ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

Dottorato di Ricerca in COMPUTER SCIENCE and ENGINEERING
XXX Ciclo

Settore Concorsuale: 09/H1 - Sistemi di elaborazione delle informazioni
Settore Scientifico Disciplinare: ING-INF/05

Middleware Solutions for Effective Cloud-CPS Integration in Pervasive Environment

Candidato:
Alessandro Zanni

Supervisore:
Prof. Paolo Bellavista

Coordinatore Dottorato:
Prof. Paolo Ciaccia

Esame finale anno 2018

website:
http://middleware.unibo.it/people/azanni

e-mail:
alessandro.zanni3@unibo.it

It's the journey, not the destination, that matters.

Contents

List of Figures

List of Tables

Abstract

The proliferation of a wide range of highly different mobile devices, from tiny sensors to powerful smartphone, with increasing connection abilities, has led to the modification of the way we interact with the surrounded environment and the introduction of several different applications in many sectors, such as energy, transportation, logistics, health-care, and so on. Several recent projects and research activities are addressing the challenging topic related to the creation of innovative mobile devices applications, leveraging the technical and economic advantages promoted by the adoption of cloud-hosted virtualized resources, i.e. cloud computing, to consolidate their service infrastructures and for elastic storage and processing infrastructure. At the same time, it is growing the manifest need of intermediate middleware solutions that can effectively integrate device localities with the global cloud resources, overcoming the issues related to their direct connection. Such relevant attention is also demonstrated by the emergence of interesting IoT-cloud platforms from industry and open-source communities, as well as by the flourishing research area of fog/edge computing, where decentralized virtual resources at edge nodes can support enhanced scalability and reduced latency via locality-based optimizations.

In this thesis work, the primary objective is to present some promising and feasible real-world solutions, trying to face and cover many different challenges and open-points of the mobile devices applications. The solutions proposed are applied at different levels of the stack, thus dividing them in relation to their internal architectures, in order to underline the intrinsic characteristics associated with the architectural solution, the requirements mainly stressed, and highlighting the most suitable scenarios they can work with. This thesis work aims to push forward the research in the field, mainly based on theoretical architecture and methodological approaches so far, introducing some industrially-relevant implementations that specifically target the issues of practical feasibility, cost-effectiveness, and efficiency of middleware solutions over easily-deployable environments. Towards this objective, the solutions are also grouped in relation to the specific applications to face, in order to promote a faster and a more correct adoption in real-world scenarios.

The described solutions are specifically designed for the support of mobile services, also in hostile environments, with the main requirements to provide and greatly increase mobile devices requirements, such as scalability, interoperability, performances, consumption-resources, reactivity via local control decisions and actuation. They show

how to efficiently tackle challenges introduced by the high amount of mobile devices, by originally extending fog computing, edge computing or cloud computing platform, in relation to the system to realize and the mobile devices used. The designs, implementations, and experimental works demonstrate the suitability of the proposed solutions to address several different open points and challenges of mobile devices applications in an efficient and effective way and, also, with applications in large-scale scenarios.

Finally, as notable side effect of the present work, in order to be able to propose really promising and useful solutions that cover lacks of current systems, the present work present a complete overview of the very recent literature about the intermediate middleware that are emerging. In fact, this thesis propose original architectures and implementations of solutions, by starting from lessons learned from the few existing experiences and by shedding new light on specific sub-fields of research where advancements are needed to effectively integrate huge numbers of geographically dispersed mobile devices and globally available cloud resources.

1 | CPS Relevance and Motivations

The recent years have undoubtedly experienced the development of an unprecedentedly huge number of devices with a very wide range of capabilities, from tiny and small-capable devices to much more powerful newer generation of smartphone. The Internet of Things (IoT) concept has introduced a great number of devices, increasingly more sophisticated and capable of differentiated forms of connectivity, that increasing the ubiquity of the applications and leads to a highly distributed network of devices communicating with human beings, other devices or systems. Along with IoT, we have also assisted to the great diffusion and social penetration of a wide range of more powerful devices, i.e. PDA and smartphone, that still offer multiple connectivity abilities and an increasingly set of functionalities and computation capabilities.

1.1 Research Challenges and Methodology

Research trends analysis estimates an exponential growth of the number of devices and connections in the next years. Cisco projects the mobile devices growth by 2020, expecting for that date up to 50 billion connected devices, with a number of machine-to-machine (M2M) device connections of 12.2 billion [1]. By the 2021, the IP traffic will be mainly generated by mobile devices, with an exponential increase of the annual global IP traffic to about 3.3 ZettaByte. Moreover, [2] extends the previous estimations, expecting 1 trillion of devices connected and 100 million of M2M device connections estimates by 2025, with a related economic impact up to more than $10 trillion.

The availability and the widespread use of such a high number of mobile devices, today equipped with multiple wireless communication interfaces, is paving the way to the all-the-time everywhere connectivity view of pervasive computing. Pervasive and ubiquitous computing is becoming the next logical step to mobile computing where services are accessed in an anytime-anywhere fashion and deployed on various kinds of mobile and stationary devices and offer a certain functionality to nearby users, changing the way we interact with the physical world, as well as the vision of established business models, with a potentially massive economic impact. Pervasive computing scenarios generally refer to an environment densely composed of ICT-enabled sensorial capabilities, to be exploited for the provisioning of context-aware, adaptable, and customizable services for better interacting with the surrounding environment. In order to build effective pervasive services, increasingly mobile devices are immersed into the environment. Many cities already have sensor networks for environmental monitoring, as well as camera or microphones for security purposes; RFID-based readers and badges can keep track of user movements and activities; smartphones are embedded with many different sensors and actuators, e.g., gyroscope, compass, accelerometer, proximity sensors, and gps localization tool.

In order to deal with sensors/actuators ubiquity, it is important to deeply understand and identify the most important challenges and open points the system needs to deal with and also to isolate the system core functionalities and production phases with the communications, data mangement, data logic, and data analytics of the system. Thus, by starting to identify the most key and primary requirements allows to define the constraints and the most appropriate improvement areas on the specific application and on similar application domains. Successivelly, by accuratelly analyzing the state of the art of the current most promising technologies on the field, it is possibile to relevantly improve widespread applications with all the massive advantages related to the sensors/actuators usage. In the following paragraphs, some relevant IoT-based systems are presented, highlighting the main challenges and issues for each specific application. Several solutions are proposed in the following sections.

1.2 CPS Overview

The diffusion of wireless technologies is identifying new scenarios of service provisioning where mobile users are willing to have ubiquitous and continuous access to both traditional and novel context-aware Internet services while they move in smart spaces. WiFi and wireless technology and sensor network technologies has been continuously evolving, i.e. 5G, trying to accommodate higher demands, speed and security, also even under very challenging conditions. The growing and increasingly powerful presence of wifi and wireless access will foster the spread of mobile devices, towards more effective and more efficient applications, in which information and intelligent services are invisibly embedded in the environment around us.

These latest advancements also lead to the diffusion of the Cyber Physical System (CPS) concept. A CPS is a system, with integrated computational and physical capabilities, that can interact with both the cyber and the physical world in which computing elements are used to coordinate and communicate with sensors, able to monitor cyber/physical indicators, and actuators, which can modify the cyber/physical environment where they execute [3, 4], as shown in Figure 1.1.

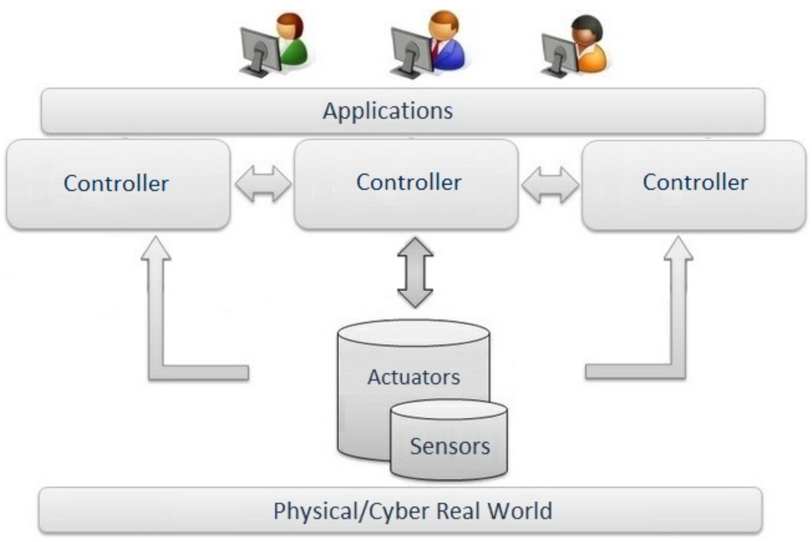


Figure 1.1: CPS General Architecture

This intimate coupling between the cyber and physical will be manifested from the nano-world to large-scale wide-area systems of systems. Similarly to how the internet transformed how humans interact and communicate with one another, e.g. revolutionized how and where information is accessed, and even changed how people buy and sell products, CPS will transform how humans interact with and control the cyber and physical world around us [5]. Computing and communication capabilities will soon be embedded in all types of objects and structures in the physical environment and the related applications, with enormous societal impact and economic benefit, will be created by harnessing these capabilities across both space and time [6].

In general, a CPS consists of two main functional components: the advanced connectivity that ensures real-time data acquisition from the physical world and information feedback from the cyber space; and intelligent data management, analytics and computational capability that constructs the cyber space [7]. Acquiring accurate and reliable data from machines and their components is the first step in developing a CPS application. The data might be directly measured by sensors or obtained from the controller, thus, is very important to manage data acquisition procedure and transferring data to a central server, and to select the proper sensors in terms of type and specifications [7]. The controller acts as central information hub where information is being pushed to it from every connected machine to form the machines network. Having massive information gathered, specific analytics have to be used to extract additional information that provide better insight over the status of individual machines among the fleet.

An emerging and technologically challenging idea is to consider Smart Cities as the most notable example of wide-scale CPS systems, where ICT solutions can work on both sensing cyber and physical indicators about the complex urban environment and favoring actuating actions that can dynamically change Smart City elements and their characteristics at provisioning time. Smart Cities have gained a central position and big hype from governmental institutions, organizations, and ICT industries, as they have the potential to increase resource consumption efficiency, improve many aspects of urban life (utilization of energy, environment, services, infrastructures, security, etc.), therefore having considerable impact on citizens quality of life.

1.3 CPS Applications towards Smart Cities

The fields of application for CPS are as numerous as they are diverse and they are increasingly extending to virtually all areas of everyday life. There are countless possible application domains where the CPS concept adoption can generate significant benefits. The most prominent areas of application include, but it is not limited to, the systems explained in the following. Several other applications belonging to various sectors, such as tele-healthcare and wellbeing, logistic operations, and many others, which are not discussed in this document, are becoming much more efficient and dynamic thanks to the increasing adoption of CPS and the use of mobile devices as key part to communicate with the surrounding environment.

1.3.1 ParticipAct Project

In the area of smart cities and smart connected communities, ParticipAct [8] is a socio/technical-aware crowdsensing platform based on a large-scale and real-world scenario, where the participation of users consists in completing collective tasks assigned them by administrators. ParticipAct is the first worldwide experiment on large scale of participation. Participants, either directly or indirectly, make data collection possible and then all data retrieved is processed and shared with other participants and researchers. The tasks cover many areas of interest and it is also possible to create group tasks that require the participation of a team of users. ParticipAct has some purposes which can be listed in four main categories:

- Quantified Self. A group of process, which elaborate data of a single person for self-monitoring activities (e.g., how much time has been spent walking).
- Information for people. A group of analytical techniques, which exploit technical and social information to control technical parameters (e.g., connections optimization thanks to the knowledge of citizens movement patterns).
- Eco-feedback. A set of analytical techniques which aims to reduce environmental impact using territory information.
- City planning. Making decisions using data retrieved thanks to users participating to the project (e.g., to know places with major social activities, students favorite jogging and cycling roots, and so on).

All data are made available by users who collaborate to the project completing tasks, which they are assigned to. A task is a job assigned to users and contains a title and a description. Tasks are classified active, if participant are involved to accomplish them, and passive, if are performed automatically by users mobile phone, e.g. triggered by geo-localization of the user position. Taking pictures, using tag, commit actions, answering a survey, and so on, are considered active tasks, whereas checking battery level, GPS localization, network identification, and so on, are considered passive tasks. Once the tasks are completed, either active or passive, all information belonging to them is sent to the ParticipAct platform to be aggregated and processed in order to obtain results on the area of interest and then used to update the profile of the user they belong to. Figure 1.2 shows the ParticipAct crowdsensing model.

This spontaneous, widespread diffusion of Internet-connected sensor-equipped devices has enabled to accurately trace world-related information and physical activities of citizens by taking advantage of people willing to collaborate toward a continuous data harvesting process, i.e. crowdsensing. That is especially true in smart cities areas where people bring almost constantly their smartphones. The crowdsensing perspective asks for a powerful sensing platform where smartphones act as data sources sparse over the city and continuously feeding fresh raw sensing data.

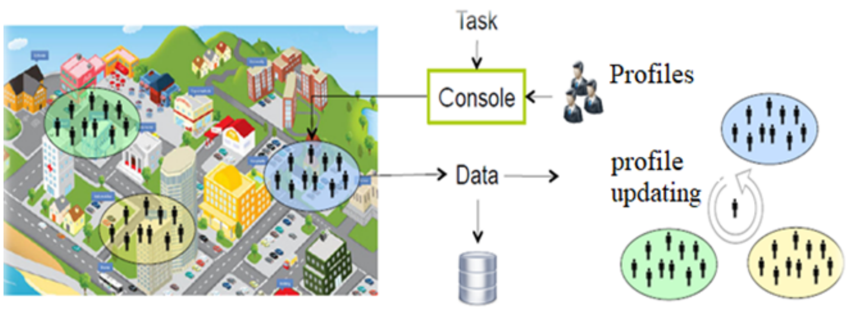


Figure 1.2: ParticipAct Overview

1.3.2 Smart Grid

Transport, communications, finance, and many other critical infrastructures depend on secure, reliable electricity supplies for energy and control. Those infrastructure are nowadays highly interconnected and a change in conditions at any one location can have immediate impacts over a wide area, with large-scale cascade failures, and the effect of a local disturbance can be magnified as it propagates through a network. Thus, traditional grids are facing many challenges that it was not designed to manage, such as congestion and atypical power flows threaten to overwhelm the system while demand increases for higher reliability and better security and protection [9].

Smart grid is envisioned to transform current grid to more intelligent one, through the usage of the available ICT-technology, to facilitate many aspects of the power supply, for both clients and providers. Smart grid aim to increase the context awareness towards a maximization of the utilization for an efficiency enhancement and a higher quality of service (free of voltage sags and spikes as well as other disturbances and interruptions) in a more controlled and secure operational environment, i.e., improving reliability and resiliency against malicious attacks, component failures, and natural disasters with autonomous control actions [10].

Smart grid also allow more strict requirements to be able to interact with the market, improving the real-time communication between consumer and utility so end-users can actively participate and tailor their energy consumption based on individual preferences (price, environmental concerns, etc.). In this way, the market efficiency improves through innovative solutions for product types (energy, services) available to market participants of all types and sizes, with benefits for both users and network operators. Users can earn or save money and generate electricity when prices are high and consume electricity when prices are low. Network operators can maintain grid stability, decrease the required capacity while improving the efficiency of power plants, and improve the service reliability mitigating peak demand and load variability.

Smart grid can be applied to different contexts with different size, with different methodologies optimized for the specific scope, such as [11]: i) Local scope (within a house): import/export into the grid optimized without cooperation with other houses, shifting electricity demand to more beneficial periods (e.g. nights) and peak shaving, towards independent house. ii) Microgrid (neighborhood). Optimization of combined import/export into the grid, shifting loads/shaving peaks, in order to better matched internally, towards perfect matching within the microgrid. It allows higher joint optimization potential and less dynamic load profile (e.g. peaks disappear in combined load) and multiple microgenerators match more demand than individual since better distribution in time of the production but with complex optimization methodology. iii) Virtual Power Plant manage large microgenerators group, replacing power plant with higher efficiency and much more flexibility (usability to react on fluctuations). It requires a complex optimization methodology, communication with individual house, with privacy and acceptance issues required.

Smart grid integrates advanced sensing technologies, control methods, and integrated communications into the current electricity grid. In smart grid, there are three main components that interact among them, each one composed of several sub-systems and of a large set of different devices. i) A smart infrastructure that include energy, information and a communication infrastructure that supports: advanced electricity generation, delivery, consumption; advanced information metering, monitoring, and management; advanced communication technologies. Within the smart infrastructure there are other subsystems, such as: smart energy, responsible for advanced electricity generation, delivery, and consumption; smart information, responsible for advanced information metering, monitoring, management; smart communication, responsible for communication connectivity and information transmission among systems, devices, applications. ii) Smart management that provides advanced management and control services. It enables a high number of functionality based on smart infrastructure to pursue various advanced management objectives, related to energy efficiency improvement, supply and demand balance, emission control, operation cost reduction, and utility maximization. iii) Smart protection provides advanced grid reliability analysis, failure protection, security and privacy protection services, to effectively and efficiently support failure protection mechanisms, cyber security issues, privacy.

1.3.3 Smart Building

Buildings are one of the largest consumers of electricity; the US Department of Energy estimates that buildings consume 70% of the electricity in the US [12] and emit approximately 40% of greenhouse gases [13]. Heating, cooling and ventilation accounts for 35% energy usage in the US and, currently, most modern buildings still condition rooms assuming maximum occupancy rather than actual usage, thus, often over-conditioned needlessly [14]. The energy usage in a building can typically be divided amongst several subsystems, including lighting, computing, server rooms and mechanical equipment used for climate control, with a significant amount of energy consumption [13]. Some typical smart building use-cases are the following. i) Energy consumption visibility: without any automation and closed-loop control, savings can be achieved by making visible the consumption patterns (current, historical), visualizing information (e.g. status, energy consumption, emission, historical trends, etc.): owners and operators can decide to optimize building operations at coarse level, while usage patterns analysis of office occupants encourage efficient practices at a fine grain level. ii) Integrated building operation: building systems are integrated, information are exchanged and there are closed loop controls. For instance: security system uses badge-in/out information to calculate the current occupancy; in an integrated building system, occupancy info is used by the control system for optimal amount of cooling/heating or to adjust the number of active elevators to balance the wait time and energy consumption by the elevator control system. iii) Demand response: energy consumption on the grid peaks/troughs in daily/weekly/seasonal cycles, wasting energy during non-peak hours. Intelligent grid must smooth out the peak of energy consumption and thus improve the overall capacity utilization. A common mechanism is to elicit demand elasticity using economic means (e.g. dynamical pricing) reducing energy peak demand by shifting non-essential consuming activities into low-demand time periods. iv) Occupant-aware building control: while reducing energy consumption and improving overall efficiency, highly advanced smart building control system considers users and uses occupants environmental preferences to perform fine-grained control and actuation of building systems (saving with minimal adverse impact on workers productivity/satisfaction).

[15] provides a reference architecture for a smart building application, shown in Figure 1.3.

Physical System contains the interface services to various sensors/actuators systems, e.g., Heating Ventilation Air Conditioning (HVAC) system, lighting control system, building fire and security system, and real time location system, linked to a local high-speed connection. System Integration domain provides physical-level integration among different physical systems and related services (e.g. data archiving, event correlation, integration and abstraction of multiple systems), managing events information, e.g. signals, measures, and commands to the subsystems. Process Integration provides pro-

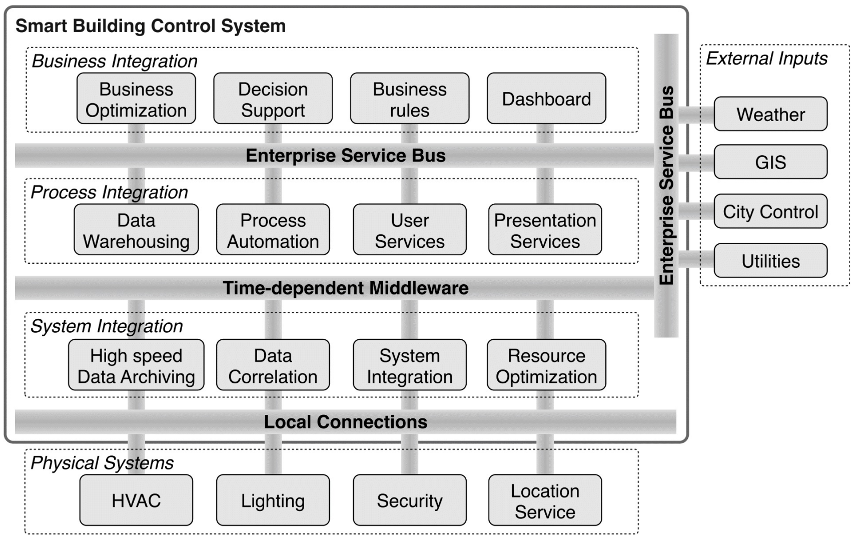


Figure 1.3: High-level System Architecture for Smart Building Control System

cess orchestration and automation, based on integrated system components, e.g., correlated events, abstract building services, historical data. Business Integration is the end-user layer and, through bulletin board, dashboard, simulation, gives information about building performance, optimization, policies/directives for improvements, by programming tools and business intelligence tools, e.g. offline analysis, online analytics, business rules. External Input connects to external data sources, giving a situational awareness of a boarder scope, e.g. real time energy price, weather information for consumption policies, to the process/business integration, thus, allowing a smart building to be integrated into bigger systems, e.g., smart grids, smart cities, and so on.

1.3.4 Smart Transportation

The worlds urban population has continuously increased over the last decades, from 30% in 1950, and now over half of the worlds population (54%) lives in urban areas. The continuing urbanization and overall growth of the worlds population is projected to add 2.5 billion people to the urban population by 2050, reaching the 66% of the total population. At the same time, the proportion of the worlds population living in urban areas is expected to increase, reaching 66 per cent by 2050. In addition, the number of mega-cities has nearly tripled since 1990; and by 2030, 41 urban agglomerations are projected to house at least 10 million inhabitants each [16]. This rapid growth has increased demand for transportation facilities and will do increasingly more in the next future. Providing more transport services to meet the increasing requests is often associated with undesirable outcomes such as traffic congestion, environmental issues like pollution and, in general, higher costs. According to [17], a study on 75 US cities, a total of 3.6 billon vehicle-hours and 5.7 billion US gallons of fuel were wasted due to congestion-related delays, resulting in a congestion cost of $67.5 billion.

Smart transport solutions, based on ICT-technologies, are required in order to face the above issues. In a managed transportation system, individual cars can travel together in fleets, with shorter following distances. Vehicles are staged and routed via main arteries as well as surface streets to match outbound flows to road capacities and fully utilize the road infrastructure. Road, air, rail transportation are coordinated to most efficiently transport people.

Smart Connected Vehicles

Smart connected vehicles (SCV) applications provide to share information among participants, detect traffic patterns and interact among them to enrich their information and, thus, improve their quality of service. SCV use a high amount of various devices, at different level of granularity, both along the road, and on-board into the car. In the road-side equipment, Road Side Unit (RSU) and Base station (BS) retrieve the information from the vehicles, communicate with the global application controller and deliver services to the participants. On the vehicle-side, on board unit (OBU) devices, composed of read-/write memory, store and retrieve information from the local vehicle or from external components, i.e. RSUs or other OBUs. OBUs allow exchanging information among different components, through wireless radio access, with reliable message transfer, network congestion control, data security and IP mobility, and to execute ad-hoc and geographical routing with the information retrieve from the road equipment. In addition, the on-board application unit (AU), that uses the OBUs for all mobility and networking functions, extends the application to users through endpoint devices, e.g. PDA. SCV applications can be different, ranging from entertainment applications, from global internet services or locally (e.g. media downloading, point of interest) to road safety and traffic efficiency. Road safety applications provide information and assistance to avoid collisions by sharing data (positions, speed, distance) between vehicles and RSUs (traffic signal violation, curve speed, emergency electronic brake, pre-crash sensing, cooperative forward collision, left turn assistant, lane-change, stop sign movement assistant). Usually, the type of messages are cooperative awareness messages (CAM), such as beacons, short messages periodically broadcast from each vehicle to neighbors to provide information of presence, position, kinematics, and basic status; or decentralized environmental notification messages (DENM), event-triggered short messages broadcast to alert road users of a hazardous event [18]. Finally, traffic efficiency and management applications (e.g. speed management, cooperative navigation) improve the vehicle traffic flow, traffic coordination and traffic assistance and provide updated local information, maps and messages bounded in space and/or time. They have no strict delay and reliability requirements, but their quality degrades with increases in packet loss and delay.

Smart Traffic Light

Another smart transportation solution is Smart Traffic Light (STL) system that manages in a dynamic and efficient way the traffic light along the road. Likewise SCV, STL among its several goals, has the two main purposes to prevent accident and to facilitate traffic flow. Accident prevention is performed with low-latency actions, by detecting pedestrians or cyclist crossing the street and measuring the distance and speed of approaching vehicles; thus, it issues alarms to approaching vehicles, changes from green to red, taking photos, and so on. To facilitate traffic flow throughout a city or a region is about how to respond to a dynamically changing traffic environment adaptively to improve controlling efficiency, under the constraint of guaranteeing fairness for each lane. The efficiency includes maximum intersection throughput (number of vehicles), and minimum vehicles average waiting time [19]. Thus, STL need to detect vehicles and calculate traffic information in real-time, determine green light sequence determination, by using the traffic information to determine the next green light to the case in the most need, and determine the light duration according with the obtained information.

In general, STL includes three layers. The bottom layer sends road traffic flow information to traffic lights and collects data from gps devices of the vehicles and from traffic light phase data from traffic controller. The intermediate layer, that consists of antennas, storage, traffic lights, receives and saves traffic flow data and sends control results to the OBUs and to the lights. The upper layer performs data processing, by using data filtering to discard data already received several times or obsolete, and traffic light control by calculating the optimal light-changing policy for this period with the lowest waiting time.

1.3.5 Wind Farm

Wind energy is the fastest-growing energy field and is becoming an important source in the modern energy supply system. The cumulative wind power capacity increased from 13.6GW in 1999 to 283 GW in 2012, with about 45 GW installed only in 2012, and this number is expected to achieve 760 GW in 2020 on moderate scenario [20].

Wind farm applications aim to tune the turbine, i.e. yaw and pitch, to the prevailing wind conditions in order to increase efficiency, thus trying to maximize the profits related to the wind power, and to stop it to minimize wear and prevent damage. They are composed of several utility-scale wind turbines with very flexible structures equipped with several closed control loops to improve wind power capture, measured as the ratio of actual to full capacity output for a given period, and power quality, affected by harmonic distortion. Typical sensors used for power measurement are strain gauges on the tower and blades, accelerometers, position encoders on the drive shaft and blade pitch actuation system, torque transducers. In addition, control loops aim to reduce structural loading and preventing mechanical breaks, keeping the turbine state always in a safe mechanical condition, hence extending lifetime and decreasing maintenance costs [21].

The wind farm system functioning is generally related to the weather conditions and the amount of wind. The wind measurements are performed by rotor speed measurement as basic control, anemometer for control purposes to determine if the wind is sufficient to start turbine operation, and wind vane to measure wind speed and wind direction [21]. In case of low or strong wind the turbines operations are more controlled in order to save money and turbine integrity. In particular: i) in low wind conditions, turbines are switched off to avoid losses because it is not economically convenient to run the turbine; ii) in high wind conditions, turbines and power are limited to the rated power to avoid electrical or mechanical load limits exceeding; iii) in very high wind conditions, turbines are switched off to prevent breakdowns. In case of normal wind conditions, the wind farm optimize dynamically the wind production.

Several controllers are used by the wind farm to manage the turbines and they operate in a semi-autonomous way at each turbine. In addition to the local optimization on the single turbine, a global coordination at the farm level is required for maximum efficiency. in fact, key to the process of assessing a potential wind farm deployment is the study of atmospheric stability and wind patterns on a yearly and monthly basis, along with turbine local operations [22]. An operational wind farm requires accurate wind forecasting at different time scales to interact with the wind market in the best way. In relation to the specific time scale the type of forecasts needed are: daily forecast used to submit bids to the independent system operator; hourly forecast to adjust the commitment, responding to events that may occur in the operating conditions (forced outages of generators, transmission lines, deviation from the forecast loads and so on); minutes forecast to dynamically optimize the wind farm operations.

