

With respect to the gossip learning architecture, in the federated learning architecture the needed nodes are many more: the edge node, the coordinator, the aggregator and the master aggregator. Fortunately, they are not so complex, therefore they can be explained shortly and clearly in the following sections.

7.2.1.1 Edge implementation

We can now delve into the implementations of the nodes just described, starting from the edge node.

```
@app.route("/execute", methods=['POST'])
def execute():
        response = request.get_json()

        file_path = os.path.join(os.path.abspath(os.path.dirname(__file__)),
'my_model.h5')
        exported_model = request.files['file']
        exported_model.save(file_path)
        model.load(file_path)

        fl_plan = json.loads(request.form['json'])
        aggregator = fl_plan['aggregator']
        operation = fl_plan['operation']
```

```
        args = fl_plan['args']

        if operation == 'train':
                delta, examples_number = model.train(dl.get_training_data(),
args)
                delta_json = []
                for w in delta:
                        delta_json.append(w.tolist())
                result = {'delta': delta_json, 'examples_number':
examples_number}
        else:
                result = model.evaluate(dl.get_training_data(), args)

        Client.push_result(aggregator, operation, result)
```

Our edge node, as explained before, is waiting for a task to perform from the coordinator. Always using Flask [34], we expose a REST API reachable with a POST request, through which the coordinator can ask the edge node to perform a certain job. In particular, the coordinator sends the actual neural network (as it can be seen from code above, it is fetched from request.files['file']), the type of operation to be carried out on that NN (from the attribute operation of the received JSON), the aggregator to which the result must be sent to (from the attribute aggregator, always from the JSON) and the arguments through which carry out the work requested (from the attribute args, from the JSON). In the first section of code, we get the neural network from the request, then we save it to a specified path (named file_path in the section above) and finally we prepare it to be used by executing the method load of the class Model (that is a simple class maintaining in memory the last received model from the coordinator). Once the loading is complete, and also the other parameters described before are read (as in the second section of the block above), we can then pass to execute the operation requested; in our use case, we modeled two kind of possible jobs: "train" and "evaluate". The "train" job simply performs a training of the received model with the arguments specified in the request by using the method fit (presented in 5.2.3), while the "evaluate" job carries out a test of the received neural network (always with the arguments specified in the request) by using the method evaluate (also presented in 5.2.3). Coming back to where we left our analysis of the block of code, we can see how, checking the current operation requested, we (of course) performs different instructions. In case the operation is "train", we invoke the method train of the class Model (that, as we were saying, corresponds to the execution of the method fit) passing the dataset of the current node and the parameters through which configure the training; when it completes the work, this method returns the corrections on the weights (delta) and the number of

examples used for training (examples_number), that are, together, finally stored in a dictionary, ready to be transformed in JSON format. If the operation is “evaluate”, the method evaluate of the class Model is called, attaching the training data and the parameters (as before) as arguments, and then obtaining the calculated accuracy on the local dataset. Finally, we send the computed result to the aggregator, specifying also the operation requested, using the method push_result that just sends a POST request to the aggregator specified.

Apart from the REST API just presented, the server also offers an endpoint to provide the prediction service: as for the gossip learning architecture, also here we implement it in the same manner as described for the cloud-centric solution, therefore we are not going to add further explanations.

7.2.1.2 Aggregator implementation

Now that the data are in the aggregator, we can analyze how they are treated there.

```
@app.route("/aggregate", methods=['POST'])
def aggregate():
        global received_weights, received_examples_number, received_evals
        response = request.get_json()
        if response['operation'] == 'train':
                weights = response['delta']
                examples_number = response['examples_number']
                received_weights.append(weights)
                received_examples_number.append(examples_number)
                if len(received_weights) == number_of_edge_nodes:
                        result_weights = 0
                        result_examples_number = 0
                        for w in received_weights:
                                if result_weights == 0:
                                        result_weights = w
                                else:
                                        for i in range(0, len(result_weights)):
                                                result_weights[i] =
np.add(np.asarray(result_weights[i]), np.asarray(w[i])).tolist()
                        for n in received_examples_number:
                                result_examples_number = result_examples_number +
n
                        received_weights.clear()
                        received_examples_number.clear()
                        Client.push_aggregated_result(master_aggregator_address,
'train', result_weights, result_examples_number)
```

```
        else:
                evaluation = response['evaluation']
                received_evals.append(evaluation)
                if len(received_evals) == number_of_edge_nodes:
                        result_evals = 0
                        for e in received_evals:
                                result_evals = result_evals + e
                        received_evals.clear()
                        Client.push_aggregated_result(master_aggregator_address,
'evaluate', result_evals, number_of_edge_nodes)
```

Also here, we distinguish the instructions between the two operations, but the core logic is the same. In both cases, we are going to save results received from edge nodes into lists, until we store all the ones we were expecting. Then, we sum results together and finally, we push the aggregates to the master aggregator, always together with the requested operation (only after having cleared the partial results from memory, to be ready to receive new ones from future tasks). The only difference is that in case of "train" operation, we send the sum of the number of examples used for training and the total weights' corrections, while in case of "evaluate" operation, we send the summation of the accuracies and the number of edge nodes that have contacted this aggregator for this operation.

7.2.1.3 Master aggregator implementation

The master aggregator implementation is really similar to the aggregator one. It simply defines another layer of aggregation of the partial results coming from the aggregators. The only differences (literally) regard the node to contact, that for the master aggregator is the coordinator (of course), and the final aggregation instruction, that in the master aggregator divides the total weights' corrections by the summation of the number of examples (in case of "train" operation) or the sum of the accuracies divided by the total number of edge nodes that participated in carrying out the task. As it can be easily understood, while for "evaluate" we carry out a simple mean of the accuracies, with the "train" operation we opted for a weighted average of the weights' corrections, giving (of course) much importance to weights' corrections coming from nodes with larger datasets: in our case, the operations are equivalent as the datasets have the same size, but we decided, for the "train" operation, to make this extension because it was originally presented this way in the federated averaging algorithm provided in [35].

7.2.1.4 Coordinator implementation

We finally arrived to analyze the coordinator, that is the center of the management of the architecture. It offers two REST APIs: one to trigger the starting of the operation (that can be called by the end user), and the other to receive the final aggregated result computed by the master aggregator. Here is the former:

```
@app.route("/start", methods=['POST'])
def start():
        global current_operation

        plan = request.get_json()
        current_operation = plan['operation']
        aggregator_addresses = plan['aggregators']
        edge_per_aggregator = int(len(edge_addresses)/len(aggregator_addresses))
        exported_model_file = model.export()
        exported_model = exported_model_file.read()
        exported_model_file.close()

        for i in range(len(edge_addresses)):
                x = threading.Thread(target=Client.push_task,
args=(edge_addresses[i], aggregator_addresses[i // edge_per_aggregator],
exported_model, current_operation, plan['args']))
                x.start()
```

As it can be easily understood, the POST request must include all the details about the operation to be executed, that are the operation type and the available aggregators; in particular, the operation type is saved to a global variable in order to recognize it later (when the task is completed and therefore the other REST API is called by the master aggregator) and execute the right instructions. Then, the coordinator calculates the number of edge nodes that each aggregator must manage, and finally, it exports the local model to be sent to the edge nodes; it is important to note that also in this solution, we did not change the original neural network defined, that remains valid also here. Once we have all these elements, we can ask the edge nodes to perform the requested task through the method push_task, that sends to them the address of the reference aggregator, the exported model, the operation requested and the settings with which the above operation must be carried out.

