Journal of Ambient Intelligence and Humanized Computing (2023) 14:6563–6584
https://doi.org/10.1007/s12652-021-03528-8

ORIGINAL RESEARCH

LE2ML: a microservices-based machine learning workbench as part of an agnostic, reliable and scalable architecture for smart homes

Florentin Thullier[1,2] · **Sylvain Hallé**[2] · **Sébastien Gaboury**[1]

Received: 17 November 2020 / Accepted: 23 September 2021 / Published online: 17 October 2021

Abstract
Over the years, several architecture of smart home has been proposed to enable the use of ambient intelligence. However, the major issue with most of them lies in their lack of high reliability and scalability. Therefore, the first contribution of this paper introduces a novel distributed architecture for smart homes, inspired by private cloud architectures, which is reliable and scalable. This implementation aims at simplifying and encouraging both the deployment of new software components as well as their reutilization to achieve the activity recognition process inside smart homes. The second contribution of this paper is the introduction of the LIARA Environment for Modular Machine Learning (LE2ML), a new machine learning workbench. Its design relies on a microservices architecture to provide a better scalability as well as smaller and faster deployments. Experiments demonstrate that our architecture is resilient to both a node failure and a total power outage. Moreover, the workbench obtained similar results, as regards the performance of the recognition, when compared to previously proposed methods.

Keywords Machine learning · Workbench · Architecture · Smart home

1 Introduction

Innovative technologies that rely on a certain degree of applied artificial intelligence (AI) have been widely adopted in the past few years. Since the introduction of the Internet of Things (IoT) (Gubbi et al. 2013), more and more of such so-called "smart" technologies have been proposed. Their particularity is their ability to acquire different sources of information from the surrounding environment and react accordingly. One of the practical applications of such smart technologies is the smart home. These residences are equipped with small pieces of electronic devices integrated in our everyday life objects to provide tailored services in a transparent way for its users (Plantevin et al. 2019; Marikyan et al. 2019).

Over the years, various architecture implementations of smart homes have been proposed in laboratories or inside real habitations built for a research purpose (Cook et al. 2003; Helal et al. 2005; Chen et al. 2009; Giroux et al. 2009; Cook et al. 2013; Bouchard et al. 2014; Lago et al. 2017; Plantevin et al. 2019). These works mainly focus on using sensors, effectors and learning algorithms to enable the use of Ambient Intelligence (Amb.I.) as an empirical method in order to support elderly autonomy and health monitoring. While each work in the field of smart homes distinguishes itself from the others, they all suggest methods to address the activity recognition problem (Chen et al. 2012). The major issue of most related smart home architectures lies in their lack of both high reliability and scalability capabilities mainly due to the presence of at least one point of failure (Plantevin et al. 2017). In that sense, Plantevin et al. (2019) have tackled this specific topic by introducing a new kind of architecture based on distributed smart sensors. While this work remains relatively new, authors plan to perform further experiments before releasing a real life implementation.

✉ Florentin Thullier
florentin.thullier1@uqac.ca

Sylvain Hallé
sylvain_halle@uqac.ca

Sébastien Gaboury
sebastien_gaboury@uqac.ca

[1] LIARA Laboratory, Department of Computer Science and Mathematics, Université du Québec à Chicoutimi, Chicoutimi, QC, Canada

[2] LIF Laboratory, Department of Computer Science and Mathematics, Université du Québec à Chicoutimi, Chicoutimi, QC, Canada

However, it already appears that the proposed architecture was designed with a strong thinking of what a consumer-ready smart home architecture should be, whereas the simplicity in the implementation of new experimental methods and protocols seems to have been put aside.

As the main objective of the activity recognition process is to obtain knowledge from data generated by the sensors present in the home, all proposed architectures rely on both sensors as well as computing units. These last ones are required in order to achieve resource-intensive tasks such as data processing and either traditional Machine Learning (ML) (Ramasamy and Roy 2018) or Deep Learning (DL) algorithms (Wang et al. 2019). As regards the use of sensors, several examples can be found in the literature. They include methods based on passive Radio-frequency Identification (RFID) (Fortin-Simard et al. 2015), Ultra Wide Band (UWB) radars (Bouchard et al. 2020) as well as vision-based approaches (Zhang et al. 2017). Moreover, in some other cases, wearable devices were introduced in smart home environments leading to the utilization of another kind of sensors such as Inertial Measurement Units (IMUs) as well as health-related sensors (Davis et al. 2016; Chapron et al. 2018). Most of these suggested methods represent built-in activity recognition implementations. In that sense, these works appear to be immutable or in other words, difficult to extend. Thus, components of the application stack generally require to be either developed again, or adapted in order to handle any change in the method. Furthermore, the replicability of related activity recognition methods are, most of the time, a pain to deal with. Indeed, depending on the dependencies, the language and the platform, these applications may be difficult to adapt from one laboratory to another.

Nevertheless, several works have mentioned the use of a machine learning workbench to process data from sensors and perform the activity recognition. From an academic point of view, these workbenches are not new and they allow fast-prototyping, data visualization as well as a better replicability and reusability of experimental methods and software components (Holmes et al. 1994; Langlois and Lu 2008). As an example, the most well known and reported of such tools in the literature is WEKA (Waikato Environment for Knowledge Analysis), an open source Java-based workbench that supports a large number of supervised and unsupervised machine learning algorithms (Witten et al. 2016). While it allows a broad range of options, its usage remains memory intensive and it may be hard to scale. With the trend currently surrounding artificial intelligence, these tools become more and more popular since public cloud computing providers such as Amazon AWS, Google Cloud Platform and Microsoft Azure have released consumer-oriented versions of such applications. As they compose the vast list of services that each provider present in their paid plans, these applications then benefit from the scalability offered by the cloud computing. However, since they are considered as general purpose services, internal mechanisms of machine learning techniques are completely hidden in order to make them accessible to all.

This work first introduces a new kind of smart home architecture which is based on a cluster and that can be seen as an on-premise private cloud. This implementation mainly aims at solving reliability issues observed with existing architectures without the need to entirely redesign an already operational system. Indeed, through such a contribution, our main objective is to simplify overly complex and unreliable implementations that lack scalability (Giroux et al. 2009; Bouchard et al. 2014) or are built on obsolete concepts (Chen and Nugent 2010; Valiente-Rocha and Lozano-Tello 2010; Bae and Kim 2011) or technologies (Cook et al. 2003; Helal et al. 2005). Hence, through a series of different experiments conducted on the proposed architecture, its resiliency as regards a complete loss of a node when either network or electric failures were simulated was successfully demonstrated.

The second objective of this work is to introduce LE2ML (LIARA Environment for Modular Machine Learning), a new kind of workbench for machine learning that makes up the glue between the smart home hardware infrastructure and the software related to Amb.I processes. In the same idea behind our smart home architecture, such a software exploits the microservices technology since it allows a better scalability, smaller and faster deployments as well as fault isolation (Dragoni et al. 2017). While, the use of microservices greatly simplifies the integration of the workbench to our proposed architecture and any other ones, LE2ML also benefits, thanks to this technology, from a modular design. This enables the implementation and the deployment of agnostic software in terms of programming languages or execution environments. Finally, we expose, through the use of this tool, that it is possible to reproduce an existing machine learning method (Thullier et al. 2017) without worsening the performance of the recognition.

The rest of the paper is structured as follows: Sections 2 and 3 present an overview of related work to both proposed smart home architecture as well as suggested machine learning workbenches. Sections 4 and 5 detail our proposed architecture and machine learning workbench respectively. Next, Sect. 6 presents some tests we have conducted. Finally, Sect. 7 draws a conclusion and Sect. 8 provides future works we will accomplish.

2 Related smart home architectures

Multiple smart home architectures have been proposed over the years. However, given the lack of any specification for an optimal architecture of such habitats, their implementations

remain different despite their common motivation to achieve activity recognition. While such a process suggests smart homes to have particular needs to improve the assistance of the residents, this section only exposes the most well-known related architectures that are fully operational within academic environments and does not take into account generic distributed systems based on Hadoop[1] or other distributed platforms. In that sense, this section aims at identifying the most important drawbacks this work intent to address.

2.1 Industrial legacy

Bouchard et al. (2014) and Giroux et al. (2009) have introduced two academic implementations of smart homes (respectively the LIARA and DOMUS laboratories) that share the same architecture built from industrial-grade hardware. They include a main server that hosts all the applications required to perform the activity recognition process, and thus are considered to be *monolithic* architectures. The transducers (sensors and actuators) present in the environment are clustered in several islands, each one being connected to an industrial automaton and they only communicate over the Modbus TCP protocol. In order to allow the applications to either read values from sensors, or updates values of actuators, a relational database is interconnected between the server and the automaton. In addition, several other transducers such as RFID readers are directly connected to the server that is in charge of updating the same database with related data.

The main advantage of these architectures lies in the robustness and the reliability of the industrial-grade hardware. Moreover, such centralized architectures simplify the interaction between data and applications since they only have to focus on the communication with the database; the complexity of the hardware layer is abstracted. Nevertheless, smart home architectures based on industrial equipment suffer from two main drawbacks. The first one refers to the cost of these architectures evaluated up to 13,500 U.S. dollars (Plantevin et al. 2017). It indubitably represents a large investment especially for elderly or people suffering from various diseases who already have to assume, partially or totally, the charge of their medication. Finally, the centralized design that facilitates the interaction with the transducers also creates several points of failure. Indeed, even if industrial-grade hardware is a guarantee of quality, it is neither impossible for the server to fail, nor for the database to corrupt or lose data. Thus, the result would be either a partial, or a total malfunction of the smart home and unsafe situations for the occupant may occur. In that sense, it appears clearly that the reliability of these smart home architecture still has to be improved even if it remains an academic implementation built for research purposes.

2.2 Service-oriented

As compared to industrial-based smart home architectures, several other implementations have been built with the objective of improving the hardware scalability capability by simplifying the integration of new transducers to smart homes such as service-oriented architectures. First works on such implementations rely on the Open Service Gateway initiative[2] (OSGi) more of less directly (Cook et al. 2003; Helal et al. 2005; Lago et al. 2017). As an example, in the GatorTech architecture (Helal et al. 2005), each transducer has its own driver allowing the communication with the rest of the system. Since transducers contain Electrically Erasable Programmable Read-Only Memory (EEPROM) where drivers are permanently stored, they can register themselves autonomously to the OSGi service definition once they are powered up. The service definition concept refers to an abstraction in order to create basic services allowing an immediate consumption of highly abstracted data. In addition, the combination of basic services makes it possible to create composite ones. For example a voice recognition service can be defined based on every basic services of microphones, or an indoor localization service through all the RFID basic services. Finally, each service and transducer are defined in an ontology in order to enable knowledge modeling. This includes, for example, the capability for the system to convert the output value of a temperature sensor from Celsius to Fahrenheit before feeding it to another service. Since these early works, several smart home service-oriented architectures have been proposed pushing forwards the use of ontologies to allow their use with the integration of newer technologies such as IoT (Hu et al. 2019) or mobile devices (Triboan et al. 2016).

The main advantages provided by these smart home architectures is their ability to handle hardware scalability. Then, through the high level of abstraction offered by the services, the development of activity recognition applications was made easier as researchers does not have to be concerned of the complexity of the underlying hardware infrastructure (*i.e.* protocols, data format, *etc.*). Nevertheless, as with the LIARA and the DOMUS smart home implementations, every related service-oriented architectures are built on a unique centralized server, creating a Single Point of Failure (SPoF). Moreover, in some implementations, it is possible to observe a bottleneck when a large amount of data is exchanged, resulting in a speed reduction on the server. Thus, it becomes overloaded and malfunctions may occur when it comes to assist smart home occupants. In that sense,

[1] https://hadoop.apache.org/.

[2] https://www.osgi.org/.

it is possible to see that efforts were placed on improving the scalability of smart homes as regards hardware components. However, it is clear that there is still a need for enhancements in terms of software scalability and overall reliability.