1.3.6 Smart Industry and Wireless Sensor Network

Smart industry, where the development of intelligent production systems and connected production sites is often discussed under the heading of Industry 4.0. In smart industry, products, components and production machines will collect and share data in real time, shifting the concept from centralized factory control systems to decentralized intelligence. This enables machines and plants to adapt their behavior to changing orders and operating conditions through self-optimization and reconfiguration with the main focus on the ability of the systems to perceive information, to derive findings from it and to change their behavior accordingly, and to store knowledge gained from experience [23]. Smart industry applications should manage both industrial information about the production status, such as energy consumption behavior, material movements, customer orders and feedback, suppliers data, and data about the market in order to be able to adapt to the continuously changing market demands, technology options and regulations, almost in real-time.

Traditional wireless sensors networks (WSN) are composed of a large number of constraints sensor devices with low computational resources, low power, low storage capacity, low bandwidth, low energy, and so on. They can only sense the surrounding environments, do some simple preprocessing data and send information to more powerful static nodes, without be able to perform some actions. Many applications, that operate in different contexts, require both sensors and actuators networks (WSAN) in order to modify the environment in relations to the current situation, towards predefines goals. Collectively, these sensors produce a huge amount of data, both in structured and unstructured form and, nowadays, WSAN need more resources available to perform more complex data processing locally, with a low latency response time, in order to activate actuators for proactive actions in timely manner.

1.4 CPS Features and Requirements

CPS bring together the discrete and precise logic of computing to monitor and control the continuous dynamics of physical environments, which are characterized of uncertainty, noise and concurrent processes. Integration of physical processes and computing is not new, and embedded systems has been used for some time to describe engineered systems that combine physical processes with computing. However, most such embedded systems are usually small-size, limited within a confined local area, and closed boxes that do not expose the computing capability to the outside with no outside connectivity that can alter the behavior [7]. The deep transformation that comes from networking ability of mobile devices introduces and poses new considerable technical challenges to face. In addition, the new CPS applications demand that embedded systems be feature-rich and networked, which makes impossible to test the software under all possible conditions and to achieve predictable timing in the face of such openness is a major technical challenge. Thus, they necessitate the introductions of innovative scientific and engineering solutions to deal with the several requirements they are characterized of. In fact, since CPS interact with the physical world, they must guarantee at least the following needs [7]: dependability, safety, security, efficiency and real-time actions; trial-and-error approaches to build computing-centric engineered systems must be replaced by rigorous methods, certified systems, and powerful tools: analyses and mathematics must replace inefficient and testing-intensive techniques; unexpected accidents and failures must fade, and robust system design must become an established domain, i.e. the lack of perfect synchrony across time and space must be dealt with; the failures of components in both the controller and physical domains must be tolerated or contained, also by addressing system dynamics; scalability and the increasing complexity must be managed; a huge amount of data are continuously collected and must be handled.

Also managing and processing large data sets is not particularly new and during the past years a range of technologies have emerged to facilitate the efficient storage and processing of big data sets, introducing tools such as Map Reduce [24], Hadoop [25], Spark [26]. Those tools has been conceived for powerful machine but Big Data analysis is quickly extending to other sectors, with additional challenges, due to the evolution and diffusion of mobile devices and IoT in particular. Typically, Big Data is characterize by three properties: Volume, Velocity, and Variety [27]. Volume refers to storing, processing, and quickly accessing large amounts of data by easily scaling in relation to the amount of data to be stored and processed. Velocity refers to the data streaming high rate into the infrastructure and to the ability to process it with minimal latency. Variety refers to the variety of data to manage from unstructured textual sources to the wide range of sensors formats, coming from heterogeneous data sources. In addition, mobile devices add the geo-distribution property, which gives the context of the data collected and takes under consideration the naturally distributed pipeline monitoring.

In other words, the mobile devices management transforms the current scenario into a more challenging one, with zillions of sensors that gather huge amount of data in a specific context that must be considered as a coherent whole. Mobile devices are also demonstrating to be a technically challenging playground for distributed supports capable of sustaining the execution and run-time requirements of advanced dynamic applications, with specific focus on mobile connectivity, openness, scalability requirements and while-on-the-move generated data [90, 29]. Future internet applications, that are raising from the development of IoT environment, are largescale, latencysensitive and are no longer created to work alone but to share infrastructure, communication resources and common platforms that manage the system. Those applications require new specifications to be satisfied, like mobility support, large-scale geographic distribution, location awareness, low latency and low traffic in order to meet new requirements [30].

1.5 Cloud-assisted CPS: Potential and Limitations

Cloud computing has emerged and gained enormous popularity as an infrastructure that eliminates the need for maintaining expensive computing hardware. The National Institute of Standards and Technology (NIST) defines cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [31]. The cloud infrastructure can be viewed as containing both a physical layer and an abstraction layer. The physical layer consists of the hardware resources that are necessary to support the cloud services provided, and typically includes server, storage and network components. The abstraction layer consists of the software deployed across the physical layer, which manifests the essential cloud characteristics. Conceptually the abstraction layer sits above the physical layer [31]. Through the use of virtualization, clouds aims to address users needs with a shared set of physical resources, located in big datacenters and accessed only when needed.

Cloud computing is a very flexible platform that allows to use the cloud resources in the way that best fit the consumers request and to this purpose. The cloud model offers multiple service models and deployment models. The service models indicate the level of abstraction of the resources provisioned: Software as a Service (SaaS) provides the capability to use the application running on cloud infrastructure, from different client interface, e.g. web browser; Platform as a Service (PaaS) provides to the consumers the capability to deploy users applications, onto the cloud infrastructure without any control of the underlying infrastructure; Infrastructure as a Service (IaaS) allow the consumer the capability to provision processing, storage, networks, and other resources where the users can run their operating systems and applications. In addition, in terms of deployment models, the cloud offers the possibility to use: private cloud, provisioned for exclusive use of single organization; community cloud, provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns; public cloud, provisioned for open use by the general public; hybrid cloud, composed of two or more distinct cloud infrastructures, i.e. private, community, public cloud, that remains unique entities but bounded together.

It starts to be recognized and that CPS for Smart Cities can be industrially deployed in real scenarios if and only if their infrastructure and applications can benefit from cloud computing, in order to allow dynamic provisioning of resources by service isolation and to enable self-scaling and managing capabilities in a cost-effective way. Typical advantages that cloud computing can bring in CPS include:

- Scalability. Cloud capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand and to the final users, the capabilities available for provisioning often appear to be unlimited. Traditional approaches to provisioning, such as worst-case capacity planning, lead to over-engineer infrastructures to ensure quality requirements during peak load conditions. This would lead to unsustainable costs for companies and municipalities willing to offer Smart City services, and is strongly mitigated by proper integration with the cloud.

- Resource pooling. The providers computing resources, e.g. processing, memory, storage, network bandwidth, are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.

- On-demand provisioning. Consumers can unilaterally provision computing capabilities, as needed automatically without requiring human interaction with each service provider, allowing to increase/decrease hardware resources only when there is an increase/decrease in their needs.

- Broad network access. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms, e.g., mobile phones, tablets, laptops, and workstations.

- Measured service. Cloud systems automatically control and optimize resource usage by leveraging a metering capability at some level of abstraction appropriate to the type of service. Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

- Costs: systems based on cloud are relatively cheap to maintain, use, and upgrade, and most importantly their resource provisioning is elastically based on demand.

Cloud computing techniques are nowadays well developed and represent an industrially mature technology, with all the potential to assist and complement mobile devices, and in particular IoT, scenarios in order to make applications industrially feasible and cost-effective. For instance, referring to IoT, sensors usually adopt a high sampling rate, in particular for critical applications, to better monitor and act instantly, thus generating huge amounts of data to be managed and stored; IoT devices are generally resource-constrained, with limited storage, processing power, communication resources, and energy; sensed data are often transmitted via multi-hop wireless communications towards few sink nodes that have to play an active role with non-negligible duties and resource consumption [32].

Virtualization techniques, like cloud computing, are enabler and key technologies for mobile devices applications and in particular in IoT environment because they assist mobile devices to overcome their constraints. In fact, they allow IoT devices to perform tasks they are not able to execute due to their very limited resource available. In addition, they are also suitable to assist more powerful mobile devices to extend their functionalities with more complex operations. Virtualization techniques allow to apply computational-intensive functionalities, i.e. based on statistical analysis or machine learning tools, to every kind of mobile devices applications, greatly extending the quality of the application provided. An effective and efficient integration between mobile devices and the cloud can significantly contribute to overall efficiency, sustainability, data quality, and industrial cost effectiveness, with the cloud being able to provide transparent and dynamic scalability to manage peak load situations and to avoid worst-case capacity planning and the associated over-engineering of support platforms.

It is widely recognize in the literature that, in order to realize their full potential, CPS need a support infrastructure and a set of technologies that make their realization feasible and meet their requirements. In particular, CPS necessitate of a much more powerful and sophisticated computational components in order to deal with the challenges and requirements related to the massive usage of mobile devices, as described previously.

A first suitable improvement and, at the first sight feasible solution, could be the introduction of a mature virtualization technique, i.e. cloud computing, to exploit the CPS controller capacity and, thus, to strongly improve its efficiency and ability to provide complex analysis and functionalities. In fact, as already described, cloud computing has the potentialities to perfectly complement CPS and solve many of their lacks associated to the mobile devices. At the same time, the integration of cloud in mobile devices applications is double-faced and not easy to manage, bringing substantial advantages to both providers and end users on one side, but raising new unsuitableness in the integration with ubiquitous services on the other side. Although cloud can import huge improvements in a system processes with its great amount of resources availability, direct exploitation of cloud resources by ubiquitous IoT devices may introduce several technical challenges and inefficiencies, such as network latency, traffic and communication overhead to the devices, and further costs. In particular, dumbly connecting a myriad of sensors directly to a global large infrastructure, with large-scope and coarse-grain functionalities is extremely demanding for virtualized resources, which are not designed, implemented, and deployed for high-frequency remote interactions, e.g. in the extreme case of one cloud invocation per each sensor duty cycle. When anything would be able to observe the surrounded environment, gathering information from it and generating data, it could be possible that some data may be not required in a certain moment and, thus, it is not required to send information to the cloud. Ubiquitous devices gather enormous quantity of data during normal execution, but can be even worst in crowded places during peak load conditions or in future applications because of the purpose of mobile devices systems to sense as much as possible and, thus, increasingly collect more data, far exceeding the bandwidth capacity of the networks. In addition, IoT sensors usually use an high sampling rate, in particular for critical applications, to better monitor and act instantly and so generates a great amount of data which should be managed and stored. The result is a continuous iteration of the support infrastructure which remains busy per each sensor duty cycle, thus the property of scale per sensor will not stand. In this context, a system where cloud and IoT devices communicate directly is not realizable because the bandwidth cannot support this data load and also future improvements of communication capabilities are not enough to face the, faster, data growing rate. Those amount of data, if letting sensors directly communicate with the cloud, might possibly slow down or even lead the cloud to crash and might also cause an extremely high energy consumption which could significantly affect the systems performance, increasing redundancy of requests and information traffic and, thus, the network could become a bottleneck for the whole system.

As a general consideration, it starts to be widely recognized that an architectural model only based on direct interconnection between IoT devices and the cloud is too simplistic [22, 30]. In other words, the trivial integration with traditional cloud resources is inappropriate because they are too globally available and far from mobile device localities to satisfy rapidly, locally, and in a decentralized way the new associated requirements and critical issues have to be faced in order to turn new mobile devices applications suitable to be deployed in real world scenario and be widespread in many contexts.

In literature many research highlight issues about direct communications between IoT devices and cloud. [33, 34] proposes new architectures that will allow the integration of any lightweight sensors with the cloud, by overcoming typical cloud issues like latency, management of continuous sensing, the ability to support periodic events and the lack of elasticity when numerous wireless sensors transmit data simultaneously. [35] addresses the problems deriving from continuous sensing that raise many challenges with cloud iterations, such as energy consumption, cost, and communications overhead. As a workaround of these issues, it is been proposed that devices collect data and only sporadically upload them to the cloud but, in this way, this delay-tolerant model of sensor sampling and processing severely limits applications effectiveness and the ability of the system to be aware of its context, adapt and react to situations. [36] focuses on network latency impact in IoT-cloud applications and some experiments were performed to observe the impact on system performance on different deployments. These evaluations show that the most sensitive latency is edge to cloud and this is especially true for cases when the edge requires accessing the core for retrieving large amounts of data which needs to be transmitted fast enough to meet response time requirements. The more the edge computations access core data the worse the system response time and performance degradation gets.

In order to effectively and efficiently develop CPS, a two-layer infrastructure model only based on i) cloud computing resources and ii) sensors/actuators may be too simplistic. For instance, letting sensors directly communicate with the cloud may cause extremely high energy consumption. As a general consideration, it is inappropriate, from both performance and economic points of view, to have each sensor communicate independently with its cloud-based applications, with no optimization at all associated with reduction of connections, possible data batching, in-CPS aggregation, and event processing.

These issues highlight that the efficient and effective cloud-CPS integration calls for innovative management solutions, integrated into a support infrastructure capable of cross-layering application-specific requirements and properly offering the most suitable tradeoff between quality and resource consumption. An effective and efficient integration between IoT and the cloud is challenging but can significantly contribute to overall efficiency, sustainability, and industrial cost effectiveness, with the cloud being able to provide transparent and dynamic scalability to manage peak load situations and to avoid worst-case capacity planning and the associated over-engineering of support platforms to ensure quality of service requirements.

2 | Edge-enabled Middleware for Scalable CPS

For the motivations described in the previous chapter, it is necessary to adopt solutions based on the introduction of a intermediate middleware to enable a new generation of mobile devices supports based on the efficient integration of IoT devices/actuators and cloud resources. The aim of the present work is to discuss, analyze and technically introduce novel middleware solutions to greatly improve the CPS abilities via the integration between CPS and cloud computing resources. In particular, those solutions are based on the integration between mobile devices and cloud resources, with the purpose of avoiding or greatly reducing the negative effects of the issues related to their direct connection. In fact, a distributed and intelligent intermediate layer, composed by multiple edges, is required to add extra functionalities to the system and in order to permit the system to work properly, efficiently, providing quality of service and capitalizing the great potential the cloud has. The intermediate layer can store mappings between physical sensors and receiving endpoints on the cloud, performing a little processing of data when devices gather them and before send them to the network and eventually to the cloud. Thus, it operates data transmissions according to application-specific requirements and overall system optimization purposes, as well as locally managing the reaction of actuators on the CPS environment, by having everything seamlessly connected as a virtual continuum of interconnected and addressable objects in a worldwide dynamic network. The result will be an underlying structure on which users may develop novel applications useful for the entire society. The intermediate middleware is considered a driver for enterprise/industrial-based IoT that brings connections to the real world in a way never reached before and tries to face new business models, introduced by the diffusion of mobile devices, rethinking about how to create and capture value.

The intermediate middleware layer aims to move part of the computation to the edge of the network, thus, modifying the traditional two-layers architecture, composed of cloud computing and mobile devices, into a three-layers architecture, as shown in Figure 2.1, where a middleware layer supports mobile devices applications to satisfy their requirements.

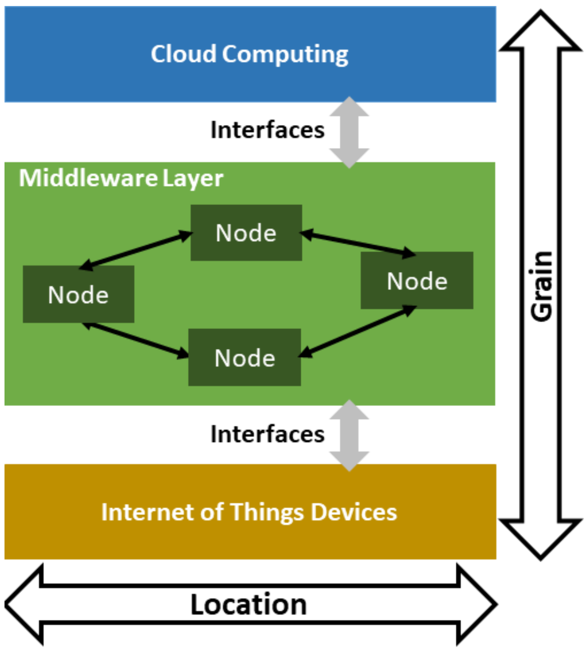


Figure 2.1: General three-layers architecture

In the following, this thesis work will detail this general architecture, specifying the middleware layer structure that best fit the category of application domains where is used, in order to address the satisfaction of the requirements needed by the specific CPS. In particular, in relation to different parameters, such as the location of the mobile devices, the distance between cloud and mobile devices, the resources available on the mobile devices, the type of information analysis required, and so on, it is possbile to indicate the most appropriate middleware to use. In this way, it is possible to create a specialization of the general three-layers architecture that support the CPS system and makes it working more efficiently, satisfying the requirements needed, and with also improvements of the system capabilities. The most promising and widespread solutions, which will be discussed in the following, are fog computing and edge computing, which will be described highlighting the main characteristics and differences between them.

The middleware introduction is starting to be widely recognized in the related state-of-the-art and other work in the literature, such as [37], [38], [39], [40], [41], has already point out the importance of a intermediate layer to satisfy the applications requirements in distributed environments. At the same time, although CPS and

Smart Cities applications concepts per se are not groundbreakingly new, the aspects to study and of potential originality are still relevant and manifold and the related research fields are very large scope and many critical technical aspects, i.e. scalability, interoperability, mobility, efficient communications, and so forth, are still to be properly investigated, thus leaving space for industrially-relevant and impactful directions of solution, as explained in the following.

2.1 Requirements Taxonomy for Edge-enabled Middleware

This section proposes a possible taxonomy for different families of solutions, that indicates how the requirements of mobile applications are spread over the architecture and which requirements are more relevant on each layer. Based on the careful analysis of all the state-of-the-art fog computing, edge computing-related literature, an original taxonomy for fog/edge solutions is proposed, based on the most proper positioning of the different features in different architectural layers and resume how the applications requirements are spread and divided into the architecture (see Figure 2.2).

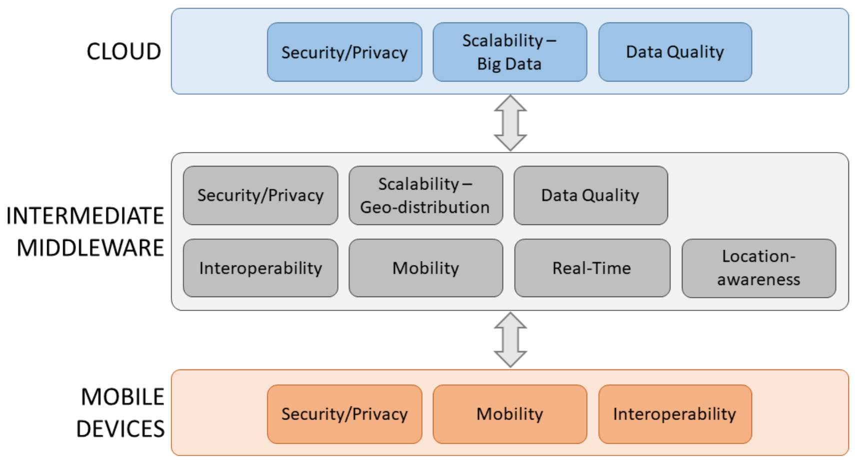


Figure 2.2: Cloud-Fog-IoT features distribution

The feature positioning are motivated with references to some clarifying application examples already explained in the previous chapter, e.g., Smart Grid, Smart Building, Smart Transportation, Wind Farm, and so on.

- **Scalability.** This feature summarizes all the mechanisms, algorithms, and tools that make IoT applications built on top of the fog/edge support deeply scalable. In particular, there are two sub-types of scalability: related to Big Data and geo-distribution. Big Data scalability support is located only on

the cloud, even if a part of computation (Small Data) is on the intermediate layer. Big Data analysis based on long-term analysis usually relates to prediction that try to capture system behaviors and to infer system evolution. Independently from the sub-type, elastic scalability is a key property of fog computing infrastructures that enables to scale up/down both the single node functions, by adding/removing internal components, and the whole hierarchy of fog nodes, by adding/removing external nodes also at different hierarchy levels. SCV/STL include the creation of efficient traffic policies to avoid traffic congestion and Wind Farm provides functions to predict and dynamically optimize bids/commitments.
Geo-distributed scalability refers to the ability to manage a large number of distributed nodes and is located on the fog layer, where several devices are spread in wide-scale scale networks, like in SCV where vehicles are dense and can move across regions.

- **Data Quality.** This property also relates to the overall ability (and associated knowledge) of the IoT deployment environment to detect anomalies and react in real time, from geo-distributed data gathered from sensors, by applying fault detection techniques. Therefore, data quality support is positioned near the edges on the intermediate middleware layer. For instance, in SCV applications, speed data quality can be improved by calculating the standard deviation of observed vehicles in relation to the average speed received in the same conditions (location, time of the day, weather, type of vehicle, etc.) and define data thresholds to discard outlier data.

- **Location-awareness.** This is a key requirement to increase the efficiency (resource consumption, network congestion) and overall quality of the system. Location-awareness-related functionality is located in the intermediate middleware layer to be closer to the targeted environment but anyway with grouping/locality visibility. In SCV/STL it is possible to divide the area of interest (intersections, roads), also based on RSU positioning; RSU can infer whether a vehicle is in danger (approaching too fast, dangerous bend), by reacting on nearby traffic light cycle or with alarms.

- **Interoperability.** Heterogeneity is an intrinsic characteristic of real world systems in this field and a property that can strongly affect system performance. In particular, in this context the interoperability feature enables the possibility for different fog nodes to communicate with and to compose hierarchical infrastructures of fog nodes, as well as to exchange internal fog node components. Mobile devices and middleware layers must provide interoperability, due to the necessity to handle different devices, in term of computational power, resource capability lifespan, and communication technologies. SCV/STL are composed of heterogeneous components (on-board sensors, RSU, traffic lights) that must work properly together, even if multiple manufacturers provide different implementations of them; they may extend beyond the borders of a single controlling authority.

- **Real-time.** Real-time requirement is characterized by computational activities with stringent timing constraints that must be met in order to achieve the desidered behavior and can be divided into hard and soft real-time. Hard real-time have the highest possible priority, are used for critical activities, and need to hit every deadline in order to keep the system to run under control and avoid even relevant consequences [42]. Soft real-time can miss some deadlines, although causing performance degradation, producing anyway some useful results [42]. Both hard and soft real-time reactions, with low-latency priority-driven responses to important interactions with incoming data, are required to react properly over the target CPS. Initial data processing and actuation must be performed within the fog layer to have sufficient resource availability and to avoid the latency due to interaction with global cloud. In SCV and STL applications, hard real-time reactions are crucial to ensure safety: for instance, [22] estimates that, in STL, the reaction time must be within few ms to be compliant with safety requirements. In Wind Farm, low-latency actuation is important to prevent turbine damage in case of strong wind or to optimize wind power forecasting.

- **Mobility.** The intermediate layer must manage device disappearance or device recoverability if a device exits its sub-network or something wrong/unexpected happens. Similarly, devices must be able to shift from one edge node to another without anomalies, thus ensuring seamless hand-off and discoverability notwithstanding device/fog node mobility. For instance, in SCV there is the need for fast mobility support to manage moving vehicles as macro-endpoints, that can move freely in the environment with the ability to discover other fog networks, and let them switch from one middleware sub-network to another.

- **Security/Privacy.** They are, probably, the most troubling requirements for real-world applications and are pervasive in all parts of a system, requiring end-to-end security/privacy, thus, the security/privacy functionality support is spread to all layers. According to [43], the threat profile for handheld devices is a superset of the profile for desktop computers due to additional threats related to size, portability, wireless interfaces, and associated services; they include theft, unauthorized access, malware, spam and electronic tracking, electronic eavesdropping and tracking, and cloning. Since fog nodes are the first to access data, they must provide contextual integrity and isolation, and control aggregation of privacy-sensitive data before that data leave the edge. In addition, all the nodes at the edge (and also the single components within a single edge node) that manage trustworthy data must be hardware and software attestable; in mature support environments, it is critical to adopt policy-based security management solutions capable of efficiently enforcing high-level rules, e.g., for authentication, strong passwords, and automated disabling of non-required services. In the intermediate layer, privacy mainly focus on the ability to monitor/protect data from information exposition. Intermediate solutions keep data in the network for better privacy and must be able to anonymize and define the ownership of user data, thus localizing intelligence but preventing to reveal protected data. In STL/SCV, security is key to avoid collisions, react in relation to the conditions of the vehicles/roads and they must also focus on privacy concerns due to the pervasive surveillance on each action, images acquisition or vehicles movements patterns.

Note that the proposed taxonomy is coherent with other first classification efforts recently emerged in the literature and, at the same time, originally addresses application requirements needs in a more focused way. For example, by comparatively considering the recognized fog computing taxonomy presented in [44], the main differences of the present taxonomy are: i) it is more application-oriented, by stressing the importance of requirements that the infrastructure should provide for the applications that run on top of it (less accent on infrastructure requirements); ii) it is more abstract and higher-level taxonomy, that does not focus on specific concerns of lifecycle management, such as automated and easy deployment, implementation, testing, maintenance, and so on; iii) it is less focused on business-related requirements, such as cost savings, adaptive infrastructure to support changing business needs and economical operations, business support in the hierarchy, and so on. Table 2.1 summarizes how the above features impact the CPS applications described in the previous section.

Table 2.1: CPS Application Requirements

	Big-Data Scalability	**Geo-distribution**	**Data Quality**	**Location-awareness**	**Interoperability**	**Real-time**	**Mobility**	**Security-privacy**
ParticipAct	Strong	Strong	Strong	Strong	Strong	Medium	Strong	Strong
Smart Grid	Strong	Strong	Strong	Strong	Strong	Medium	Strong	Strong
Smart Building	Strong	Strong	Strong	Strong	Strong	Medium	Strong	Strong
SCV	Strong	Strong	Strong	Strong	Strong	Strong	Strong external - Limited in vehicle	Strong
STL	Strong	Strong	Strong	Strong	Strong	Strong	Strong	Strong
Wind Farm	Strong	Limited (in farm)	Medium	Medium	Limited	Medium	Limited	Medium (expecially physical)
Smart industry	Strong	Medium	Strong	Medium	Medium	Medium	Medium	Strong
WSN	Strong	Strong	Strong	Strong	Strong	Strong	Strong	Strong

2.2 Fog Computing

Fog computing is a relatively new concept and already popular term, coined by Cisco [30], that indicates a part of computation moved near to end-users with the purpose to let off their computing load and speed up response/performance. In other words, fog computing identifies a horizontal architecture that distributes computing, storage, control, and networking functions closer to the users along a cloud-to-thing continuum [44]. Fog computing is a highly distributed solution that, in terms of infrastructure, is located between the network backbone, where there are the global virtualized resources and services, and the mobile devices at the edge, allowing to interface and connect the two sides towards advanced pervasive applications.

Fog computing can be profitably introduced to let IoT applications interwork efficiently with cloud resources: in fact, fog can act as the intermediation layer between the cloud and IoT, at the same time by extending the applicability domains of cloud solutions and by increasing resource availability in MIoT. In this perspective, fog can be considered as a significant extension of the cloud computing concept, capable of providing virtualized computation/storage resources and services with the essential difference of the distance from utilizing endpoints. While the cloud exploits virtualization to provide a global view of resources everywhere available and consists of mostly homogeneous physical resources, far from users and devices, the fog tends to exploit heterogeneous resources that are geographically distributed (often with the addition of mobility support) and situated in proximity of data sources and targeted devices. In other words, fog computing has been designed to put intelligence close to IoT devices in order to perform decentralized tasks as soon as data are generated from sensors and can affect all different levels of the IT development, from device management to network traffic control, from data aggregation to resource management, from application development/coordination to security and fault recognition.

The related literature [45] uses in this context the term softwarization of network and service management to highlight the evolution pushed by the fog. A primary idea emerging from existing fog solutions in the literature is to deploy a common platform that supports a wide range of different applications, and the same support platform, with multi-tenancy features, can be used also by a multiplicity of client organizations that anyway should perceive their resources as dedicated, without mutual interference [46]. Figure 2.3 shows a high-level architecture that summarizes the above vision by positioning the IoT, cloud, and fog computing layers.