When the task is completed, that means that the master aggregator has the final results, the REST API below is called:

```
@app.route("/finish", methods=['POST'])
```

```
def finish():
        global current_operation
        request_json = request.get_json()
        if current_operation == 'train':
                model.update(request_json['result'])
        else:
                print('Accuracy: ' + str(request_json['result']))
```

With this REST API, we simply use the obtained results to update the weights of the model through the method update of the class Model (that simply adds the weights' corrections to the current weights of the stored model), if the current operation is "train", otherwise it simply prints out the mean accuracy calculated.

7.2.2 Validation

We finished describing the implementation: now our architecture is ready to be created. Prior to use it in order to provide the prediction service needed, we need to validate it by creating a test architecture (resembling the one made for gossip learning) and then checking its maximum reachable accuracy.

7.2.2.1 Test settings

First, we lay out how our test environment is configured. For what regards the topology, we use 40 edge nodes (to resemble the environment used in the gossip learning test), 5 aggregators (therefore receiving 8 results each), 1 master aggregator (managing 5 aggregators, of course) and 1 coordinator, for a total of 47 deployed nodes. For each edge node, we prepare a dataset with 3 records in the same way as we did for the edge nodes of the gossip learning architecture: therefore, also here we have that the datasets are independent of each other (the intersection of each one of them with any other dataset results in an empty set) and that each existing class is represented by exactly one record. Also, recalling the considerations made when considering the dataset size in the gossip learning architecture section, we define 3 as batch size and 5 as number of epochs as parameters for the training rounds. Finally, to better evaluate the performances in the testing process, we provide each edge node with the same test set, composed by 100 records, independent from the training sets used by the same nodes (each record present in the test set cannot be found in any training dataset); to not be confused with the actual "evaluate" operation of this architecture, that calculates the mean accuracy mixing together results among edge nodes, each one of them computing the performance on their training dataset (in our testing process we are going to use the

“evaluate” operation, but not to get the result of the operation per se, but just to send the actual model to the edge nodes).

7.2.2.2 Test results

Given the settings above as our test environment, we proceed to execute a test plan consisting in alternating, throughout the iterations carried out, test sessions with training sessions: for even iterations, we are going to have a test session (using the operation “evaluate”) to send the actual coordinator’s model to edge nodes and compute its accuracy on the test set, while for odd iterations, we are going to have a training session (using the operation “train”) to send the actual coordinator’s model to edge nodes for a training round with the edge node’s training dataset and then, as in the other operation, compute its accuracy on the test set. We manage to keep this test going for about 30 iterations, but as we will see in a moment, it was enough to see the complete behavior of how the accuracy progresses as iterations pass by. Looking at figure 22, we can see how the node converges to the final accuracy in 23 iterations, assessing it at 95% (as in the gossip learning architecture, but much more rapidly). Moreover, we can see how, already at iteration 4, we reach the maximum accuracy of 97%; we can remember how, in the gossip learning solution, the mean index of reaching the maximum accuracy (of the same value of 97%) was 15. Moreover, we are not having wide fluctuations at all, with respect to the ones encountered in the gossip learning architecture. Overall, we can say that the federated learning approach is faster and more robust with respect to the gossip learning solution, but this was to be expected: each time we issue a training session in our network using federated learning, the neural network is “trained” on a total of 120 examples, while in gossip learning, each created model makes random walks across the network, and this makes covering all the datasets for training slower and more difficult. On the other hand, we cannot forget that to make the federated learning architecture running we need many more nodes with respect to the gossip learning solution. Overall, we can conclude by saying that, if we need fast convergence and we have additional nodes ready to host the needed entities, we can use the federated learning approach, but if we do not mind to wait more to reach convergence and/or we need to pay more than what we are willing to accept to allocate more nodes to deploy the federated learning solution, we can prefer the gossip learning architecture.

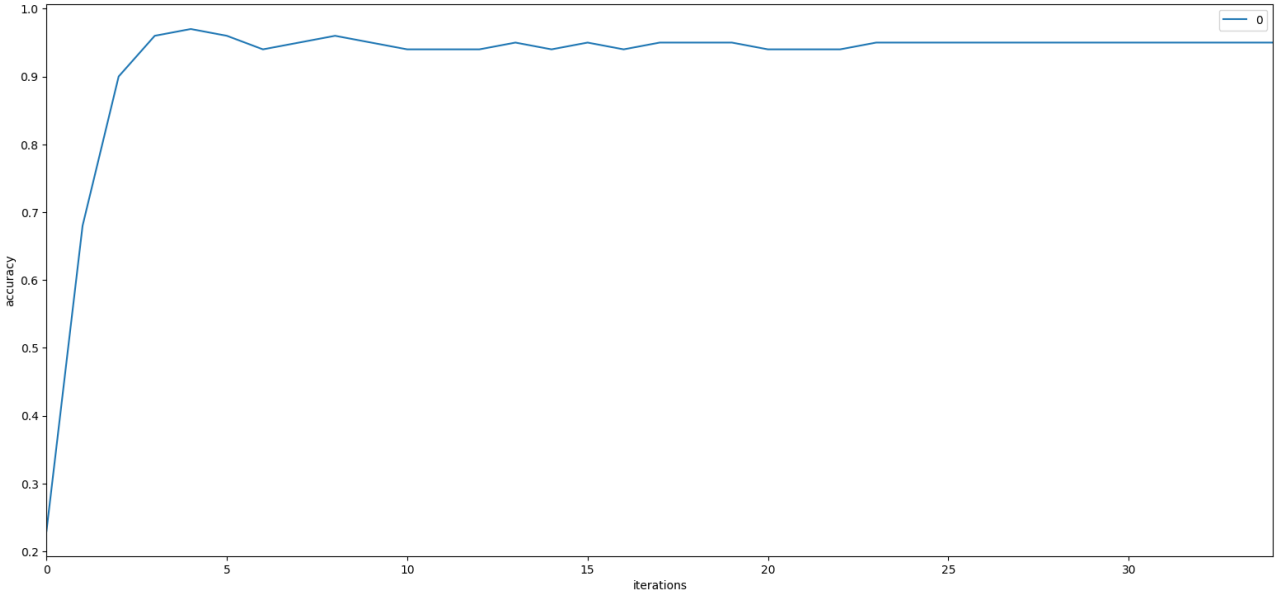

Figure 20. Progress of the accuracy of a single node.

8. Deployment and maintenance

Now that the architectures are ready to be used, we can employ them to solve our problem at hand. But in order to do that, we need to deploy them on the available nodes, but a manual deployment approach could be too complex and prone to errors to be carried out successfully. Moreover, for what concerns their maintenance, if we imagine a situation in which the entire architecture finds itself in a faulty state (while it was serving the prediction service) and must be rebooted or replaced by another solution as fast as possible, we would fall short again trying to meet such requirements, if we do not change the underlying method to manage them. Fortunately for us, we can use fogØ5 to solve the above concerns, as it seems to perfectly meet our necessities for this use case. But, in order to use fogØ5, we first need to map each node of the two architectures to an entity manageable by fogØ5, and then effectively use fogØ5 to deploy and manage them: as for the first necessity, fogØ5 already provides to us all the instruments we need, but for the second objective, we have to extend the management paradigm as it is offered today by the platform, and build a sort of orchestrator that is going to cover our own necessities and also the ones related to problems similar to ours. As our orchestrator could influence the way we map our architectures, we are going to start from defining it (phase five) and then transform the architectures the way it expects them to be (phase six), and finally deploy them through a script that exploits the created orchestrator (phase seven).