2.3 Mesh network

Through the recent advancements in wireless communication technologies, a new type of smart home architectures based on wireless mesh networks appeared (Cook et al. 2013). The most significant implementation of such architectures is CASAS (Cook et al. 2013), which was designed with a strong emphasis on low price and a simplified installation. The proposed architecture relies on four main components. The first one is a physical layer composed by a mesh network that includes every transducer communicating with each other through the ZigBee protocol. Then, all the events of the mesh are sent to the second main component, a messaging service using a ZigBee bridge that is in charge of converting messages in order for them to be compatible with the eXtensible Messaging and Presence Protocol (XMPP). Since the messaging service is a publish/subscribe one, it is possible for some other layers to be integrated to the architecture to exploit transducers. As examples, CASAS includes an application layer to perform the activity recognition as well as an archiving service that stores all messages generated by the environment.

The principal advantage of CASAS and every other architectures based on mesh networks remains their low cost and their simple installation process. Indeed, the cost of CASAS has been estimated at 2765 U.S. dollars—five times less than an industrial-based architecture since it does not include a costly main server. However, while ZigBee is an open standard run by the ZigBee Alliance, there are few devices, others than consumer IoT "smart devices", compatible with the ZigBee protocol out of the box, without requiring a hardware expansion such as a USB dongle for computers. In that sense, the hardware scalability of such implementations seems to be constrained to ZigBee compatible devices only; this represents an obstacle for heterogeneous environments such as smart homes. Moreover, suggested mesh networks architectures are also vulnerable to various points of failure (*i.e.*, the bridges between each layer component) putting their reliability in question.

2.4 Distributed smart transducers

As exposed in the previous sections, the main drawback shared by the different existing smart home architectures lies in their lack of reliability due to their implementation design. As demonstrated, each of them creates from one to many points of failure that may lead to either a partial or a total malfunction of the activity recognition process made inside these habitats. However, this particular issue was addressed several years ago in different fields of computer science such as distributed computing applied to big data with the help of redundancy and clusters (Dikaiakos et al. 2009; Zaharia et al. 2010; Chen et al. 2014; Jafarnejad Ghomi et al. 2017).

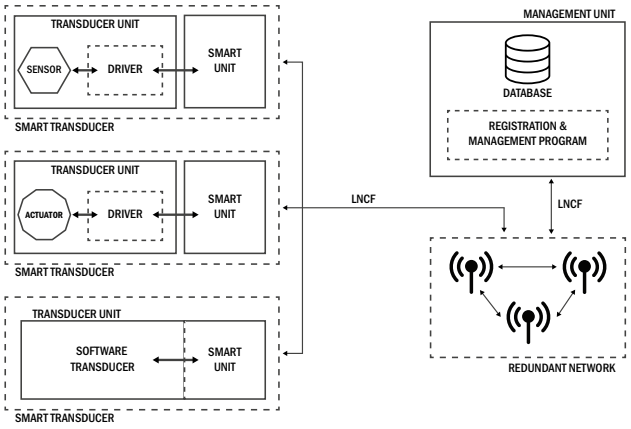

Fig. 1 Implementation design of the smart home architecture based on distributed smart transducers suggested by Plantevin et al. (2019)

Recently, Plantevin et al. (2019) have proposed a distributed smart home architecture in order to remove all possible points of failure present in the other suggested architectures. To that end, they have suggested to implement smart transducers relying on the IEEE 1451.4 standard (Institute of Electrical and Electronics Engineers 1999). As shown in Fig. 1, smart transducers are divided in two different types: physical and software transducers such as Application Programming Interfaces (APIs) consumption. Moreover, every transducer is composed of two distinct units where the communication between them is performed through either a serial link for hardware transducers, or a ZeroMQ[3] inter-process for software ones. The first unit (*i.e.*, the transducer unit) is in charge of making the interface between real world and high-level data. The second unit (*i.e.*, the smart unit) refers to the main computing piece of hardware. Its role is both to communicate with the entire environment and to host the application stack required to perform the activity recognition process through data generated by the transducer. All the smart transducers are interconnected across a redundant wireless network infrastructure and communicate in a distributed manner through the Light Node Communication Framework (LNCF) protocol, a custom protocol previously introduced by the same authors (Plantevin et al. 2017). Since a transducer requires a driver to operate properly, it registers itself at the first boot from a centralized entity (*i.e.*, the management unit). This entity then sends back

[3] https://zeromq.org/.

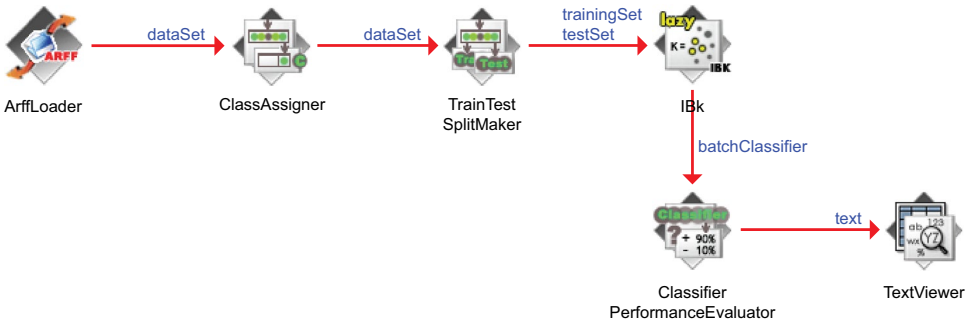

Fig. 2 WEKA *knowledge flow* example of a `k-NN` classification using `k=1` neighbour on a 80/20 split of the iris dataset (Dua and Graff 2017)

the corresponding driver and its description file indicating how the driver has to be flashed on the transducers unit. While the management unit is mandatory in order to set up new smart transducers, it is less important afterwards since it is mainly used to keep track of transducers present in the environment as well as emit alerts when a smart transducer fails.

This recent smart home architecture implementation has the main advantage to be scalable as regards hardware as well as reliable since the high availability of the entire system is guaranteed due to the smart transducers. However, since it is possible to consider the manager unit as a SPoF, authors have mentioned that such a central entity may be hosted in the cloud to ensure its high availability. Moreover, since its role is only important either at the setup stage or in case the architecture need to be modified, a possible fail of this unit does not compromise the proper functioning of the activity recognition process and does not endanger the residents of the home. Nevertheless, the support for the heterogeneity of software components required to perform activity recognition processes seems to have been put aside in the design of this architecture.

2.5 Conclusion

To conclude on these related smart home architectures, it is clear that the centralization of these implementations entails a significant lack of reliability. Moreover, some of these architectures such as CASAS or Gator-Tech also appear to concede difficulties in integrating new types of sensors and different software components, even though they were originally built with a focus placed on improving the scalability and the flexibility of smart home architectures. More recently, Plantevin et al. (2019) have proposed an architecture where efforts have been made to answer the questions of hardware scalability and global reliability of smart homes. Although this work remains a first step towards what the design of a smart home architecture should be, this implementation still suffers from the requirement for a central unit as well as a lack of support for the heterogeneity of software components used in smart home processes, such as activity recognition.

3 Related machine learning workbenches

According to Langlois and Lu (2008), a machine learning workbench is a tool providing a unified interface to a number of machine learning algorithms in order to handle more than one machine learning related problems. Over the past few years, several of these tools have been developed and exploited in various fields of research such as bioinformatics (Larrañaga et al. 2006), cybersecurity (Handa et al. 2019), health care (Rajkomar et al. 2019) and more specifically to perform activity recognition inside smart homes (Chen et al. 2011). This section describes in detail three of the most employed open-source machine learning workbenches of the literature in order to expose the needs of introducing our workbench: LE2ML.

3.1 WEKA

WEKA is an open-source collection of machine learning algorithms written in Java that has been introduced in 1994 to perform data mining tasks (Holmes et al. 1994). It contains tools for data preprocessing, classification, regression, clustering, association rules, deep learning and visualization. While WEKA has its own native file format, it also supports several other ones such as Comma-Separated Values (CSV), Matlab files as well as connectivity to several databases through JDBC (Java Database Connectivity). Then, a vast number of methods allow to filter such data, from removing some attributes to advanced operations such as Principal Component Analysis (PCA). In its first versions, WEKA was only a command-line tool. Nowadays, while it is possible to include it as a dependency of a project to use it directly in both Java or R code (Hornik et al. 2009), it additionally offers to be used through several types of Graphical User Interfaces (GUI). These contain the *explorer*, where the user is allowed to set up quick data manipulation techniques (*i.e.*, filtering, classification, clustering and visualization) without the need of writing code. The *experimenter* is a tool allowing to run in parallel, over different computers in a network, several machine learning experiments in order to evaluate classification and regression methods. The *knowledge flow* interface offers to perform the same tasks as the *explorer*

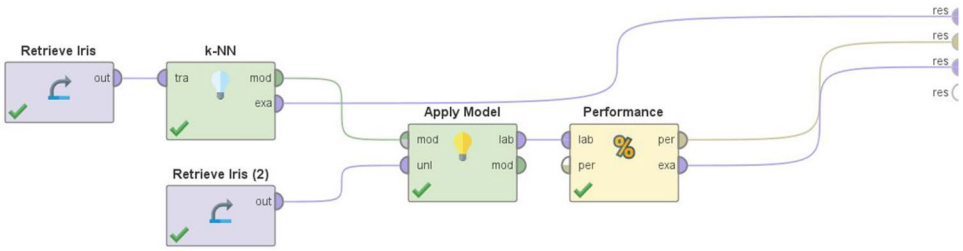

Fig. 3 RapidMiner *operator chain* example of a `k-NN` classification using `k=1` neighbour on a 80/20 split of the iris dataset (Dua and Graff 2017)

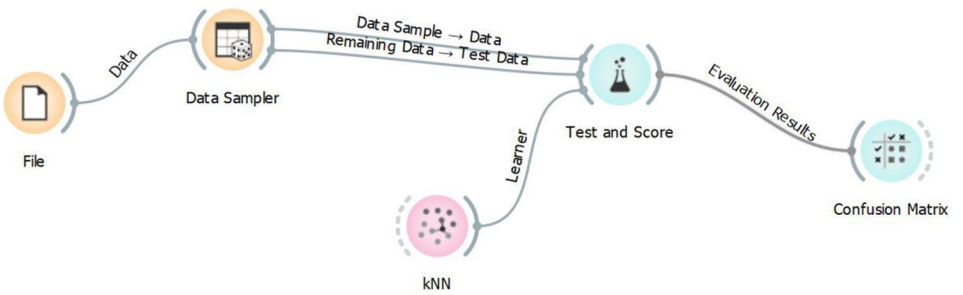

Fig. 4 Orange *schema* example of a `k-NN` classification using `k=1` neighbour on a 80/20 split of the iris dataset (Dua and Graff 2017)

through the manipulation of blocks to build a workflow process as shown in Fig. 2.

WEKA is probably the most well known and used machine learning workbench since it is a portable tool (Bouckaert et al. 2010) offering a wide range of possibilities to achieve machine learning related processes with a unsettle ease. However, it suffers from several drawbacks. Firstly, since it is written in Java and relies on an old codebase, its ability to handle a vast amount of data and the rapidity of various operations are deeply affected by a large computational resource consumption. More recently, a package manager was introduced in order to allow third parties to extend the core features of WEKA (Hall et al. 2009). While such additional functionalities have allowed this tool to become more flexible, these packages can only be developed in Java to suit the *workbench* environment needs.

3.2 RapidMiner

RapidMiner, also known as YALE (Yet Another Learning Environment) (Ritthoo et al. 2003; Hofmann and Klinkenberg 2014) was developed starting in 2001 then rebranded as RapidMiner in 2007. In the same way as for WEKA, RapidMiner provides an integrated Java-based environment for machine learning techniques. Conversely, this tool only offers a GUI to create a workflow—in the same way as the *knowledge flow* interface in WEKA. Indeed, processes are described from elementary blocks called *operators*. It is then possible for the user to compose an *operator chain* by placing *operators* on a canvas and wiring their input and output ports, as shown in Fig. 3. While WEKA is embedded in RapidMiner as a dependency used in several blocks, the majority of proposed blocks inside RapidMiner are extending many aspects of machine learning that are not covered by WEKA.

RapidMiner also includes the possibility to add packages developed by third parties to improve its core capabilities and a script block where it is possible to write code through either a Java or a Groovy syntax. In that sense, RapidMiner appears to be slightly more flexible than WEKA. However, since it relies on a comparable software design and an identical base programming language, this workbench suffers from the same drawbacks. In addition, it is possible to say that RapidMiner is not suitable for the deployment of experimental applications inside a laboratory since there is a serious lack of either a Command-Line Interface (CLI), or a library version allowing its usage inside a program.