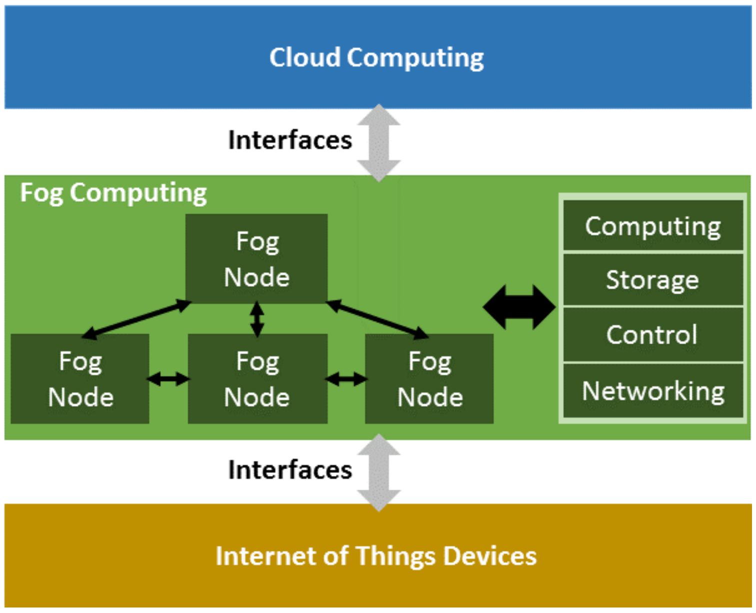


Figure 2.3: The high-level picture of the integration of cloud, fog, and IoT technologies

Due to cloud technology maturity, cloud-side interfaces are more defined and it is currently easier to make cloud service platforms interact; on the contrary, IoT-side interfaces and, even more, fog ones are more various and heterogeneous nowadays and much work should be done to homogenize the different approaches and implementations that are very recently emerging.

The fog-cloud integration literature, in particular in the MIoT domain, is relatively limited and still in its infancy. In [30] a hierarchical partitioning of fog computing is proposed. The bottom layer is designed for M2M communications with ubiquitous devices; it collects/processes data flows and issues control commands to actuators; it also filters the data to be consumed locally, by delegating the management of the remaining dataflows to the higher-level tiers in the hierarchy. The second and third tiers deal with visualization and reporting towards human-to-machine interactions (explicit operators involvement), as well as systems/processes (M2M-oriented).

The concepts of fog and cloud computing can be integrated in a single infrastructure to achieve the best from both technology. Fog nodes can access more energy, can be physically larger, can storage terabytes of data and have more capabilities than IoT devices, thus, can also support physical endpoints to overcome their constraints in terms of energy, space, environmental issues and reliability.

Cloud computing improve IoT applications, creating better services more efficiently and, with the fog computing, communication can be made real-time for latency-sensitive applications like video streaming, gaming, augmented reality, and so on. This is possible because fog computing has been designed to put intelligence in the network in order to perform tasks as soon as data arrive from sensors, on the contrary of solutions which use a single physical component, like a gateway, where network is not physical and it cannot perform as fog. Fog provides an easier and more efficient execution of tasks like storage, data aggregation, pre-processing, data security and privacy tasks near where information is gathered, so that the performance can improve significantly. Since fog is localized, the proximity to devices guarantees location-awareness and thus the possibility to aggregate data easier or also to increase security of the system detecting fault data due to its knowledge of the surrounded environment and the data domains.

2.2.1 Architecture Proposal

In the literature, primarily due to the novelty of the fog concept, only few works propose detailed architecture about how to implement the fog layer and which components it should consist of. Figure 2.4 illustrates an original reference architecture that details the high-level architecture of Figure 2.3 and go into more in-depth technical details of the fog computing analysis.

This novel architecture can guide fog computing solution implementations and can contribute to create a common understanding to advance fog research and to leverage the deployment of real-world fog implementations. Note that the intermediate fog layer in the proposed architecture may consist of a single node in simple deployment scenarios, as well as of a hierarchy of coordinated fog nodes, each one with a possibly different and more specific task (heterogeneous roles), also depending on the complexity and scalability requirements of the targeted IoT application.

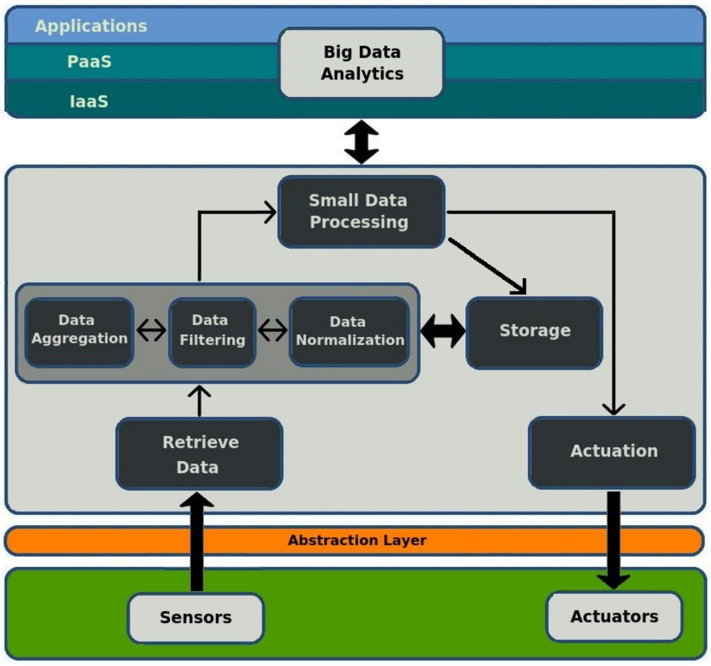


Figure 2.4: The proposed cloud-fog-IoT reference architecture

Local Sensing and Data Handling

In fog computing, sensing is a critical aspect, because directly affecting the quality (e.g., in terms of precision, accuracy, confidence level, and so on) of the generated data, which typically is the primary input for successive application steps. In the usual case of sensors without sufficient intelligence, location awareness or computational power to perform local filtering operations on data, they send all generated data to the fog layer; the fog is responsible for providing automatic data acquisition mechanisms to extract useful data and, consequently, for saving resources during later stages.

This abstraction layer allows sensors/actuators to easily interface with the fog node and to efficiently send data for higher-level system analytics and software functions. It also allows to share meta-data about fog architectural elements, for example to the purpose of data and multi-vendor interoperability, service composability, or cross-layer optimizations. Notable examples are to optimally route data between fog nodes by exploiting information-centric network (ICN) solutions and to create dynamic fog topologies through software-defined network (SDN) mechanisms and tools [44].

In the proposed architecture, a first component is used to retrieve data from all sensors, based on an abstraction layer, that acts as data sink for sensors. Initial data retrieved from sensors need to be handled and automatically transformed into useful data for further elaborations, through processes like i) data aggregation, ii) data filtering, and iii) data normalization, that transforms the aggregated and filtered data into a standard and commonly agreed format.

Big and Small Data Processing

After a first step of raw data managing, middleware components process data, through both more traditional data analytics on the cloud (named Big Data processing in the following) and more lightweight data processing techniques on the fog layer (named Small Data processing in the following). Big Data analysis and processing are typically used to perform more strategic and system-wide decisions, or for policy management; they usually need a non-negligible amount of runtime resources to support the execution of data intensive operations. As widely accepted in the related literature, for instance in [47], Big Data are data characterized by variety, volume, velocity, veracity, volatility and they should properly be hosted by cloud resources, thus relieving scalability, cost, and performance issues.

On the contrary, fog computing must perform short-term analysis and lightweight processing with relatively limited amount of data. Small Data can be considered as an extension of the general Big Data concept, specifically targeting resources near the devices, suitable to be handled in terms of fog computing support and able to perform low-latency actions, e.g., by taking operational decisions as soon as data can be turned into meaningful context. In the related literature the Small Data term starts to be used to refer to a limited quantity of fine-grained data; [48] points out the main differences between Big and Small Data in specific application domains.

Actuation

One of the most interesting components in our architecture proposal relates to the actuation phase. In IoT applications, after improvements in collecting the environmental data by sensors, fog can also play the role of strongly improving the actuation phase with timely reactions to sensed aggregated/filtered information. Many IoT applications require to timely react on the execution environment to boost their responsiveness quality, in particular when operating in critical contexts and if the monitored behavior relevantly deviates from the desired state; in this perspective, fog computing might considerably turn on new opportunities of effective and efficient implementation.

Storage - Cloud distribution

Another original element of the proposed architecture is the storage functionality, made up by a group of resources that act as a small set of distributed cloud-like storage resources inside the fog. This facility can bring limited cloud services closer to the edge of the deployment targets or temporarily store some data, in particular Small Data, and periodically upload them to the cloud, reducing unnecessary global-scope interactions. Non-negligible benefits are expected, also in terms of scalability, reliability, data integrity, and boost of performance, with improvement in application responsiveness and user quality of experience.

2.3 Mobile Edge Computing

The mobile edge computing (MEC) is a category of different solutions that differ from fog computing, mainly from the application and the architectural point of view, because it is targeted for more powerful mobile device and is based on support servers that work in a more standalone way rather than highly connected and distributed ones, with specialized nodes than usually depend from other nodes.

[49] indicates MEC as small-scale data centers deployed by the telco operators in close proximity with end-users, and may be colocated with the existing infrastructures, i.e. wireless access points, such as macro base stations. [50] and [51] highlight the necessity of new infrastructures, with low-latency connection to large-scale resource-rich cloud computing infrastructures within the network edge and backhaul/core networks, that extend traditional cloud-based infrastructure, deployed by telco operators, for fast interactive response, high reliability, and in general to satisfy the stringent needs of telecom services.

Starting from the previous definitions, we claim edge computing concept emerged from telco operators that can benefit from the virtualization techniques introduction at a closer location to the edge of the network, by exporting cloud capabilities to the users proximity. Mobile edge computing solutions fall into a same definition based on the extension of typical telco services, which are global services providers that supply services into the main network, usually far from the mobile devices, toward the edge of the network. With the proliferation and wide-spread adoption of mobile telephony and data, service providers have been eager to exploit context information or customized services, i.e. location-based service, and the mobile edge computing is the most suitable concept to extend the typical telco network. Mobile edge computing introduces servers that are used by providers to extend the global telco services approaching mobile devices with closer servers that have the services deployed on them. In this perspective, edge computing nodes can be considered as the access points to the main network backbones used by the telco companies to distribute services or as smaller base stations with the ability to propagate the services near end points.

The main solutions that fall into the mobile edge computing category are Follow-Me-Cloud and ETSI Edge Computing.

2.3.1 Follow Me Cloud

Follow-me Cloud (FMC) proposes a mobility management scheme, defined as a technology developed to support novel mobile cloud computing applications, by providing both the ability to migrate network end-points and to reactively relocate network services depending on users locations, and migrated to follow their movements. In relation to the users location, FMC provides the services execution from optimal data center for the current locations of the users and the current conditions of the network. Following this concept of mobility, it is possible to introduce an analytical model for FMC that provides the performance related to the user experience and to the cloud/mobile operator, underlining the importance of careful consideration when triggering the service migration [52]. The general goal is to guarantee adequate performance for the client-server communication and localize network traffic generated by applications to have a precise control on the use of network resources.

2.3.2 ETSI Mobile Edge Computing

ETSI Mobile Edge Computing (in the following called MEC) is the general MEC concept defined and standardized by the Industry Specification Group (ISG) within the European Telecommunications Standards Institute (ETSI [53]). It is defined as an emerging technology that provides cloud and IT services within the close proximity of mobile subscribers, moving applications, data, and services from cloud towards the edge of the network. It enables mobile subscribers to access IT and cloud computing services at the close proximity within the range of Radio Access Network (RAN) and can be defined as a model for enabling business oriented, cloud computing platform within the radio access network at the close proximity of mobile subscribers to serve delay sensitive, context aware applications [54]. In fact, MEC offers real time RAN information, such as network load and users location, to provide context aware services to the mobile subscribers, thereby enriching users satisfaction and improving Quality of Experience (QoE).

MEC is based on a virtualized platform, with an approach complementary to Network Function Virtualization (NFV): in fact, while NFV is focused on network functions, the MEC framework enables applications running at the edge of the network. The infrastructure that hosts MEC and NFV or network functions is quite similar; thus, in order to allow operators to benefit as much as possible from their investment, it will be beneficial to reuse the infrastructure and infrastructure management of NFV to the largest extent possible, by hosting both Virtual Network Functions (VNF) and MEC applications on the same platform [53].

MEC platform increases the edge responsibility, bringing the computation and storage capacity to the edge of the network and allows computation and services to be hosted at the edge, which reduces the network latency and bandwidth consumption of the endpoint. Network operators can allow the radio network edge to be handled by third-party partners, this will allow to rapidly deploy new applications and edge services to the mobile subscribers, enterprises. MEC applications are characterized by proximity of data sources that determines a significant reduction in data movement across the network. Thus, the main benefits result in more location awareness, reduced congestion, cost and latency, elimination of bottlenecks resulting from centralized computing systems, improved security of encrypted data as it stays closer to the end user reducing exposure to hostile elements and improved scalability arising from virtualized systems. All of this can be translated into value and can create opportunities for mobile operators, application and content providers enabling them to play complementary and profitable roles within their respective business models and allowing them to better monetize the mobile broadband experience [55].

Note that nowadays in most literature the edge computing and fog computing terms are commonly used as synonymous to specify an intermediate layer to support mobile devices, due to their similar objectives and some common features. However, the terms indicate different concepts and should be used more precisely to indicate each own a different approach, given that edge-based and fog-based solutions exhibit some key differences that make it different technologies. For instance, fog computing has been conceived as an extension of cloud computing and, thus, works in association and in a complementary way to the global cloud resources, while edge computing is not necessarily linked to cloud computing and can work standalone. Fog computing is based on a hierarchical architecture where node has different tasks in relation to their level, while edge computing minimizes the number of layers used and usually is within a flat organization. In relation to that, fog focuses on scalability and allow to scale the number of nodes into the architecture while edge computing is not particularly focused on scalability. Fog computing extends edge computing concept because it includes, along to the computation, also networking support facilities, storage, control and acceleration components.

2.4 Other Edge-related Solutions

There are other solutions, with the same purpose to support mobile devices, extending their capabilities and meeting their requirements, which are based on different concepts of fog end edge computing. The most relevant approaches in the field are cloudlet and mobile computation offloading.

2.4.1 Cloudlet

The cloudlet approach, from one side, is very similar to the MEC concept, exploiting a cluster of multi-core computers, with gigabit internal connectivity, by using virtualized resources, near endpoints, thus, aiming to bring the computing power of cloud data centers closer to end devices. On the other side, it differs from the edge computing that is specifically defined as an extension for the telco infrastructure, because it is not considered a development in the evolution of mobile base stations or linked to the convergence of IT and telecommunications networking.

Cloudlet is a general-purpose approach that can host every kind of third-party services, with the primary goal to satisfy real-time and location-awareness requirements. If no cloudlet is available nearby, mobile devices can degrade to a fallback mode that involves a geographically distant and globally available cloud; full functionality and performance can return later if a device discovers a nearby cloudlet [46]. A cloudlet has only soft state, thus, the management burden is kept considerably low [56] and is completely transparent under normal conditions, giving mobile users the impression of directly interacting with the cloud [57]. Cloudlets offer numerous advantages over the global cloud, such as lower latency, higher bandwidth with less generated traffic, offline availability, cost-effectiveness.

Although the typical advantages related to the movement of the computation near the edge of the network, cloudlet is less effective compared to cloud computing characteristic: it covers a small region and is less resourceful than the cloud, thus, it is not scalable in service and resource provisioning. To overcome this cloudlet challenging issues, the MEC paradigm has been proposed [54].

2.4.2 Mobile Computation Offloading

Mobile applications are constantly growing in functionality and complexity in order to offer a wider set of high-level features that challenge execution time and energy usage in even top-end mobile smartphones. This is especially true for mobile applications aimed to seamlessly augmenting users cognitive abilities such as speech recognition, natural language processing, computer vision, and augmented reality applications [46]. The computation ability required to execute the functions needed in such applications can be beyond the computation ability of the used mobile devices, or can drain their battery life at an unacceptable pace (e.g., a few percentage points per minute). This raises major concerns for users because, as the complexity of computations in mobile applications increases, the performance of their smartphones degrades, as well as their energy consumption increases. In other words, the gap between the demand of resource-intensive applications and the more slowly growing availability of resources at mobile devices becomes critical for offering a fully satisfactory users experience.

Mobile computation offloading [58] has attracted much interest and is widely accepted as a powerful concept that can overcome the resource constraints of mobile devices and low-power IoT devices, relieving them from heavy computation duties that conflict with their constrained and limited available resources, primarily in terms of battery power. Mobile computation offloading allows dynamically migrating computation-intensive applications from resource-limited devices to external powerful servers; it can provide reduced execution time and, most relevant, reduced energy consumption at the cost of the process of transferring computation-related information. In particular, it starts to be recognized that the offloading decision should be context-dependent, i.e., deciding when and which computation task to offload depending on current context data such as execution time, energy consumption, task complexity, expected network latency, etc.

Traditional mobile computation offloading is designed to be performed towards virtualized computing resources over the globally available cloud. However, due to the unsuitability of cloud resources to deal with the strict latency requirements of mobile devices, also mobile computation offloading is starting to consider the possibility to dynamically offload computations from mobile devices towards fog or mobile edge nodes. Mobile applications can benefit from computation capability of fog and edge nodes to remain available and responsive while processing large volumes of data [59].

Table 2.3 resumes the main characteristics of the previous described proposals.

Table 2.3: Middleware Solutions

Solution	**Main Characteristics**	**Strengths**	**Weaknesses**
Fog Computing	Hierarchy, node specialization	Middleware features	Cloud support required
Follow-Me Cloud	Dynamic services mobility	Less cloud-devices distance	No middleware support
ETSI MEC	Flat organization, telco extension	Middleware features	IoT support
Cloudlet	Small standalone cloud	General-purpose	Scalability, resource provisioning
Computation Offloading	No predefined architecture	Client computation relief	Offloading tasks definition

2.5 Edge-based Architectural Proposals

In this section, some intermediate middleware-based solutions are grouped into macro-areas to highlight possible categories of solutions, divided for type of use cases, that support a wide range of applications. This section aims to extend the discussion on the application features, outlining solutions that may increase/decrease the ability of the system to meet certain needs of IoT applications and indicating the benefits to use one solution rather than another. In particular, there are three categories of solution: i) middleware moved towards cloud, ii) middleware moved towards edge, iii) multiple middleware levels.

Note that, although the fog computing solutions have been expressively conceived to work in combination with the cloud computing, in this section the generic term intermediate middleware is used because also edge computing can be properly used in solutions that require the usage of cloud computing, falling in the following categories of solutions.

2.5.1 Middleware Moved Towards Cloud

In system where the intermediate middleware is moved closer to cloud and farer from the mobile devices, the cloud features are reinforced within the middleware layer. Since the distance from the edge is higher, the intermediate layer performs tasks more similar to the global cloud, exploiting the cloud-related features, and is responsible to control a wider area and can coordinate more mobile devices. To this purpose, the middleware layer must be reinforced with more powerful hardware and more computational-resources available to enable the ability to perform intensive and complex analysis. Therefore, scalability related to both Big Data and geo-distribution, and analysis to improve data quality are stronger. In addition, also mobility support must be reinforced in order to face wider and more frequent movements due the higher number of mobile devices to manage. On the other side, due to the increased distance, real-time response and location-awareness are more neglected features and cannot be central requirements within the considered applications. Responses take more time to arrive to middleware nodes and then to devices and low-latency requirements can be compromised. Since the middleware has to manage mobile devices widespread in the larger portion of the environment located in different areas, that operate in different contexts, the middleware layer cannot accurately handle contextual analysis due to less environmental details available and, thus, also location-awareness feature cannot be detailed. Finally data are no longer collect to stay close to users, thus, more precautions must be evolved to protect data privacy.

From the architectural point of view, the infrastructure is composed of less but more powerful middleware nodes that already contains enough computational resources to perform the most of the tasks and analysis required. For this reason, the architecture is more centralized and less distributed, with the middleware nodes that work more as standalone servers, rather than in a hierarchy/cluster of connected nodes, with less frequent communications among different middleware nodes and with the global cloud. Table 2.5 resumes the feature more related to a mobile devices-centric application and cloud-centric application. Of course this architectural solution exploits the cloud-related characteristics and gives less importance on the devices-related ones.

Table 2.5: Mobile Devices vs Cloud related Characteristics

Devices Characteristics	**Cloud Characteristics**
Real-time	Big Data Analysis
Location-awareness	Geo-distribution
Communication Inter-Middleware	Data Quality
Interplay Cloud-Middleware	Mobility
Distributed Architecture	Centralized Architecture
Privacy	

This solution may be suitable for applications that can handle higher latency and where location-awareness is not crucial for analysis. For instance, some applications that may benefit from this configuration are wind farm or smart grid use cases because a limited response-latency can be tolerated without compromise the system functionalities. In fact, a more relaxed application of the real-time and location-awareness requirements may lead to: less timely reaction related to wind power forecasting/analysis and a higher delay to apply optimization of wind energy prices in wind far; higher home energy consumption and less adjustments of supply-demand energy balance in smart grid. Although these small disadvantages, this solution can significantly improve the strategic analysis of the system, by more complex and accurate data analysis closer to the edge, where the cloud is not necessarily needed, and, at the same time, minimizing the communications required.

This architectural solution is more suitable to be applied in context with more resourceful mobile devices, e.g. smartphone, rather than constraint or tiny IoT devices that need to send most of the data collected due to the amount of traffic generated in the network. In addition, it is a more suitable model to realize edge computing solutions, rather than fog computing solutions, because edge solutions are typically based on standalone servers with limited interactions with the global cloud computing.

2.5.2 Middleware Moved Towards Edge

This type of solution is opposed to the previous one, as well as the implication bring by this architectural model. In this case, middleware layer is closer to devices and farer from cloud, with more constraint hardware capabilities and they can perform lighter computation. Thus, features related to mobile devices-related features are reinforced while cloud-like features are weaker and, with reference to Table 2, the devices-related characteristics are more central than the cloud-related ones.

From the architectural perspective, the intermediate layer is composed of multiple nodes, highly distributed, and strongly connected to both the other middleware nodes and the global cloud, due to the more limited computational resources available. Thus, more data are sent to the cloud and to other nodes, with an increase amount of network traffic. Since each node controls a smaller portion of the environment with less devices monitored by a single node, geo-distribution scalability and mobility support are less relevant. On the other side, moving the computation closer to the edge improves many IoT-related features. In particular real-time response, location-awareness and privacy are directly related to the distance between middleware and mobile devices. An intermediate layer closer to the edge can be used in latency sensitive and critical scenarios applications, where real-time processes are required in order to guarantee the correct execution. For instance, SCV and STL applications can really benefit from this solution and overcome many issues related to the critical contexts where they act, by using nodes very close the edge, in order to perform immediate data elaborations and actuations and make real-time executions.

This architectural solution is more suitable to be applied in context characterized by tiny and constraint devices, i.e. IoT, that need frequent communication with the intermediate middleware, due to negligible computational power. In this case, although the amount of data sent is high, the generated traffic is confined into a small portion of the network and does not affect the other sub-networks. In addition, this solution is more suitable to realize fog computing solution, rather than edge computing solutions, because fog computing solution typically use multiple nodes densely connected node in a hierarchical way, thus, also very close to the mobile devices, with frequent connection with among the other fog computing nodes. However, this solution is feasible only in very limited and small scenarios due to the difficulties to perform advanced data analysis and the distance with the cloud, that, in busy or real-world applications contexts, can introduce an enormous amount of data into the network and slow down the overall system.

2.5.3 Multiple Middleware Levels

This type of solution is the combination and the extension of the two previous solutions and consider an intermediate middleware composed of multiple levels of nodes densely connected, with possible internal organization of sub-groups of nodes. The idea is to create a multi-levels organization, where each middleware node is no longer a general-purpose node with a wide range of functionalities but, rather, it has a particular role, location responsibilities and is specialized to efficiently execute few tasks, reporting the results to another node. The node may be organized, as shown in Figure 2.5, into hierarchical organization or mesh/cluster of nodes, in relation to the nature of the scenario where the system works, and systems can work properly, efficiently, eventually balancing computational load and with a stronger scaling ability.

In multi-level architecture, whatever the organization, each node is specialized to perform a specific work, in relation to the level and its assignment and it is optimized to handle a specific task, with different functionalities and capabilities to do that. In fact, nodes are equipped with different hardware, that gives the specific type and amount of resources to perform the tasks assigned in the best way, i.e. powerful gpu hardware for image processing tasks, powerful ram for streaming processing applications, and so on. Nodes closer to the edge, in lower levels of the hierarchy, exploit real-time interactions and location-awareness, and have the ability to highlight IoT-related

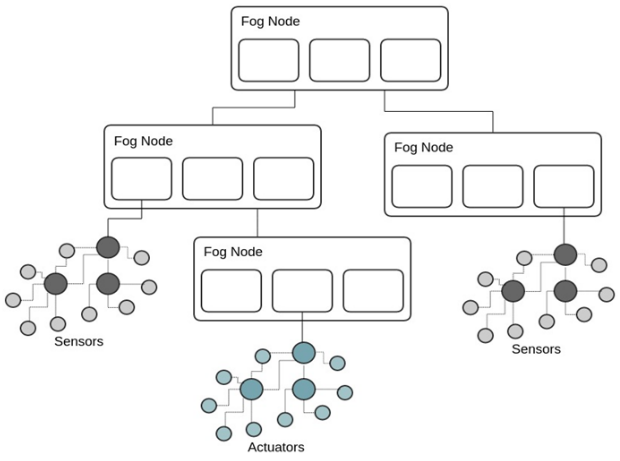


Figure 2.5: Multi-levels Intermediate Middleware Architecture

features. Nodes closer to cloud, in higher levels of hierarchy, are more powerful and perform resource-intensive operations and highlight the cloud-related features. By using specialized nodes at different levels, it is possible to get the best from both the previous approaches described. Table 2.7 resumes stronger and weaker points of this solution.

Table 2.7: Multi-Levels Intermediate Middleware Architecture

Strong Characteristics	**Weak Characteristics**
Architecture Definition	General-Purpose Architecture
Specialized Nodes	General-Purpose Node
Different Node Capabilities	
Hierarchical Task Partition	

This architectural solution is suitable for a wide range of contexts applications, because it mixes and exploits both mobile devices-related and cloud-related features, taking both advantages and minimizing their limitation. In particular, this model adoption is necessary in wide and large scale systems that deal with tiny and constraint IoT devices that need a support close to the edge, in order to create a support infrastructure close to IoT devices but, at the same time, with powerful capabilities and advanced functionalities. This solution is usually adopted in large-scale and real-world fog computing solution characterized of a sophisticated hierarchical organization of fog computing nodes.

2.6 Solutions for Mobile Services

In the following chapters, this thesis work propose different solutions that aim to face the challenges of mobile applications, in particular in case of a high amount of devices. The solutions differ considerably from each other because they try to cover as much as possible the wide spectrum of open points and challenges on the mobile devices applications, each one targeting specific requirements. In particular, the solutions for mobile services are divided in relation to the type of mobile devices and the architecture used in the system, following the explanation in Section 2.5, into:

devices are enough powerful but, at the same time, the edge nodes are required in order to guarantee higher performance or reliability, i.e. the application works in a hostile environment;

- cloud computing-based solutions, if the middleware is moved towards the cloud computing platform, the mobile devices are enough powerful and we do not need to stress particular requirements inside a large-scale system.

Starting from this general list and by delving into a more detailed explanation, the different solutions focus on the most relevant challanges of mobile devices applications and propose how to face the issues related to each category of solution and at different level of the stack, by focusing more on the protocols, on the infrastructure, or on the final user application.

Chapter 3 proposes fog computing-based solutions that address multiple requirements of mobile devices following the fog computing concept. In particular, it faces the satisfaction of the following requirements, by proposing solutions both from protocol and from infrastructure perspectives. Scalability and performance requirements are provided by analyzing and modifying popular protocols and frameworks and combining them for the specific usage in pervasive environments. Resource usage is optimized, by minimizing the traffic generated both on middleware-mobile devices interface, by combining different communication protocols to use them in the best way, and on cloud-middleware interface, by highly increasing the autonomy of the middleware nodes that are able to perform standalone jobs and communicates with the cloud only in case of anomaly or out-of-scope activity. Finally, they specifically address the system customization, providing the configurability of the infrastructure and the applications to deploy, in order to be able to modify and update the infrastructure/applications in an easy way, also at runtime, without compromise the system/applications functionalities. Mobility is provided by migrating the application and the infrastructure from one node to another to guarantee the service continuity and by considering the possibility that mobile devices moving can leave or join the system without compromize the operations. Interoperability is addressed by using an abstraction of the specific mobile devices phisical specification provided by the communication protocols and by the infrastructure, i.e. through bundles or container-based virtualization. Finally, reliability is faced in order to recover from unexpected behavior, typical in mobile devices environments, without compromize the system functionalities.