8.1 Orchestrator

The main objective for our orchestrator is, for sure, to effectively and efficiently deploy the architectures we created by employing fog05 APIs. Simultaneously, our secondary objective should be to define this program as dynamic and flexible as possible, to permits its usage with minimum changes also with other architectures similar to the ones we created in this thesis.

In order to achieve these purposes, our proposed orchestrator is composed by four main classes: the ArchitectureRepository, the ArchitectureDataRepository, the StrategyRepository and the ActiveStrategyManager.

8.1.1 ArchitectureRepository

This entity simply stores the architectures that the user creates. It uses a dictionary where the keys are the names of the classes implementing the architectures and the values are the instances of those classes. If a user has an architecture that he wants to use, he can call the method add_architecture of the class ArchitectureRepository, passing as argument an instance of an architecture that is an instance of a class extending the Architecture interface (that we will explore in a moment).

For architecture, we mean a complex composition of entities or atomic entities (we are referring to fogØ5 jargon, now) in terms of dependencies between them that need to be solved prior to deploy them: these dependencies can be of many types, but, for example, in our case, it referred principally to nodes needing IP address of certain other entities or to the specification of parameters to adjust the entity's behavior as requested from the user.

Now, delving into our concept of architecture, we can analyze our Architecture interface. It defines a lifecycle for any Architecture-extending class, exposing 6 methods (plus other 3 utilities methods) through which the entities composing the architecture are instantiated and terminated on the basis of the current state. These 6 methods are (presented in the same order as the lifecycle that the architecture must follow):

1. Configure: each architecture has unique configuration parameters it offers to control its deployment. Through this method we can set them, by passing as argument a dictionary containing as keys, the properties we want to set, and, as values, the desired behavior we expect from that property. At the end of the successful execution of this method, the architecture reaches the configured state;
2. Associate: each architecture can be used in different environments (that, in fogØ5, is equal to say that can be used with different YAKS instances). As such, we need an operation that ties the instance of a needed architecture to a specific environment, in order to be able to use there the considered architecture. This can be done by calling this method and passing as arguments an instance of the FIMAPI class from fogØ5 module (a class that provides APIs to enable interactions with a certain YAKS instance), a dictionary containing other useful information about the environment (that can help mapping the entities to the most appropriate nodes) and another dictionary storing more specific directives about the deployment of the architecture (it differs from the one at the previous point as this

dictionary should contain indications imposed by the environment we are going to associate to). If this method executes with success, the architecture arrives to the associated state;

3. Deploy: it is for sure the main method of the lifecycle of an architecture. It contains the deployment logic, instantiating all the associated entities and providing them with all the needed information and additional data they need to work properly. It only takes one argument, being the directory path that contains the needed files to be loaded on one or more nodes of the architecture. This operation uses the fundamental utility method get_mapping, that exploiting the instance of the class FIMAPI, the environment dictionary, the current state of the architecture and the requirements needed by the current architecture (defined in the "private" utility method __get_requirements), returns for each entity the assigned node; the current state of the architecture is needed for the method __get_requirements, in order to let it known for which entity we need to find a mapping (and therefore a corresponding node). If this method executes with success, the architecture reaches the deployed state. At this point, we can check if the architecture is effectively working or not by calling the utility method check_status, that contains the control logic for the current architecture and returns a boolean indicating if it is online or offline;
4. Serve: strictly related to the previous point, there is the serve method. This method deploys the entities that are going to exploit the service offered by the architecture (activated through the previous method) always using the method get_mapping as explained in the previous point and providing them with all the settings and additional data needed. This instantiation is separated from the previous one, because the architecture could need some time to prepare itself for providing the service, therefore with this separation we are free to supply the service when we think we are ready. Finally, if this method executes with success, the architecture arrives to the served state;
5. Undeploy: when we need to stop serving our service or when we have to replace the current architecture with another, we call this method. It terminates all the running entities (both the ones associated with deploy and with serve) and offload all the descriptors loaded by this architecture into the environment. At the end of the successful execution of this method, the architecture reaches the undeployed state; at this point, it can only come back to the deployed state by executing the deploy method.

When we are managing an architecture, we have to execute the above methods one after the other. As stated before, only from the undeployed state we can come back to the deployed state, while

from any other state we can only proceed forward; if we need a similar architecture but with different configurations and parameters we have to restart from the configure method using a new instance of the same architecture.

8.1.2 ArchitectureDataRepository

This repository keeps in memory associations between an architecture and its needed data for deploy or serve operations. In particular, it offers the add_architecture_data method to which we feed the name of the architecture, the identifier of the data we are loading and the path to files needed at deployment time and at serve time: if we successfully call it, we can memorize data to fetch them later on and from all the different environments, and use them at deploy or serve time of our architectures.

8.1.3 StrategyRepository

This repository keeps in store strategies, that are instances of the class Strategy, that is an abstraction created to memorize a combination of architectures that provides as efficiently and resiliently as possible the same service. In particular, this combination is defined in terms of two elements: one main architecture and zero or more replacement architectures; each architecture indicated provides the same service, but with different performances, as the main architecture is the preferred choice and therefore should offer the best performance, while the replacement architectures are the ones that should be activated when the main one does not work, and therefore should be less expensive (in terms of nodes needed to deploy it) but also provide less quality with respect to the main one. Ultimately, we can say that the purpose of this abstraction is to ensure continuity and efficiency in the provisioning of a business service defining a strategy (from here it comes the name of the class) composed by different architectures ready to be activated when needed.

Coming back to the repository, it is important to note that we can store strategies by calling the method define_strategy and passing as argument a dictionary, containing the name of the main architecture and its configuration, and a list of dictionaries, containing for each entry the name of a replacement architecture and its configuration: in fact, it is in the StrategyRepository that the configuration of the architectures takes place.

8.1.4 ActiveStrategyManager

This class manages active strategies, that are strategies whose main architecture (or one of its replacement architectures) has been deployed. In particular, the class ActiveStrategy permits to store the main architecture, the replacement architectures, the arguments needed to execute association operations, the identifier to access additional data for deployment or serve operations, the strategy to which this active strategy refers to and a method (more on this later).

The offered operations are four, also defining a lifecycle for an ActiveStrategy:

1. Start: it permits to activate a strategy, by associating and then deploying that strategy's main architecture. Thus, we first carry out the association operation as described before, by passing as arguments the instance of the class FIMAPI (created by this class through the address of the YAKS instance provided by the user), the environment dictionary and the additional association arguments. Then, we deploy the architecture by using the method deploy as described above, thus passing the path from which the architecture has to fetch the additional files for its deployment. Once the main architecture is deployed, this method stores the information about this activated strategy in a list (memorizing all the parameters described before) and, contextually, through the class Watcher, activates a thread that, on a certain frequency (as defined by the user), checks the status of the deployed architecture (that can be either the main architecture or one of the replacement architectures) to see if it is still working or if it needs to be replaced (if one of the replacement architectures is active, it is also checked if the main architecture can substitute the replacement one, in order to come back to provide the best performance possible) and triggers the execution of the manage method (described below) when a change in the status is detected;
2. Serve: if a strategy is activated, we can make it provide its service by executing this operation that is going to call the serve method of the deployed architecture by passing it the path containing the needed data;
3. Stop: it undeploys the current active architecture of the specified strategy (being it either the main architecture or one of the replacement architectures). As for the current implementation, a strategy cannot be reactivated after the stop, as it is also deleted from the local list of active ones: if we want to reactivate it, we need to call the method start and therefore allocate a new ActiveStrategy;
4. Manage: as anticipated before, this is the reference method for the Watcher, to be called when a status change in one of the architectures is identified. In particular, there are two

kind of changes that can be detected, main-ready and needs-replacement: the former means that a replacement architecture is providing the service but the main one is ready to come back online, while the latter means that the main architecture is having a problem and needs to be substituted. In case the first occurs, the active replacement architecture is undeployed and the main architecture is deployed again and served (there is no chance of delaying the serving phase, but it should be a priority to continue providing the service needed, as a replacement is something that usually happens when the architecture is already fulfilling the service that it needs to stay active). Otherwise, if the second occurs, the main architecture is undeployed and then the algorithm tries to find if one of replacement architectures can be deployed and, if possible, effectively deploy that architecture.