3.3 Orange

In the same way, Orange is also considered to be a machine learning workbench (Demšar et al. 2004, 2013). First introduced in 1997, the core functionalities of the tool were initially conceived in `C++` and its upper layer (*i.e.*, modules and GUI) was a Python binding. However, since 2015, Orange has entirely been redeveloped—the `C++` core has been replaced in favor of Python code and the GUI was also redesigned. Such as WEKA, Orange either provides the ability to be imported as a library in a Python script or to be employed as a component-based GUI. Indeed, it is similarly possible to use *widgets* placed on a canvas, to build a *schema* representing the machine learning workflow to achieve, as shown in Fig. 4. Moreover, Orange offers the possibility to use custom *widgets* through a package manager as well.

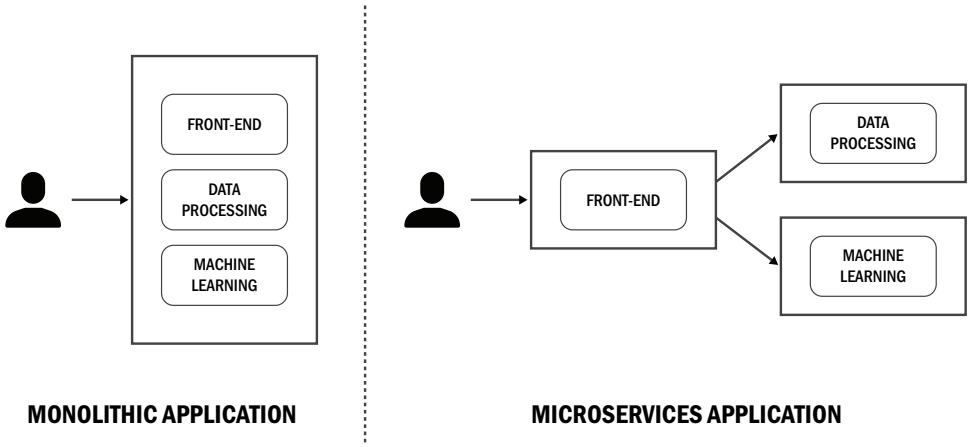

Fig. 5 Comparison between monolithic and microservices architectures for an application that aims at resolving a machine learning problem through an interaction with a GUI

In the same way as WEKA, Orange appears to be more likely to be employed for academic applications since it is possible to integrate it in a Python program. Since the most recent version of Orange (*i.e.*, Orange3) now relies on several modern Python tools such as NumPy,[4] SciPy.[5] and scikit-learn,[6] there is no end to the possibilities for creating custom *widgets*. Moreover, from our perspective, we have observed a more reasonable amount of computational resources consumed when compared to WEKA. However, the guidelines to design custom *widgets* is not trivial and limited to Python code. Finally, both WEKA packages and Orange custom *widgets* have to be manually installed on each end-user devices (*e.g.*, computers or servers) that may become hard to manage between experimenters and that may lead to reusability issues.

3.4 Conclusion

The presentation of the three main workbenches largely employed in academic environments allowed us to identify the main drawbacks of each of these solutions. Thus, it appeared that all of these tools share the same main disadvantage: a strong dependence on their own context. While all of them enable the integration of third-party software components, these components must be developed according to the requirements of the particular tool they are built for. However, such additional packages can become difficult to maintain by a software development team in a research laboratory. Indeed, they must be installed manually on each device composing the environment (*e.g.*, server or computers), which may undoubtedly lead to deployment and reusability issues.

[4] https://numpy.org/.

[5] https://www.scipy.org/.

[6] https://scikit-learn.org/.

4 The proposed architecture

As the main drawbacks of currently proposed smart home architectures lie in their lack of reliability and scalability in terms of software components, the first contribution of this paper is to introduce a new architecture design for smart homes that is based on a cluster implementation to cope with these inconveniences. Moreover, since related architectures also lack flexibility, our architecture has been built to allow interoperability between applications developed in different programming languages, running on various platforms. In addition, this implementation has been made compatible with the majority of existing smart home architectures. It may be seen as an extension of the work of Plantevin et al. (2019) which has been focusing on removing points of failure to enhance the whole reliability of the smart home by introducing distributed smart transducers. The main idea of this work is to go further and suggesting an implementation that relies on microservices instead of a monolithic approach. Thus, we aim at removing completely the remaining points of failure in every related smart homes architecture.

The first section of this fourth part provides required basic knowledge as regards microservices architectures and their various practical implementations in recent years. Then, the second section describes in depth our own implementation that relies on such a microservices technology.

4.1 Basic knowledge

Microservices architectures (Dragoni et al. 2017) recently appeared as a new paradigm for programming applications based on the concept of Service Oriented Architecture (SOA) (MacKenzie et al. 2006). As exposed in Fig. 5, such an approach suggests to split an application into a set of smaller, independent and interconnected services that perform fine-grained functions instead of building a single monolithic application, where all of its components are in

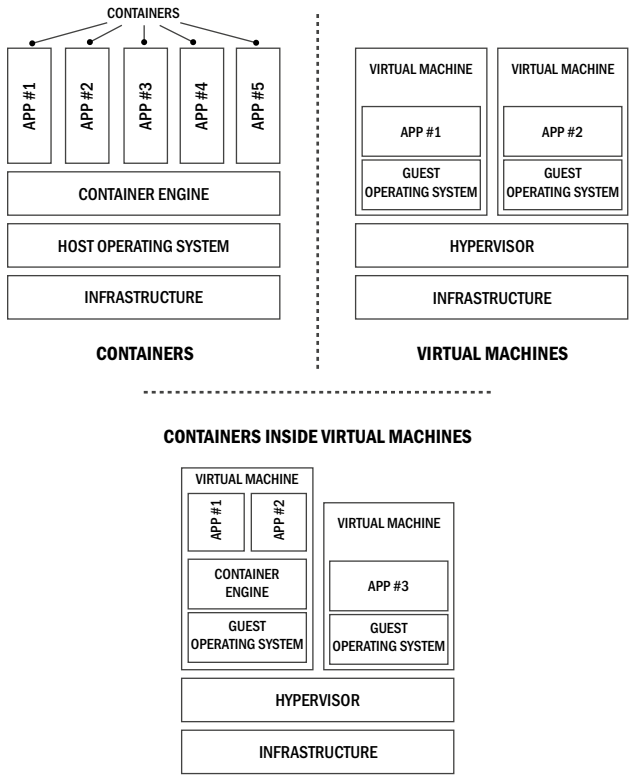

Fig. 6 Detailed illustration of the three main virtualization techniques

a single instance. As compared to monolithic applications, a microservices architecture presents several advantages. Since the applications are fragmented into the most basic level of their functionalities, each service becomes individually easier to maintain as well as much faster to develop and deploy (*Agility*). In addition, such a division allows developers to create applications by employing existing fragments for multiple purposes (*Reusability*). Moreover, the barrier between the technologies such as programming languages is reduced since services are independent (*Technology Agnosticism*). Another benefit of microservices is that as the demand for an application increases, it is possible to either extend resources allocated (*Dynamic Horizontal Scalability*) or to increase the number of instances (*Dynamic Vertical Scalability*) for the most needed services rather than for the entire application. Finally, since the components of an application are distributed, they can handle a service failure by degrading this functionality specifically and not crashing the entire application. Thereby, it is possible to put various mechanisms in place to overcome unexpected failures, increasing the resiliency of the whole system (*Reliability*).

Currently, there are many ways to set up a microservices architecture—the most common one being the use of virtualization tools such as traditional Virtual Machines (VMs), containers, or containers inside VMs. Figure 6 provides an illustration of these three different implementations. Even though the choice of a virtualization method does not affect the resulting architecture benefits, there are significant differences between traditional VMs and containers that are important to detail. The major difference is that containers virtualize at the Operating System (OS) level, whereas traditional VMs virtualize at the hardware level. On the VM side, a hypervisor makes siloed slices of hardware available to allocate resources to each VM. There are generally two types of hypervisors. Some are directly executed on the bare metal of the hardware, while others run as an additional layer of software within the host OS. Conversely, containers allow virtual instances to share a single host OS. Thus, they produce much more lightweight virtual images than traditional VMs. Moreover, since containers are isolated, they execute in separate spaces from each other and from certain parts of the host OS. Such an isolation first allows a more efficient resource utilization and since they only have to terminate the processes running in their isolated spaces, they remain fast to create and destroy. There is no need to boot and shut down an entire OS.

To conclude, a VM is an emulation of a computer system while containers are a standard unit of software that packages up code and all its dependencies (*e.g.*, libraries or binaries) required for the application to run quickly and reliably from one computing environment to another. These two technologies are not incompatible since it is possible for them to be used together. However, executing containers inside virtual machines is, most of the time, the result of the evolution of mature virtualization environments. Generally, containers are run directly on bare metal servers to optimize performance and latency and/or to reduce the licensing costs of both virtualization tools and operating systems.

4.2 Proposed practical implementation

This section exposes in detail the proposed practical implementation of the microservices-based architecture for smart home environments. Figure 7 describes the hardware organization that is mainly built around five main nodes (`m0, m1, m2, w0, w1`). These nodes form a distributed cluster supporting the microservices architecture. In addition, our implementation suggests two more nodes that were intentionally separated from the others (`f0, f1`). A Redundant Array of Independent Disks (RAID) storage is attached to each of these two nodes and they have been configured to provide a distributed file system that is used by the nodes composing the cluster. Indeed, since containers have the ability to mount several volumes, such a file system offers a reliable way to share data between each container regardless the node on which they are running. Moreover, every node is connected to a switch to enable network communication and the entire architecture is isolated inside a local network. This was a recommendation stipulated by our laboratory in order

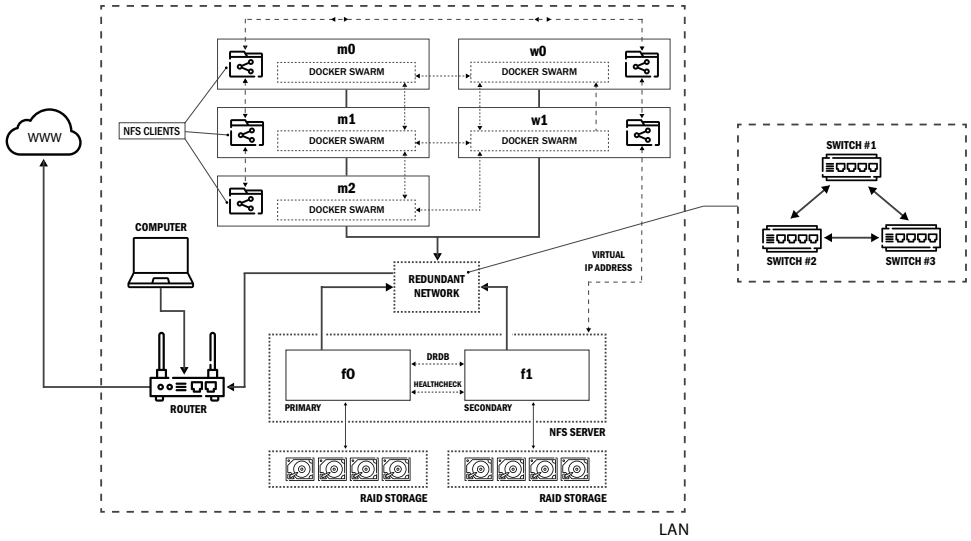

Fig. 7 Hardware organization of the proposed smart home architecture exposing the clustered implementation for microservices as well as the distributed file system

to prevent security issues that may compromise experiments and to preserve the anonymity of participants. However, to facilitate the set up and the maintenance, the architecture is accessible in an ad hoc manner with a computer through a router connected to both the web and the local network.

4.2.1 Distributed file system

The distributed file system offered by the nodes `f0` and `f1` relies on the Network File System (NFS) protocol. The principle of NFS remains simple as it is implemented in a client/server computing model. An NFS sever manages the authentication, authorization, the administration of the clients as well as all the data. Once authorized, it is possible for the NFS clients to access remote data in the same way they are accessed locally. In that sense, each node composing the cluster is allowed to manipulate files (create, read, update, delete) and all the other nodes will be aware of the resulting changes.