Chapter 4 proposes mobile edge computing-based solutions that address the following mobile devices requirements, both from the applicative and the infrastructure perspectives. Mobility is provided by the ability to migrate the application from the mobile devices to the edge nodes and from one edge node to another one more powerful or better located to serve the mobile devices, also in case of execution in hostile environments. Similarly, scalability and performance requirements are carefully considered by the synergic migration of the application in case of high workload or to execute intensive tasks. Resource consumption is minimized by a fine-grained analysis of the resource usage on both the mobile devices and the edge node that indicates where is more feasible to run the application in terms of energy. The solution proposed are fully automated to decide dynamically, at runtime and without human intervention, which application (or part of it) migrate and where is more efficient to execute.

Chapter 5 proposes cloud computing-based solutions that address the following mobile devices requirements, from both applicative and the infrastructure perspectives, by using the cloud platform without the support of a middleware layer. Mobility is provided by extension of cloud computing techniques enabling dynamic network function and self-adaptation in order to ease the deployment and operations of mobile telco services through self-management, self-maintenance, on premise design and operations control functions. System performance and scalability are carefully considered to create dynamic cloud solution, by introducing the service state migration which allows to move all the infrastructure and data that compose a service, in order to reactivate the service on the destination location with the same state conditions of the origin service. Finally, interoperability is deeply analyzed by introducing a semantic-based approach, based on a federation web service, to overcome specific organization complexity and to hide heterogeneity and distribution of data sources.

Let note that multiple solutions, independently from the architecture used, propose migrations, i.e. of the application or of components of the infrastructure, between different nodes, both in a proactive and reactive way. This is mainly related to the need to provide system operations continuity, as well as performances, that cause to migrate system functionalities and applications from one edge node (or cloud host) to another, to serve the mobile services from the most suitable and adapt nodes/host. This is particularly true in case of middleware usage where edge nodes usually have limited resources avaialble, thus may need further computation ability in high workload contexts and, in addition, can highly improve the tasks execution by exploiting the location-awareness requirement.

3 | Scalability and Containerization for Fog Computing

This chapter presents two solutions, that face some common and relevant challenges described previously, with the purpose to significantly push forward the realization of efficient, effective and dynamic fog computing infrastructure. In fact, it is manifest the growing research and industrial interest in scalable solutions for the efficient integration of large amounts of deployed IoT sensors and actuators and cloud-hosted virtualized resources for elastic storage and processing. Such relevant attention is also demonstrated by the emergence of interesting IoT-cloud platforms and protocols from industry and open-source communities as well as by the flourishing research area of fog computing, where decentralized virtual resources at edge nodes can support enhanced scalability and reduced latency via locality-based optimizations. Many works have been proposed lately, to improve the scalability at different level of the stack, e.g. [60, 61, 62, 63, 64] but, at the same time, the scalability is still in its infancy with many existing IoT frameworks (targeting both the academic and industrial communities) that still do not address the scalability requirements properly and adequately. To this purpose, the first solution will focus on the communication optimizations that enable the creation of a scalable infrastructure able to guarantee high performance also in high workload conditions, by using IoT framework optimizations and the interwork of popular communications protocols.

In addition, the second solution of this chapter investigates more the possibility to deeply innovate the future fog computing infrastructure proposals, alleviating most of the efforts related to the implementation and deployment of fog computing solutions. In fact, the solution introduces a highly manageable and interoperable way to create fog nodes on-the-fly via the adoption of containerization techniques, whose advantages are deemed prevalent to disadvantages also in the case of IoT gateways with limited resource availability.

Both solutions aims to address the most influential and popular properties related to fog computing, that can enable fog computing to be applied in a widespread way, also in industrial contexts.

3.1 Related Work

3.1.1 IoT Federation

Some research activities in the literature have recently proposed architectures and solutions to address IoT sensors/actuators federation, in particular within large-scale deployment scenarios, with particular focus on how to effectively and efficiently achieve scalability. Even if they explored relevant solution guidelines and were someway inspiring for the community of researchers in the field, they only partially solved some aspects related to this hard technical challenge, thus leaving still open space for additional, especially industry-mature, solutions.

[65] proposes an architecture based on the base protocol that provides a generic self-organized Peer-to-Peer (P2P) overlay network service, by using CoAP as the transfer protocol to access RESTful services. Here CoAP is used as the common application layer protocol for heterogeneous and constrained devices, while RELOAD is employed primarily on proxy nodes with higher amount of resources. This work is also relevant to point out the relevance of open standard-based solutions for interoperability and the unsuitability of CoAP alone, e.g., because it is inappropriate to scalably aggregate IoT resources into a hierarchical organization.

[66] proposes a 3-layers architecture for cloud-integrated IoT services based on CoAP. The work proposes a Web integration platform that collects data from IoT devices and can outperform high-performance general-purpose Web servers that are not optimized for the IoT. [66] uses the Californium framework for CoAP integration, by showing its good throughput performance also in high-concurrency situations. Along with CoAP, the solution in [66] employs multi-threaded sockets, but shows that this implementation option is very resource-demanding while growing the number of clients, in particular in terms of memory.

Instead, [67, 68] mainly highlight successful examples of MQTT usage in some application domains. In particular, [67] proposes an architecture for adaptive periodic many-to-one communication in large-scale cyber-physical systems. The work shows how MQTT can guarantee good flexibility in highly dynamic execution environments where publishers and subscribers can join/leave frequently. [68] reports the experience of a smart home management system where MQTT is used over a hierarchical architecture similar to ours, but with no integrated exploitation of CoAP.

3.1.2 Containers

Some work in the literature highlight the necessity to introduce virtualization techniques on the middleware layer in order to face several challenges. [69] highlights the importance to provide flexibility and isolated environments. [70] underlines the challenge of fast mobility that imposes on mobile devices platform to provide and keep the compute and storage resources close to the devices to be able to rapidly reconfigure the switching context, and seamlessly orchestrate the allocation of resources and the migration of the state to the new location.

Since container-based services and applications share their underlying OS, the associated deployments are significantly smaller in size than VM-oriented hypervisor deployments, thus making it possible to store hundreds of containers on a physical host, as well as restarting a container without rebooting the OS, which is very relevant in several application domains [71]. [72] highlights how containers, differently from VMs, are more flexible for packaging, delivering, and orchestrating both software infrastructure services and application-level components, i.e. typically for tasks performed by a PaaS.

Although resource virtualization techniques based on containers offer a promising approach for middleware solutions since they have the potential to enable lightweight migrations, i.e. just the container state, unfortunately, the work on this topic focus on theoretical analysis so far rather than propose practical effective and novel solutions. For example, [72] is limited to the study of the container deployment performance analyzing the standard Docker pull operation. Similarly, [73] proposes the design of programmable CPS by deploying containerized applications that run close to the devices on fog nodes, by deploying programmable nodes capable of inter-node P2P communication and services orchestration from a centralized control instance.

Among the very few work that address, in a practical way, the virtualization of applications on the middleware layer, [74] proposes a solution that relies on LXC containers, which provide exibility and lightweightness if compared with traditional virtual machines, that are orchestrated with OpenStack. Although this approach is valuable, it requires more powerful nodes able to run OpenStack components, i.e. nova and cinder, partially loosing the benefits related to the containerizations and requiring a master node very powerful.

On the contrary, the container-based solution, proposed in this chapter, aims to promote a further step of advancement of the research in the field by giving the opportunity of exploiting container-based virtualization on top of IoT gateways. In fact, it explains the design, implementation, and experimental results of an innovative middleware solution, complete of a full infrastructure support (i.e. download, update, and management of virtualized images), based on Docker containerization over resource-limited RaspberryPi devices at the edge of the network. To the best of our knowledge, this is one of the first cases of implementation and experimentation of virtualization techniques over fog nodes, in particular while working with IoT gateways with very limited resources, such as RaspberryPi nodes.

3.2 Scalable IoT-Cloud Interactions

This solution describes a fog computing architecture, based on a wide hierarchy that manages a large-scale environment composed of tiny and constrained devices.

The purpose is to greatly improve the management of a large fog computing infrastructure, facing current and typical challenges and open points, mainly related to: communication efficiency, handling a high number of data exchanged from and to IoT devices and among other nodes into the hierarchy, with minimal amount of messages; scalability, to enable the infrastructure to manage workload peaks keeping the high performance; interoperability, that allows different typology of mobile devices to use the hierarchy.

An innovative distributed architecture, combining M2M industry-mature protocols, i.e., MQTT and CoAP, is proposed in an original way to enhance the scalability of gateways for the efficient IoT-cloud integration. The combined and synergic integration of such emerging standard protocols of wide industrial interest, which are recognized to be generally lightweight, flexible, asynchronous, and secure are exploited when efficiently integrated with Secure Sockets Layer (SSL) and Datagram Transport Layer Security (DTLS) mechanisms.

This research topic is relevant because there is no solution yet in the related literature that has adopted a gateway-oriented architecture where gateways jointly exploit MQTT and CoAP to achieve highly scalable IoT device management through dynamic hierarchical tree organizations.

3.2.1 M2M Communications

MQTT

Message Queue Telemetry Transport (MQTT) [75] is a lightweight message-oriented protocol following the publish/subscribe model. It allows achieving good scalability and can dynamically support a wide range of applications, especially in the IoT and M2M domains. Every MQTT resource is modeled as a client and can connect to an MQTT broker over TCP. MQTT has intrinsic characteristics that make it a valuable option in IoT environments with low-latency, low-bandwidth, and power efficiency requirements, e.g. small header overhead, topic-oriented management, and automatic message forwarding when clients reconnect.

MQTT uses hierarchical topics and nodes may subscribe to a topic and then observe the whole hierarchy by using wildcards. In addition, MQTT provides communication reliability, if needed, by enabling secure transmission for hierarchy control messages: reliable communications exploit TCP and differentiated QoS levels. The TCP handshake to the public broker address and the persistent connection establishment allow reaching hosts behind Network Address Translation (NAT) without problems, a common configuration when dealing with real IoT devices in many practical deployment environments. Moreover, it provides reliability with the possibility to dynamically select one of three QoS level options for message delivery:

- QoS0 - At most once semantics, with the best-effort mode of the network. The delivery is not acknowledged and the message is not stored. The message could be lost or duplicate. It is the fastest mode of transferring messages.

- QoS1 - At least once. Each message might be delivered multiple times, with possible duplications, if failures occur before an acknowledgment is received by the sender. With this option, messages must be stored locally at the sender, until they have been delivered to their receiver, so to enable possible retransmissions.

- QoS2 - Exactly once, with the guarantee that no duplication of messages occurs. It extends QoS1, by storing messages also at receivers in order to avoid any duplications.

Finally, MQTT allows every node to register a broker-side message (Last Will and Testament) that the broker sends to all subscribed clients on the topic, when the node disconnects unexpectedly. This provides a basic automated mechanism for the monitoring and management of disconnections in highly dynamic IoT environments.

CoAP

Constrained Application Protocol (CoAP) [76] is a document-transfer protocol optimized for M2M communications between constrained devices. CoAP provides a request/response interaction model, exchanging small messages on the UDP transport layer. It is suitable for constrained nodes and constrained low-power and lossy networks: it uses a short fixed-length binary header of 4 bytes, possibly followed by compact binary options and a payload [77]. In particular, CoAP allows i) adding UDP support for unicast and multicast requests with optional reliability, ii) exchanging asynchronous connection-less datagrams, iii) low header overhead on packets, iv) low complexity of parsing operations, and v) supporting simple mechanisms for proxy and caching actions. About reliability, CoAP supports four options in terms of different types of messages, namely Confirmable, Non-confirmable, Acknowledgement, Reset.

CoAP works asynchronously over UDP, which is connection-less, with higher performance, smaller packets, and reduced overhead. In fact, it starts to be recognized that CoAP characteristics make it specifically suitable for low-energy consumption application domains [77].

From the implementation perspective, Californium framework is used to introduce the CoAP protocol functionalities into the fog infrastructure and to combine it with the MQTT protocol. Californium [66] is an open source framework, written in Java, that implements CoAP functionalities and provides web service RESTful API for web-like IoT mashups. Californium focus on service scalability with a flexible concurrency model for the implementation of large-scale IoT applications and provides implementations for:

- Observe draft [78], a CoAP extension that enables CoAP clients to retrieve a representation of a resource and keep this representation updated.
- Blockwise transfers draft [79], a CoAP extension for transferring blocks of information to handle block transfer separately and, thus, truly stateless behavior.
- Resource directory draft [80], a CoAP extension called Resource Directory (RD). All the results are returned with CoRE Link Format [81] payload.
- Datagram Transport Layer Security (DTLS) protoco [82], to provide privacy for datagram protocols and security, based on the Transport Layer Security (TLS).
- CoAP-HTTP cross-proxy support, through HttpCore-NIO [83] and Guava [84].

Californium has been used as CoAP-support for multiple reasons. It is currently the most complete implementation of CoAP protocol, with many drafts extensions support. It is easily integrable with Kura and extensible with third-parties Java libraries to further improve the implementation of some functionalities. It is easily extensible with different types of usage to create also complex functionalities, in relation to the purpose.

MQTT-CoAP Combination

MQTT and CoAP are protocols both open, lightweight, and typically used in constrained environments but with different characteristics. Both protocols, due to their intrinsic composition and implementation, include advantages and disadvantages in relation to the different contexts of application.

In particular, MQTT thanks to the usage of the publish/subscribe model and the wildcards, perfectly fits the application into a hierarchical organization and can be used to dynamically organize an infrastructure to efficiently manage large scale via hierarcFhy- and locality-oriented optimizations.

Notwithstanding its many advantages, MQTT also presents some drawbacks in its pervasive IoT employment, mainly related to the usage of TCP transport protocol in an IoT environment, due to their different purposes of communication that combine a reliable communication medium in a very unstable and unreliable environment. In fact, TCP is a reliable connection-oriented protocol that establish the connection, using a multi-step handshake procedure, and maintain a connection until the exchanging messages are finished and assures that the messages have been delivered with the acknowledgement messages. On the other side, IoT environment is typically based on tiny devices that sense everything from the surrounded environment, including a high quantity of noise or useless data, with limited capabilities that, thus, cannot process information locally and send all the data towards the edge layer, with a high sampling rate. In this context, MQTT it is more suitable for resource-full nodes than for constrained IoT devices, due to: the multi-step handshake needed every time new nodes are establishing a TCP connection in a devices-crowded environment; the TCP connections kept always open, with the management of a persistent node-broker session, thus often generating useless resource consumption, in particular at IoT endpoints. To partially solve this issues, a possible solution may involve to set the MQTT clean session flag on the broker to choose to either keep the connection open or remove it after the transmission. Unfortunately, this approach is not usually viable in practical IoT environments because not using TCP persistent sockets means having to re-establish connections through the multi-step handshake every time an endpoint sends data. That would lead to unacceptable performance decrease in several application scenarios due to more data traffic, decrease of battery-operated life, also given the usual high sample rate of IoT endpoints.

Therefore, an MQTT standalone exploitation is not enough and CoAP is the right protocol to complement MQTT in order to overcome its limitations when applied to large-scale IoT deployment scenarios. In particular, for IoT endpoints locally connected to a node, MQTT introduces overhead for functions that are not crucial: using a less reliable but also less resource-demanding connection is often more suitable, especially if the sporadic loss of partial data is not critically impacting the overall system behavior, as it regularly occurs for cyber-physical monitoring scenarios.

The MQTT-oriented communication has been extended with more lightweight CoAP-based coordination functionality, thus achieving scalable interactions, especially in the challenging case of node/resource discovery. Thus, MQTT is exploited for inter-node communications between node equipped with CoAP servers in order to retrieve resource information and for node synchronization in our tree management procedures. The MQTT publish/subscribe model allows reaching multiple nodes by limiting the overall number of exchanged messages, thus achieving good performance also when our hierarchy consists of a large number of nodes. Publishers and subscribers are completely decoupled from each other thanks to the usage of the MQTT broker, thus simplifying the addition of new nodes and making any node able to seamlessly interact with the others. In our solution, each participant node can publish messages independently to the receiver nodes state and the broker may send messages to nodes when they turn active; also in the case a subscriber is performing a non-interruptible operation, the broker can queue messages to be processed later (temporal decoupling). Moreover, nodes can exchange messages without any prior knowledge of each other (spatial decoupling).

Along with MQTT, it is used CoAP for interactions where it is needed direct and very responsive lightweight communications, with low reliability constraints, i.e., between the CoAP server and the IoT endpoints due to the high data exchange rate between them. CoAP allows achieving these goals in single localities, where NAT drawbacks are ineffective and communication reliability is anyway adequate. Vice versa, although CoAP may be a standard-base for scalable architecture, as already stated, it has some limitations stemming from UDP usage, lack of advanced quality support, and NAT tunneling and port forwarding. In addition, [85] underlines the unsuitableness of using CoAP alone because it does not allow to aggregate IoT resources into a hierarchical organization. For these reasons, a

CoAP-based solution has been integrated with MQTT, as detailed in the following parts of the paper, thus complementing the functions and strengths of both protocols.

In particular, by following the seminal work in [77], CoAP has been exploited to easily enable IoT devices to be discovered and to expose them externally as resources accessible with REST-call methods, using the /.well-known/core URIs [86] with resource descriptions in the CoRE link format. By delving into finer technical details, each CoAP endpoint has associated the CoRE link attributes, which extend those in Web Linking [87], e.g. resource type (rt), interface description (if), content-type (ct), maximum estimated size (sz) [81]. This allows clients to discover which resources are provided and their associated information before accessing them.

3.2.2 Kura Gateway-based Architecture

Kura Overview

Kura [88] is an open-source framework for IoT applications that provides a platform for building IoT gateways. In particular, it abstracts the design and implementation of gateways from the complexity of real-world industrial scenarios consisting of heterogeneous hardware/network devices. To this purpose, Kura aggregates and controls device information, as well as it supports the simplification of the overall development and deployment process. Kura is based on Java OSGi to support, in a widely accepted way, dynamic management of software components with no need of operation suspension, to simplify the process of writing reusable software building blocks and to create self-contained pluggable packages (i.e., bundles) specifically suitable for IoT applications. In particular, Kura has been employed as implementation basis because it can serve as a suitable container for machine-to-machine applications running in service gateways. Kura uses MQTT as its central protocol: it provides different features for message publication towards MQTT brokers and the subscription to specific broker-supported topics. In addition, Kura components can support routing functionality, by managing wired/wireless communications and by allowing Virtual Private Networking (VPN) connections, firewall usage, and NAT operations. Kura APIs offer easy access to the underlying hardware including serial ports, GPS, watchdogs, USBs, gpio-s, etc. Finally, Kura is open-source, with a fervent and growing community supporting the initiative and continuously proposing extensions and innovative Kura-based solutions, thus guaranteeing maintenance/evolution support for the years to come (very relevant for supports and applications of industrial interest). Figure 3.1 shows the base IoT-cloud architecture pushed by the existing Kura framework, with the regular way of using Kura in such an architecture.

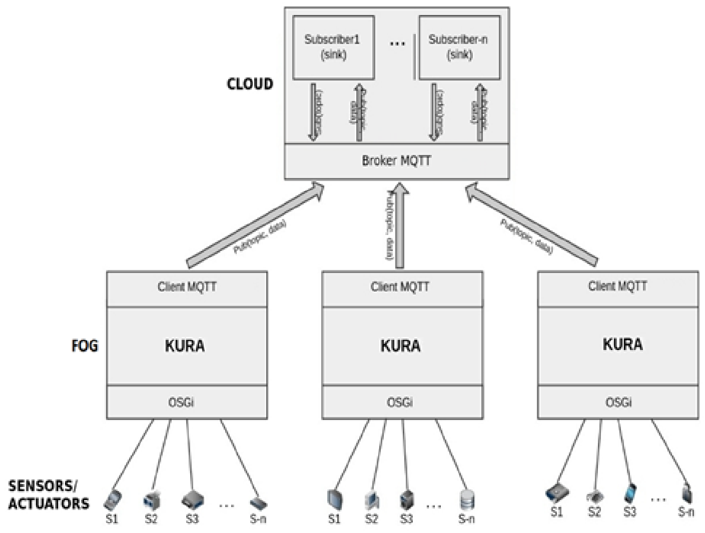


Figure 3.1: The Proposed Cloud-Fog-IoT Reference Architecture

Kura Extension Towards Fog Computing Platform

The proposed solution extends, and has been designed and implemented on top of, the Kura framework, a platform for IoT-cloud integration of strong industrial relevance, and on the role of infrastructure gateways for efficient message brokering [89, 90, 91]. In particular, the proposal aims to introduce a fog computing infrastructure that can constitute a basis for further solutions to improve scalability and reduce latency for communication/coordination among wide-scale sets of geographically distributed IoT devices interworking via gateways.

By delving into finer technical details, by referring to the default Kura architecture, this research work hoghloghts some non-negligible weaknesses that motivate the need for some relevant extensions, at least in the perspective of efficient fog computing exploitation:

- **Single MQTT broker on the cloud.** Kura allows to communicate only with a single broker located on the cloud and, thus, tends to send all the collected data from IoT devices to that broker. This raises non-negligible concerns, such as i) the single broker deployment option may cause a performance slowdown in case of high load, enabling only to deploy small applications with a limited number of devices or with very limited sensor sampling rates; ii) Kura gateways can only dispatch sensed information towards the cloud, with no fog-oriented processing operation performed locally; and iii) Kura exploits persistent sockets to connect its gateways to the integrated cloud, with a non-negligible waste of resources in case of intermittent/interrupted communications or of sporadic interaction.

- **Flat topology.** Gateways are usually hosted on hardware equipment with a limited amount of resources, hence a single gateway or a set of gateways organized in a flat topology can perform only relatively limited operations, i.e., processing, storage, inference, etc., which might be insufficient for several application domains of interest and in many envisioned IoT applications.

Gateway-side MQTT Brokers

This sub-sections tries to concisely illustrate how Kura can be extended to overcome the above weaknesses, thus making it more suitable as a basic building block for fog computing infrastructures to support three-layer cloud-integrated architectures for IoT and consequently wide-scale IoT applications.

The first relevant extension is towards the inclusion of an MQTT broker, e.g., Mosquitto [20], on each gateway, as depicted in Figure 3.2, in order to collect sensed information at the gateway side and, at a later stage, possibly after local processing and inferencing, to send filtered/processed/aggregated data to the cloud.

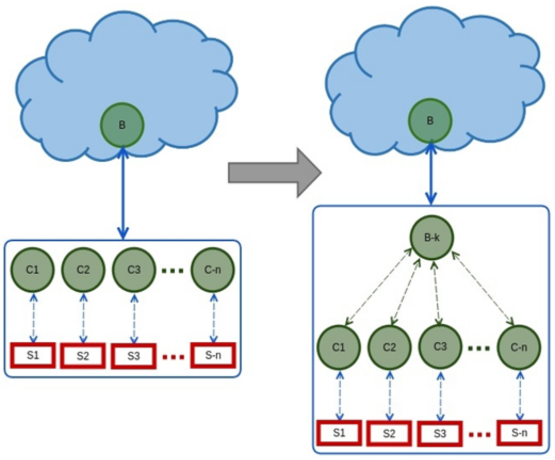


Figure 3.2: Adding Gateway-side MQTT brokers

The main advantages deriving from this extension, in the fog computing perspective, are:

- **Enabling hierarchical topologies.** The internal topology turns, from a flat structure where all the clients can send data to the cloud-side broker, to a hierarchical structure where the gateway-side broker can serve as a local root. Further extensions of the hierarchy can consider using multiple brokers or more client/broker levels to further strengthen the before-the-cloud processing capabilities. This potential augmentation of the infrastructure should be dynamically fitted, however, against the currently available resources at the IoT gateway side, thus further pushing for the opportunity of advanced dynamic deployment support.

- **Gateway-level MQTT message aggregation.** It is possible to perform and implement typical fog computing functionality on the gateway, e.g., basic data aggregation and filtering actions, Small Data processing [48], and alike. The proposed extension still exploits the standard MQTT protocol, as the original Kura framework, but in two following separated segments, i.e., sensors-to-gateway and gateway-to-cloud-hosted brokers.

- **Real-time message delivery and reactions.** The introduced ability of the extended IoT gateways to collect data and perform actions can significantly speed up system reactions. Reactions with hard/soft real-time constraints may then be delegated to IoT gateway components closer to the edges of the network, with improvements in time and quality of reactions (fresher sensed data).

- **Actuation capacity and message priorities.** A peculiar ability of fog computing is to dynamically determine the situations when it is necessary an immediate actuation or when it is possible to send data to the cloud for intense postponed analytics. Each IoT application, in relation to the context and the type of information sensed from the environment where it is immersed, must define sets of messages with the relative priority and consequently differentiating the type and the promptness of reactions, as well as its desired level of fog-cloud interplay. For instance, in smart city-oriented vehicular applications, cloud analytics are particularly useful and tight coupled outside vehicle endpoints for the evaluation of traffic patterns and to detect best ways to reach destinations. However, this is limitedly related to operations inside vehicles (intra-vehicle applications) where fog nodes have to deal more frequently with real-time actions.

- **Locality awareness and locality-oriented optimizations.** The information sensed by IoT devices are processed by each gateway in its locality and thus with a usually more complete knowledge of the local environment from which the sensed data are generated.

- **Gateway-cloud connection optimization.** The cloud-side broker is no more required to stay alive at any time and to continuously receive sensed data collected from each part of the overall deployment environment. Given that it receives data after that gateways have performed processing/aggregation/filtering/etc. operations over them, the usage of persistent sockets is often not required and inefficient (persistent sockets are the most widespread communication mechanism in available MQTT brokers). Therefore, in the proposed extension, non-persistent sockets have been exploited, which are dynamically established only when necessary, to exchange data between the cloud-hosted broker and the lower-level brokers at IoT gateways.

Enabling Cluster/Mesh Topologies for Kura Gateways

Another significant IoT gateway extension, originally proposed, relates to virtually strengthening, in a dynamic way, the available gateway resources via the combination of multiple physical gateways and the aggregation of their resources. For example, Figure 3.3 depicts a possible cluster-oriented physical topology for Kura gateways realizing a virtual and strengthened higher-layer gateway, while Figure 3.4 shows a similar concept through virtualization of an underlying mesh topology. In fact, the support to a more powerful intermediate layer, based on either cluster or mesh topologies, is demonstrating to be central to enable scalability in large IoT applications.

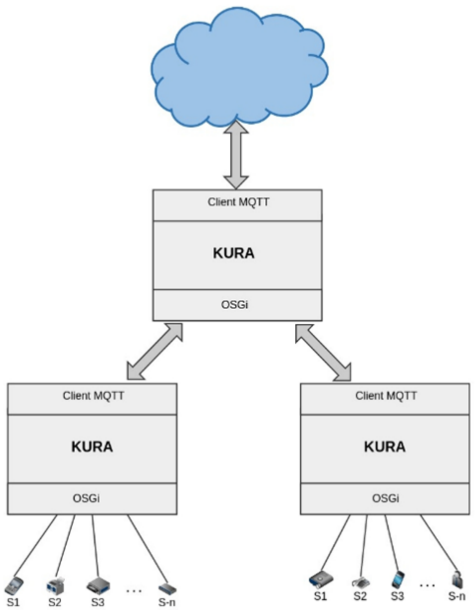


Figure 3.3: The Supported Cluster Organization of IoT Gateways

The most significant advantages associated with full and seamless

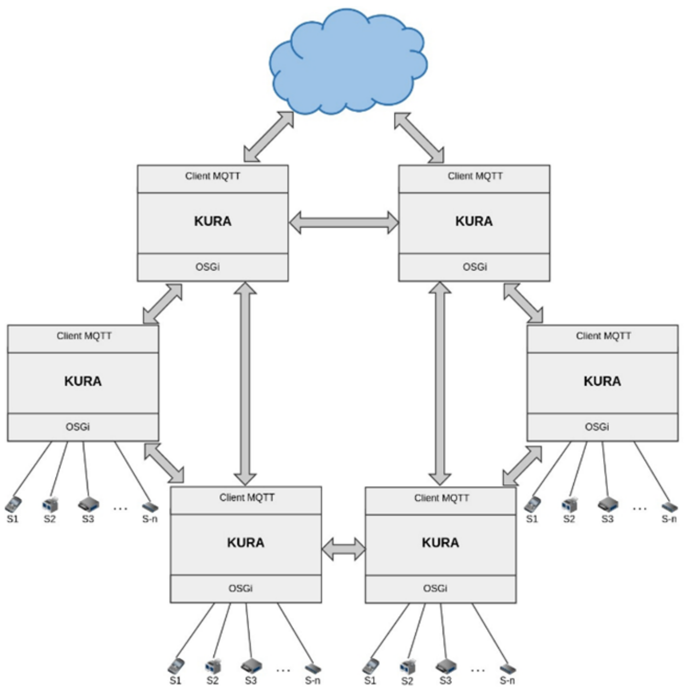


Figure 3.4: The Supported Mesh Organization of IoT Gateways

support to cluster/mesh organizations of IoT gateways are:

- **Kura gateway specialization.** The cluster/mesh topology consists of different gateways with different properties and each gateway is more suitable to perform some tasks or to cover some functionalities rather than others. For example, some gateways may be deployed to be more specialized to aggregate data, others for context-aware processing/filtering operations, others for locality-efficient data storage and indexing, and so on.