8.2 Mapping the architectures

Now, the orchestrator is ready and the architectures are implemented: the remaining phases correspond to map the architectures' nodes to fogØ5 atomic entities and deploy them through the orchestrator just defined.

In order to map the architectures, we first need to identify the required entities and then analyze how they interact between them, and, finally, create a class (extending Architecture) containing all the deployment, serve and undeployment logics.

8.2.1 Gossip learning architecture mapping

We started the mapping process from the easiest architecture, the gossip learning one, to better explain both the schemas and the underlying functionalities exposed by fogØ5.

8.2.1.1 Descriptors

In that architecture, we have only one type of node, the edge node (as we called it): this type of node needs to expose a server and to communicate with other edge nodes and to the coordinator (to receive additional data) using the TCP/IP stack, all while having access to libraries as NumPy, Pandas, Keras and Tensorflow (whose related operations and footprints on memory are everything but negligible). Therefore, we are going to need a VM instance or a container: given the capabilities of the test environment and the actual features offered by fogØ5, we decided to map this entity to a container, in particular to a Linux container (managed by LXC that, as already highlighted in the

chapter about fogØ5, is well supported at the moment). Here is an excerpt from the entity descriptor:


```
{
    "uuid": "c4f7ae9f-4c9f-4f6a-b179-ba10c03141d4",
    "name": "gossip-edge",
    "configuration": {
        "conf_type": "CLOUD_INIT",
        "script": ""
    },
    "image": {
        "uri":
"file:///home/osboxes/Scrivania/one_node_deployment/architecture_repository/architectures/gossip_learning/descriptors/gossip_edge.tar.gz",
        "checksum": "",
        "format": "tar.gz"
    },
    "hypervisor": "LXD",
    "migration_kind": "LIVE",
    "interfaces": [
        {
            "name": "eth0",
            "if_type": "EXTERNAL",
            "virtual_interface": {
                "intf_type": "BRIDGED",
            }
        },
        {
            "name": "eth1",
            "if_type": "INTERNAL",
            "virtual_interface": {
                "intf_type": "VIRTIO",
            }
        }
    ]
}
```


The first parameters we see are, of course, “uuid” and “name”: the former is the identifier used to reference the descriptor, while the latter is a simple variable offered to provide human-readable information about the entity. Then, as we can see from the “image”, “configuration” and “hypervisor “properties, we are effectively configuring a LXC container: in the “image” property, we define the image to be loaded through the “uri” parameter (in our case, an image containing all the

necessary code to make the node properly work, parametrizable by indicating the maximum number of peers that each node can contain), in the “configuration” setting, we states in “conf_type” that we are going to initialize the container through the cloud_init method (the “script” parameter is empty as we are going to define it dynamically, we will see in a moment how), and, as it can be seen from the “hypervisor” property, we have LXD as the supervisor (a common choice when it comes to manage LXC). Then, through the migration_kind parameter, we impose that, in case of migration, the state of this container should be preserved (in order to not lose the progress made by the training process). Finally, we come to the network interfaces. As we said before, this type of node needs to talk to other edge nodes, but also to the orchestrator (we will see in a moment why), therefore, in order to separate the two traffic flows and ensure a greater security, we create two different network interfaces: “eth0” for the coordinator, therefore reachable from the external world (value “EXTERNAL” of the parameter “if_type”) and made available by bridging the virtual interfaces of the container with the one hosting the container (value “BRIDGED” of the parameter “intf_type”), and “eth1” for the other edge nodes, not reachable from the outside (value “INTERNAL” of the parameter “if_type”) and made available through a particular software interface (value “VIRTIO” of the parameter “intf_type”). But while the external interface is already working out-of-the-box (being bridged), when we create an internal interface we need to deploy a network entity to provide IP addresses and make entities communicate using those interfaces; it is defined as follows (the same schema will be used in other occasions):

```
{
    "uuid": "923b951d-7de6-4317-bdb1-e325ca10d026",
    "name": "edge-edge-network",
    "net_type": "ELAN",
    "ip_configuration": {
        "ip_version": "IPV4",
        "subnet": "192.168.234.0/24",
        "gateway": "192.168.234.1",
        "dhcp_enable": true,
        "dhcp_range": "192.168.234.2,192.168.234.100",
        "dns": "8.8.8.8,8.8.4.4"
    }
}
```

Apart from the “uuid” and “name” parameters, the first thing we do is set the “net_type” to be an ELAN (that following the ETSI MANO specification, is a multipoint service interconnecting a set of VNF; in our case, a set of FDU instances). Then, we have “ip_configuration”, describing all the

needed properties to properly set up communications between these interfaces, like “ip_version”, “subnet”, “gateway”, “dhcp_enable”, “dhcp_range” and “dns”, all very self-explanatory: through this configuration, fogØ5 sets up a DHCP server and indicates DNS addresses, then it provides, when requested, IP addresses to the interfaces connected to this network and route traffic as needed to make entities communicate. To make interfaces use networks like the one above, we need to connect them using connection points (not reported in the descriptor for brevity): in the descriptor of the edge node presented before, we need to set a property “connection_points” where we have to set a list containing an object for each network we want to connect to; these objects are defined by specifying the UUID of the connection point offered by the interface (to be specified in the interface object through the parameter “cp_id”) and the UUID of the target network.

We have defined the mapping for the edge node and how it can exchange information with its peers: the only step remaining is to define the serve phase, that means to decide which are the entities to be created that are going to take profit from the service provided. In our use case, we have a prediction service, therefore we need entities that could send to our making-predictions edge nodes the data collected on the field; since we are analyzing a hydraulic system, we have sensors distributed all over that machine, therefore we can think of deploying entities simulating the behavior of sensors, sending read data to edge nodes that compute predictions. In our case, we do not have real sensors, but we need to read data from files and invoke the REST API provided by edge nodes for prediction, therefore we also simulate them by using containers like with the edge nodes. Concluding, we defined a descriptor like the one presented above but with a different image (containing a Python script that, for each line read from files, send a prediction request to an edge node) and with just one internal interface (as they only need to communicate with edge nodes, they do not need additional data as they are already provided with the image); of course, we also defined another network like the one described above, to manage edge and sensor communications, and added an internal interface to the edge descriptor to connect to this newly-created network.