However, since we aimed at suggesting a reliable architecture, having only one NFS server did not satisfy such constraint. In that sense, to enable a clustered version of the NFS server, the Distributed Replicated Block Device[7] (DRDB) tool was used to create a distributed and replicated storage system. This tool was configured to perform a full replication of data stored between a primary server (`f0`) and a secondary server (`f1`) that may be seen as a networked RAID 1 storage. In order to favor data integrity over replication speed, a constant synchronous replication between both the primary and the secondary node was chosen instead of either an asynchronous or a semi-synchronous replication. Hence, local write operations on the primary node are considered completed only after both the local and the remote disk writes on the secondary node have been confirmed. However, DRDB does not provide high availability functionalities. In that sense, the Heartbeat[8] tool has also been adopted to achieve such a mechanism. As a background process, the software aims at watching the state of both DRDB servers. In case of failure of the primary node, the virtual IP (VIP) address assigned to the clustered file system and that was attributed to the primary node is automatically reassigned to the secondary node. Thus, the access to the data is not compromised.

4.2.2 Container orchestration

The implementation of the microservices architecture proposed in this paper first relies on Docker[9] as the container engine that was installed on each node composing the cluster (`m0, m1, m2, w0, w1`). In addition, a container orchestrator was necessary to enable a distributed cluster between many nodes. Container orchestration refers to a process that allows to manage the life cycles of containers and their dynamic environment. More precisely, orchestration tools automate several tasks such as the configuration, the scheduling, the provisioning and the scaling of containers as well as load-balancing, resources allocation and the security of the interactions between containers. Currently, there are several popular container orchestration tools such as Kubernetes,[10] Apache Mesos[11] and Docker Swarm.[12]

[7] https://www.linbit.com/drbd/.

[8] http://www.linux-ha.org/doc/man-pages/re-heartbeat.html.

[9] https://www.docker.com/.

[10] https://kubernetes.io/.

[11] https://mesos.apache.org/.

[12] https://docs.docker.com/engine/swarm/.

Docker Swarm is the tool that has been adopted to perform the container orchestration for our architecture. Several factors motivated such a choice. First of all, this tool is simpler to set up since it is packaged, distributed and fully integrated with the container engine we already used. Besides, while it may be considered as less extensible than other tools, it offers all the most important features we needed in order to create and manage our cluster, as described below.

A Docker Swarm cluster is composed of many nodes on which the Docker engine is installed. It is possible for these nodes to have two types of roles, being either a manager or a worker role. A manager node receives the service definitions and dispatches the tasks to worker nodes accordingly. They are also in charge of both the orchestration and management of the cluster. In such a context, services refer to the definition of one or many tasks (*i.e.*, containers) to be executed on the worker nodes. Nevertheless, it is possible to determine manually the deployment policy. Hence, services may be either global or replicated. When a service is global, it is deployed on every available node once. Conversely, when it is replicated, the manager is in charge of the distribution between nodes according to the number of desired replicas. Moreover, every worker node composing the cluster includes an agent that aims at reporting the state of its tasks to the manager nodes. By default, manager nodes are also worker nodes, but it is possible to configure them to not accept any workload.

As regards management, only one manager node is elected as a leader by all the other manager nodes using the Raft consensus algorithm (Ongaro and Ousterhout 2014). Hence, this particular node is in charge of all the decisions for the complete cluster. While there is no limit on the number of manager nodes, the decision about how many managers to implement is a trade-off between performance and fault tolerance. Indeed, if the node elected as the leader of the cluster dies, any other manager may become the leader if the Raft quorum (*i.e.*, a majority agreement between manager nodes) is possible. More precisely, Raft tolerates up to $(N-1)/2$ failures and requires a quorum of $(N/2)+1$ members to allow a decision (N referring to the number of manager nodes). Thus, it is clear that such a process will create a higher network traffic load with a larger number of managers, inducing an overall reduction of the cluster performance. In that sense, we opted for a five node cluster composed of three managers and two workers in order to tolerate a complete failure of one node, whatever its role. Moreover, such a number of manager nodes respect the requirement of having an odd number of manager nodes in order to continue respecting the quorum in case of failure.

To enable a container-to-container communication inside a Swarm cluster, a specific network driver, called overlay, is employed. Since the implementation of this driver relies on the VXLAN technology (Mahalingam et al. 2014) where the required configurations were automated, a distributed network may be created among all the nodes composing the cluster. In that sense, when a Swarm cluster is initialized, an ingress network is automatically created on each node. It is a specific overlay network that aims at facilitating the load balancing of the different services between the cluster nodes. Then, it is possible to create as many overlay networks as needed to connect containers that need to exchange information.

To that end, a bridge network is also created automatically on every node at the cluster initialization. These networks are required in order to link all the overlay networks (including the ingress network) to the physical network of the cluster nodes. Hence, since bridge networks sit on top of the host-specific networks layer, a communication between containers interconnected by an overlay network but executing in different nodes is made possible. By default, these communications are encrypted using the AES-GCM algorithm (Salowey et al. 2008). Manager nodes of the cluster rotate the encryption key every twelve hours to ensure an acceptable level of security without compromising the performance of network communications inside the cluster.

Moreover, overlay networks induce a network isolation by dynamically assigning a VIP address to every container at their creation. Such a process simplify the service consumption. However, when the number of services becomes important, it may appear relevant to know some VIP addresses in advance, which is obviously not possible. To solve this issue, the Swarm orchestrator provides a mechanism named "service discovery". This feature relies on a DNS server embedded in the container engine of each node that is in charge of resolving services name. Then, network queries are routed to the corresponding containers through their associated virtual IP (VIP) addresses. In other words, the name of a given service can be used in replacement of its VIP in a configuration file for example and this reference will be replaced by the service discovery process once the VIP is known.

Nevertheless, when services are replicated, the service discovery alone is not sufficient. In that sense, the orchestrator provides a load balancer—a mechanism that allows to determine which replica to use. The Docker Swarm load balancer relies on the IP Virtual Servers (IPVS) protocol that already implements transport-layer load balancing as part of the Linux kernel. This protocol thus allows to redirect requests for TCP/UDP-based services to the containers. In a Swarm cluster specific case, every node listens on the exposed port of services that require to be remotely accessible from the outside (*e.g.*, a web dashboard listening on port `8080`). Then, IPVS uses a Round-Robin load balancing (Ghaffarinejad and Syrotiuk 2014) to forward the request to one of the active replicas composing the service. As illustrated in Fig. 8, if the requested node does not have a replica for the service, IPVS will route the request to an active

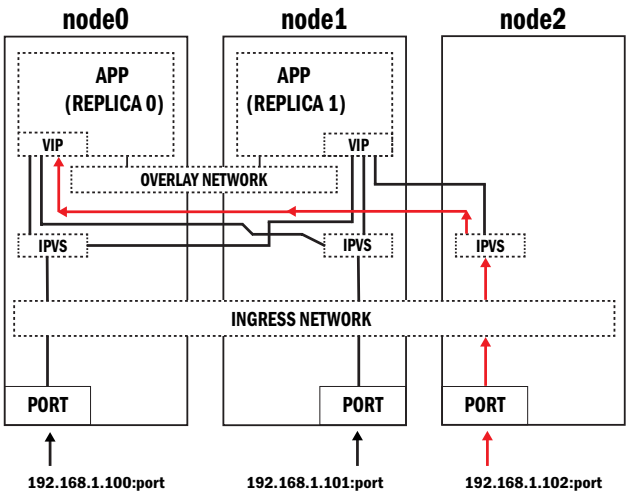

Fig. 8 Example of Docker Swarm orchestrator implementation of the IPVS Round-Robin load balancing that shows a possible route (identified by red arrows) of a request made on the `App` service which has two replicas placed on two separate nodes

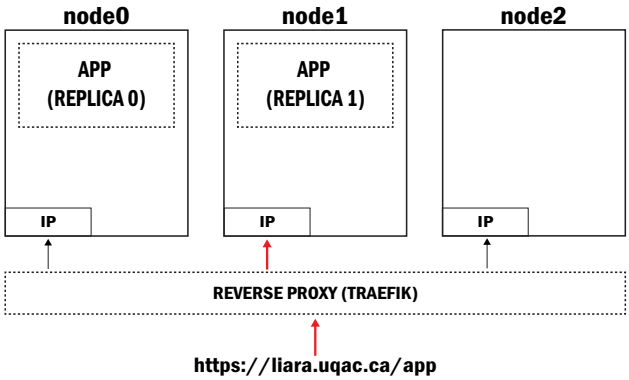

Fig. 9 Example of operation of the Træfik reverse proxy to route a client request made through a secure `https` URL in order to access the `App` service that has two replicas placed on two separate nodes

one present in another node, which is possible thanks to the ingress overlay network.

4.2.3 Reverse proxy

Since our architecture is composed of a five node cluster, all the services deployed on these nodes are accessible through five different entry points, thanks to the orchestrator that publishes the ports of these services across all nodes in the cluster. However, while it simplifies the access to such services, it may become hard to manage which service is published on which port. Moreover, as the orchestrator provides a load balancing at the service level, it appeared important for our architecture to include a node-level load balancing as well and a unified and secure external access to the internal services through a Uniform Resource Locator (URL) such as `https://app.domain.com`. For this purpose, the open-source tool Træfik[13] has been chosen over several other options such as NGINX[14] or HAProxy[15] because it is best suited for clustered architectures based on microservices. Indeed, the main advantage of Træfik is its capability to automatically discover information about the network and services available in the cluster and to dynamically update its configuration as the environment changes. These features are particularly important with microservices as they tend to be stateless and short-lived. Moreover, it is possible to deploy new versions frequently, and instances may be scaled dynamically as the demand increases.

Primarily, Træfik is considered to be a reverse proxy, a component which is the main entry point of a private network. These components are typically placed behind a firewall and they aim at routing client requests to the appropriate node or service. Moreover, reverse proxies can also fulfill a number of other important functions to improve efficiency and the security. First of all, Træfik provides a node-level load balancing that allows requests to be distributed among a pool of active nodes to prevent any of them from being saturated. In addition, reverse proxies such as Træfik, allow the anonymity of the internal architecture by providing an additional layer of obfuscation by never revealing the IP address of the node that actually handles requests. Finally, they enable the use of Transport Layer Security (TLS) encryption for clients using one or more certificates for different services or subdomains. Figure 9 illustrates an example of such a process. Indeed, the reverse proxy (*i.e.*, Træfik), provides a unified main entry point to access the `App` service through a secured HTTPS URL. Then, the internal load balancer of Træfik routes the client request to one of the active replicas of the `App` service. In that scenario, the Swarm service-level load balancer is not shown, but it still remains active. Indeed, while the node-level load balancer may route the request to `node 1`, it is possible for the service-level load balancer to internally route the request to the `replica 0` of the `App` service placed on `node 0`.

4.2.4 Cluster management

Since the Docker engine is a CLI tool, the initialization of the Swarm cluster defining our architecture may be achieved in several ways. Actually, the most appropriate way is to

[13] https://traefik.io/.

[14] https://www.nginx.com/.

[15] https://www.haproxy.org/.

use a configuration management tools such as Ansible[16] that enable automation for systems administration. However, as our architecture remains quite simple and small, we opted to initialize the cluster manually through various shell scripts and configuration files. Indeed, while automation tools offer a better management of highly complex systems administration workflows as well as a better global flexibility, shell scripts appeared to be more flexible as the proposed architecture is still deployed in an experimental environment.

In addition, after its initialization, the cluster and all its Docker containers still require to be managed. To complete such a task, the open-source software Portainer[17] has been deployed as part of the cluster. Through a nice and clean GUI, this tool allowed us to greatly simplify the cluster management task by abstracting the complexity of some management-related commands of the Docker engine and removing the risk of errors that may occur when using long command-line strings. A second reason for the choice of Portainer is its wide range of functionalities. The most important ones include the management of the images, networks and volumes of containers; a visualization of the nodes of our cluster with the placement of each container; the access to every log file for each container as well as the possibility to scale services by increasing or decreasing the number of replicas depending on the load for such services.

4.2.5 Database replication

As this architecture implementation was designed for smart homes, it was important to suggest a reliable way to store sensor values as well as actuator states. To this end, three services have been deployed on the cluster, each one containing an instance of the MongoDB[18] DataBase Management System (DBMS) in order to enable data replication with a three-member replica set architecture.