- **Locality exploitation and data quality.** Locality-based optimizations can be improved by performing more accurate and complex analytics and by taking advantage of larger resource availability.

- **Geo-distribution** is another feature whose performance/effectiveness advantages depend on the number of interworking gateways (and their total amount of available resources). With the increased number of cluster/mesh-organized gateways it is possible to manage dense sensor localities and to make the overall distributed deployment scale better.

- **Scalability.** Cluster/mesh of gateways can scale more easily, according with the total amount of resources available, also by facilitating the realization of load balancing operations and dynamic failover management.

- **Security and privacy.** Security and privacy improvements are connected to locality-based operations and full locality visibility, which allow having more complete and accurate knowledge of the environment where gateways are operating, towards the creation of more efficient support features. At the same time, the proximity between environment and data elaboration places, partially moved to the layers of gateways, can improve privacy and simplify decentralized ownership management.

Tree-based Hierarchy Architecture

Based on the fog computing infrastructure obtained from the Kura extension, this solution adopts a dynamic tree structure organization, composed of many gateways and IoT devices that exploit MQTT and CoAP in a combined way to improve the efficiency of i) node communications for efficient resource look-up; ii) IoT device discoverability; iii) resilience to device disappearance or unexpected disconnection; and iv) dynamic and lightweight hierarchy reconfiguration, triggered by ii) and iii).

In particular, the proposed CoAP-based gateway-oriented distributed architecture is sustained by five support services, each one implemented in terms of an OSGi bundle. Every node is suited by the Kura framework that executes the bundles. The service base bundle is used to provide libraries to external bundles or to retrieve information about network addresses currently used by the node. The bundle contains a CoAP server, using UDP to have full access to the whole functionality of the open-source and widely adopted CoAP Californium framework. It retrieves data from either local resources or remotely through the Remote Query Resource (RQ) service, and uses a Resource Directory (RD) data structure to store all the information about resources.

Externally, each node is connected with the others in a tree-structure hierarchy. The tree structure consists of multiple OSGi bundles replicated in each node (with the associated RDs) and is based on two main components: CoAPTreeHandler (CTH) that manages all the CoAP servers hierarchy; MQTT broker, to exchange both resource and control messages to synchronize CoAP servers. Figure 3.5 depicts our high-level architecture proposal.

CoAP servers need to exchange messages for resource requests and sometimes, if needed, they update the hierarchy knowledge by disseminating information to all involved nodes. It is a multi-layers hierarchy and each node specifies a domain name and a group, or sub-group if necessary, name that identify the node into the hierarchy. Thus, it is possible to limit the interest towards external resources

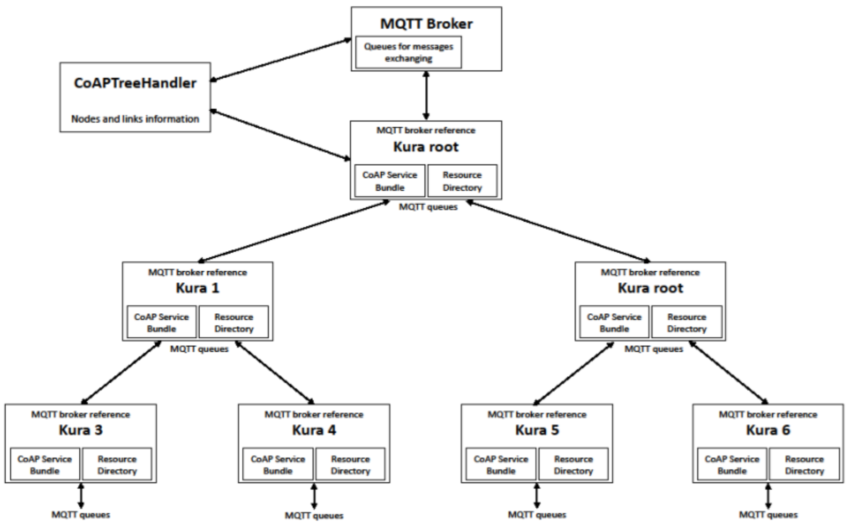


Figure 3.5: Architecture Integrating MQTT and CoAP for Better Scalability

to store into the RD, e.g., only to a specific sub-group or domain, thus making the message exchange between brokers more efficient. As depicted in Figure 3.6, the hierarchy is divided into three levels:

- **Level 0** includes only the root node and enables inter-localities communications. Every message sent to a different locality passes through the root node.
- **Level 1**includes all the nodes belonging to a specific locality. It permits quicker update than level2 and receive updates directly from CTH. If the root node is added/removed, level1 updates level2 nodes when they subscribe to the related topics, without any private additional communication by CTH.
- **Level 2** includes all the nodes of a given locality and of a given group. It might also be divided into specific subgroups. It has less temporal constraints than the other levels and, in case of parent disconnection, it receives periodically updates by level1 with a private communication.

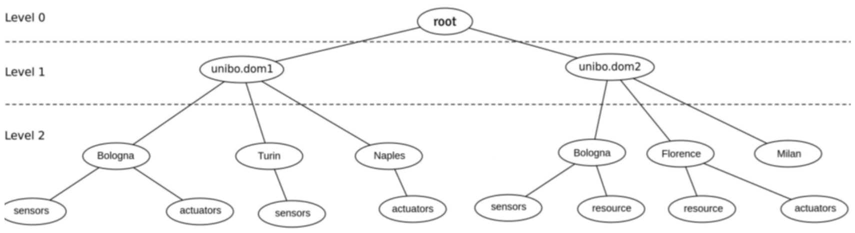


Figure 3.6: The Hierarchical Tree Structure of our Extended Kura Gateways.

Implementation Insights

This tree-based resource management solution is composed of tree-based hierarchy management components that connect the different nodes in our dynamic hierarchy and, from the internal point of view, the nodes have a common internal structure replicated into all nodes. The CoAP-based extension of the Kura IoT gateway is modularly and flexibly implemented via OSGi bundles, executed by the Kura framework on each IoT node. In addition, Kura allows to exploit a native MQTT support to select differentiated QoS levels: exactly-once semantics can be accessed via the so-called QoS2 mode and it is used for most relevant hierarchy management messages; MQTT QoS1, instead, offers at-least-once semantics that it is adopted for more common inter-node communication, with overall performance benefits.

The management of our hierarchical tree is performed by MQTT broker bundle and CoAPTreeHandler (CTH) bundle. The MQTT Broker bundle is a simple extension of a regular MQTT broker to exchange messages between nodes within the hierarchy, with the implementation based on Mosquitto [92].

The CTH bundle dynamically manages the hierarchy and dispatch node requests: it provides hierarchy metadata to all nodes, returns the references to create a tree-structure organized with domains/-groups; manages multiple node replications; and handles children nodes in case of parent disconnections. Internally, CTH consists of CTHListener and CTHCollections. CTHListener exposes an interface to interact with CTH, triggered by request from nodes that require actions on the hierarchy. The CTHCollections class stores information about the overall hierarchy and consists of two collections optimized for lookup and node substitutions: a Map-extension data structure to store node paths and properties; a data structure to map parent-children associations by using the Guava library in order to simplify collection updates, i.e., dynamic creation of one-to-many associations. To practically exemplify how the hierarchy management works, here there are some common use cases:

- **New root request.** CTH returns the root reference and stores into CTHCollection the node address (root duplicates management). When root disconnects, CTHCollection contains references to suitable new root nodes to replace it and notify level1 nodes.

- **Node addition** (Figure 3.7). By providing its path, a node requests root information: if root is present, the node is notified with the root node reference (root_response); if root unavailable, the node is added as root (root_added) or updated (root_not_available). CTH checks the path provided: if new path, the node is added to the hierarchy; otherwise, the node becomes the child of the original node and will be able to perform resource lookup or return a local resource.

- **Node removal.** i) If the node to be removed is the root, CTHCollection provides the reference to the new root, otherwise CTH searches a new root on level1. ii) If leaf, CTH only updates the hierarchy information. iii) Otherwise (Figure 3.8), children are associated to the root while CTH looks for a new parent, then children are restored.

- **CTH failure.** During CTH inaccessibility the root is not aware of hierarchy updates. Once CTH reconnects, an emergency MQTT topic (cth_online) triggers a stateless recovery with hierarchy node soft-reboot: i) parents remove all properties attached; ii) children remove parents and resend the request to join the hierarchy; iii) resource collections are sent to the parents and to CTH to be aligned again.

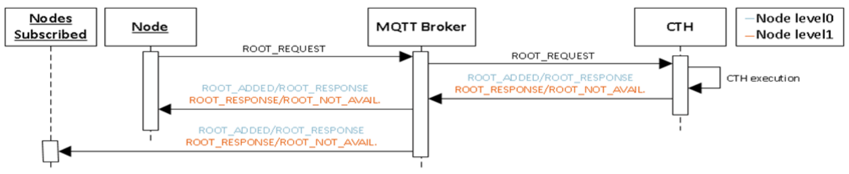


Figure 3.7: Node Connection Iterations

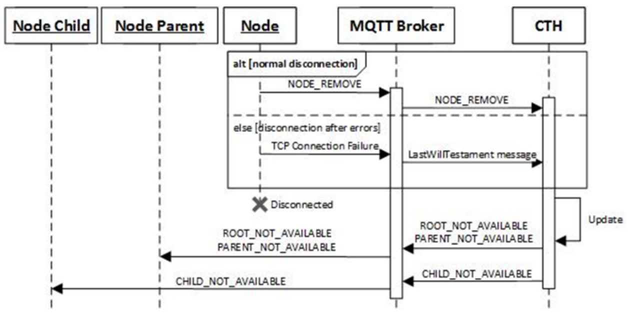


Figure 3.8: Node Removal Iterations

For the sake of modularity and loose coupling, any node is structured into multiple components:

- **CoAP Server** bundle is the central component within a node, manages all the properties of registered resources and uses Californium to offer REST API operations. In particular, can be coordinated with other servers to perform callback methods in relation to the received messages. It works locally on a node to create a high-performance CoAP support inside Kura, thus enabling the possibility to send CoAP REST requests to discoverable devices. The CoAP server implementation is provided of a DTLS support, using the existing Scandium [93] subproject inside Californium, to protect and limit the access on the resources, in particular on actuators that can modify the environment, allowing the execution of REST operations (e.g. POST, PUT, DELETE) only to authenticated users.

- **Resource Directory** (RD) bundle is a data structure used to store endpoints and resources belonging to different domains, groups, or subgroups. It supports usual discovery operations, e.g., to register, maintain, lookup, remove endpoints and resource descriptions. It includes an automated support to node disconnection management: if an endpoint does not update the RD periodically, it is automatically removed.

- **Remote Query resource** (RQ) bundle supports resource lookup by CoAP servers in the hierarchy, thus allowing nodes to retrieve information from CoAP servers by interfacing CoAP requests with MQTT brokers, with CoAPPublisher and CoAPSubscriber classes. The resource lookup is performed by a set of iterative requests towards other hierarchy nodes, starting from the parent node, that reply with the direct link to the resource or with a list of CoAP servers that may contain the resource for further search (see Figure 3.9). RQ and CoAP servers are

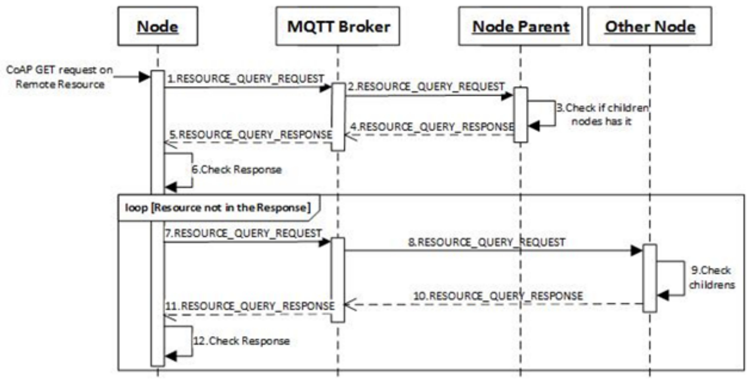

Node
MQTT Broker
Node Parent
Other Node
CoAP GET request on Remote Resource
1.RESOURCE_QUERY_REQUEST
2.RESOURCE_QUERY_REQUEST
3.Check if children nodes has it
4.RESOURCE_QUERY_RESPONSE
5.RESOURCE_QUERY_RESPONSE
6.Check Response
loop [Resource not in the Response]
7.RESOURCE_QUERY_REQUEST
8.RESOURCE_QUERY_REQUEST
9.Check childrens
10.RESOURCE_QUERY_RESPONSE
11.RESOURCE_QUERY_RESPONSE
12.Check Response

Figure 3.9: Resource Lookup Iterations

completely decoupled to enable servers to call RQ only in case of queries on remote resources; for local resources, they can exploit optimized and direct CoAP requests with no RQ interaction.

- **CoAP Resource** (RQ) bundle is a service dedicated to the external exposition of discoverable resources. It can add, delete, or modify the exposed attributes.

MQTT and CoAP Optimization

Although Kura and Californium are widespread and industry-mature frameworks, their default configurations have demonstrated to be not specifically optimized for the hierarchical IoT gateway organization. Thus, introducing some non-negligible implementation/configuration improvements is necessary, focusing on the following primary critical scalability aspects: reducing the number of MQTT messages exchanged among levels and nodes; reducing the number of POST requests to RD.

Object Serialization (OS). MQTT is content-less and only support byte array as payload content. Thus, serialization is performed very often during regular operation: each CoAP message is serialized, then used into the Kura DataService class for MQTT communication, and de-serialized during responses. Performance have been improved by integrating serialization based on the efficient Kryo [94] framework. Kryo is an open-source object graph serialization framework for Java language that provides performance, efficiency and API easy to use for multiple serializations, outperforming normal Java serialization.

DataService (DS) and DataTransportService. The DataService class is the Kura component used to manage MQTT communication and offers several configuration options, delegating to the DataTransportService the implementation of the transport protocol to interact with the associated MQTT broker. When DataService receives a publish request, it stores the message into a DataStore and submits the message on the internal executor of DataTransportService. DataStore is a heavy storage structure that, in case of many messages published concurrently, may even cause unavailability of the device because of its high CPU consumption. Therefore, to reduce associated overhead, it is been exploited only the lower-level DataTransportService class, by removing the support to the features, irrelevant for the proposed solution, of message priority management and message storage on behalf of temporarily disconnected devices. In addition, the DataTransportService performs event propagation by accessing methods and client listeners of the MQTT Paho implementation [95].

The experimental work has pointed out that the default configuration of the Californium framework exhibits some performance drawbacks for highly scalable scenarios, in particular in terms of efficient message parsing. The related aspects have been optimized by providing original extensions for i) efficient data structures for CoAP payload parsing, ii) efficient support to most frequent regular expressions to identify message sections, and iii) exploitation of message setting patterns available into the CoRE Link Format.

RD Parsing (RP) and String Refactoring (SR). To optimize regular expression management and to speed up parsing, any CoRE link has been represented with resource path in the first position of the RD attribute list (fixed static position). In addition, the adoption of Guava [84] libraries for efficient string management can optimize the solution: for instance, by replacing the Java Scanner class with the specialized Guava Splitter or using the optimized StringBuilder for string declaration.

Requests Aggregation (RA). Since every node can send POST messages, it is likely that some nodes may send information about multiple resources. Thus, message management must be optimized whenever possible (in absence of strict latency requirements) by aggregating multiple resources together into a single CoRE link format composition.

3.2.3 Performance Evaluation

To assess and validate the feasibility of the tree-based IoT gateway organization proposed, several tests have been performed in realistic application/deployment scenarios, in order to qualitatively and quantitatively evaluate MQTT and CoAP, in particular in terms of performance and scalability. The performance evaluations have been performed on the two critical scalability elements of the solution: the number of MQTT messages exchanged and the POST requests performed on RD. In fact, every time a node needs to communicate, e.g., to manage the tree hierarchy or to retrieve an endpoint attribute, the solution exploits MQTT messages, whose transport is natively integrated in Kura. The number of MQTT messages has demonstrated to depend linearly on both the number of levels and the number of nodes in our hierarchy. The number of CoAP requests to RD is generally high as well, due to the central role of RD in our architecture for intra-node communications and endpoint attribute retrieval.

The tests use growing numbers of sequential MQTT requests and CoAP POST ones, distributed over a realistic testbed environment consisting of 10 nodes organized in a 4-layer tree, 20 service bundles per node, 2 devices per service bundle, and 10 sensors per device. Each regular (non-root) node is equipped with RaspberryPi 1 model B+, with Raspbian Whezzy, Kura version 1.1.1; Parallels VM, with 512 MB Ram, single core CPU, Xubuntu OS. The root node is 2.2 GHz Intel i7 CPU, 16 GB RAM, 1600 MHz DDR3. The integrated MQTT Broker is Mosquitto. The setup of the testbed deployment is shown in Figure 3.10.

Extensive tests, on MQTT and RD evaluation, have been performed by sending growing numbers of sequential MQTT requests

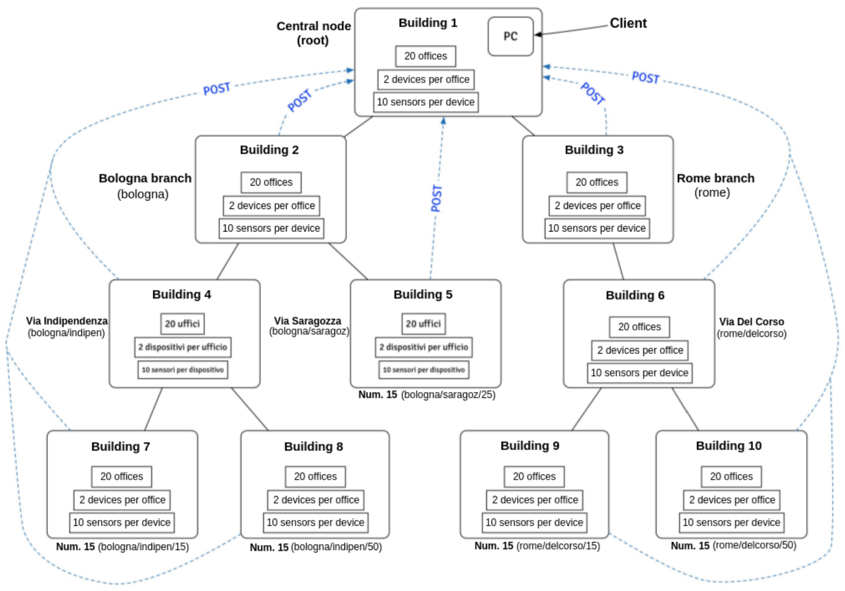


Figure 3.10: Testbed Scenario

and CoAP POST ones, with the purpose of evaluating the hierarchical solution under different load conditions. Starting by sending a peak of 1000 concurrent requests between two RaspberryPi, without any of the optimizations described above, the prototype is not able to handle them; exceptions related to too many messages stored and too many messages waiting the ACK are observed. Thanks to the object serialization optimization, it is possible to observe a significant decrease of the total time needed to serve all the requests (around 49%), but anyway this is still insufficient to manage such a large peak with no exception occurrences. By adding also our DataService optimization, tests show another significant performance improvement, as well as the ability not to enter in exception situations due to overload. After that preliminary experimentation, the behavior of the prototype has been evaluated while further increasing the number of MQTT exchanged messages. Table 3.1 reports the total time to complete MQTT transmissions in relation to the optimizations introduced. Note that, due to their limited capabilities and the high number of operations to perform, sometimes RaspberryPi nodes are affected by connection lost errors with the MQTT broker; this has shown to be prevalently due to the inability of sending MQTT heartbeat messages in time to keep the associated connection alive.

Table 3.1: MQTT Performance Results

Senser/Receiver	Opt.	MQTT Req.	Time	Errors
Rasp/Rasp	–	1000	211 s	KuraStoreCapacity Reached
Rasp/Rasp	OS – DS	1000	14 s	–
VM/Rasp	OS – DS	1000	5 s	–
VM/VM	OS – DS	1000	1 s	–
Rasp/Rasp	OS – DS	10000	120 s	Lost Connection
VM/Rasp	OS – DS	10000	60 s	–
VM/VM	OS – DS	10000	7 s	–
Rasp/Rasp	OS – DS	60000	723 s	Lost Connection
VM/Rasp	OS – DS	60000	377 s	–
VM/VM	OS – DS	60000	34 s	–

By passing to some relevant results about RD (CoAP-centered) evaluation, the test on RD started by sending 500 POST requests: already with this number of requests, the default Californium configuration has shown non-negligible performance issues; for instance, when working on top of a limited gateway such as the RaspberryPi one, this load peak may frequently generate errors. Therefore, the RD optimizations have been applied, also in different partial subsets; Table 3.3 reports the related performance results in terms of total time to complete the POST requests on RD, with the different possible subsets of optimizations introduced.

Table 3.3: RD Performance Results

Senser/Receiver	Opt.	POST Req.	Time	Errors
Rasp/Rasp	–	500	99 s	POST Timeout
VM/Rasp	–	500	12 s	–
VM/VM	–	500	6 s	–
Rasp/Rasp	RP	500	68 s	Possible POST Timeout
VM/Rasp	RP	500	7 s	–
VM/VM	RP	500	4 s	–
Rasp/Rasp	RP – RA	500	55 s	–
VM/Rasp	RP – RA	500	5 s	–
VM/VM	RP – RA	500	3 s	–
Rasp/Rasp	RP – RA – SR	500	19 s	–
VM/Rasp	RP – RA – SR	500	4 s	–
VM/VM	RP – RA – SR	500	0.8 s	–
Rasp/Rasp	RP – RA – SR	1000	51 s	–
VM/Rasp	RP – RA – SR	1000	11 s	–
VM/VM	RP – RA – SR	1000	1.8 s	–
Rasp/Rasp	RP – RA – SR	5000	255 s	POST timeout, not responsive
VM/Rasp	RP – RA – SR	5000	54 s	–
VM/VM	RP – RA – SR	5000	3.3 s	–
Rasp/Rasp	RP – RA – SR	10000	491 s	POST timeout, not responsive
VM/Rasp	RP – RA – SR	10000	104 s	–
VM/VM	RP – RA – SR	10000	6.9 s	–

Introducing the security DTLS security support, some tests have been provided on RD performance. DTLS may slow down the system in case of many new nodes, because every time a new node sends a message must perform the DTLS handshake to create the DTLS session. Figure 3.11 illustrates the results considering both normal iteration between Kura devices inside the hierarchy and, the worst case, about communications only from external independent clients.

In addition to the above results for communication performance, the solution has been thoroughly investigated in term of resource consumption on IoT gateways, which is crucial also to maintain full responsiveness of the participating nodes. CPU and RAM usage on RaspberryPi nodes, when 500 resources are registered (four descrip-

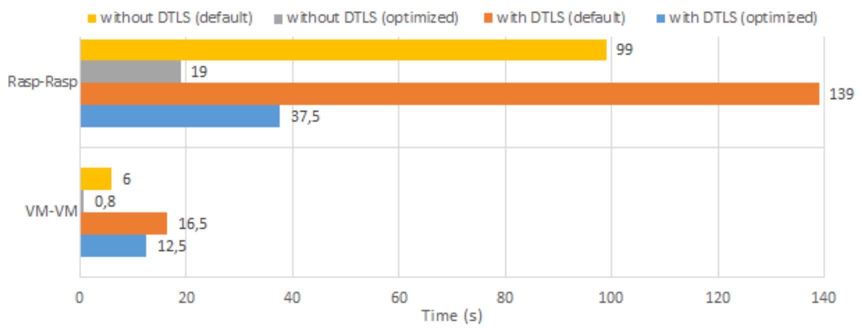


Figure 3.11: Performance RD with(out) DTLS

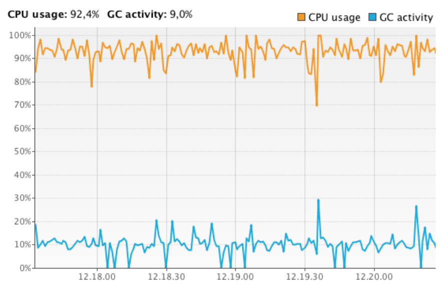


Figure 3.12: Initial CPU

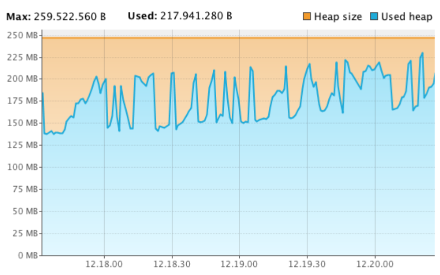


Figure 3.13: Initial Memory

tion attributes each, on average) via POST REST invocations, is respectively 92% and 80%, as shown in Figure 3.12 and 3.13, with around 550 active threads. In fact, every time RD stores a resource, the Californium-based CoAP implementation associates a new thread to it in order to monitor its validity time and delete the registration once it expires. With a high amount of concurrent requests, this multi-threaded support is clearly not adequate. Therefore, the associated multi-threaded support has been modified via an optimized set of worker threads, each of them handling a partition of resources (the cardinality of the partition is dynamically determined and can change over time self-adapting monitoring). In particular, registration update periodicity may vary depending on desired responsiveness and available resources at gateways; consider also that in the considered application scenario, a false positive in resource registrations only generates the waste of a useless lightweight CoAP request that will not receive any response. Figures 3.14 and 3.15 show the performance results about resource consumption in the thread-optimized case, with only 35 active threads on average.

3.3 Fog Deployment via Containerization

This section presents the enhancing of the fog computing middleware via dynamic IoT gateway configuration through i) the creation

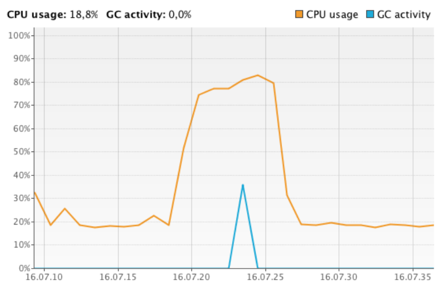


Figure 3.14: Final CPU

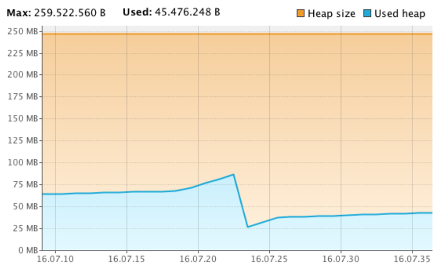


Figure 3.15: Final Memory

of standard gateway base configuration; ii) the creation of container-based (typically small and atomic) applications/services, each with very specific functions; and iii) the dynamic orchestration of fog middleware services by the global cloud, with the possibility to install, replace, or extend the currently installed configurations and available middleware services.

3.3.1 Containers Overview and Motivation

Containers are nowadays a very effective alternative solution to more traditional Virtual Machines (VMs), also allowing the deployment of virtualized resources on less powerful server hosts than in regular cloud computing [96], with relatively limited performance impact. Container-based virtualization is an industrially mature technology that provides real virtualization at the OS level, rather than a full OS on virtual hardware, with all the primary virtualized properties achievable via VMs, e.g., own network interfaces, own filesystem, isolation in terms of security and resource usage, but with much more lightweight operations. Since container-based services and applications share their underlying OS, the associated deployments are significantly smaller in size than VM-oriented hypervisor deployments, thus making it possible to store hundreds of containers on a physical host, as well as restarting a container without rebooting the OS, which is very relevant in several application domains [70]. [71] highlights how containers, differently from VMs, are more flexible for packaging, delivering, and orchestrating both software infrastructure services and application-level components, i.e. typically for tasks performed by a PaaS.

The solution is based on this general idea and perspective, and originally extends it to the applicability domain of fog computing middleware. In addition, containers provide an abstraction that makes each container a self-contained unit of computation [97], thus allowing easier portability and interoperability, with lightweight components and good suitability for distributed applications. Several tools provide the abstraction and implement the container-oriented models, e.g., LinuX Containers (LXC), Docker, CRIU, systemd-nspawn, rkt, runC, OpenVZ, and many others.

Containers Available

LXC (LinuX Container) [97] is a virtualization technology to create multiple Linux virtual environments, which provides low-level kernel features to guarantee sandboxing processes and to control resource allocation. LXC provides the isolation of processes on the shared OS via kernel namespaces, which are the basic isolation mechanism to separate containers in LXC, and via control groups, which allocate and manage container resources.