8.2.1.2 Architecture-extending class

Now that we have the images and the descriptors ready, we can code the architecture logic inside a class extending the Architecture interface. Leaving aside configure and associate operations (that we are going to see briefly), we are going to present all the other methods to show how we decided to manage this architecture. The configure operation permits to define the number of edge nodes that we want to deploy and the maximum number of peers’ addresses that each edge node can

store, while the associate operation does not contain any relevant difference with respect to the normal associate operation (setting the FIMAPI instance and the environment args), except for the setting of the process_port parameter, that defines the port on which the server (created by the edge node) have to listen. Now, we can analyze the main operation, deploy, examining the following code:

```
nodes = self.get_mapping('deploy')
if not nodes:
    raise('Mapping not found!')

for n in self.networks_d.values():
    self.fog05_api.network.add_network(n)

edge_nodes = []
self.entities_i['edge_nodes'] = edge_nodes
edge_uuid = self.entities_d['edge']['uuid']
for i in range(0, self.edge_nodes_number):
    command = "#cloud-config\nruncmd:\n  - [ sh, -xc, 'python3 -u
/home/ubuntu/code/server.py " + str(self.peers_per_node) + " "
    if i < self.peers_per_node:
        for j in range(0, i):
            index = random.randrange(0, i)
            address = get_address(self.fog05_api,
self.entities_i['edge_nodes'][index], 'eth1')
            command += address + ":" +  self.process_port + " "
    else:
        for j in range(0, self.peers_per_node):
            index = random.randrange(0, i)
            address = get_address(self.fog05_api,
self.entities_i['edge_nodes'][index], 'eth1')
            command += address + ":" +  self.process_port + " "
    command += "> /home/ubuntu/log.txt' ]"
    self.entities_d['edge']['configuration']['script'] = command
    self.fog05_api.fdu.onboard(self.entities_d['edge'])
    edge_nodes.append(self.fog05_api.fdu.instantiate(edge_uuid,
nodes['edge'][i]))

edge_files = get_data_files(data)
for i in range(0, len(self.entities_i['edge_nodes'])):
    address = get_address(self.fog05_api, self.entities_i['edge_nodes'][i],
'eth0') + ':' + self.process_port
```

```
    files = {'profile.txt': open(edge_files['edge'][i]['profile.txt'], 'r'),
'TS1.txt': open(edge_files['edge'][i]['TS1.txt'], 'r'),
    'TS2.txt': open(edge_files['edge'][i]['TS2.txt'], 'r'), 'TS3.txt':
open(edge_files['edge'][i]['TS3.txt'], 'r'),
    'TS4.txt': open(edge_files['edge'][i]['TS4.txt'], 'r')}
    load_data(address,files)
```

In the first section, as anticipated while explaining the deploy operation, we try to find which are the best nodes to deploy our entities to, based on the current situation of the associated environment (that corresponds to a YAKS instance). If we find this mapping (meaning that there are enough nodes in the network to host all the entities of the current architecture), we first deploy the networks whose descriptors are stored in “networks_d” through the method add_network (as indicated in the second section of the block above), and then, we start deploying also our edge nodes too (in the third section of the block). For this second purpose, we first put in the dictionary “entities_i” (a dictionary storing the identifiers of all the running entities, divided by type) a list containing the identifiers of the running edge nodes, then we save the UUID identifier from the edge node’s descriptor accessing the dictionary “entities_d” (another dictionary storing all the descriptors needed for this architecture); we are going to use both of them later. Then, we create as many edge nodes (as specified in the configure operation) through the for loop, using first the onboard operation and then the instantiate operation (that is equivalent to call, in order, define, configure and run methods explained in the chapter 3) obtaining, at the end, the identifier of the running instance to be stored inside “entities_i”. But prior to make these operations, we define the command that each edge node must execute at its boot (modifying the “script” property) imposing them to run the server passing by arguments the maximum number of peers (always specified in the configure operation) and a list of random peers’ addresses (of course, with equal or less size with respect to the maximum one specified). Once instantiated all the entities, we load the data to the edge nodes (in the last section of the code) by first fetching them from the architecture data repository, then by sending them through a POST request to the address of the edge node considered (recovered by using the function get_address that exploits fogØ5 APIs, as when instantiating edge nodes, where we needed to send peers’ addresses to the edge node to be instantiated); in our case, this is necessary in order to efficiently load a different training dataset for each available edge node.

Now, we can describe the serve operation. There is nothing specific here to be analyzed, as we simply try to find a mapping to instantiate a single sensor and, if we find it, we onboard the descriptor associated and instantiate the container (always using “entities_d” for the first operation

and memorizing the result of the second operation in “entities_i”). Of course, also here we modified the “script” parameter to define a command to start, at boot, the script that calls the prediction service as explained before (therefore, we called get_address too, to fetch the address of the interface that the edge node exposes to communicate with the sensors).
Finally, we can describe the undeploy method, where we just terminate (it is a method offered by FIMAPI that is equivalent to call, sequentially, stop, clean and undefine methods explained in the chapter 3), using the information contained in “entities_i”, and offload, using the data stored in “entities_d”, all the edge nodes and the sensor; at the end, we delete the information just used from “entities_i” and “entities_d”.
One last note prior to wrap up this section regards the method check_status, that for this architecture has been defined to return False (and therefore signal that the architecture is offline) if half of the deployed edge nodes plus one are down: to check this condition, we try to use the method get_address (described before) to obtain the address of the interface that the edge node exposes to the orchestrator; if we get a valid address, the node is up and running, otherwise it is labeled as faulty.

8.2.2 Federated learning architecture mapping

The last step, prior to deploy the solution through the orchestrator, remains the mapping of the federated learning architecture’s nodes to fogØ5 entities and the definition of the related class.

8.2.2.1 Descriptors

As we described before, we have four types of node: the coordinator, the aggregator, the master aggregator and the edge node; as stated for the previous architecture, also here we need to have a sensor, that uses the prediction service in the same identical manner as described before (therefore we are not going to further analyze it). For what concerns the other nodes, we have no doubts: each one of them is a server that also does push operations, and in particular, the coordinator and the edge node also need to manage a neural network; overall, there is no node as big as the edge node of the gossip learning architecture, but they still have a level of complexity such that a container is employable to map these nodes. The descriptors for these nodes use the same schema presented for the edge node of the gossip learning architecture: the only differences regard the interfaces used by each one of them; the coordinator uses three interfaces, as it needs to communicate with the master aggregator, the edge nodes and the orchestrator (the only external interface), the edge

nodes use four interfaces, as it exchange information with the coordinator, the aggregator, the sensor and the orchestrator (the only external interface), and the master aggregator and the aggregator need two interfaces each, one to communicate between them, another for communications between aggregators and edge nodes and the last for communications between master aggregator and coordinator. Another minor difference regards the migration type for the entities: while the aggregator and the master aggregator have “COLD” set as “migration_kind”, meaning that there is no need to save their state before migration (as their function runs out when the task is carried out), the coordinator and the edge node have “LIVE” (like the edge node of the gossip learning architecture), because we are of course interested in preserving the progress of the training. For the sake of completeness, it is to be noted that, for each type of node, we prepared (as before) a container with the code of the considered node, ready to be launched in the deployment and serve phases (as we are going to see in a moment). Obviously, we also need to create networks to make the entities communicate: one for sensor and edge nodes communications, another for edge nodes and aggregators, another for aggregators and master aggregator, another for master aggregator and coordinator and finally one for edge nodes and coordinator; also in this case, we used the same schema as the network definition explained before, obviously changing for each network the subnet and the DHCP configurations to assign unique IP addresses.