MongoDB is an open-source document-oriented and non-relational (NoSQL) DBMS used for high volume data storage. Such a choice has been made since its design is focused on delivering high performance, high availability, and automatic scaling. Since MongoDB is a document-oriented DBMS, data is stored inside collections as sets of documents. These documents refer to data structures composed of field and value pairs. They are formatted in BSON (Binary JavaScript Object Notation or Binary JSON), a file format similar to JSON where fields may include other documents, arrays, and arrays of documents. Such a semi-structured model is a database model where there is no separation between the data and the schema. The main advantage of the document-oriented approach first remains the possibility of representing complex hierarchical relationships with a single record. Moreover, such models also makes data repartition across multiple database nodes easier, thus simplifying the scaling process.

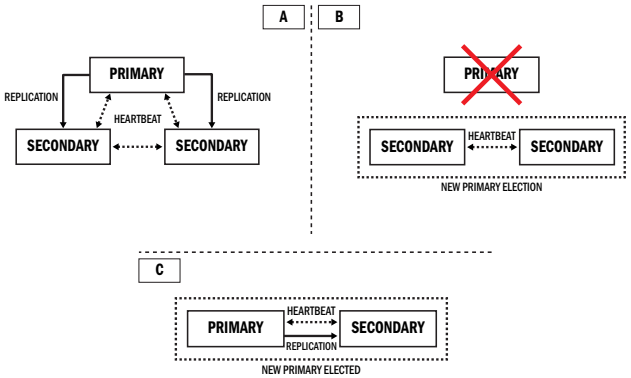

Fig. 10 The three-member replica set database architecture that shows the election of a new primary node when a complete failure of the first primary node occurs

MongoDB replica sets are a group of database instances that maintain the same data set providing redundancy and high availability in order to enable fault tolerance. To deploy a replica set, two different architecture options are possible. Firstly, replica sets with a replication cluster that contain a primary node and a minimum of two secondary nodes. Secondly, replica sets with arbiter that also involve a primary node, and secondary nodes but require an arbiter node as well. A primary node is the only member in a replica set that receives write operations while secondary nodes are in charge of maintaining a copy of the data set of the primary node. To replicate data, every secondary node asynchronously apply instructions from the operation log of the primary node to their own data set. Conversely, an arbiter node does not have a copy of the data set and it cannot become a primary node. However, an arbiter participates at electing a primary when required. In that sense, in the case of a three-member replica set, the replication cluster architecture implementation has been preferred since having an arbiter instead of two secondary nodes does not guarantee such a good load-balancing for read-only operations over the data.

Figure 10 illustrates the operation of our Primary with two Secondary members (P-S-S) replica set architecture. Every 2 s, all nodes send heartbeats to each other. If a heartbeat does not return within 10 s, the other servers mark the unresponsive server as inaccessible. When the primary server is unavailable, an election is triggered for one of the remaining secondary servers to become the new primary server. In the same way as for the Docker Swarm cluster orchestrator, the guideline to deploy a replica set is to have

[16] https://www.ansible.com/.

[17] https://www.portainer.io/.

[18] https://www.mongodb.com/.

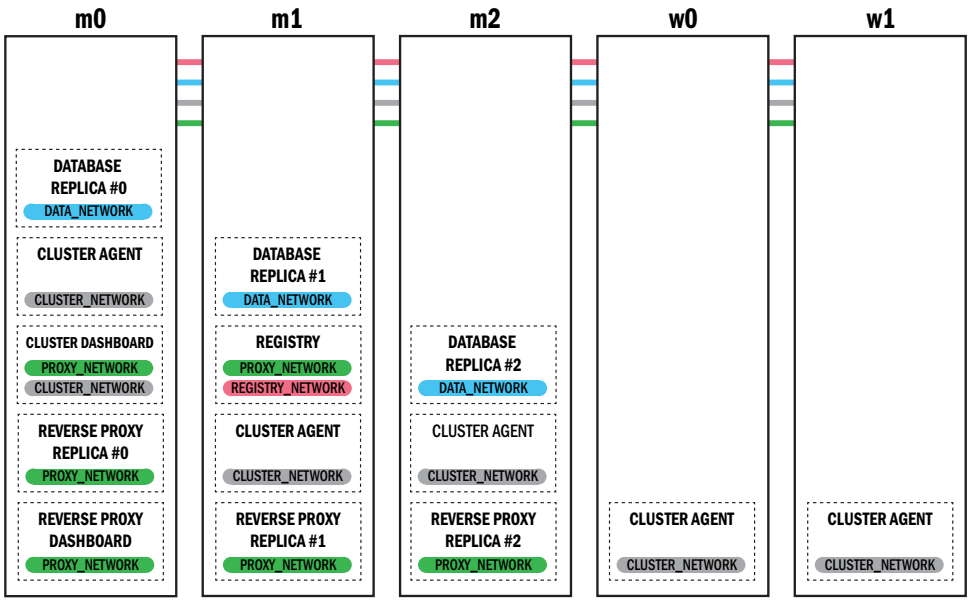

Fig. 11 Illustration of the placement of containers on the nodes composing the cluster that have been introduced in the description of the practical implementation of our architecture for smart homes

an odd number of nodes allowing, in our case, one complete node failure since the fault tolerance quorum is identical.

4.2.6 Container images *Registry*

The main benefit of using a microservices architecture, especially in experimental environments such as research laboratories where efforts are placed on developing smart homes, is to provide a simple means for deploying experimental software components running in containers. Hence, to facilitate the versioning, the distribution as well as the reuse of such components, a storage system for container images was deployed on the cluster.

Container images are immutable files that are composed of several layers containing the source code, libraries, dependencies and tools required for a container to run. In other words, they represent a service and its virtual environment at a specific point in time. Ultimately, a container is just a running image. In that sense, these images may be stored on a system called a *registry* where a repository holds all the versions of a specific image. Thanks to the *registry*, it is possible for the experimenters using our architecture to download container images (`pull`) and to publish (`push`) new images containing their software components. By default, the Docker engine embedded in each node of our clustered architecture is configured with a public *registry* (*i.e.*, DockerHub[19]). However, since experimental applications may contain vulnerabilities, confidential information or innovative algorithms, it was important to change the configuration of the engine to make it point to our private *registry*.

[19] https://hub.docker.com/.

4.2.7 Container organization

As a summary of the various aspects that describe the design of the proposed architecture introduced in this section, Fig. 11 provides a graphical representation of the placement of containers on the nodes composing the cluster. In addition, overlay networks for each container are also exposed.

The Træfik application is composed of two distinct services. The first one is the reverse proxy service that has three replicas deployed on manager nodes. This placement was preferred since it ensures a proper functioning of such an important piece of software. The second service is the web dashboard that allows the visualization of the current active routes and some relevant metrics. Since a failure of this service does not compromise the whole reliability of the architecture, it is deployed as a single container on a manager node. However, it has a restart policy set to always restart the container if it stops for whatever reason. In that sense, if a complete failure of the `m0` node occurs, the Træfik dashboard container will restart either on `m1` or `m2`. The same deployment instructions have been set for both the container *registry* and the cluster management dashboard (*i.e.*, Portainer) services for the same reasons.

In the same way, the MongoDB replica set was deployed across manager nodes to continuously provide a safe replication of the data and tolerate a maximum of one node failure. Finally, since the role of the Portainer application is to enable the possibility of managing the entire cluster, a global service (*i.e.*, `cluster_agent`) is required in order to allow the containerized dashboard to send requests to the docker engine of each node. As regards network communication, one overlay network has been created for each service and the reverse proxy dashboard is automatically

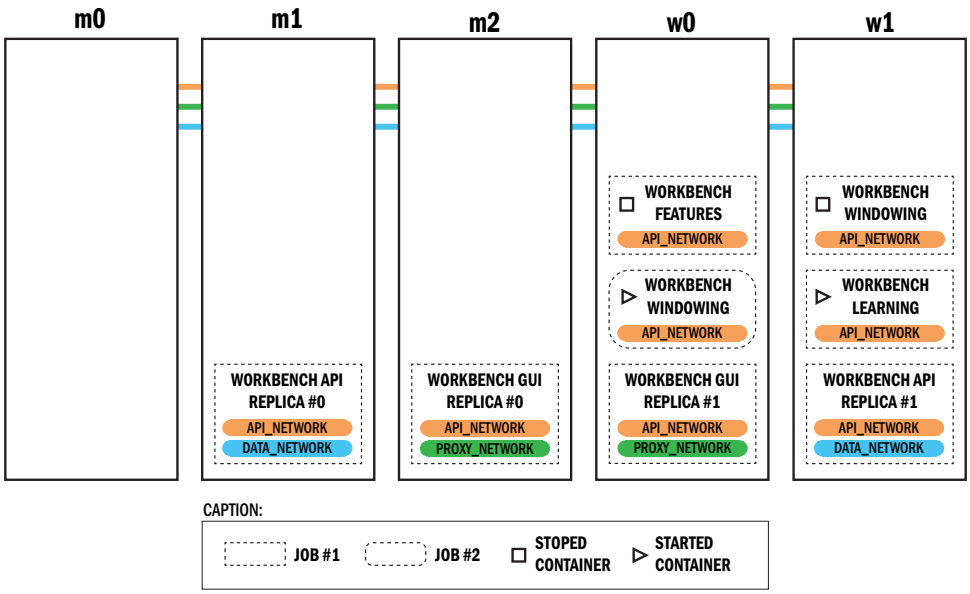

Fig. 12 An example of a possible deployment organization on each node of our clustered architecture for the workbench, related containers as well as their respective overlay networks

exposed. However, both the cluster management dashboard and the *registry* also require to be accessible from outside the cluster. In that sense, these services are connected to the `proxy_network` allowing the reverse proxy to handle the routing of client requests for such services accordingly.

5 The LE2ML workbench

The second contribution of this paper is LE2ML[20] (LIARA Environment for Modular Machine Learning), a new kind of machine learning workbench that deeply relies on microservices. While this workbench has been specifically designed to suit our architecture, it is perfectly possible for it to be run on more traditional environments such as a single main server in a monolithic smart home architecture or a simple desktop computer, as long as they have the Docker engine installed. Moreover, since LE2ML has been built with a strong focus placed on its modularity, the core component of its design architecture is a stateless RESTful API (REpresentational State Transfer API). This API is principally in charge of driving every other software components that are considered to be modules which anyone may develop and integrate to the core software based on the needs. Finally, as REST APIs are accessible through HTTP requests, a web application has also been developed in order to facilitate the interaction with the API. Figure 12 presents an example of a possible deployment. It illustrates the organization of the containers and their associated overlay networks that are required for the workbench to run on our clustered architecture. This figure aims at guiding our explanation for the rest of this section.

5.1 RESTful API

As the centerpiece of the workbench, the LE2ML RESTful API[21] has been developed in JavaScript using both Node.js[22] and Express.js.[23] These two open source tools offered respectively a powerful JavaScript runtime environment and a web application framework that allowed us to create a simple and modern REST API. Moreover, as shown on Fig. 12, the service of the API has been deployed as a service using two replicas that run either on manager or worker nodes. Obviously, this is the suggested base requirement that ensures the API remains accessible even if a node fails. However, it is possible to scale the service up when the API needs to handle numerous requests in order to avoid a slowdown of the entire workbench. In addition, while the API service provides its own overlay network (*i.e.*, `api_network`), each of its replicas also have to be attached to the `data_network` since some information required to be stored inside the replicated database. These include user and job related data; application keys as well as data sources, machine learning algorithms, windowing and feature functions that are available in the LE2ML modules.

The main objective of the LE2ML API is to execute several concurrent machines learning jobs thanks to the definition of machine learning pipelines. To this end, the API is

[20] https://github.com/FlorentinTh/LE2ML.

[21] https://github.com/FlorentinTh/LE2ML-API.

[22] https://nodejs.org/.

[23] https://expressjs.com/.

```
version: '1'
pipeline: machine_learning
source: inertial
process: train
model: model-202009222020
cross-validation: true
input:
  file:
    type: raw
    filename: soil_9_axis_imu_lsm9ds1.csv
windowing:
  enable: true
  parameters:
    length: 360Hz
    function:
      label: rectangular
      container: core-windowing
    overlap: 0
features:
  save: true
  filename: saved-features.csv
  list:
    ...
    - label: kurtosis_adjusted
      container: core-inertial-features
    ...
algorithm:
  name: k_nearest_neighbors
  container: core-py-sk
  parameters:
    search_algorithm: linear
    num_neighbors: 1
    distance: manhattan
```

Fig. 13 Example of a machine learning pipeline definition file that describes the training process of a traditional machine learning job to complete

in charge of orchestrating the launch of modules running inside containers in order to complete the execution of the pipeline and output some results. Moreover, it also provides several non-modular built-in features that are complementary to the process of machine learning. These include the import of data files; the management of either imported or newly created data files; data visualization; authentication and authorization management for users and applications; user and role management and module management.