CRIU [98] is an emerging software tool, used to checkpoint/restore a tree of running processes: CRIU can easily freeze a running application, or part of it, and checkpoint it to persistent storage as a collection of files. CRIU can automatically enrich the code of running containers in order to get the dump of the state container when proper triggers are fired; associated with that, a typical and interesting CRIU employment nowadays is as the base for implementing container live migration.

Docker [99] is an industry-popular and mature initiative to build independent containerized applications by extending LXC with a high-level tool with powerful facilities, along with a very easy to use interface and deployment process. For instance, among the others, Docker tries to overcome dependency issues easily by packaging each component and its dependencies in order to solve conflicting or missing dependencies and to overcome platform differences automatically by the Docker engine. The available Docker implementation provides:

- Portability. It defines a format for bundling an application with all its dependencies into a single object, benefits from LXC process sandboxing, and extends it with an abstraction for machine-specific settings, e.g., networking, storage, logging, distribution, etc.
- Component reuse. Any container can be used as a base image to create more specialized components, in a very easy and assisted way.
- Versioning. It supports the tracking of container versions, by inspecting the differences between versions automatically, as well as committing new versions and rolling back. This easily enables the possibility to perform only incremental uploads/downloads based on differences between two versions, with limited bandwidth usage.

- Large set of supporting tools. Docker defines API for automating and customizing the creation and deployment of containers and many tools may be integrated to extend its capabilities.
- Container sharing. There is the rich support to a public registry where anyway can upload/download containers.
- Application orientation. Docker is optimized for the deployment of applications, rather than LXC that is more OS-oriented with focus on containers as lightweight machines.
- Automatic build mechanisms. Docker includes a tool to automatically assemble a container from source code, with full control over application dependencies, build tools, packaging, and so on.

In short and roughly speaking, Docker is gaining momentum, in both the academic and the industrial communities, because it provides a high-level API and good overall performance, by adding a limited overhead if compared with LXC containers. Docker is the containerization technology used for the proposed solution described below because it provides a high-level extension of LXC capabilities, by guaranteeing at the same time a reasonably lightweight usage of resources, with performance results that are comparable with LXC. In addition, Docker is based on a fervent large community of developers, thus being a significant advantage if compared to competitors. Moreover, CRIU has not been considered because at the moment it only supports process migration (not a full-support containerization solution).

Docker Storage Drivers

Docker has a pluggable storage driver architecture to give the flexibility to integrate and configure the most suitable storage driver for the specific application requirements and deployment characteristics, ranging in a relatively wide set of different technologies, the most widespread of which are AUFS, Device-mapper, and OverlayFS.

AUFS [100] is a layered filesystem that can transparently overlay one or more existing filesystems, by merging multiple layers into a single representation of a filesystem [101]. This allows to reuse layers, i.e., multiple containers that require the same base image, and to support efficient versioning of the used images, i.e., by including and exporting only differences between different versions of the same image. AUFS uses the Copy-on-Write (CoW) technology that creates a snapshot of a file every time a process needs to modify it. AUFS, based on CoW support, has demonstrated to have a non-negligible overhead in case of large size files or with high numbers of folders; however, its performance can be tuned and optimized dynamically via different configuration mechanisms, which typically are used to achieve maximum performance for container creation and I/O operations [101].

Device-mapper introduces (and works at) a further layer, i.e., block level: it is based on the so-called thin provisioning [102] where blocks realized via a sparse file allow to use Docker in a very easy-to-use way with no need for static configuration definition. The primary drawback is related to associated performance: every time a container writes to a block in the allocated pool of blocks, this block should be copied to the sparse file, thus introducing non-negligible latency.

OverlayFS [103] is a filesystem based on two main layers: the upper filesystem layer is visible to applications and readable/writable; the lower filesystem modules are instead not visible and read-only; they realize the base layer to be merged with the upper layer (containing latest data modifications) to have the final updated vision. OverlayFS provides good performance thanks to page caching and the two-layers design, e.g., AUFS comparatively introduces more latency due to the need of searching among more layers; however, OverlayFS can exhibit significant overhead in the case of high dynamicity and high numbers of modification operations.

Finally, for the sake of completeness, here there is a brief description of the less popular BTRFS and VFS: BTRFS, similarly to the AUFS driver, is based on CoW and offers better scalability and reliability in cases of non-extreme modifications rate; in VFS each layer is implemented as a different folder because VFS does not support CoW, thus leading to a simple and very portable solution, but with non-negligible drawbacks in terms of achievable performance.

3.3.2 Management and Orchestration

In a general fog computing middleware, container management and orchestration are performed by the cloud computing layer, by taking advantage from its potentially global visibility and from its key ability to perform predictive long term data analysis, thus trying to capture system behaviors and to infer system evolution [104]. In fact, in this perspective, cloud computing has the global view of resource distribution and application usage that allows to better manage the efficient management and orchestration of the distributed containers over the fog nodes.

A cloud orchestrator is necessary to deal with multi-containers management to handle: i) the creation of the cluster of containers that compose an application; and ii) the distributed deployment of an application over several IoT gateways. Container scheduling loads containers into multiple gateways, indicating how to run them, e.g., in which order to run containers, their dependencies, the minimal resources needed, or grouping all containers for a specific application for optimized transmission to gateways. More advanced tools may check if the destination hosts are correctly configured to run containers, by possibly trying to solve the associated issues. The combination of containers, which abstract from platform heterogeneity, and orchestration mechanisms, which enable logically centralized container management, allows to create a loosely coupled architecture between the configuration capability and the underlying middleware. This promotes the reuse, automation, and decoupling of allocation algorithms from deployment, by allowing to focus more on which tasks to send to the IoT gateways instead of how to deploy it on the target platform. In order to integrate container management and orchestration functions in this fog computing middleware, several existing solutions in the literature, e.g. Docker Swarm, Kubernetes, Apache Mesos, have been thoroughly evaluated; the rich set of existing container-oriented orchestrators is a manifest sign of the academic/industrial interest in the approach that has been applied to fog computing nodes.

Docker Swarm [105] is a native tool to manage Docker clustering. It is easy to use and very flexible; in addition, it exposes Docker API to be used by other Docker tools, e.g., Docker Compose. Docker Swarm is a relatively novel tool, that is developing quickly and gaining momentum, but also still needs further improvements to support complex scheduling, to be used in a production environment or for a large-scale system due to some limits. For instance, at the moment, Docker Swarm does not support any sophisticated load-balancing mechanism and lacks interoperability with other industrial tools in the field.

Kubernetes [106] is an orchestration framework, typically used as a clustering engine to define containers organization within an application. Kubernetes has demonstrated to achieve good performance and in a scalable way, without adding significant overhead to existing containers. In addition, thanks to its plugin architecture, it easily integrates with different vendors tools and technologies. In addition, Kubernetes self-healing, auto restarting, replication, and rescheduling mechanisms make it more robust and suitable for container-based applications [97]. Primary Kubernetes drawbacks relate to complexity because the setup is quite complicated and installations differ from platform to platform.

Apache Mesos [107] is a low-level, portable, and very reliable orchestrator. It has been designed to achieve high performance and to scale to very large clusters, even if it has high resource overhead. While being very interesting for other deployment scenarios, due to its overhead load it is not recommendable for usage in some IoT application domains and in fog computing middleware.

Note that, even if not detailed here and out of the specific scope of this paper, in the present fog computing middleware it has been integrated Docker Swarm as the provider of the basic container orchestrator mechanisms. In fact, at the moment, Docker Swarm represents the best tradeoff between good industry-level solidity, high performance, and limited overhead: in the preliminary experimentation work it has demonstrated to provide good overall performance; it is improving rapidly, with a relatively large community of users behind it; from the perspective of resource consumption, it has shown to be more lightweight than Mesos and Kubernetes. Probably, Mesos and Kubernetes can provide more complete and robust orchestrators, not only limited to Docker containers; however, at the moment, it is more suitable a lightweight solution, also easier to be integrated with additional mechanisms and policies, specifically designed for the targeted goal of orchestrating distributed fog resources based on global visibility of application deployment environment.

3.3.3 Configuration and Management of IoT Gateways

A defined standard gateway configuration is needed as a base to create a fog-oriented IoT gateway. Each fog participant must be compliant with that configuration in order to promote the deployment of general-purpose fog nodes, which can then be dynamically extended with additional and optional middleware services. To this purpose, it is possible to define macro-operations that compose the skeleton of any application powered by this fog computing middleware. Namely, in the proposed solution it is possible to define the following base macro-operations: I/O input operations, storage facilities, service computation, networking capabilities, and output operations. The base configuration must only manage the macro-operations flow (called skeleton in the following) and system lifecycle from a high-level and abstract point of view in order to push for and facilitate openness and portability. For example, every fog node must specify the macro-services list and the skeleton to follow to achieve the final desired behavior, as shown in the simple example in Figure 3.16.

In this way, every fog node has the same base configuration and the same skeleton, with no relationships on how the macro-service will be composed further in single services and their specific implementations. In addition, the base skeleton does not give any indication about the applications that will use the supporting fog node as their IoT gateway (dynamic association).

The IoT gateway design and implementation include the definition of macro-service composition and the single service implementa-

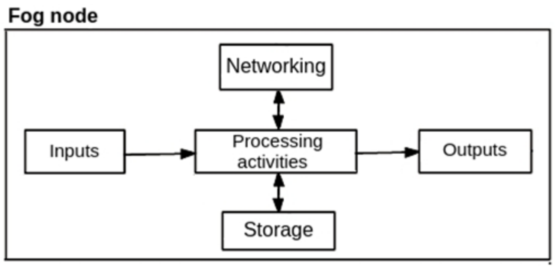


Figure 3.16: Fog Node Skeleton

tion through containers. Contrarily to macro-services, single services are specific to the application domain where they are used in. The proposed process in which a system administrator can build a complete functionality, e.g., data aggregation, data filtering, data normalization, data processing, database access, output management, etc., within a container, can package it for deployment, and then send it to the needed IoT gateways. In this way, by exploiting application composability, it is possible to configure a given IoT gateway in relation to the specific application(s) where it is used, by loading on it all and only the containers related to such specific application(s). In addition, every time a new component is implemented or a new version of a functionality is released, the new container-based version of the needed functionality can be uploaded to the gateway. Moreover, each container may contain a time-to-live value, particularly suitable for long-term container management, that specifies the validity of a container version: if not refreshed, the expired version of the container is automatically discarded in the fog middleware solution. Each containerized component is isolated and independent from the others, thus, it is possible to update and/or upload only specific functions/containers in a fully independent way. In addition, it is possible to use every node interchangeably to create an application because all the nodes have the same skeleton (skeleton-based homogeneity). This property allows to dynamically create fog nodes for a particular application, e.g., in relation to hardware-specific properties of an IoT gateway and to application relevance/priority in a given time interval. For instance, for compute-intensive applications it is possible to employ more compute-powerful nodes, rather than storage-intensive applications where it is necessary to enhance database operations, or I/O intensive applications where the focus is on communication capabilities. In addition, some applications are particularly used in relation to the time of the day, with rush hours with high peak load conditions and the need of more gateways (or resources on the available gateways), and off-peak times when many resources may be released. For example, smart traffic light systems have quite predictable stress patterns, by permitting to optimize IoT gateway resource consumption and deployment by sharing them with other applications far from rush traffic hours.

3.3.4 Containers-base Fog Computing Solution

In order to validate and evaluate the feasibility of the containers-support to realize an efficient and dynamic fog computing architecture, this solution is based on the containerization of the applications and the orchestration of them by the global cloud. In particular, it has been designed and implemented a solution that, by using containerization, it offers the characteristics of mobile presence and scalability, in order to guarantee, respectively, i) continuity of service in case of users movements and ii) highly scalable support with performance independently from the workload.

Overall Architecture

Specifically, as shown in Figure 3.17, the solution is based on a multi-layers architecture, composed of mobile devices layer, fog computing layer, and cloud computing layer.

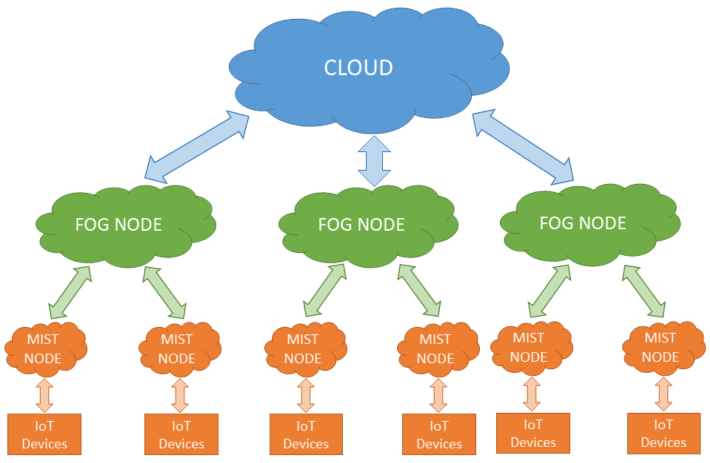


Figure 3.17: High-Level Architecture View

Mobile Devices Layer

The mobile device layer consists of both IoT endpoints immerse into the environment and mist computing nodes.

IoT devices are considered as the smallest possible entity, with no internal computational power and only network ability, to send the data towards the nearest mist node.

Mist computing [108] pushes processing even further to the network edge, involving the sensor and actuator devices, to decreases latency and to support IoT devices with no self-capabilities. Mist computing nodes are tiny devices directly connected to IoT devices at the extreme edge of the network that feed the fog nodes at the upper layer. They are constraint devices with very little computation and storage capacity and act as small sink of information and as vehicle for IoT devices to connect towards the corresponding fog node. In this solution, the mist computing part gathers data from the IoT devices, send it through the fog node wifi network and receives management instructions to change the destination fog node in case of high traffic to on the intermediate layer.

Fog Computing Layer

The fog computing intermediate layer consists of multiple distributed nodes that creates a cluster to provide functionalities for endpoints devices under the orchestration of the cloud computing layer.

From the application perspective, fog nodes provide, to mobile devices layer, a containers-based self-intelligent application able to perform calculations and statistical analysis on the information sent. In fact, it retrieves data from the bottom layer, stores it into a database, performs analysis and communicates the results back to the mobile devices layer. The fog layer manages the mobile devices layer providing them the functionalities needed for IoT endpoints to analyze environmental information and to act on the environment accordingly to the goal of the application scenario usecase considered.

From a monitoring point of view, the fog nodes have a resource-monitoring component that checks the available resources locally and perform lightweight analysis on the evolution of the resources usage on the node, determining if a node could become congested. The fog nodes contains the resources and the ability to perform limited but, at the same time, sufficient calculations that turn them into autonomous nodes able to take decisions in an independent way and with no external communications, under normal workload circumstances.

Cloud Computing layer

The cloud computing layer is the coordinator component that handles high-level and large-scope management functionalities for the overall system. It performs the containers-based application orchestration, by taking advantage from its system global visibility of resource distribution and application usage that allow to better manage the efficient management and orchestration of the distributed containers over the fog nodes. It orchestrates the fog nodes as a dynamic cluster of nodes that changes accordingly to the evolution of the environment to monitor and the system functionalities to provide. It handles the deployment of the application, deciding when to move the application, in relation to the position of the endpoints to monitor and the fog nodes to use to provide the mobile devices services. In addition, the cloud provides support during high-busy application scenarios, when the fog nodes have to serve more requests than their capabilities, by scaling up the fog nodes cluster or rebalancing the workload among the fog nodes. Similarly, it manages the fog nodes cluster in case of low node workload, by scaling down the fog cluster to minimize the resources consumption.

A cloud orchestrator is necessary to deal with multi-containers management to handle: i) the creation of the cluster of containers that compose an application; and ii) the distributed deployment of an application over several fog nodes.

Mobile Presence Feature

Mobile presence is a key concept of location-based services and pervasive applications that enables to continue to supply the mobile services independently to the final user movements and positions. Usually, mobile presence is the base from which a range of advanced applications that are innately associated with the mobility of users can be deployed in a personal and pervasive way [109].

In this solution, the concept of mobile presence applied to IoT endpoint devices is considered and extended by providing continuous service to them with a minimal unavailability time, similarly to what happens when dealing with final users. In particular, an efficient mobile presence solution, that autonomously detects the mist nodes nearby, is provided to connect with the fog nodes, and the application moves accordingly, following the mist nodes position, in a reactive manner. A reactive application migration has been designed and implemented, where the overall containers-based application moves from one fog node to another, if a new mist node is detected to connect to a different fog node from the previous connection. The application follows the movements of the IoT devices to monitor guaranteeing mobile services continuity with persistent IoT support and minimal unavailability time.

A reactive application migration has been implemented between different fog nodes, without any proactive capability in order to deal with the IoT layer, which is composed of a possible high number of mist nodes that can gather a potential huge amount of data from IoT devices. Thus, from a resource point of view, it is much more suitable a lightweight reactive behavior with a small unavailability time rather than high-computation analysis that predict next movements in advance but without add significant information to the system operations.

Scalability Feature

Scalability is a key concept in mobile services provisioning and even more central when integrating presence-aware applications, that must guarantee necessary efficient and scalable platforms upon which to deploy the mobile presence services, and in order to manage and exchange presence information in restricted networks [109]. In particular, in large-scale mobile-presence service, the increasing of the number of devices to monitor or the possible frequent mobile presence updates may lead to a scalability issue. In addition, in the near future is expected a relevant growing trend of mobile devices and mobile presence applications into mobile networks, thus, a scalable mobile presence service is deemed essential for future Internet applications [110].

To address these concerns, the mobile presence solution is extended with the scalability requirement, which enables mobile presence services to be applied also in busy contexts, large-scale or real-world application scenarios. The solution is based on a monitoring component per each fog node, coordinated by the cloud layer accordingly to the geometric monitoring algorithm, that continuously monitors the node locally and it communicates with the cloud to scale the application up or down, in case the available resources are running out or decreasing drastically. Scaling the application up and down is provided in a lightweight way, using the geometric monitoring approach, by trying to internally balancing the fog nodes cluster first, recalculating and adjusting workload estimation parameters at each resource usage violation, before to modify the topology of the cluster with the introduction of new fog nodes to extend the cluster or the disposal of some of them. In addition, the solution scalability reaction is provided within a very limited timeframe, as described in the experimental results section, with minimal unavailability time and, thus, with negligible loss of context information.

3.3.5 Implementation Insights

The containerization tool used to create the containers to build the application is Docker. The application is composed of three Docker containers, as shown in the Figure 3.18.

(i) Mosquitto container serves as MQTT broker and enables to exchange messages between the fog node and the mist nodes connected to its local network. The mist nodes send, through the MQTT broker, all the sensed data and receive the request to change fog access point connection, if available, in case of high workload. (ii) MongoDB [111] container stores all the data sent from the mist nodes

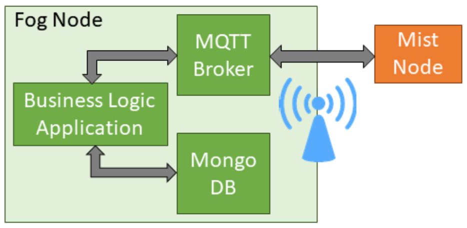


Figure 3.18: Fog Node Internal Structure

into its NoSQL database. (iii) The container with the business logic of the application listens on the MQTT broker for messages from the mobile devices layer and retrieves the data stored into the database to analyze it and generate the proper reactive actuation for the IoT devices. Each fog node is equipped with wifi access point capabilities to allow the mist nodes to connect to it and send data to the local MQTT broker.

The Docker images, the binary files that include all of the requirements for running Docker containers, are built on the cloud platform. Usually, the Docker image build process is both resource-consuming and time-consuming process to build a Docker image on a limited-resources device, e.g. fog node device. Thus, the cloud is employed to build the Docker images to take advantage from a much more powerful platform. In this way, it is possible to obtain a twofold advantage: minimize the time necessary to build them, increasing the images building speed, and avoid any additional workload on the fog device, keeping the resource consumption as much lightweight as possible, only related to the real application execution. Since Docker images are generally architecture-dependent and it is possible to run an image only on the same architecture from which the images has been created, Docker cross-platform images are built. QEMU [112] is used to build Docker images on the cloud but targeted to be executed on a different architecture, e.g. RaspberryPi, respectively x86_64 and ARMv7 processors. QEMU is an open source standalone machine emulator and virtualizer, able to emulate ARM boards, used to run an unmodified target operating system and all its applications in a virtual machine, that uses a dynamic translator to convert the code into native host instructions to improve performances. QEMU is used to build both the base image shared among the other Docker images and the upper-layer images that specify the Docker images adding to the base image the application-specific code. Finally, after the images are built with QEMU, they are pushed towards the Docker Hub registry to distribute the newly created image among the fog nodes. Docker Hub is a stateless cloud-based registry service that provides a centralized resource for Docker images discovery, distribution and change management, user and team collaboration accessing to shared image repositories.

Geometric Monitoring

Along with the containers-based application, a monitoring component, based on geometric monitoring approach, has been implemented to monitor the resource consumption locally on each fog node, at runtime, and proactively modify the system configuration dynamically, before a node run out of resources.

Geometric monitoring is an approach that reduces monitoring the value of a function, compared to a threshold, to a set of constraints applied locally on each of the streams. The constraints are used to locally filter out data increments that do not affect the monitoring outcome, thus avoiding unnecessary communication with the coordinator, in order to enable monitoring of arbitrary threshold functions over distributed data streams in an efficient manner [113]. An extended version of the geometric monitoring approach has been implemented in order to increase the fog nodes autonomy because it provides constraints that allow fog nodes, easily and in a lightweight way, to check if the current resource usage satisfies the requirements, i.e. within the defined threshold. Thus, the communication between fog nodes and the cloud coordinator are extremely minimized and happen only if a resource violation occurs. For sake of briefness, more details of the geometric monitoring procedure can be found in [113, 114], where it has been extensively explained.

In this solution, to monitoring the resources available on the fog nodes, the cpu, memory and disk values are considered about the percentage of resources used on the total available. In order to improve the estimation accuracy of the monitoring components during its lifecycle, each fog node monitoring system is initialized creating a safe zone to use during the following resource consumption analysis to check local violations. To this purpose, at the startup, each fog node sends to the cloud some resources measurements that will be elaborated by the cloud and returned as constraints that define the safe zone for each specific fog node. The number of samples to send to the cloud varies in relation to the required estimation accuracy and the time available on the fog nodes: more data gathered more the estimation is accurate because based on a more solid training set and more time the fog nodes are idle and cannot respond to the coordinator. In order to have a good tradeoff about accuracy and time, the node is sampled every 1s to create a training set of 100 values per resources. Note that this procedure does not affect the system performance because it is executed only once, at the startup, within the whole node lifecycle.

Figure 3.19 shows the points measured into a 3-dimensional space.

The blue plane is the 2-dimensional optimal hyperplane that best approximate the points in the training set and is used as target to

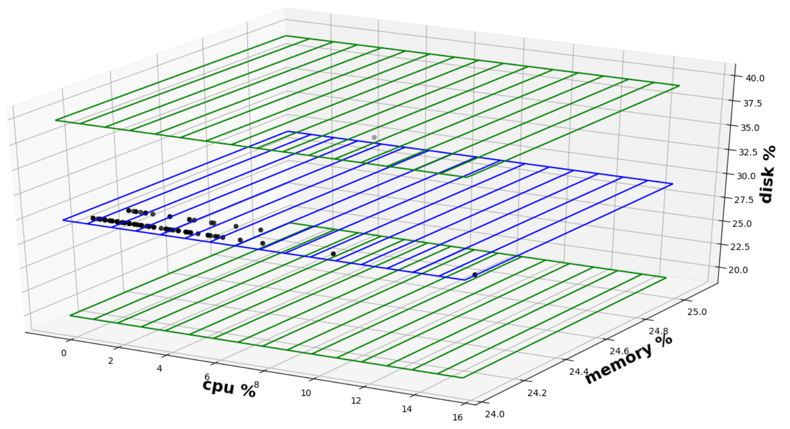


Figure 3.19: Data Distribution and Related Hyperplanes

evaluate the distance of the future resources measurements. In particular, the optimal hyperplane is found solving the following optimization problem, based on a least square problem:

- Given a generic plane equation 3.1, it is possible to write it as 3.2, where the z is the target vector to approximate:

$$a * x + b * y + c * z + d = 0 \tag{3.1}$$

$$\frac{a}{c} * x + \frac{b}{c} * y + \frac{d}{c} = z\,, \;\; with \;\; c \neq 0 \tag{3.2}$$

- Given a point i and set A=$\frac{a}{c}, B = \frac{b}{c}, C = \frac{d}{c}$, $\forall i$, it is possible to calculate the least squares function ω as 3.3:

$$\omega = A * x_i + B * y_i + C \tag{3.3}$$

- The goal is to retrieve the minimal distance between the least square function ω (3.3) and the related target value z_i, in order to find the plane that best approximate the points retrieved, which indicates the optimal solution of the minimization problem. Thus, equation 3.4 represents the formulation problem of the least square method where $min_{distance}$ is the minimization sum of the Euclidean norms of the squared residuals ω and z_i:

$$min_{distance} = \sum_{i=1}^{N} \frac{1}{2} ||\omega - z_i||_2^2 \tag{3.4}$$

  where, N is the number of values in the dataset, i.e. 100.

Resolving this optimization problem it is possible to obtain the coefficients of the solution vector A, B, C, that are the coefficient of the least square fit plane, and they will be applied on the future monitoring data points measured to check their position in the 3-dimensional space.

From a geometric point of view, the minimized distance equation in (4), on the measured resources at runtime, gives an evaluation about the evolution of the resource usage because: a positive result means the measured point is located above the optimal hyperplane, thus, the resource usage is increasing; a negative result means the measured point is located below the optimal hyperplane, thus, the resource usage is decreasing. The distance between the monitoring results obtained and the optimal hyperplane indicates the distance from the optimal resource usage and from the threshold.

The green planes are the threshold hyperplane defined as the threshold values applied as offset to the optimal hyperplane. Thus, the space between the two green hyperplane is considered safe zone and, on the contrary, all the space outside the green hyperplane raises local violation on the fog node during the monitoring. While the monitored resources define a point within the threshold hyperplane borders, no communication with the cloud is required.

In the case, during the usage of the application, a new resource measurement point goes outside the safe zone, the fog node notifies a local violation to the cloud. The cloud, thought the so-called scalability orchestration, explained in detail in the following, will re-balance the cluster or deploy a new fog node. This process continues until the fog node measures resources consumption data is outside the safe zone.

Containers Orchestration

Docker swarm mode implements Raft Consensus Algorithm [115] to manage the global cluster state in order to make sure that all the manager nodes that are in charge of managing and scheduling tasks in the cluster, agree on the shared state and are storing the same consistent state. In case of failure, any manager node can pick up the tasks of scheduling and re-balance tasks to match the desired state, recovering autonomously and restoring the services to a stable state.

In addition, Docker Swarm is complemented with Docker Compose [116], a tool for defining and running multi-container Docker applications, based on a Compose yml file to configure applications services that helps building, starting, monitoring, and stopping microservices. It groups all the containers for a specific application and defines how to build a container, i.e., how many instance of each container to create, the deployment parameters, restart policies, the minimal resources needed, and so on. In the present solution, Docker Compose, that defines applications composed of multiple containers and manage their deployment, is jointly used with Docker Swarm, that combines and orchestrates multiple nodes as a cluster and the Docker services component created. Here, there is a portion of code of our Docker Compose yml file:

```
version: '3'
services:
  business-logic-app:
    image: alez/test-app-raspi:3
    deploy:
      mode: replicated
      replicas: 1
      restart_policy:
         condition: on-failure
         delay: 5s
         max_attempts: 3
      placement:
        constraints:
       - node.labels.rasp1==true
```

It is been used Docker Compose version 3, that extends the older Compose versions with options related to the Swarm service deployment on multiple nodes. For each container to deploy, it is defined the name, the Docker image tag, and the deploy settings, e.g. mode, number of replicas, the restart policy condition (on-failure because the business logic container depends on MongoDB and Mosquitto containers, thus, it fails if they are not fully started and attempt to deploy it other 3 times), and placement constraints. Finally, the Docker Swarm is applied and, with the command docker stack deploy –compose-file, the Docker engine parses the Compose file to deploy the containers specified and creating Docker services that get directly deployed in the Swarm mode cluster.