8.2.2.2 Architecture-extending class

Now, we are ready to define the class in which describe the deployment, undeployment and serve logic. As before, the configuration and association operations are rather simple: in the former, we define the number of edge nodes we want to instantiate and the number of edge nodes that each aggregator must manage, in the latter, we just associate to the architecture the instance of the FIMAPI class and the environment dictionary and specify the process port (as before). For what concerns the deploy operation, we can analyze it in depth:

```
nodes = self.get_mapping('deploy')
if not nodes:
    raise('Mapping not found!')

for n in self.networks_d.values():
    self.fog05_api.network.add_network(n)

command = "#cloud-config\nruncmd:\n  - [ sh, -xc, 'python3 -u
/home/ubuntu/code/server.py > /home/ubuntu/log.txt' ]"
```

```
self.entities_d['edge']['configuration']['script'] = command
self.fog05_api.fdu.onboard(self.entities_d['edge'])
edge_nodes = []
self.entities_i['edge_nodes'] = edge_nodes
edge_uuid = self.entities_d['edge']['uuid']
for i in range(0, self.edge_nodes_number):
    edge_nodes.append(self.fog05_api.fdu.instantiate(edge_uuid, nodes['edge']))

edge_files = get_data_files(data)
for i in range(0, len(self.entities_i['edge_nodes'])):
    address = get_address(self.fog05_api, self.entities_i['edge_nodes'][i],
'eth0') + ':' + self.process_port
    files = {'profile.txt': open(edge_files['edge'][i]['profile.txt'], 'r'),
'TS1.txt': open(edge_files['edge'][i]['TS1.txt'], 'r'),
    'TS2.txt': open(edge_files['edge'][i]['TS2.txt'], 'r'), 'TS3.txt':
open(edge_files['edge'][i]['TS3.txt'], 'r'),
    'TS4.txt': open(edge_files['edge'][i]['TS4.txt'], 'r')}
    load_data(address,files)

edge_addresses = []
for edge_node in self.entities_i['edge_nodes']:
    edge_addresses.append(get_address(self.fog05_api, edge_node, 'eth1') + ':' +
self.process_port)
command = "#cloud-config\nruncmd:\n  - [ sh, -xc, 'python3 -u
/home/ubuntu/code/server.py "
for address in edge_addresses:
    command += address + " "
command += "> /home/ubuntu/log.txt' ]"
self.entities_d['coordinator']['configuration']['script'] = command
self.fog05_api.fdu.onboard(self.entities_d['coordinator'])
self.entities_i['coordinator'] =
self.fog05_api.fdu.instantiate(self.entities_d['coordinator']['uuid'],
nodes['coordinator'])

create_server(self.start_task_execution, self.finish_task_execution)
```

The first two sections are identical to the ones from the deployment of the gossip learning architecture: we find a mapping and create the needed networks. Then, in the third section, we create the edge nodes by: defining the command to execute at boot (starting the server), onboarding the descriptor and instantiate an entity as many times as specified in the configuration phase. Next, in the fourth section, we load data to edge nodes with the same algorithm used in the

previous architecture (also in this case we need this step to load the datasets to the edge nodes). Finally, we can instantiate the coordinator: first, we fetch all the edge nodes' addresses, then we define the command to instantiate the server at boot (by passing as arguments all the found addresses) and we set it into the descriptor, next we onboard the modified descriptor and at the end, we instantiate the coordinator (always using "entities_i" and "entities_d" as explained before). Now that both the edge nodes and the coordinator are ready, we can create a server (through the last instruction) to remotely control this architecture; the REST APIs that this server offers are two, start and stop: with the former, we can ask (from the orchestrator) the architecture to execute the task passed with the request, with the latter, we can signal to the orchestrator that the task has been carried out. When a request arrives to the start endpoint, the following method is executed:

```
nodes = self.get_mapping('task')
if not nodes:
    raise('Mapping not found!')

aggregators_number = round(self.edge_nodes_number /
self.edge_nodes_per_aggregator)
coordinator_master_address = get_address(self.fog05_api,
self.entities_i['coordinator'], 'eth1') + ':' + self.process_port
command = "#cloud-config\nruncmd:\n  - [ sh, -xc, 'python3 -u
/home/ubuntu/code/server.py " + str(aggregators_number) + " " +
coordinator_master_address + " > /home/ubuntu/log.txt' ]"
self.entities_d['master_aggregator']['configuration']['script'] = command
self.fog05_api.fdu.onboard(self.entities_d['master_aggregator'])
self.entities_i['master_aggregator'] =
self.fog05_api.fdu.instantiate(self.entities_d['master_aggregator']['uuid'],
nodes['master_aggregator'])

aggregator_master_address = get_address(self.fog05_api,
self.entities_i['master_aggregator'],'eth1') + ':' + self.process_port
command = "#cloud-config\nruncmd:\n  - [ sh, -xc, 'python3 -u
/home/ubuntu/code/server.py " + str(self.edge_nodes_per_aggregator) + " " +
aggregator_master_address + " > /home/ubuntu/log.txt' ]"
self.entities_d['aggregator']['configuration']['script'] = command
self.fog05_api.fdu.onboard(self.entities_d['aggregator'])
aggregators = []
self.entities_i['aggregators'] = aggregators
for i in range(0, aggregators_number):
        aggregators.append(self.fog05_api.fdu.instantiate(self.entities_d['aggre
gator']['uuid'], nodes['aggregator']))
```

```
aggregators_addresses = []
for aggregator in self.entities_i['aggregators']:
    aggregators_addresses.append(get_address(self.fog05_api, aggregator, 'eth1')
+ ':' + self.process_port)
task['aggregators'] = aggregators_addresses
coordinator_mgmt_address = get_address(self.fog05_api,
self.entities_i['coordinator'], 'eth0') + ':' + self.process_port
endpoint = 'http://' + coordinator_mgmt_address + '/start'
r = requests.post(endpoint, json=task)
```

The first operation is always to request a mapping, specifying the operation for which we are requesting this mapping. Then, in the second section of the block above, we instantiate the master aggregator by: finding the number of the aggregators (dividing the number of edge nodes by the number of edge nodes per aggregator) and the address of the coordinator, setting the command of the descriptor to execute the server at boot by passing as arguments the aggregators' number and the coordinator's address, and finally onboarding and instantiating the resulting descriptor memorizing the instance identifier in "entities_i". Then, with the same algorithm, we instantiate as many aggregators as requested, defining the command to execute their server by passing as argument the number of edge nodes they manage and the master aggregator's address. Finally, the last section of code explains how the task is started: first, the aggregators' addresses are fetched, then we attach them to the request we are going to send to the coordinator, and at the end, we get the coordinator address and, using it, we send the request to start executing the task. When the coordinator receives the results from the master aggregator, it signals to the orchestrator (through the REST API finish) that the task has been executed, and then the orchestrator terminates both the aggregators and the master aggregator (with a method like the undeploy we analyzed in the gossip learning architecture).

For what concerns both the serve and undeploy methods, they are managed in the same way as shown in the gossip learning architecture: for the former, we instantiate a sensor that contacts the edge to get predictions based on data that it reads on files, while for the latter, we simply terminate the edge nodes, the coordinator and the server (emptying the related information in "entities_i" and "entities_d" as seen before).

The last note regards (as before) the method check_status: in this case, we signal the architecture as offline if the coordinator is down or if there is no active edge node; to check if the coordinator or the edge node is active, as before, we call the method get_address to see if there is an actual address associated to the specified entity.