The definition of a machine learning pipeline is made through a YAML (Yet Another Markup Language) file. This approach was chosen since such a file type remains easy to read and write. These files are also ideal for Version Control System (VCS) as well as simple to share between several experimenters in a research team. Once provided to the API, the pipeline definition is converted in a JSON format in order to be validated thanks to a JSON schema, then stored on the distributed file system. Next, the content of the file is parsed in order to start consecutively every required tasks composing the machine learning job to complete. An example of such a pipeline definition file is presented in Fig. 13.

At first, the user needs to specify the type of pipeline the job refers to. The possibilities are `machine_learning` or `deep_learning` to enable the computation of either traditional machine learning or deep learning techniques. The next mandatory property that has to be set is the `source`, referring to the type of input data to be computed (*e.g.*, vocal or inertial data). When a traditional machine learning pipeline is defined, the `process` property is also required and possible values are `train`, `test` or `none`. The `none` value was introduced in order to allow users to complete only some tasks that precede the learning process, such as feature extraction. As shown in our example, when the process is set to `train`, a `model` property must be provided in order to save the trained model in the file system for further use in other `test` process jobs. Moreover, it is also possible to enable a 10-fold cross-validation evaluation to output an estimation of the performance for the newly trained model.

Next, the input data file that is required to either train a model or make predictions needs to be defined beginning by the specification of the type of the input. Currently, only two options are allowed in the first version of definition files, being either a file or a WebSocket connection. In the case of a WebSocket input source, the URL of the server has to be indicated. Otherwise, files required to be referenced by both their filename and the type of data they contain since it is possible to use either raw data or computed features as input. Finally, the remaining content of the file defines the tasks and associated parameters that are required to complete the job.

To complete the execution of jobs created thanks to pipeline definitions, the API is responsible to launch the containers according to the provided module definition of each task. Hence, since some containers depend on the output of the preceding task, they are started one after the other as soon as their process is terminated. However, when a task is defined to use more than one container to be completed, containers are started at the same time so they may run in parallel. In addition, multiple jobs may be started concurrently, allowing the possibility for many identical containers to be started and run at the same time. As an example, Fig. 12 shows two jobs started at different moments. As regards the first job, it is possible to observe that both the windowing and the feature extraction processes are finished while the learning task is still running. Conversely, the windowing task for the second job is just started and the other containers are not visible because they have not been started yet. We have assumed that both jobs have been defined identically.

In order to make a container a module of LE2ML, three environment variables have to be set at its start. These are the unique job identifier, the unique user identifier as well as a unique token for each container. The token is employed to let the API make the link between a given container and the task it is attached to. Through this knowledge, it is

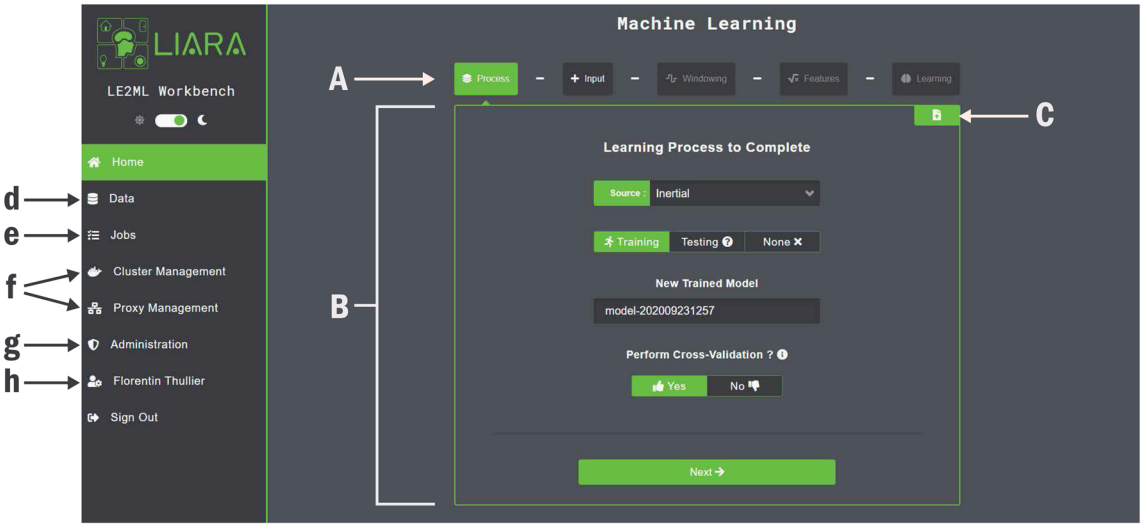

Fig. 14 Screenshot of the graphical user interface that shows one of the tasks to configure in order to create a definition of a traditional machine learning pipeline

possible to update appropriately the associated job entry in the database. Indeed, once terminated, a LE2ML module must communicate its state through either a success or an error endpoint to the API. To call these routes, the container requires to be authorized thanks to an application key generated beforehand.

Finally, as the execution of machine learning jobs involve the creation of a few files, results produced by each task of the pipeline that require to write data on the file system are isolated in separate folders. In that sense, the folder for a given job is mounted as a workspace volume inside each container defined for the tasks of the pipeline.

5.2 Web application

To facilitate the interaction with the API, we considered appropriate to provide an interface. A graphical user interface[24] was preferred over a CLI (the development of such an interface is not excluded in the near future). Indeed, since research teams may be multidisciplinary, offering a GUI has been prioritized, because it is easier to learn and use for neophyte users. This interface is a web application developed with modern technologies. It relies on a custom-made JavaScript front-end framework that supports ECMAScript 6 and higher versions of the standard (`ES6+`). In addition, the deployment strategy on the clustered architecture that has been adopted for such an application remains the same as for the API—a replicated service having two instances that may be placed either on a manager or a worker node.

Figure 14 shows a screenshot of the main view that constitutes the interface allowing to define a new pipeline to start a machine learning job. The breadcrumb (`A`) exposes to the users each step they have to go through in order to define all the tasks of the pipeline. The view that comes just before (not shown) offers the choice of the type of pipeline to define (*i.e.*, either a traditional machine learning or a deep learning pipeline). For each step, a set of web components (`B`) is offered to allow users to configure each task in a convenient way. While it is possible to build a pipeline definition file through the interaction with the GUI, a file import feature (`C`) is also provided and enables the reuse of existing pipeline definitions. In that case, all the components at each step are pre-filled according to the content of the uploaded file.

In addition, this interface also provides the users with a way to interact with every other features offered by the API. The "Data" menu (`d`) gathers all the features related to input data files. These include a file uploader; a file manager to navigate through the files hosted on the distributed file system; a data visualization tool as well as access to both data preprocessing and data fusion modules. The "Jobs" menu (`e`) offers a list of both started and completed machine learning jobs created by the current authenticated user. This list is updated in real time thanks to Server-Sent Events (SSE) which allow the server to send push notifications to our front-end application. Moreover, through this view, it is possible to download the files resulting from a completed job. These files may either be a report that includes relevant metrics computed during the training task, or the list of predictions determined through the testing phase. Then, the "Cluster Management" and "Proxy Management" menus (`f`) redirect to the Portainer and the Træfik dashboards respectively. The "Administration" menu (`g`) allows to administrator users to

[24] https://github.com/FlorentinTh/LE2ML-GUI.

manage modules, application keys and registered users of the workbench. Finally, authenticated users also have access to their profile (`h`) where it is possible for them to edit most of their information, including their password.

5.3 Provided modules

LE2ML is designed with a strong effort placed on its modularity. In that sense, it can accommodate several modules specifically built to achieve fine-grained tasks as part of different machine learning processes. The development of these modules has been made simple, so that it is possible, for anyone, to extend the functionalities of the workbench. Modules are able to handle data-related tasks such as data preprocessing or data fusion methods as well as feature extraction mechanisms. In addition, they are also capable of offering either traditional machine learning and deep learning algorithms as modules.

The first main advantage that motivates the use of modules is that they provide a simple way to cope with the diversity in terms of programming languages and implementations that form the vast world of machine learning related techniques. For example, researchers have the ability to develop a new machine learning algorithm to test its performances on a well-known dataset using the technology they feel most appropriate. Thus, they do not have to worry about the compatibility of their new implementation and they can benefit from the power of the tools (*e.g.*, libraries, frameworks) offered by a technology in particular. The second benefit of employing modules is that they significantly facilitate the management of both the scalability and the concurrent execution of many processes since they are containerized.

While such a modular design offers a wide range of possibilities, this section only describes three LE2ML modules provided for this work, as they were identified as the most important ones for our laboratory.

5.3.1 Windowing

Machine learning processes sometimes involve either signals or time series as input data. In that sense, it may appear relevant to apply a window function in order to split these raw data into smaller segments. Since several works from our laboratory have been introduced based on the use of different types of signals (Thullier et al. 2017; Chapron et al. 2018; Bouchard et al. 2020), we decided to provide a windowing module[25] as part of the LE2ML workbench. This module has been developed in `ES6+` JavaScript and it offers the most well-known window functions that are typically applied on such type of data (*i.e.*, rectangular, triangular, Hamming, Hann and Blackmann functions). Moreover, such a module also allows tuning several other parameters which the windowing process depends on. First of all, the length of data splits is mandatory. However, since some datasets may not include timestamps, it is possible for the length parameter to be expressed either in a frequency unit (*e.g.*, hertz), or in a unit of time (*e.g.*, seconds and minutes) for timestamped data. In addition, it is possible to set an overlap parameter expressed as a percentage of the window size. Finally, to ensure the fastest possible execution as well as its capability to handle gigabytes, or even terabytes of data without consuming a disproportionate amount of memory, our implementation of the windowing process mainly relies on both readable and writable streams provided by the Node.js environment.

5.3.2 Feature extraction

The feature extraction process remains an important task to achieve in order to complete a machine learning process. The objective of such a task is to compute a vector of discriminating characteristics based on each fragment of a raw data signal obtained from the previous windowing process. In that sense, since some research of our laboratory have employed Inertial Measurement Unit (IMU) sensors (Thullier et al. 2017, 2018; Chapron et al. 2018), the proposed module[26] introduced in this work has been implemented to provide the most widely used features for inertial data. It allows the calculation of 11 different features that came from both temporal and frequency domains, such as the *kurtosis*, the *skewness*, the *DC component*, the *energy* and the *entropy*. This module have also been developed in `ES6+` JavaScript. While JavaScript is not a strong typed programming language and appears not suitable to compute complex mathematical formulas, we have surprisingly observed a same level of precision and better performances as with a similar program written in Go.[27] In the same way as for the windowing module, the feature extraction module is memory efficient through stream-enabled file processing. Moreover, it also provides a certain level of flexibility since the transformation from the time domain to the frequency domain is made thanks to an implementation of Bluestein's FFT algorithm (Bluestein 1970).

The first version of the machine learning pipeline definition file allows to list a set of features to extract on input raw data (cf. ln. 22 of Fig. 13). This list may either be empty or populated with objects composed of both their respective labels and the name of the container (*i.e.*, module) to use for

[25] https://github.com/FlorentinTh/LE2ML-Windowing-Module.

[26] https://github.com/FlorentinTh/LE2ML-FeatureExtractor-Module.

[27] https://github.com/LIARALab/GoLang-FeatureExtractor.

Table 1 Summary of the experiments that were conducted on the architecture in order to determine its overall reliability regarding one-node failures

Scenario	Node	Type	Failure	Tasks restarted	Leader election	Cluster state	Recovery time
N_0	w0	Worker	Network	Yes	No	Available	$\approx 1\,s$
$\mathbf{E}_0$	w0	Worker	Electrical	Yes	No	Available	≈ 1 min
N_1	m2	Manager	Network	Yes	No	Available	≈ 1 s
E_1	m2	Manager	Electrical	Yes	No	Available	≈ 1 min
N_2	m0	Leader	Network	Yes	Yes	Available	≈ 1 s
E_2	m0	Leader	Electrical	Yes	Yes	Available	≈ 1 min

its computation. In the first case, the feature extraction task is skipped. Otherwise, such a definition enables to declare the use of multiple modules in order to complete this task. Thus, each module is responsible for the processing of their related features. They operate in parallel and they are independent of each other. Then, the task is stated complete once the last container has terminated its execution. As the state of the feature extraction task is updated in the database, a merge operation is triggered by the API. This operation then assembles each file produced by all the containers, which is finally supplied as input to the machine learning algorithm.