In the following, the orchestration related to the mobile presence and scalability features and managed by the cloud computing layer is described more in detail. Before to start the system, all the fog nodes available are inserted manually into the swarm as worker nodes and the node is set on the cloud computing layer as the master of the swarm. At the startup, all the fog nodes are reachable from the cloud layer but in drain availability state to prevent from receiving new tasks from the swarm manager until a node is in the active availability state. Each fog node continuously executes two threads, checking: new devices available in the neighborhood that connect with its access point (mobile presence orchestration); the amount of resource available on the host (scalability orchestration).

Mobile Presence Orchestration

Mobile presence orchestration aims to move at runtime all the containers that compose the application from one fog node to another, following the movement of the related mist node. The mobile presence orchestration is triggered on the fog node by the thread that checks the wifi interface to check if new devices connects to the node access point. Figure 3.20 details the sequence diagram related to the mobile presence orchestration with the steps needed for the procedure.

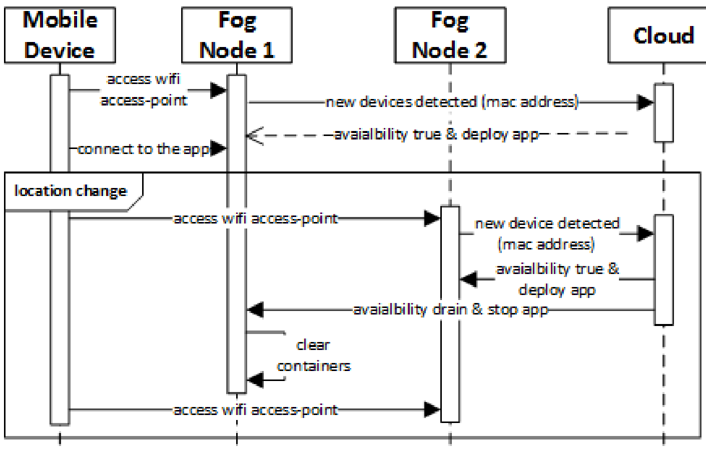


Figure 3.20: Mobile Presence Procedure

Initially, a new mobile device connects to the fog node access point, the fog node reads its MEC address checks if it is a new connected device or a pre-connected device. If it is a new connected device, the fog node sends the MEC address to the master node on the cloud to notify it, through a webserver located on the cloud side, and the mobile presence procedure takes place. The cloud sets the fog node into active availability state, modifies the Docker Compose file to deploy the application accordingly, and deploy the containers that compose the application following the Docker Compose specifications. In the meantime, the mobile device tries to connect to the application, in particular with the MQTT broker, until the Mosquitto container has started up.

When the mobile device changes location, it connects to the access point of a different fog node that detects it as node device and redo the procedure above. During the mobile presence orchestration, the cloud set the Docker Compose file to deploy only one replica for the application because the application is deployed only on one node. Thus, during the deployment of the new containers, the old application, which was running on the previous fog node, is stopped and no more available. In addition, the previous fog node is set to drain availability and its exited containers are deleted to minimize the resource usage and optimize the space available, since after some mobile presence procedures the node could have many exited and useless containers.

Scalaiblity Orchestration

Scalability orchestration aims to manage borderline workload situations in order to adjust the amount of resources available required to serve the clients. The solution handles both to scale up the system, in case of relevant increase of the resource needed, and to scale down the system to manage efficiently the resources available and do not waste them.

The scalability orchestration is triggered on the fog node by the thread that checks the resource available and perform the geometric monitoring on the resources consumption, if the amount of resource available on the node is diverging from the expected values. Scalability orchestration, following the geometric monitoring rules, is composed of two main actions, executed at runtime in sequence: calculate and re-estimate the vectors needed by the geometric monitoring to try to re-balance the fog node resource consumption within the cluster; extend the application onto more fog nodes, by increasing the number of containers replicas and deploying all the containers of the application on new available fog node. Figure 3.21 details the sequence diagram related to the scale up orchestration and the steps needed for the procedure.

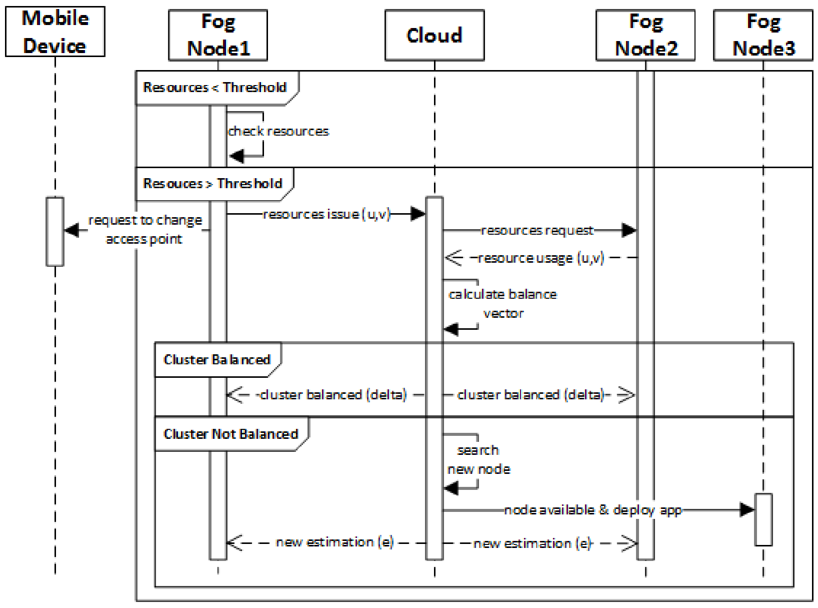


Figure 3.21: Scalability Procedure

Each fog node is constantly monitored to check if its resource consumption is within the threshold defined by the cloud. If the monitor component detects a violation on the amount of resource consumption, it send a message to the Mist node through the MQTT broker to search another access point because the current one is overloaded, and notifies it to the cloud, specifying the drift vector (u) and the statistics vector (v) of the geometric monitoring.

If the requesting node is alone into cluster, the cloud deploys another fog node to extend the cluster.

On the contrary, if the requesting fog node belongs to a cluster on nodes, the cloud, as a first action, tries to re-balance the cluster. Re-balancing the cluster is a procedure that involves the cloud to retrieve all the drift and statistics vectors from all the nodes into the cluster. The cloud, then, tries to create a group of nodes, called balancing nodes, such that the average of the drift vectors held by the nodes in the balancing group, called balancing vector, creates a monochromatic area with the same estimated vector [113]. The cloud tries iteratively to balance the cluster adding, to the balancing group, a cluster node until the balancing vector is monochromatic or all the cluster nodes are in the balancing group. If the cluster balancing succeed and the balancing vector results monochromatic, the cloud calculates the delta vector, that is an adjustment of the offset vector (slack vector) and sends it to every nodes in the balancing group.

Vice versa, if the balancing process fails and there are no more node, inside the cluster, to add to the balancing vector, the cloud searches for a new suitable node to add to the cluster. The set of suitable nodes to extend the cluster may be all the fog nodes currently ready but idle in the drain availability state. Successively, the cloud sets the new fog node into active availability state, modifies the Docker Compose file to deploy the application on all the cluster nodes, and deploy the containers. Finally, the cloud calculates the new estimation vector (e) for each node inside the cluster and send it to them.

In case the fog nodes detects a violation related to a too little amount of resource consumption the system scales down, with a similar procedure, by removing from the cluster the requesting node that notified the local violation, and re-estimating the new estimation vector (e) for the rest of the fog nodes remained in the cluster.

3.3.6 Performances

As performance evaluation of the container-based solution described, the tests on both container performances and on the overall solution proposed are provided. The container performance tests represent preliminary evaluations to validate the containers usage and the show the containers feasibility to be used into the realization of a fog computing infrastructure, with a particular focus on the overhead introduced. The tests on the overall solution proposed show the realization of an efficient and dynamic fog computing solution by orchestrating containers-based applications in the most efficient way, guaranteeing the solution to be scalable, dynamic, autonomous with a minimal amount of communications with the global cloud and continuity of service.

Containerization Overhead Performance Results

An application running on a fog node has been implemented and experimented. It behaves according to the skeleton of i) gathering data from sensors, ii) storing gathered information, iii) periodically (for energy consumption motivations) reading local storage, and iv) processing the read data to identify alerts/warnings. The application is implemented in Golang, which is the most supported language in Docker, and is an example of application that, in different time intervals, has relevant requirements in terms of both computing and I/O operations. As already anticipated, to collect quantitative performance indicators in a particularly challenging deployment environment, the fog nodes has been realized with RaspberryPi 1 Model B+ devices, with 512MB RAM and a single core CPU.

As the realistic testbed to test container-based applications, a simple use case application in the domain of SCV has been developed and deployed. In this SCV application, a fog node located on a smart bus, following pre-determined routes, assists the other distributed SCV participants; for the sake of simplicity, in the example illustrated here to show the performance results of our fog computing containerization, the fog node has the main goal of elaborating and aggregating the data gathered from multiple sources. In other words, the fog node acts as a mobile sink collecting data from a dynamically determined set of heterogeneous sensors, that can be installed onboard of the transit vehicles or on other infrastructural components, e.g., base stations with car detection modules at road crossings; the fog node keeps track of gathered data by storing and sorting information, e.g., according to priorities, originating locations, previous warnings or error signals, and so on. In addition, the mobile sink can decide to spread valuable concise information to other SCV participants opportunistically encountered during its travel, thus contributing to overall system awareness and to the emergence of cooperative behaviors.

Here some interesting results about the experimentation work are reported by focusing on Docker-based containerization and on the impact of filesystem configuration on performance. In particular, the performance of single containers are compared in relation to native code execution, by estimating the overall overhead and delays introduced by the exploitation of Docker-based containers with either AUFS, Device mapper, or OverlayFS. In addition, the usage of multiple concurrent containers have been thoroughly evaluated over a single fog node (which is highly typical in envisioned future applications), also to verify the capability of even very constrained devices to manage several containers simultaneously. All the results reported in this section do not include the latency and effects of orchestration mechanisms, which are shown in the next section, in order to exclusively focus on the performance characterization of the extended IoT gateway on the fog node.

Table 3.5 reports the most significant results in terms of average time, over a wide set of hundreds of runs, to complete the execution of the sample SCV application in the case of native code execution vs. container-based execution with the exploitation of different storage drivers, i.e. AUFS, Device mapper, and OverlayFS. In addition, the table highlights the time contributions associated with the most relevant operations performed during the SCV application execution: container creation time; I/O operations (file opening and reading size of 1.6 MB); CPU operations (double sorting, first alphabetically and then by words length). The maximum standard deviation experienced for the reported results is within 0.2 s.

Table 3.5: Native-Code and Container Execution Time

Operation	**Native**	**Docker + AUFS**	**Docker + DevMapper**	**Docker + OveralyFS**
Start Container	–	3.5 s	9.1 s	3.3 s
I/O Operations	1.6 s	4.3 s	4.7 s	4.3 s
CPU Operations	3.1 s	3.4 s	4.2 s	3.5 s
Total Operations	4.7 s	12.5 s	21 s	11.8 s

The build time of the container-based version of the SCV application has not been considered in the reported results because not relevant for runtime performance analysis. Anyway, an image build time is in the order of a few minutes even on RaspberryPis, which will be rarely exploited as the nodes where the image building is performed; the reasonable process in a realistic production environment will be to delegate build operations to cloud resources and successively to migrate the ready images towards the interested fog nodes.

By coming back to the results of Table 3.5, as expected, the execution of native code outperforms the container-based one. This has demonstrated to be mainly due to the more efficient usage of file allocation table (fat) disk filesystem, of course in addition to the time for container creation. Among the case of Docker-based execution, the measured performance has shown to have a strong dependence on the filesystem used: Device mapper demonstrated to have the worst performance in this SCV application, mainly because of the high number of write operations it performs on the sparse file, with the introduction of a significant delay also in this case of single container execution; AUFS and OverlayFS have achieved comparable results on every operation, with a slightly better behavior of OverlayFS.

Finally, Figure 3.22 reports the results about the concurrent execution of multiple containers over a single fog node. Also in this case, AUFS and OverlayFS outperform Device mapper, with a growing performance gap that has shown to be proportional to the number of containers in execution, thus demonstrating the practical unsuitability of using Device mapper over resource-constrained fog nodes in the relevant case of multiple concurrent containers scenario. Most relevant, Figure 3.22 clearly shows that, in terms of scalability while growing the number of running containers, also a resource-constrained IoT gateway such as a RaspberryPi node can achieve very good performance (linear dependency of execution time on the number of concurrently running containers), thus demonstrating the practical feasibility of the proposed approach.

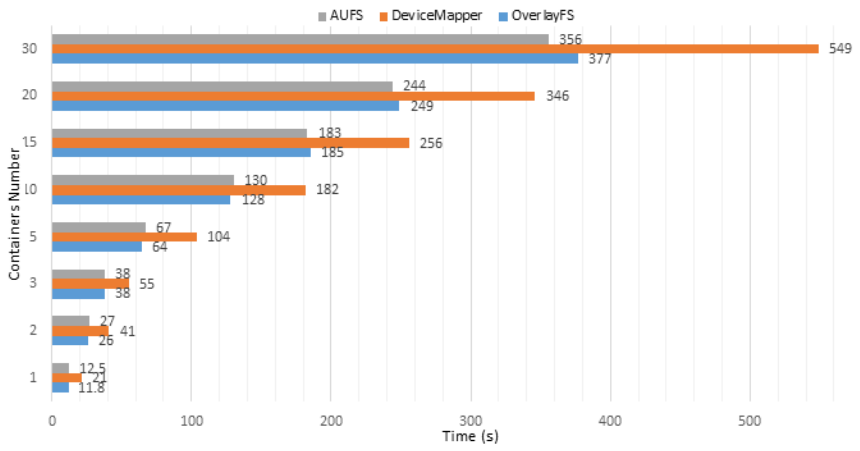


Figure 3.22: Execution Time over Multiple Containers

Elastic Provisioning of Mobile Services

Since the previous tests on the container-based application are encouraging and show good preliminary results, in particular in terms of scalability of multiple containers execution, also over very resource-constrained nodes such as RaspberryPis1, extending the tests by applying allocation algorithms for the Docker Swarm-based orchestrator. In this way, it is possible to exploit global visibility of resource status and of available fog nodes, and experiment the application of the fog computing middleware over a real application domain and a geographical distributed deployment environment coordinated by either fixed or mobile IoT gateway fog nodes. The geographical-distributed testbed is shared between Sweden and Italy and among different providers, as shown in Figure 3.23.

The master node, on the cloud layer, that coordinates the overall system, is located at University of Bologna, Italy. The fog layer

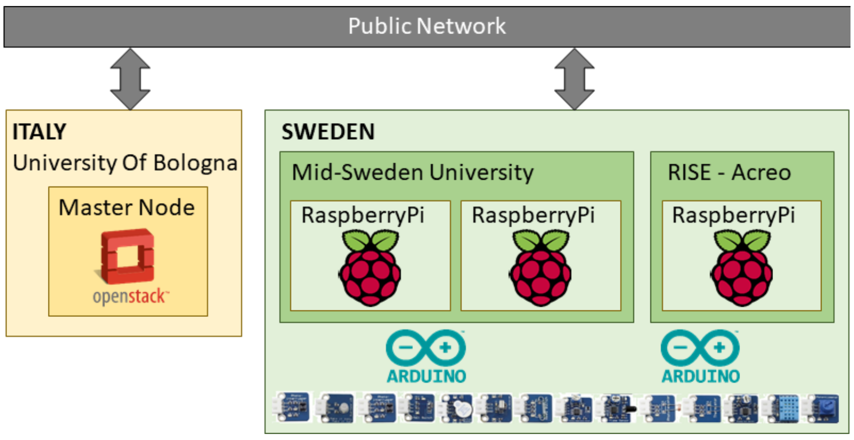


Figure 3.23: Geo-distributed Testbed Used

is composed of three RaspberryPi3 located in Sweden, within the networks of Mid-Sweden University and the RISE-Acreo Research Center. Each RaspberryPi3 is equipped with 64-bit quad-core ARM Cortex-A53 processor, 1 GB of RAM and 16 GB of storage space. The mist nodes are Arduino nanoESP devices that are able to move freely into the environment. Each Arduino nanoEsp device is equipped with ATmega168 processor, 16 kB of flash memory and 1 KB of SRAM memory. The IoT layer is composed of different types of sensors, i.e. temperature, movements, and so on.

It is worth noting that the network connection between the master node at University of Bologna and the fog nodes located in Mid-Sweden University has a round trip time of 61 ms, with a bandwidth of about 90 Mbits/s.

From the final user point of view, the evaluated time for the Arduino to disconnect from the access point of a fog node and reconnect to the access point of another node is 3.91 s, with a standard deviation of 1.49 s. In addition, the results about the moment when the Arduino device disconnect from the MQTT broker and reconnect successfully to the MQTT broker on another node is 11.06 s, with a standard deviation of 2.04 s. Note that the standard deviation of the experimental results obtained is quite high, in particular related to the time needed to disconnect and reconnect to the access point. This instability is caused by to the imprecision and unreliability of the Arduino nanoESP integrated wifi that takes different time to establish a connection, with the MQTT broker and, in particular, with the access point.

Investigating both the mobile presence and scalability orchestration of the solution proposed, Table 3.7 shows the average times related to orchestrate, deploy and startup the application composed of containers, with a standard deviation within 0.18 s for the total time

Table 3.7: Containers Orchestration and Deploy Time

Mode	Orchestration	Deploy	Startup	Total
Mobile Presence	0.10 s	1.50 s	0.92 s	2.52 s
Scale Up	0.45 s	1.50 s	0.92 s	2.87 s
Scale Down	0.55 s	1.50 s	0.92 s	2.97 s

where, mode is the type of deployment the master node orchestrate; orchestration is the time needed to the master node to perform the calculations needed, e.g. re-balancing the cluster, selecting which fog node use as target node to deploy and modify the Docker Compose file accordingly; deploy is the time needed to deploy all the containers on the selected node; startup is the time required by the containers, after the deployment, to startup and serve the clients; total time is the time to complete the overall phases of the corresponding mode.

Table 3.9 shows the set of times during orchestration in the case all the fog node already has locally the Docker images needed to run the containers that compose the application. Table 3.9 shows, after hundreds of runs, the average times related to pull the Docker images form the Docker Hub registry, with an average standard deviation of about 6 s

Table 3.9: Pull Docker Images from Docker Hub Registry

Container Name	Base Image	Upper Layer Image	Total Time
Mosquitto	243.38 s	70.27 s	313.65 s
MongoDB	248.38 s	72.00 s	315.38 s
Business Logic App	248.38 s	83.16 s	326.54 s

where, base image is the time to pull the QEMU image that is used to create for all the other containers; the upper layer image is the Docker image that contains the code of the specific container and is applied to the base image to specialize it; the total time is the time to pull the complete Docker image for the containers. Pull the images from the Docker Hub registry is a heavy and time-consuming action, that can require more than 5 minutes, thus, fog nodes must assure to get the Docker images in advance before to receive the master node request to deploy the service.

On the cloud side, the time to build Docker images varies, in average, from about 205s for MongoDB image to 340s for the Mosquitto image. Note that it does not slow down the system performance because the Docker images building process: is executed in an offload mode, independent from the system lifecycle and completely transparent from fog nodes, which only continue to perform request towards the Docker Hub registry on the same images name; is performed by the cloud computing layer, which is per definition powerful and can manage it without any performance drops, also considering the low amount of requests from the fog computing layer has to handle.

Finally, on the cloud layer, the time to calculate and re-estimate the vectors necessary for the geometric monitoring process are almost negligible and they are within 21ms and 29ms to calculate, respectively, a new estimation vector e and a new delta vector, to re-balance the cluster.

4 | Live Migration and Automated Offloading for Edge Computing

This chapter presents two implemented solutions based on MEC concept that exploit a quite powerful intermediate layer to support mobile devices. The considered mobile devices, i.e. smartphones, have more resources that IoT devices, thus, only a single layer of nodes within the intermediate middleware infrastructure is required without the necessary need to employ a hierarchy structure, with nodes very close to the edge, e.g. in the case of tiny IoT devices. A powerful standalone server, with few interactions with the global cloud resources, within the middleware infrastructure is needed, in order to satisfy the requirements of mobile devices and MEC is the ideal solution.

Both solutions address the technical problem related to the impossibility of mobile devices to execute some tasks/applications properly in terms of time, as well as the possibility to save energy resources. They adopt edge computing solutions to support the execution of those resource-intensive applications without compromise the performance. In particular, the first solution exploits an autonomous and transparent mechanism that guarantee service continuity and minimize the unavailability of the service by moving the service proactively ad reactively. The second solution extends the MEC concept with the mobile computation offloading that allows to dynamically migrate applications, or part of them, to a remote server.

4.1 Related Work

Many works that address both the general topic of VM/container migration or support mobile devices and mobile computation offloading already exist in the related literature. Although the hot topic and the proliferation of several work on those topics, only a very few research activities have focused on VM/container migration and mobile computation offloading towards MEC middleware for mobile services. In addition, the existent solutions are quite limited and they usually do not provide extended capabilities to work in hostile environments, which is a relevant challenging aspect of modern CPS, or in a complete autonomous way.

4.1.1 VM/Containers Migration

From the VM/containers migration side, [117] highlights the limitations of traditional live VM migration on edge devices, by proposing live migration in response to client handoff in cloudlets, with less involvement of the hypervisor and by promoting migration to optimal offload sites, adapting to changing network conditions and processing capacity. [118] presents the foglets programming infrastructure that provides APIs for a spatio-temporal data abstraction for storing and retrieving application generated data on the local nodes, and primitives for communication among the resources in the geo-distributed computational continuum fog-cloud and it handles mechanisms for quality/workload-sensitive migration of service components among fog nodes. [119] proposes the integration between cloudlet and base station subsystem to provide proxy functionality closer to mobile devices in order to support mobile multimedia services and adjusts resource allocation of resources triggered by runtime handoffs. [120] evaluates handoff conditions in relation to various parameters (e.g., waiting time, energy requirement of the communication, signal strength, bit rate, number of interactions between cloudlets and associated devices) and decides to offload the computation based on fuzzy techniques. In addition, some aspects of the above activities, such as for [120, 121], are also suitable and specifically targeted for hostile environments. In particular, [121] highlights the usage of some different cloudlet provisioning mechanisms, such as optimized VM synthesis for interoperability purpose, to demonstrate the middleware layer suitability in hostile environments, by highlighting the importance of automated handoff of data and computation. [122] describes a reference architecture for code offload that aims to decrease large application overlays and to achieve rapid VM/application startup time, by carrying the overlay on mobile devices and transferring it at runtime to a discovered cloudlet. From the mobile computation offloading side, [123] proposes a multi-agent-based code offloading mechanism, by using reinforcement learning and code blocks migration, to reduce both execution time and energy consumption of mobile devices. It adopts multi-agent based distributed method for offloading computation and Markov decision process to model dynamic environment appropriately.

Note that in this paragraph has been presented only the work conerning the migration at middleware layer. More work related to VM migration will be described in section 5.1.

4.1.2 Mobile Computation Offloading

The related literature about the general topic of mobile computation offloading is already quite rich, thus demonstrating the strong interest of the community in the field. MAUI [124] proposes an architecture for code offloading using a profiler which measures energy consumption and data transfer requirements and a solver which decides to offload a method based on the measurements performed by the profiler. ThinkAir [125] benefits from cloud computing elasticity, performing on-demand resource allocation, and exploiting methods execution parallelism by dynamically creating, resuming, and destroying multiple VMs in the cloud when needed. COMET [126] uses a distributed shared memory on and VM synchronization primitives to augment mobile devices with machines available in the network, by using thread offloading among distributed devices. Cuckoo [127] focuses on the implementation of a communicating library between mobile device and Ibis communication middleware, by offloading bundles that contains the compiled code. COSMOS [128] proposes a component-based framework for managing context data in ubiquitous environments and focuses on solving offloading decision problem and resource allocation on VM, providing computation offloading as a generic service underlying computing and communication resources transparent to mobile devices, by allocating offloading demands from the mobile devices to a shared set of compute resources that it dynamically acquires from a commercial cloud service provider. The ULOOF framework [129, 151] introduces an improved offloading decision mechanism by refining the assessment of the available bandwidth as well as energy consumption, providing a realist running time and energy consumption estimation towards a more accurate offloading decision.

While all the above works have tried to address computation offloading in different ways at different granularity, i.e. methods, threads, components, they focus on offloading execution, rather than selecting the tasks to offload. The code selection decision is one of the primary relevant and still open technical challenge is to partition, in a general-purpose way, the application code into offloadable and non-offloadable parts [130] but still not-well explored topic in the related offloading literature. [131] underlines the tasks selection algorithm as a main issue in offloading execution solutions and it highlights the granularity selection and the dynamic application partition as the main challenges to face in order to offload on fog/edge and cloud platforms. The lack of an autonomous code selection mechanism is a very limiting issue that do not allow applying the computation offloading to a wide set of applications, in particular large-scale applications, due to the obvious impossibility to scan the whole applications structure manually and introduces big inefficiencies. A selection algorithm is required in order to analyze every kind of application dynamically and detecting the list of methods suitable to be offloaded in an efficient way.

In fact, the above work generally require the developers to add annotations to indicate which portion of application to offload, modifying the application code manually. Doing this partitioning manually requires significant human effort: methods may be unsuitable for offloading due to several non-trivial motivations, such as frequent interaction with final users, difficulties in replicating device-specific resource instances at cloud/edge nodes, impossibility to serialize used resources, only to mention a few; moreover, this required human effort slows down the acceptance of mobile offloading techniques in industrial scenarios for obvious cost motivations. In addition, note that the complexity of identifying all the offloadable methods push in the direction of determining only a subset of the really offloadable methods, to avoid false positives at runtime, thus limiting the impact of offloading on observed latency and power consumption improvements. Therefore, to maximize the benefits of offloading, a sophisticated and automated task selection algorithm is required. Developing a general-purpose task selection algorithm is not trivial because it requires to detect offloadable parts of tasks from the entire Android application code without a-priori knowledge of the analyzed application and has not been explored so far.

In this perspective, CloneCloud [132] uses static code analyzer to automatically mark possible migration points and, thus, to partition the binary of an application with a set of execution points. CloneCloud uses a thread-granularity for the offloading execution selection and each execution point decides between where the application migrates the thread towards the cloud or the local execution on the device. CloneCloud, in this moment, is the only proposal that implements an autonomous code selection mechanism without hard-coding into the specific application but it specifically targets a powerful remote platform, i.e. the cloud computing, to execute the offloaded code, because it needs to re-create a VM with the same hardware and OS used locally.

On the contrary, in the following, the proposed solution focus on a more decoupled and lightweight solution that has no platform constraints and can be used in many remote servers independently from the available resource. In fact, the solution aims to offload code at the method level that can run on every server equipped with a JVM without the same hardware/OS of the mobile device and, thus suitable also for less-powerful platforms, i.e. fog/edge nodes.

4.2 Migration-enhanced Support for Mobile Services

This solution proposes a practical contribution to overcome the challenges of limited-resource mobile devices in hostile environments by using a MEC intermediate middleware layer, by using, along with typical reactive functionalities, effective strategies for proactive migration. The primary focus is to efficiently design and implement a MEC layer to assist devices to preserve their full functionalities and to supply service provisioning to final users also in case of high mobility in hostile environments. In particular, in the designed and implemented support platform, highly demanding computation tasks on mobile devices can be delegated to the MEC layer that executes the tasks and returns the related results by preserving service continuity, also in case of end users mobility, thanks to proper virtualized function migration between MEC nodes. In addition, typical reactive migration has been complemented with predictive handoff mechanisms that move a specific virtualized function into the most suitable MEC node, based on the consideration of multiple aspects, such as network statistics, availability, and recoverability. In particular, the proposal is a dynamic proactive handoff scheme with motion prediction, by integrating proactive virtualized function migration in it based on predicted users movement, towards the next forecasted MEC node, without waiting for users requests in the new MEC locality, in order to prepare in advance composed service provisioning and to drastically minimize the unavailability time due to virtualized function migration.

The explained solution is a valuable solution also for the several mobile devices that are forced to operate in the so-called hostile environments, characterized by very high uncertainty of the available resources, limited bandwidth, unreliable networks, and rapid deployment needs [134], that need to execute computation-intensive functions and would benefit from compliance with strict quality requirements. Thus, although some recent research has enabled the possibility for mobile devices to offload computations towards more powerful resources, typically hosted in the global cloud, mobile services in hostile environments cannot always rely on cloud computing. The main reasons are the unsuitability of cloud to support the needed quality, for instance quick actuation in case of anomalies, disconnection management and recovery or high mobility, and scarce availability of network infrastructures that can cause unexpected disconnections. In such a challenging deployment environment, it starts to be recognized that a three-layer device-middleware-cloud architecture is necessary to move part of the resources towards the edge and overcome the limitations of direct cloud-device iterations.