8.3 Providing the prediction service

Our architectures are mapped and ready to be used: through the following script we can exploit the architectures in order to serve as efficiently and effectively as possible the prediction service.

```
yaks_address = ''
if len(sys.argv) < 2:
        print('[Usage] {} <yaks_address ip:port>'.format(sys.argv[0]))
        exit(0)
        yaks_address = sys.argv[1]

fla = FederatedLearningArchitecture()
gla = GossipLearningArchitecture()
architecture_repository = ArchitectureRepository()
architecture_repository.add_architecture(fla)
architecture_repository.add_architecture(gla)

architecture_data_repository = ArchitectureDataRepository()
architecture_data_repository.add_architecture_data(fla.__class__.__name__,
'hydraulic-system',
        '/home/osboxes/Scrivania/one_node_deployment/architecture_data_repositor
y/data/FederatedLearningArchitecture/hydraulic-system/deploy')
architecture_data_repository.add_architecture_data(gla.__class__.__name__,
'hydraulic-system',
'/home/osboxes/Scrivania/one_node_deployment/architecture_data_repository/data/G
ossipLearningArchitecture/hydraulic-system/deploy')

strategy_repository = StrategyRepository(architecture_repository)
fla_config = {'edge_nodes_number': 1, 'edge_nodes_per_aggregator': 1}
gla_config = {'edge_nodes_number': 4, 'peers_per_edge': 3}
strategy_index = strategy_repository.define_strategy({fla.__class__.__name__:
fla_config}, [{gla.__class__.__name__: gla_config}])

asm = ActiveStrategyManager(architecture_data_repository, strategy_repository)

environment = {}
association_args = {fla.__class__.__name__: {'process_port': '5000'},
gla.__class__.__name__: {'process_port': '5000'}}
active_strategy_index = asm.start_strategy(yaks_address, association_args,
environment, 'hydraulic-system', strategy_index)
time.sleep(240)
```

```
orchestrator.serve_strategy(active_strategy_index)
time.sleep(240)
orchestrator.stop_strategy(active_strategy_index)
```

First of all, it is important to note that this script is parametrized by the address of the YAKS instance that it wants to manage, therefore we need to execute it by specifying both an IP address and a port on which a YAKS instance is listening.

Then, we begin instantiating the architectures: we call the constructors of FederatedLearningArchitecture and GossipLearningArchitecture, thus making them load their descriptors from paths statically provided in their constructors and initializing the variables "entities_i", "entities_d" and "networks_d", that we saw before.

Afterwards, we initialize the ArchitectureRepository, and call the add_architecture method to store the previous initialized architectures.

Next, we also load data for each architecture by creating an instance of ArchitectureDataRepository and we call the method add_architecture_data by passing as arguments the architecture for which we are loading data, the data identifier and the folder path containing the data.

Now, it is the turn of instantiating a StrategyRepository, passing as argument the architecture repository, and use it to call the method define_strategy, that expects a dictionary and a list of dictionaries as arguments: the single dictionary represents the main architecture and contains, as key, the name of the architecture class, and as value, the configuration for that architecture, while the list of dictionaries represents the set of replacement architectures available (as for the main one, also with the replacement architectures we have as key the name of the class and as value the associated configuration). As for the configurations, we are forced, for the federated learning architecture, to set 1 as the number of edge nodes and 1 as the number of the aggregator, while for the gossip learning architecture, to set 4 as the number of edge nodes and 3 as the number of peers per edge node; this is given both by the computation limits of the current test environment and by bugs still affecting YAKS that sometimes prevents the correct instantiation of well-defined entities because of deadlocks derived from the data distribution approach it employs. Once we call the method above, we obtain the index referring this strategy (memorized in the variable strategy_index), to be used with ActiveStrategyManager later.

Finally, we can instantiate the ActiveStrategyManager passing by arguments the ArchitectureDataRepository and the StrategyRepository and then, use the method start_strategy to initiate the training process. In order to do that, we need to define the environment dictionary and the association arguments, then pass them together with YAKS instance's address, a data name and

an index of the strategy that we want to instantiate with the above method. As explained, through this method we deploy the main architecture and start monitoring it, obtaining the index of the ActiveStrategy just started.

At this point, we wait some time (simulating that the learning process has been completed in the meantime) through the method sleep offered by the module time, and then we serve the architecture through the method serve_strategy, passing as argument the ActiveStrategy's index just obtained.

Finally, after another pause, we use the method stop_strategy, always passing the index above, to stop serving the prediction service and to undeploy the active architecture.

Concluding, it is understandable how, by using our proposed orchestrator that combines the architectures that supply the same specific functionality, a service can be resiliently and efficiently provided. In fact, once we start a strategy, we also begin a continuous monitoring of the status of the active architecture and have the orchestrator automatically react to status changes, by deploying one of the replacement architectures in place of the main one: there is no need for manual intervention. This is not only a gain in terms of reaction rate to the problems the architecture is encountering, but also in terms of automatizing and carrying out faster the operations that a user should do to deploy and undeploy the used architecture. In fact, we cannot forget about the problem of meeting the requirements of an architecture in terms of storage and computation capabilities when deploying it: a user should first check manually for compatibility with the current environment (in terms of available nodes versus required ones) and then deploy the entities, possibly choosing the best node for each entity to deploy; this orchestrator automatizes also this aspect, by using the method get_mapping presented before.

Another important aspect is for sure the flexibility that this orchestrator offers: we defined a general Architecture interface, that each architecture that wants to be managed by this tool needs to extend. The only requirement needed, apart from the one just described, is to map the nodes, composing the architecture, to fogØ5 atomic entities, but given the many management possibilities offered by fogØ5, this requirement is not so compelling, as we demonstrated in the previous sections with the architectures we created. Moreover, the division between ArchitectureRepository, containing the "pure" form of the architecture, StrategyRepository, containing the configured state of the architecture, and ActiveStrategyManager, containing the associated state of the architecture, permits the easy reusage of the same architecture in many different strategies and active strategies; together with ArchitectureDataRepository, that permits

to associate, for the same architecture, many different additional sets of data to further personalize the behavior of the needed architecture.

Finally, one last perk to be noted is for sure the ease-of-use: each one of the orchestrator's classes is very easy to instantiate and use (apart from ActiveStrategyManager, that presents a little more complexity), making the core logic easy to understand and control: all the complexity is relegated to the actual architectures, where the end user needs to define the logic for the various operations presented before, but this should not be a problem, since the APIs and the schemas for the descriptors offered by fogØ5 are simple enough to permits a smooth mapping of the needed architectures even without having a deep comprehension of fogØ5 itself.

9. Conclusions and future developments

Our aim for this master thesis was to provide a prediction service for a condition monitoring problem principally exploiting emerging concepts from the fog computing research field. In particular, our first purpose was to highlight the importance and the validity of distributed machine learning in general and also, especially, in light of the future available deployment environments, in which compute, storage and networking resources can be found in any place of the network and therefore exploited to execute operations where it makes the most sense. As a second objective, we wanted to confirm fogØ5 as the tool that can, in the near future, effectively and efficiently tackle the upcoming management problem of massive distributed deployments across networks, as its primary goal (in anticipation of the wide diffusion of fog networks) is to offer a simple but rich interface to control and manage all types of resources that can be found across networks.

First, we demonstrated that, in machine learning domain, there are some use cases (like ours) where the distributed solutions can come up nearly on par with the local centralized solutions, considering the reached accuracy. But we have to remember that accuracy, in use cases like this, is not the only parameter to take into consideration when comparing the solutions: with distributed architectures like the ones proposed, we also gain privacy, control over data, reduced resource consumption and low latency. Delving into the proposed distributed architectures, we have that the federated learning, in terms of time to convergence and fluctuations in the training process, overcomes the gossip learning, but we need to keep in mind that the former needs more resources to be deployed with respect to the latter; moreover, focusing on the gossip learning architecture, we also demonstrated how, with settings principally oriented to avoid faulty nodes, the capability of the nodes to discover new peers remains remarkable. Concluding, it could be of great use to test the created architectures on different use cases and, if possible, with many more nodes, to compare how the training progress changes. In addition, another important aspect to be tested could be the concept drift, to observe how fast these architectures can react to change in the underlying statistical properties.