5.3.3 Machine learning

The machine learning module[28] distributed with this first version of the LE2ML workbench has been made to enable the use of traditional machine learning algorithms only. This module has been developed in Python 3 as its design offers a software overlay for the popular scikit-learn Python library. This module aims to translate automatically the parameters of the pipeline configuration file (cf. ln. 30 of Fig. 13) into scikit-learn API code. In that sense, it first allows training models thanks to several traditional machine learning algorithms that are provided by the library, among which are the `k-Nearest Neighbors` (`k-NN`) and the Random Forest algorithms. It is then possible to enable the evaluation of such trained models through a 10-fold cross validation method in order to output the confusion matrix as well as a few relevant metrics such as the accuracy, the `F-measure` and Cohen's kappa. Since the machine learning module offers to save trained models on the distributed file system, they may be applied to perform test processes to compute and output predictions. The training operation has been configured to automatically exploit all the available resources allocated to the container in order to execute this process in parallel.

6 Experiments

In order to evaluate and validate our distributed architecture, a preliminary set-up has been deployed in our laboratory. In addition, some experiments have been performed to confirm the capability of our system to be resilient to failures. Finally, based on a previous research that was conducted by our research team (Thullier et al. 2017), an entire machine learning job has been supplied to our workbench in order to outline its non-limiting capacities. In addition, the objective of such an experiment is to demonstrate the reproducibility of previously proposed recognition method without the need to write or adapt any code.

6.1 Materials

The implementation of our distributed architecture has been achieved by using open-source hardware in order to maximize its reproducibility. To this end, we chose to use of seven Raspberry Pi 3 Model B, a conventional wireless router (LinkSys WRT1900AC) running the OpenWRT[29] operating system as well as a simple network switch. The main advantage of using Raspberry Pi boards is that they offer an acceptable price–performance ratio. While their unit cost is US$35, they embed a 1.2 GHz quad-core ARMv8 processor and 1 GiB of SDRAM. Moreover, to keep such an experimental deployment as cheap as possible the storage was not done on RAID systems. Instead, two USB flash drives were connected to the two Raspberry Pi boards employed to implement the distributed file system. In addition, since this model of Raspberry Pi has a processor based on the ARMv8 architecture, the use of 64-bit operating systems was possible. In that sense, we opted for a specific version of a Debian distribution compatible with such an architecture rather than the Raspberry Pi OS,[30] which is still not currently available in a 64-bit version. The installation of a 64-bit system was mandatory. Indeed, as containers virtualize environments at the OS level, several tools that were

[28] https://github.com/FlorentinTh/LE2ML-Learning-Module.

[29] https://openwrt.org/.

[30] https://www.raspberrypi.org/downloads/raspberry-pi-os/.

Table 2 Summary of the experiments that were conducted on the architecture in order to determine its overall reliability regarding multiple node failures

Scenario	Nodes	Node types	Failure	Tasks restarted	Leader election	Cluster state	Recovery time
N_{10}	`m1 w0`	Manager worker	Network	yes	No	Available	≈ 1 s
E_{10}	`m1 w0`	Manager worker	Electrical	Yes	No	Available	≈ 1 min
N_{21}	`m0 m1`	leader manager	network	no	no	not Available	–
E_{21}	`m0 m1`	Leader manager	Electrical	No	No	Not available	–
E*	All	–	Electrical	No	No	Available	≈ 10 min

required in the design our architecture were only available for ARMv8 architectures such as the MongoDB database.

6.2 High availability

Following the initialization of the Docker Swarm cluster on each five nodes of the architecture, the four application stacks that describes the policy detailed in the Sect. 4.2.7 have been deployed. The distributed cluster architecture is then fully operational within minutes. This deployment policy has been specifically determined to handle a total failure of at most one node. This choice has been made considering that adding more nodes to the cluster will necessarily improve fault tolerance. However, having all of them on the same geographical localization will not provide a better reliability of the whole system and it will create a much more complex environment to manage and maintain. Hence, to confirm that our proposed architecture is reliable, it appeared relevant to perform some experiments. To this end, the application stack containing both the workbench API and GUI services has also been deployed.

First of all, in order to evaluate the reliability of the entire cluster architecture according to several failure scenarios for a single node, two types of failures have been simulated. These include network (`N`) and electrical (`E`) failures that may occur on either worker (`0`), manager (`1`) or leader (`2`) nodes. Therefore, a set of six failures was created and results obtained are summed up in Table 1. For each of these scenarios, we observed the expected operations to be performed by the cluster. The first operation is the ability for our achitecture to dispatch and restart automatically the tasks that were running on the failed node to the remaining operational nodes. The second operation is the election process of a new leader node. In addition, we also observed the general state of the cluster to determine whether it was functional despite the loss of a node. Finally, the last observation provided in this experiment concerns the recovery time required to return to the initial state.

Then, the loss of two nodes was also considered according to the same simulated failures as in the first experiment. Obtained results are exposed in Table 2. Through these results, a proper operation of the cluster has been observed in both scenarios N_{10} and E_{10}, even though we stated that

Table 3 Results obtained in the previous research (Thullier et al. 2018) for both `k-NN` and random forest machine learning algorithms applied to the recognition of different soil types

	Accuracy	`F`-measure	Cohen's kappa
`k-NN`	0.93	0.93	0.89
Random Forest	0.92	0.92	0.88

Table 4 Results obtained with our proposed workbench (LE2ML) for both `k-NN` and random forest machine learning algorithms applied to the recognition of different soil types

	Accuracy	`F`-measure	Cohen's kappa
`k-NN`	0.91	0.91	0.86
Random Forest	0.92	0.92	0.88

our architecture only tolerated a single node failure. This can be explained by the fact that the malfunction has been simulated for both a manager and a worker node. Indeed, reproducing these two simulated failures by involving both the leader and any single manager node (*i.e.* N_{21} and E_{21}) led to a compromised operation of the entire architecture. While services started on worker nodes continued to run, the ability for the cluster to perform management tasks including starting new services was lost. In that sense, since losing two nodes may corrupt the operation of the architecture in such a specific case it is safer to keep in mind that given our implementation, a one node failure remains the maximum tolerable limit. Finally, Table 2 also shows results obtained when a total power outage was simulated (`E*`). To do this, all the Raspberry Pi boards have been shut down using the breaker on the power bar to which they were plugged in. However, both the router and the network switch remained powered. Since the state of the cluster is preserved by the engine, it was able to self recover from this situation after every node was restarted and each manager was able to communicate again to each other. The observed full recovery period lasted several minutes.

To conclude, these two experiments have illustrated the global reliability of our proposed architecture, through its

ability to handle various cases of node failures. However, as electrical failures may require a longer time to recover the initial state of the cluster, we suggest that all the equipment employed in this implementation be electrically supplied through an Uninterruptible Power Source (UPS).

6.3 Workbench evaluation

This section presents an experiment that was conducted to show the versatility of the workbench for an academic use in smart homes. Thanks to LE2ML, we were able to reproduce an experiment introduced in a previous work conducted by our laboratory (Thullier et al. 2017). To do this, there was no rewritten or adapted code—only pipeline definition files and the raw data obtained from this previous work were required. Our previous research proposed a traditional machine learning method to allow the recognition of different soil types based on inertial data generated by a user's gait. To achieve this objective, inertial data have been obtained through a wearable device at 60 Hz. Then, several features related to the processing of inertial signals that came from both time and frequency domains were computed over a non-overlapping rectangular time window function of 6 s. Finally, two machine learning algorithms have been compared to achieve the recognition of soil types. These assessments were achieved thanks to WEKA that has been used as a library in a command line interface application. In order to comply with this proposed method, two distinct jobs were created via the workbench, each one corresponding to a specific definition for the two algorithms. The first file defines the `k-NN` algorithm tuned with $k = 1$ neighbor to search through a linear computation of the Manhattan distance. The second file describes the random forest algorithm tuned with $B = 300$ trees, $F = \lfloor \log_2(m) + 1 \rfloor$ random variables and the computation of the information gain. The evaluation of each trained model has been achieved through a 10-fold cross validation method and the results that were obtained (Thullier et al. 2018) are presented in Table 3.

The definition file of the pipeline used to complete such a recognition with the `k-NN` algorithm is exposed in Fig. 13. Results obtained for both algorithms using the LE2ML workbench are provided in Table 4. As regards such results, it is possible for us to state that our workbench offer the same level of precision than WEKA. Indeed, identical recognition performances have been observed for the three evaluation metrics when using the random forest algorithm. Although the resulting performance obtained with the `k-NN` algorithm using our workbench was less efficient, the difference is not significant enough. This is explainable by the variation in implementation between scikit-learn and WEKA. Nevertheless, despite a number of modules that are still limited, we believe that LE2ML shall prove an indispensable tool to promote reusability of software components between research laboratories as well as between different projects and their stakeholders.

7 Conclusions

In this paper, we have first introduced a novel architecture for smart homes that aims at being agnostic, reliable and scalable in opposition to existing architectures. The design of the proposed implementation relies on the concept of microservices which are supported by a distributed cluster of five nodes. The use of microservices improved both the interoperability between machine learning applications that are developed in various programming languages and their reusability. Moreover, such an implementation has been made compatible with the majority of existing smart home architectures, since it may easily replace the central computing unit (server or broker) that is present in their design. Therefore, it contributes to remove completely any remaining single point of failure in order to provide a higher level of reliability. Finally, the capacity to add more resources, both hardware and software, allows this architecture to offer a greater level of scalability. Through the experimentation that has been performed to evaluate the high availability of the proposed architecture, we showed that the proper functioning of the entire system was not affected by a complete failure of one node of the cluster. Moreover, such an experiment also revealed that the entire architecture was also resilient to a total power outage.

In addition, the second contribution introduced in this paper is a new kind of machine learning workbench named LIARA Environment for Modular Machine Learning (LE2ML). This workbench has been implemented to be fully compatible with our proposed architecture through a modular design. However, such a tool is also capable of working on more conventional monolithic architectures. Its implementation deeply relies on a RESTful API which is its core component. The API is responsible for the orchestration of the whole system and most particularly the modules. These modules are either provided with the platform or they may be specifically developed given any technology according to the needs. To interact with this API, a web interface has been specifically developed. This interface allows, among other things, to simplify the creation of pipeline definition files that describe the entire process of machine learning to be computed. Currently, all modules provided with the suggested workbench were mainly built to handle inertial data, except for the traditional machine learning module. Hence, the experiment that has been achieved to evaluate LE2ML was based on the replication of the method from our previously proposed research. This method presents a machine learning process to recognize different soil types based on

a wearable device which embed an IMU sensor. The results provided by this research have been compared with the ones obtained with our workbench and no significant difference was shown in the resulting overall performance.

8 Future works

Future works will focus first on performing a deployment of the proposed architecture in a more professional way inside our laboratory. To this end, more suitable and dedicated materials than the ones used in the presented deployment will be employed, since such a set-up was intended for experimental purpose only. In another time, our focus will be placed on designing the deep learning pipeline to integrate in the LE2ML workbench. Moreover, we expect to suggest more modules to the workbench in order to enable the support of more type of data; include several other processes such as data preprocessing and data fusion techniques as well as providing the ability to handle different types of input sources such as a WebSocket connection.

Acknowledgements The authors would like to thank the Canadian Foundation for Innovation (CFI) for providing the laboratory infrastructure and the equipment.

Declarations

Conflict of interest The authors declare no conflict of interest.