Then, for what regards fogØ5, we proved that it can be a valid solution (even if it is still in an early development stage) by managing the deployment of the complex architectures presented before. But for difficult use cases like ours, we could not use it in its “pure form”: we needed to encapsulate its functionalities inside a management software to better control the architectures. Therefore, we defined an orchestrator, that filled in this gap and solved the necessity to automatize the operations

of deployment and undeployment of those architectures. Of course, there is still plenty of room for improvement: one enhancement could be to refactor the orchestrator's codebase to define each class as a microservice and let them communicate remotely. This could greatly improve the management possibility offered by the orchestrator, as we can imagine having, for example, a centralized node hosting the ArchitectureRepository, that can load the requested architectures to another node hosting the StrategyRepository, that, in turn, can load the needed strategies to ActiveStrategyManager's instances. Through this deployment, we can have two centralized entities from which ActiveStrategyManager's instances hosted on nodes scattered all over the world can fetch the needed strategies: in particular, we can imagine using an ActiveStrategyManager for each different environment (in our case, a YAKS instance or a set of interconnected YAKS instances) that can correspond, in the real world, for example, to a set of nodes belonging to a company. Concluding, as fogØ5 advances through the development stages, combining it with the proposed orchestrator (updated with the suggested ideas) could result in a very useful instrument to effectively tackle massive parallel management of distributed architectures across networks.

References

[1] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, E. Riviere. Edge-centric Computing: Vision and Challenges. In ACM SIGCOMM Computer Communication Review, Volume 45, Number 5, October 2015;

[2] A. Corsaro, G. Baldoni. fogØ5: Unifying the computing, networking and storage fabrics end-to-end. In 3rd IEEE Cloudification of Internet of Things, Conference Paper, July 2018;

[3] OpenFog Consortium Architecture Working Group. OpenFog Architecture Overview. In technical report, February 2016;

[4] OpenStack Foundation. What is OpenStack? In https://www.openstack.org/software/, 2019;

[5] A. Corsaro, E. Boasson, O. Hecart. zenoh: The zero network overhead protocol. In http://zenoh.io/download/pdf/zenoh.pdf, ADLINK Technology, July 23, 2018;

[6] R. Warren. Data Distribution Service (DDS) Brief. In OMG Specifications, August 1, 2011;

[7] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, B. Ohlman. A survey of information-centric networking. In IEEE Communications Magazine, volume 50, issue 7, July 2012;

[8] G. Baldoni. eclipse/fogØ5. In https://github.com/eclipse/fog05, Eclipse, July 13, 2019;

[9] G. Baldoni, A. Corsaro, J. Enoch. atolab/yaks-python. In https://github.com/atolab/yaks-python, ATOLAB, July 13, 2019;

[10] GlobalPlatform. The Trusted Execution Environment: Delivering Enhanced Security at a Lower Cost to the Mobile Market. In White Paper, June 2015;

[11] D. Happ, N. Karowski, T. Menzel, V. Handziski, A. Wolisz. Meeting IoT Platform Requirements with Open Pub/Sub Solutions. In Annals of Telecommunications, July 2016;

[12] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu. Edge Computing: Vision and Challenges. In IEEE Internet of Things Journal, Vol. 3, No. 5, October 2016;

[13] C. Lemaréchal. Cauchy and the Gradient Method. In Documenta Mathematica, Extra Volume ISMP 251–254, 2012;

[14] T. E. Oliphant. Guide to NumPy. In http://web.mit.edu/dvp/Public/numpybook.pdf, Dec 7, 2006;

[15] D. Cournapeau. scikit-learn user guide. In https://scikit-learn.org/stable/_downloads/scikit-learn-docs.pdf, May 24, 2019;

[16] W. McKinney & PyData Development Team. pandas: powerful Python data analysis toolkit. In https://pandas.pydata.org/pandas-docs/stable/pandas.pdf, July 18, 2019;

[17] CNTK team. microsoft/CNTK. In https://github.com/microsoft/CNTK, July 26, 2019;

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro et al.. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In TensorFlow Preliminary White Paper, November 9, 2015;

[19] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito et al.. Automatic differentiation in PyTorch. In https://openreview.net/pdf?id=BJJsrmfCZ, October 28, 2017;

[20] IEEE. IEEE 1934-2018 - IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing. In https://standards.ieee.org/standard/1934-2018.html, August 2, 2018;

[21] M. Caudill. Neural Network Primer: Part I. In AI Expert, Volume 2 Issue 12, December 1987;

[22] I. Goodfellow, Y. Bengio, A. Courville. Deep Learning. In MIT Press, 2016;

[23] F. Chollet et al.. Keras: The Python Deep Learning library. In https://keras.io, 2015;

[24] S. Ruder. An overview of gradient descent optimization algorithms. In https://arxiv.org/pdf/1609.04747.pdf, June 15, 2017;

[25] A. Corsaro. Breaking the Edge -- A Journey Through Cloud, Edge and Fog Computing. In https://www.slideshare.net/Angelo.Corsaro/breaking-the-edge-a-journey-through-cloud-edge-and-fog-computing, May 2, 2019;

[26] M. Matsugu, K. Mori, Y. Mitari, Y. Kaneda. Subject independent facial expression recognition with robust face detection using a convolutional neural network. In Neural Networks 16 (2003) 555–559, June 2003;

[27] T. Wang, D. J. Wu, A. Coates, A. Y. Ng. End-to-end text recognition with convolutional neural networks. In Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012), 11-15 November 2012;

[28] S. Kido, Y. Hirano, N. Hashimoto. Detection and Classification of Lung Abnormalities by Use of Convolutional Neural Network (CNN) and Regions with CNN Features (R-CNN). In 2018 International Workshop on Advanced Image Technology (IWAIT), 7-9 January 2018;

[29] A. van den Oord, S. Dieleman, B. Schrauwen. Deep content-based music recommendation. In Curran Associates, Inc. pp. 2643–2651, December 2013;

[30] N. Helwig, E. Pignanelli, A. Schütze. Condition monitoring of a complex hydraulic system using multivariate statistics. In IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings, 2015;

[31] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, D. J. Inman. 1D Convolutional Neural Networks and Applications – A Survey. In arXiv e-prints, May 2019;

[32] R. Ormandi, I. Hegedus, M. Jelasity. Gossip Learning with Linear Models on Fully Distributed Data. In Concurrency and Computation Practice and Experience, June 2012;

[33] M. Jelasity, S. Voulgaris, R. Guerraoui, A. M. Kermarrec, M. van Steen. Gossip-based peer sampling. In ACM Trans. Comput. Syst. 25, 3, Article 8, August 2007;

[34] Pallets Organization. Flask. In https://flask.palletsprojects.com/en/1.1.x/, September 2019;

[35] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. McMahan, T. Overveldt, D. Petrou, D. Ramage, J. Roselander. Towards Federated Learning at Scale: System Design. In arXiv e-prints, March 2019;

[36] WeBank AI Group. Federated Learning White Paper 1.0. In https://www.fedai.org/static/flwp-en.pdf, September 2018.