References

Bae IH, Kim HG (2011) An ontology-based approach to ADL recognition in smart homes. In: International Conference on future generation communication and networking, Springer, Jeju Island, Korea 266:3 71–380. https://doi.org/10.1007/978-3-642-27201-1_42

Bluestein LI (1970) A linear filtering approach to the computation of discrete Fourier transform. IEEE Trans Audio and Electroacoust 18(4):451–455. https://doi.org/10.1109/TAU.1970.1162132

Bouchard K, Bouchard B, Bouzouane A (2014) Practical guidelines to build smart homes: lessons learned. In: Opportunistic networking, smart home, smart city, smart systems. CRC Press, Taylor & Francis, pp 1–37

Bouchard K, Maitre J, Bertuglia C, Gaboury S (2020) Activity recognition in smart homes using uwb radars. Proc Comput Sci 170:10–17. https://doi.org/10.1016/j.procs.2020.03.004

Bouckaert RR, Frank E, Hall MA, Holmes G, Pfahringer B, Reutemann P, Witten IH (2010) WEKA—experiences with a java open-source project. J Mach Learn Res 11:2533–2541

Chapron K, Plantevin V, Thullier F, Bouchard K, Duchesne E, Gaboury S (2018) A more efficient transportable and scalable system for real-time activities and exercises recognition. Sensors. https://doi.org/10.3390/s18010268

Chen L, Nugent C (2010) Situation aware cognitive assistance in smart homes. J Mob Multimed 6(3):263–280

Chen L, Nugent C, Mulvenna M, Finlay D, Hong X (2009) Semantic smart homes: towards knowledge rich assisted living environments. In: McClean S, Millard P, El-Darzi E, Nugent C (eds) Intelligent patient management. Springer, Berlin, pp 279–296. https://doi.org/10.1007/978-3-642-00179-6_17

Chen L, Nugent C, Biswas J, Hoey J (2011) Activity recognition in pervasive intelligent environments. In: Atlantis ambient and pervasive intelligence. Atlantis Press. https://doi.org/10.2991/978-94-91216-05-3

Chen L, Hoey J, Nugent CD, Cook DJ, Yu Z (2012) Sensor-based activity recognition. IEEE Trans Syst Man Cybernet. https://doi.org/10.1109/TSMCC.2012.2198883

Chen M, Mao S, Liu Y (2014) Big data: A survey. Mob Netw Appl 19:171–209. https://doi.org/10.1007/s11036-013-0489-0

Cook D, Youngblood M, Heierman E, Gopalratnam K, Rao S, Litvin A, Khawaja F (2003) MavHome: an agent-based smart home. In: Proceedings of the First IEEE International Conference on pervasive computing and communications (PerCom), IEEE, Fort Worth, TX, USA, pp 521–524, https://doi.org/10.1109/PERCOM.2003.1192783

Cook DJ, Crandall AS, Thomas BL, Krishnan NC (2013) CASAS: a smart home in a box. Computer 46(7):62–69. https://doi.org/10.1109/MC.2012.328

Davis K, Owusu E, Bastani V, Marcenaro L, Hu J, Regazzoni C, Feijs L (2016) Activity recognition based on inertial sensors for ambient assisted living. In: FUSION 2016—19th International Conference on information fusion, proceedings, pp 371–378

Demšar J, Zupan B, Leban G, Curk T (2004) Orange: from experimental machine learning to interactive data mining. Lecture Notes Comput Sci (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 3202:537–539. https://doi.org/10.1007/978-3-540-30116-5_58

Demšar J, Curk T, Erjavec A, Gorup Č, Hočevar T, Milutinovič M, Možina M, Polajnar M, Toplak M, Starič A, Štajdohar M, Umek L, Žagar L, Žbontar J, Žitnik M, Zupan B (2013) Orange: data mining toolbox in python. J Mach Learn Res 14(1):2349–2353. https://doi.org/10.5555/2567709.2567736

Dikaiakos MD, Katsaros D, Mehra P, Pallis G, Vakali A (2009) Cloud computing: distributed internet computing for IT and scientific research. IEEE Internet Comput 13(5):10–11. https://doi.org/10.1109/MIC.2009.103

Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L (2017) Microservices: yesterday, today, and tomorrow. In: Mazzara M, Meyer B (eds) Present and ulterior software engineering. Springer, Berlin, pp 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

Dua D, Graff C (2017) UCI machine learning repository. http://archive.ics.uci.edu/ml. Accessed 17 Nov 2020

Fortin-Simard D, Bilodeau JS, Bouchard K, Gaboury S, Bouchard B, Bouzouane A (2015) Exploiting passive RFID technology for activity recognition in smart homes. IEEE Intell Syst 30(4):7–15. https://doi.org/10.1109/MIS.2015.18

Ghaffarinejad A, Syrotiuk VR (2014) Load balancing in a campus network using software defined networking. In: Proceedings—2014 3rd GENI Research and Educational Experiment Workshop, GREE 2014, IEEE, Atlanta, GA, USA, pp 75–76, https://doi.org/10.1109/GREE.2014.9

Giroux S, Leblanc T, Bouzouane A, Bouchard B, Pigot H, Bauchet J (2009) The praxis of cognitive assistance in smart homes. BMI Book, Ormond Beach, pp 183–211. https://doi.org/10.3233/978-1-60750-048-3-183

Gubbi J, Buyya R, Marusic S, Palaniswami M (2013) Internet of Things (IoT): a vision, architectural elements, and future directions. Futur Gener Comput Syst 29(7):1645–1660. https://doi.org/10.1016/j.future.2013.01.010

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA data mining software: an update. ACM SIGKDD Explor Newslett 11(1):10–18. https://doi.org/10.1145/1656274.1656278

Handa A, Sharma A, Shukla SK (2019) Machine learning in cybersecurity: a review. Wiley Interdiscipl Rev Data Min Knowl Discov. https://doi.org/10.1002/widm.1306

Helal S, Mann W, El-Zabadani H, King J, Kaddoura Y, Jansen E (2005) The Gator tech smart house: a programmable pervasive space. Computer 38(3):50–60. https://doi.org/10.1109/MC.2005.107

Hofmann M, Klinkenberg R (2014) RapidMiner: data mining use cases and business analytics applications. CRC Press, Taylor & Francis, Boca Raton

Holmes G, Donkin A, Witten IH (1994) WEKA: a machine learning workbench. In: Australian and New Zealand Conference on intelligent information systems—proceedings, pp 357–361, https://doi.org/10.1109/anziis.1994.396988

Hornik K, Buchta C, Zeileis A (2009) Open-source machine learning: R meets Weka. Comput Stat 24(2):225–232. https://doi.org/10.1007/s00180-008-0119-7

Hu P, Ning H, Chen L, Daneshmand M (2019) An open internet of things system architecture based on software-defined device. IEEE Internet Things J 6(2):2583–2592. https://doi.org/10.1109/JIOT.2018.2872028

Institute of Electrical and Electronics Engineers (1999) IEEE Std 1451.1-1999, IEEE Standard for a smart transducer interface for sensors and actuators—network capable application processor (NCAP) information model. In: IEEE, https://doi.org/10.1109/IEEESTD.2000.91313

Jafarnejad Ghomi E, Masoud Rahmani A, Nasih Qader N (2017) Load-balancing algorithms in cloud computing: a survey. J Netw Comput Appl 88:50–71. https://doi.org/10.1016/j.jnca.2017.04.007

Lago P, Lang F, Roncancio C, Jiménez-Guarín C, Mateescu R, Bonnefond N (2017) The contextact@a4h real-life dataset of daily-living activities. In: Brézillon P, Turner R, Penco C (eds) Modeling and using context, vol 10257. Springer, Berlin, pp 175–188. https://doi.org/10.1007/978-3-319-57837-8_14

Langlois RE, Lu H (2008) Intelligible machine learning with malibu. In: Proceedings of the 30th Annual International Conference of the IEEE engineering in medicine and biology society, EMBS'08—"Personalized Healthcare through Technology", pp 3795–3798, https://doi.org/10.1109/iembs.2008.4650035

Larrañaga P, Calvo B, Santana R, Bielza C, Galdiano J, Inza I, Lozano JA, Armañanzas R, Santafé G, Pérez A, Robles V (2006) Machine learning in bioinformatics. Brief Bioinform 7(1):86–112. https://doi.org/10.1093/bib/bbk007

MacKenzie CM, Laskey K, McCabe F, Brown PF, Metz R (2006) Reference model for service oriented architecture 1.0. OASIS Standard. OASIS Open 12:1–31

Mahalingam M, Dutt DG, Duda K, Agarwal P, Kreeger L, Sridhar T, Bursell M, Wright C (2014) Virtual eXtensible Local Area Network (VXLAN): a framework for overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, RFC Editor, https://www.rfc-editor.org/rfc/rfc7348.txt

Marikyan D, Papagiannidis S, Alamanos E (2019) A systematic review of the smart home literature: a user perspective. Technol Forecast Soc Change 138:139–154. https://doi.org/10.1016/j.techfore.2018.08.015

Ongaro D, Ousterhout J (2014) In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, pp 305–319

Plantevin V, Bouzouane A, Gaboury S (2017) The light node communication framework: a new way to communicate inside smart homes. Sensors 17(10):2397–2416. https://doi.org/10.3390/s17102397

Plantevin V, Bouzouane A, Bouchard B, Gaboury S (2019) Towards a more reliable and scalable architecture for smart home environments. J Ambient Intell Humaniz Comput 10(7):2645–2656. https://doi.org/10.1007/s12652-018-0954-5

Rajkomar A, Dean J, Kohane I (2019) Machine learning in medicine. N Engl J Med 380(14):1347–1358. https://doi.org/10.1056/NEJMra1814259

Ramasamy Ramamurthy S, Roy N (2018) Recent trends in machine learning for human activity recognition—a survey. Wiley Interdiscipl Rev Data Min Knowl Discov 8:4. https://doi.org/10.1002/widm.1254

Ritthoo O, Klinkenberg R, Fischer S, Mierswa I, Felske S (2003) Yale: yet another learning environment. Tech. rep., Universität Dortmund, Dortmund, Germany, https://doi.org/10.17877/DE290R-15309

Salowey J, Choudhury A, McGrew D (2008) AES galois counter mode (GCM) cipher suites for TLS. RFC 5288, RFC Editor, https://www.rfc-editor.org/rfc/rfc5288.txt. Accessed 17 Nov 2020

Thullier F, Plantevin V, Bouzouane A, Halle S, Gaboury S (2017) A position-independent method for soil types recognition using inertial data from a wearable device. In: 2017 IEEE SmartWorld, ubiquitous intelligence & computing, advanced & trusted computed, scalable computing & communications, cloud & big data computing, internet of people and smart city innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), IEEE, San Francisco, CA, USA, pp 1–10, https://doi.org/10.1109/UIC-ATC.2017.8397511

Thullier F, Plantevin V, Bouzouane A, Halle S, Gaboury S (2018) A comparison of inertial data acquisition methods for a position-independent soil types recognition. In: 2018 IEEE SmartWorld, ubiquitous intelligence & computing, advanced & trusted computing, scalable computing & communications, cloud & big data computing, internet of people and smart city innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), IEEE, Guangzhou, China, pp 1052–1056, https://doi.org/10.1109/SmartWorld.2018.00183

Triboan D, Chen L, Chen F, Wang Z (2016) Towards a service-oriented architecture for a mobile assistive system with real-time environmental sensing. Tsinghua Sci Technol 21(6):581–597. https://doi.org/10.1109/TST.2016.7787002

Valiente-Rocha PA, Lozano-Tello A (2010) Ontology-based expert system for home automation controlling. In: International Conference on industrial, engineering and other applications of applied intelligent systems, Springer, Córdoba, Spain 6096:661–670. https://doi.org/10.1007/978-3-642-13022-9_66

Wang J, Chen Y, Hao S, Peng X, Hu L (2019) Deep learning for sensor-based activity recognition: a survey. Pattern Recognit Lett 119:3–11. https://doi.org/10.1016/j.patrec.2018.02.010

Witten IH, Frank E, Hall MA, Pal CJ (2016) Data mining: practical machine learning tools and techniques, 4th edn. Morgan Kaufmann Publishers Inc., Burlington. https://doi.org/10.1016/c2009-0-19715-5

Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: ACM (ed) 2nd USENIX Workshop on hot topics in cloud computing, HotCloud 2010, USENIX Association, Boston, MA, pp 1–10, https://doi.org/10.5555/1863103.1863113

Zhang S, Wei Z, Nie J, Huang L, Wang S, Li Z (2017) A review on human activity recognition using vision-based method. J Healthc Eng. https://doi.org/10.1155/2017/3090343