4.2.1 Design

The three-layer architecture, based on the extension of emerging MEC technologies, is illustrated in Figure 4.1.

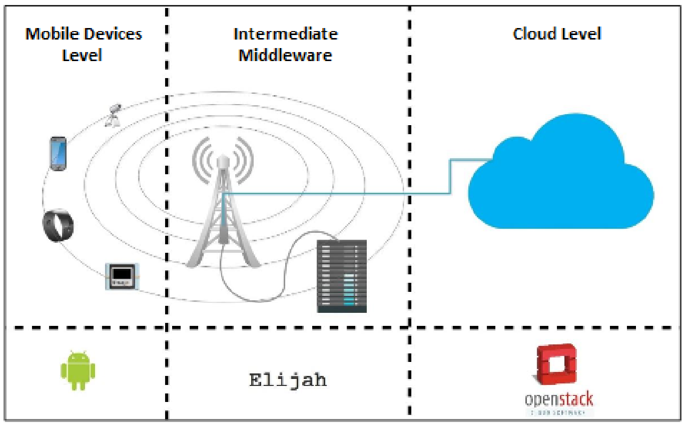


Figure 4.1: General Three-Layers Architecture

Mobile Devices Layer

The mobile device layer consists of all the endpoints that need to perform high-resource demanding executions of mobile services and do not have enough capabilities to do that. For instance, this includes heavy image and video analysis/processing that can be performed uniquely with computationally intensive techniques that require resources beyond the capability of mobile devices, at least taking into consideration possible application-specific requirements on response time. The solution fits a very wide spectrum of heterogeneous mobile devices, with the only constraint to run Android OS.

In the application case running example used in the following sections, the target application is a face recognition application, for security purposes, able to monitor an environment with cameras that capture photos and videos of the surrounded area. To enable such a service to be practically useful in real-world scenarios, the processing time for any sensed media must be limited in the order of a very few seconds, to be able to raise alarms promptly in the case of suspicions.

In a context like this, mobile devices often do not have enough capabilities to satisfy strict requirements on response time, in particular if considering their possible immersion in hostile environments; therefore, it is a must that they have to delegate most analysis functions to the MEC middleware layer.

MEC Middleware Layer

The MEC middleware layer consists of two primary components: i) the standard Elijah platform, one of the current most promising and complete open-source project to realize the middleware layer, that allows moving computational resources near the edge of the network, by providing and orchestrating VMs near mobile devices; and ii) a platform extension module, called ServerManager, that completely decouples mobile devices and Elijah, linking them by receiving and forwarding the users requests towards the Elijah platform and vice versa, thus intercepting and coordinating all interactions between mobile devices and MEC nodes.

MEC nodes can be provisioned in either a proactive or a reactive way.

Proactive provisioning allows to minimize and automate in an efficient way virtualized function migration, by pre-loading the needed functions in advance on the target MEC node that, presumably, will be the next visited by the served user; this is managed before receiving explicit migration requests, thus limiting the costs in terms of unavailability and performance during the procedures of mobile device handoffs and associated VM/container synthesis. The global cloud (third layer) triggers virtualized function migration (because it is the cloud where predictions are typically performed when connectivity is available), by interacting with the ServerManager.

On the contrary, reactive migration is triggered when a mobile device requests explicitly to move a virtualized function; this is the only possible solution when connectivity to the cloud is unavailable. Once triggered, the ServerManager checks the possibility to execute the virtualized function required on the MEC node and forwards to Elijah the request for the creation of a VM/container to execute the task. Finally, the ServerManager returns to the mobile device the reference to invoke the specified VM/container for future direct VM/container-to-device communications.

Cloud Layer

The cloud layer is used to assist the MEC intermediate middleware to provide proactive analysis about users movements. In addition, it provides base VM images and overlays used by the service, acting as a backup repository where newly created base images and new/updated overlays files are stored. In fact, in this solution, the cloud layer has seldom updated snapshots of the complete status of the overall deployment environment, by receiving periodical updates by MEC nodes about users movements; it stores all users location history and performs high-computation predictions on cloud-stored location data. The location prediction performed is probabilistic (over neighbor MEC nodes). If the detected probability of a user to change its current MEC node is higher than a configurable threshold, the cloud layer sends a request to the ServerManager on the new target MEC node to start the migration timely.

In addition, the cloud layer is used during the initial service setup operations and in case of anomalies, to send missing files (see below) to the involved MEC nodes, thus acting as a global repository for MEC nodes. In fact, the cloud layer stores, with periodical snapshots, all the base VM/container images to use for node setup and all the overlays specifically created by MEC nodes for any supported mobile services. In the proposed architecture, it is the cloud layer to have the responsibility to react and recover MEC nodes when anomalies or errors occur.

In particular, the proposed solution is based on three main guidelines, that characterize its effectiveness and originality: i) mobility prediction, that keeps track of the history of users movements to predict the next movement and triggers the proactive migration in a timely manner; ii) proactive VM/container migration, that allows moving stateful virtualized functions at runtime among different MEC nodes to meet users high mobility requirements and to support service continuity; and iii) Elijah platform usage into a MEC architecture to allow the execution of resource-intensive tasks close to mobile devices also in hostile environments.

4.2.2 Elijah/Openstack++

Openstack [133] is a project started in 2010 as a joint project of Rackspace Hosting and NASA and currently managed by Openstack Foundation. It is a well-known and widely diffused open-source IaaS that provides many services that interact with each other to deliver the full feature set and to be able to manage computation, storage, and networking resources to supply dynamic allocation of VMs. Elijah [134] is a notable MEC-oriented extension of the Openstack infrastructure, with a relevant and growing community of MEC developers working on top of it. Elijah configuration is based on the Openstack installation, thus with very similar functionalities, and specifically targeted to run the intermediate middleware layer, by the load of the Elijah extension libraries to specify the MEC platform. In the proposed extended platform, the standard Elijah nova.conf configuration file has been modified with the specification of the CPU models, so to enable the resource-aware handoffs between heterogeneous hardware and to allow interoperability among different MEC nodes. In fact, since a deployed VM gets the characteristics of the host CPU through the hypervisor and the flags for hardware virtualization, through the virsh APIs it is possible to interact with the libvirt driver to manage the KVM/QEMU iterations and determine whether CPUs are compatible, then specifying them into the enforced nova.conf file. It is also possible to dynamically add or remove features to the platform, in the same way used in Openstack to add extensions on the default version, by adding custom files into standard paths, e.g. cloudlet.py, cloudlet_api.py, etc. In addition, since Openstack is a complex tool that contains all the typical functionalities required by cloud environments, Elijah also provides a more lightweight version, called standalone version [135]. Elijah standalone version is completely uncoupled from OpenStack, allowing to test features in a much easier way and could be very suitable particularly for testing purpose and providing tools to perform VMs synthesis, without the handoff mechanism, starting from an overlay, independently from the other platforms.

The most relevant Elijah features at the base of the proposed work are: base image import, base image resume, overlay creation, and VM synthesis.

BASE IMAGE IMPORT. A VM base image can be imported offline into Glance to load in advance the base image that will be used to build each VM. Each base image is a compress file that contains [136]: base disk image with the related hash value list; a memory snapshot with the related hash value list; is_cloudlet flag that indicates that is not a standard cloud image; libvirt configuration with the metadata that indicates the characteristics of the VM generated with the base image. The Elijah command to import a base image is cloudlet import-base <base_image_path> that decompresses the base image and stores it into Elijah database with the assignment of a unique hash to be identified unequivocally.

BASE IMAGE RESUME. A base image resume is still an offline operation and usually follows the import base image. During resume base image, a developer prepares a back-end server at the middleware layer and typically this phase includes: preparing dependent libraries, downloading and setting executable binaries, and changing OS and system configurations [136], as analogy happens in Openstack when a snapshot is resumed. To resume a base image, the Elijah platform uses a cloudlet hypervisor driver class, called CloudletDriver, that inherited the original LibvirtDriver and check if the metadata associate to the virtual disk image base has the is_cloudlet flag. In this case the driver resumes the base VM from the snapshot, rather than boots a new VM instance. Usually the first time a base image is resumed it takes a long time, in the order of a few minutes in relation to the hardware capability of the host, but since it can be executed offline, it is performed in advance preparing the MEC node before to receive the users requests. In this way, Elijah imports the base image into the cache of the compute node, thus, it does not slow down the system and is not significantly perceived by the users for further base image resumes. At the end of this operation, there is a VM ready to execute the service.

OVERLAY CREATION. This feature aims to create a minimal VM overlay starting from a resumed or running instance and then compress and save the VM overlay in Glance storage for later download. VM overlay is able to create snapshots used later to resume the VM from a specific moment, by containing the delta between the client VM and the base image VM. It contains all the changes needed to add on the base VM to reproduce the client VM environment at the moment of the migration. This functionality has been added with the extensions mechanism, defining a new virtualization driver CloudletDriver class that inherits nova rpc.ComputeAPI [136]. The Elijah command to create a customized VM based on top of the base VM is cloudlet overlay <overlay_path>.

VM SYNTHESYS AND HANDOFF. VM handoff allows VMs to migrate between different Openstack nodes. Since it involves two independent nodes, it is necessary that the user has the permission to access them, in order to call the APIs and they are contained into the message payload together with the destination URL [4]. The command to execute the handoff is through a Python file, called cloudlet_client, that requires the UUID of the VM to migrate and the credential to access both the Openstack. It is possible to perform VM handoff only if the VMs have been synthesized. VM synthesis launches a new VM instance to the Openstack cluster. It uses an HTTP POST message with the overlay_ulr parameter and this message is handled at CloudletDriver hypervisor driver that manages the VM spawning methods to perform VM synthesis using the VM overlay and the VM base image [136]. The synthesis mechanism is invoked with the commands synthesis_server for the server that starts to listen locally and synthesis_client with the specification of the server IP and the overlay URL.

4.2.3 ServerManager

The ServerManager component interfaces mobile devices and cloud requests with the Elijah platform, by managing all resource and communication requests, and by acting as a dispatcher for requests towards the MEC node. The ServerManager advertises its features by sending multicast periodical messages containing the type and the description of the available MEC functions through a discovery server and can be autonomously discovered by devices. The discovery service uses Avahi [137], an open-source Zero Configuration Networking (Zeroconf) [138]. Zeroconf is composed of a set technologies that allows to automatically configure IP networks, in absence of configured information from either a user or infrastructure services, e.g. DHCP and DNS servers. Zeroconf implements some main functionalities [139]: i) automatic network address assignment, by introducing a link-local method of addressing coupled with the IPv4/IPv6 auto-configuration mechanism; ii) automatic distribution and resolution of host names, with Multicast DNS (mDNS); iii) automatic location of network services to find services over the network though the DNS-SD process; iv) multicast address allocation, using the Zeroconf Multicast Address Allocation Protocol (ZMAAP). Avahi is one of the major Zeroconf implementation, based on Linux, that allows to locate the services inside the local network and a new host to view other hosts and communicate with them in the network. The Avahi daemon uses the core libraries to implement a DNS multicast stack accessible through the advertise of XML files in the /etc/avahi/services folder that exposes a service and its characteristics. It also uses nss-mdns and avahi-dnsconfd libraries to access the discovery system and to interact with the DNS via command line. In addition, Libavahi-client library adds to the original libavahi-core APIs the DNS multicast service, e.g. avahi-publish, avahi-brose, avahi-resolve.

When the devices or the cloud contact the ServerManager by specifying, respectively, the virtualized functions needed for mobile service execution and the proactive migration invocation, the ServerManager checks on the MEC node: i) service availability, by verifying the possible presence of useful VMs (VM caching) to be used for the running service request. In case there are not suitable VMs available, it forwards the request to Elijah by specifying the overlay to start the VM provisioning; ii) the base image availability of the target service, usually located in Glance and pre-loaded by Elijah.

When a mobile device send requests, the cloudlet server checks on the MEC node how to serve them:

- Base image availability request. The cloudlet server checks the presence of the base image inside Glance with JCloud [140] and returns to the client the availability or not of the target service. Usually the base images are in Glance and are pre-loaded by Elijah. Anyway, this control is present in case of anomalies that may force Elijah to contact the cloud to import the base image from it.

- Service availability request. The cloudlet server checks the presence of VMs already created previously to be use for the service request. In case there are not any VMs available, the cloudlet server forwards the request to Elijah specifying the overlay and Elijah synthetizes a new VM starting the provisioning operations. At the end of the synthesis, the cloudlet server returns to the cloudlet client the IP address and port of the VM.

Finally, the Network Time Protocol (NTP) [141], a widely-used synchronization mechanism among a set of distributed computer clock, is used on every ServerManager synchronize different MEC hosts and to allow a good reciprocal communication.

4.2.4 Migration

The solution is based on the ability to automatically perform virtualized function migration in a proactive way by pre-loading and pre-configuring the needed VMs in advance on the right MEC node. In this way, when mobile devices actually perform their handoff, the ServerManager checks the VMs available and returns the IP and port to access those VMs by avoiding any communication/load to Elijah. The predictive handoff is complemented by a reactive behavior, necessary to cope with the few cases when mobile devices request for the migration and the needed VMs have not been migrated in advance, e.g., in the case of unexpected users movements: service handoff is performed in this case via requesting the discovery of a nearby MEC node.

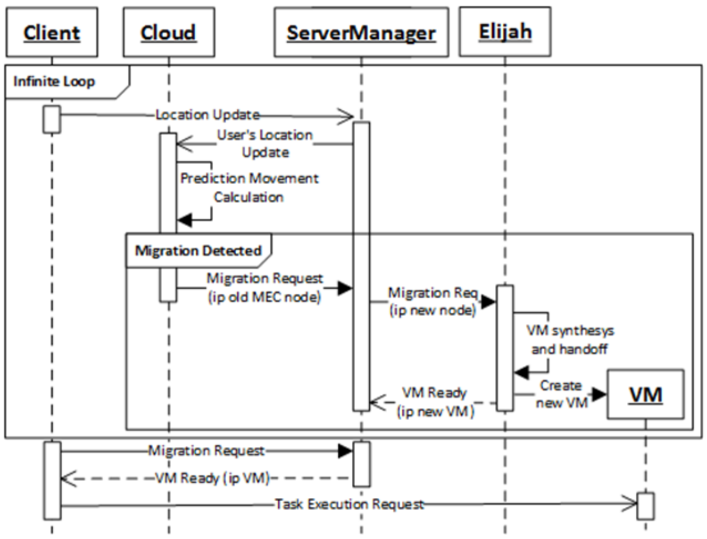


Figure 4.2: Our Proactive Migration Procedure

As depicted in Figure 4.2, the proactive migration stores location data (latitude and longitude) on the global cloud to compose temporal data series, by passing through the ServerManager. Periodically the cloud performs data analysis on those historical series, in order to predict next users locations. To this purpose, it is applied a polynomial non-linear regression, a type of regression where the relationship between X and Y is modeled as an N degree polynomial expression and fits the nonlinear relationship by using the LibSVM [142] machine learning toolkit, which makes available a set of libraries for Support Vector Machine:

$$K(xi, xj) = (x0ixj + coef0) * d \tag{4.1}$$

where d is the degree of the hyperplane.

A regression-oriented technique is used, instead of other discrete statistical models, e.g., a Markov model widely used in the literature for migration predictions, because it has a more accurate forecast, depending on historical data and implicitly considering other elements, such as direction/speed of movement, temporal/spatial data locality, and delta between consecutive sensed locations.

The regression analysis uses a data training set composed of the historical data associated to the movements of a specific user and is based on an initial training phase and the prediction phase. The training phase analyzes the first part of the dataset (approx. first 80% of the training set), to find the best input combinations in relation to the correlation coefficient and mean squared error. The prediction phase (approx. last 20% of the training set) compares the prediction with the target results known in advance and apply the one-step ahead forecast where after each value predicted the previous model is updated to forecast the next one.

Note that regression analysis requires more resources than discrete models, for both storing large amounts of data and processing them; however, the MEC infrastructure does not suffer of performance degradation because this processing is performed on the global cloud and in offline mode.

In particular, the regression analytics results are used to predict the likelihood percentage of a given user to move under a given MEC node, by considering the subset of MEC nodes in the users proximity. Finally, when the cloud sends a migration request to the ServerManager, the VM migration takes place and Elijah is seamlessly integrated to execute both VM synthesis and handoff procedures through it.

4.2.5 VM Synthesis and Handoff

ServerManager receives from either users or the cloud the migration request and forwards it to Elijah, with the indication of the base image to instantiate and the possible overlay(s) to add. In this way, Elijah receives the specification of the service to create, though the base image, and the users modifications, though the overlay(s), thus, allowing to perform the live migration of stateful virtualized functions at runtime.

On the one hand, the time needed to synthesize a virtualized function may significantly vary in relation to which configuration and files are already available on the MEC node: i) Elijah has the base VM and retrieves the overlay locally at runtime, thus, can start to execute tasks to serve the mobile devices just after having added the overlay at the present base VM; ii) Elijah has the base VM but not a local overlay of the service and it needs to retrieve it from either clients or the global cloud; iii) or Elijah needs both the base VM and the overlay. VM synthesis allows a VM, that is running on one node, to freeze its state and resume at any node, at runtime, to execute near the mobile device. The synthesis is performed with the delivery of an overlay, specified with an URL by the mobile device, that applies the deltas over the base VM on the MEC node. Figure 4.3 shows the synthesis procedure of the proposed solution.

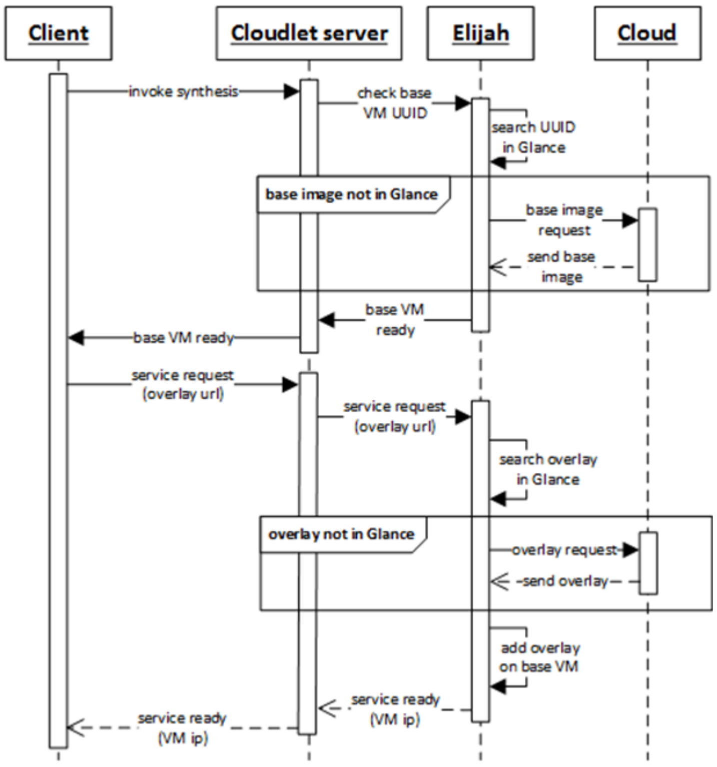


Figure 4.3: Synthesis procedure

On the other hand, VM handoff is the higher-level MEC platform feature that allows running VMs to migrate between different MEC nodes at runtime, typically to better support service continuity and to preserve service state notwithstanding MEC-related mobility. The handoff procedure is completely transparent to the client and only needs to invoke the handoff functionality by specifying the IP of the current MEC node to the new MEC one (to enable the retrieval of service state). Figure 4.4 shows the handoff procedure of the proposed solution.

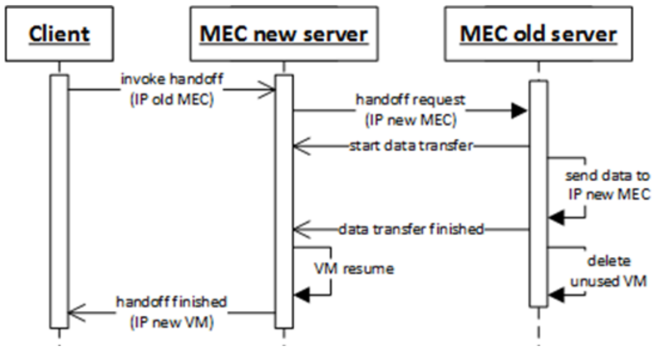
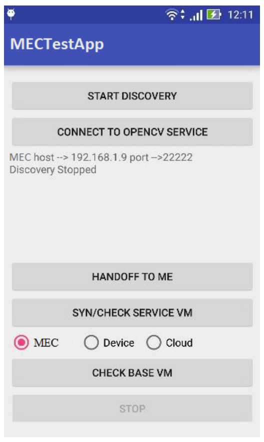


Figure 4.4: Handoff Procedure

4.2.6 Mobile Services Usecase

Figure 4.5 and 4.6 illustrate the two Android applications that have been developed: MEClient, to manage the communications between MEC layer and mobile devices; openCV application, that is used by the users and wants to perform face recognition analysis.

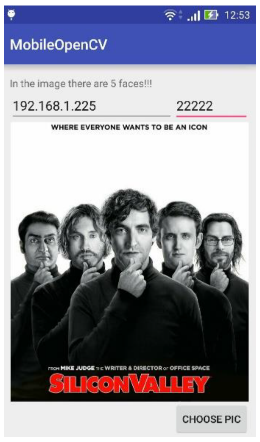
The MEClient has many functionalities to manage the communication between mobile device and the ServerManager. The start discovery button enables the reception of multicast messages from the cloudlet servers to be aware of the presence of a MEC node. The



MEC host --> 192.168.1.9 port -->22222
Discovery Stopped

[✓ MEC] Device Cloud

Figure 4.5: MEClient Application



In the image there are 5 faces!!!

192.168.1.225 22222

Figure 4.6: OpenCV Application

MEClient scans the suitable services exposed at MEC layer and retrieves the IP address and the port to connect to them. MEClient application uses the Network Service Discovery (NSD) [143] that allows users to identify MEC servers in the local network. The NSD ServiceInfo class, provided by Android, sets all the service parameters, e.g. name, type of transport protocol, and port, then the RegistrationListener detects any events related to the discovery and, in case the service found is correct, the connection information are retrieved with the resolveService method.

Successively, with the check base VM and the syn/check service VM buttons, the MEClient requests to the MEC server the availability, respectively of the image base VM and the target service, on the MEC node and obtains the parameters to access the remote VM and interact with the OpenCV servers. Once the service is ready on the MEC node, the MEClient enables the "connect to OpenCV service" button, usually disabled unless a VM has been already used and is still ready to serve requests. By clicking on this button, it is shown a list of possible Android application that can be executed on the VM created select the application to use.

In the hostile scenario considered, the service OpenCV can be interrupt unexpectedly at anytime by the device mobility that can lose the connection with the MEC node. In this case, the MEClient stores the history of all the MEC servers used previously by the mobile device and with the handoff to me functionality allows to select the old MEC node containing the VM to migrate and invoke the handoff towards the new MEC node.

Finally, the implemented OpenCV application selects a picture and sends it to the target VM specifying the IP address and port previously obtained. After the application establishes the connection once, it does not require the MEClient anymore because it is already aware of the parameters to access the MEC node and it connects directly to the VM. The VM receives and processes the picture detecting the faces inside the picture, with OpenCV framework, returns the number of them and wait for the next request.

The usecase developed consider the usage of mobile service for hostile environments as the combination of two Android apps: ClientManager and DetectApp. On the one hand, ClientManager handles the communications towards the MEC node, through the reception of multicast messages from the distributed ServerManagers to be aware of the presence of a MEC node in the neighborhood.

It requests the ServerManager to check availability and retrieves the parameters to access the suitable remote VM. In addition, to enhance availability in hostile environments, ClientManager stores the last MEC servers used in the past in order to restore remote connection to the nearest MEC node and promptly+locally invoke the migration towards a MEC node in the case of anomalies such as cloud connectivity loss.

On the other hand, DetectApp has the role of commanding high-computation face recognition analysis and sending the related picture to the target VM, by specifying IP address and port obtained from ClientManager. After DetectApp establishes the connection, it works independently from ClientManager and receives updates directly from the used VM.

Virtualized functions has been used to specialize a VM with the functionalities required by the user, by applying the needed overlay (mainly consisting of the libraries characterizing the specific mobile service) on the VM created with the base image resume. In particular, in the running case study OpenCV [144], i.e., an open-source library including a set of optimized algorithms to process media files, has been extensively used to detect and highlight human faces inside an image. OpenCV already contains the implementation of the Haar Classifier library for face recognition, based on several weak classifiers, which perform simple checks, e.g., pixel difference operations, combined to produce the decision. The related haarcascade_frontalface_alt.xml file is used to build an instance of the CascadeClassifier class, specializing it into face detection. The classifier is first trained and then applied on the region of interest and in the target image, moving it on all the image to check if any location is likely to show the searched object, i.e. a face. The CascadeClassifier applies a resultant classifier that consists of several simpler classifiers at different stages that are applied subsequently to the region of interest until at some stage the candidate is rejected or all the stages are passed. In particular, the detectMultiScale method, that detects objects of different sizes in the input image and returned as a list of rectangles, uses the MapOfRect matrix to store the data related to the face detection. The server stores locally a copy of the image with a red box around faces, while the number of faces detected are returned to the mobile devices.

The resultant composed classifier, composed of the weak classifiers applied subsequently, is applied to all the region of interest of the target image to check if any location is likely to show the searched object, i.e. a face. Finally, the server stores locally a copy of the image with a red box around faces, while the number of detected faces are returned to the interested mobile clients.

4.2.7 Experimental Results

The operational environment used to validate and assess the performance of both MEC platform and the running case study consists of: one Android smartphone, one server on the cloud, and two MEC nodes with heterogeneous CPU, i.e., AMD Phenom II X4 965 and Intel Core i7 2640M, each with 8GB RAM, 100GB HDD. Each VM managed by the MEC platform in this operational environment is equipped with 1VCPU, 1GB Ram, and 8GB HDD.

Interesting results relate to the many tests performed about synthesis and handoff. The overlay considered and retrieved locally in Glance, as it happens during normal system execution, has a size of 106MB for newly created services with approx. 26MB of users data. During handoff, the MEC platform automatically splits this overlay into several chunks, of limited size to facilitate and parallelize their transmission. The vicinity of adjacent MEC nodes, highly distributed in the execution environment, and the generally powerful and stable connectivity among MEC nodes, as assumed in [134] and also in several other research activities like [145, 146], make the overlay transmission an operation with relatively small overhead in the performed experiments if compared with data compression/ decompression; thus, below the focus is mainly on computing capacity and overall node workload.

The average measured times of the main steps that compose the synthesis procedure are resumed in Table 4.1 and the handoff procedure in Tables 4.3 and 4.5. The results have been obtained while migrating equivalent VMs, i.e., processing the same number of requests from clients, and using each node both as sender and receiver to simulate the randomness related to the clients usage. The reported results are average values over hundreds of runs, with a standard deviation always within 2%.

Table 4.1: Synthesis Measurements

Process	Time (s)
VM Migration Check	0.29
Image Creation	2.32
VM Restored	1.54
VM Resumed	0.09
VM Resumed	2.28
Synthesis Procedure Total Duration	98.62

Table 4.3: Synthesis Measurements

Process	Time (s)
Data Compression	121.64
Data Transmission	120.02
Handoff Procedure	165.74

Table 4.5: Synthesis Measurements

Process	Time (s)
Data Reception	125.52
Spawning	168.37
Data Compression	120.23

The overall time to complete handoff has resulted to be 172.52s, with a total VM downtime (generating mobile service unavailability and service interruption) perceived by the client of only 1.60s thanks to the proactive approach.

About the resource usage, Table 4.7 and 4.9 resume the number of threads, RAM and CPU percentage used by, respectively, the two main processes executed during synthesis procedure, i.e. cloudlet_vmnetfs used to enable on-demand fetches of VM disk/memory and cloudlet_qemu-system-x86_64 to start VM before having entire memory snapshot, and the sender/destination hosts during handoff.

Table 4.7: Resource Usage

Process	Thread Number	RAM (MB)	CPU (%)
Cloudlet_vmnetfs	6	65	25
Cloudet_qemu-system-x86_64	4	240	3

Table 4.9: Handoff Processes Measurements

Process	Thread Number	RAM (MB)	CPU (%)
Handoff Sender	14	2.06	25
Handoff Destination	6	1.04	25

Tests have also compared the overall performance between the case of MEC-assisted execution and the case of global cloud-assisted execution. As in realistic cases, the MEC layer and mobile devices are on the same local network at one-hop distance, the available network bandwidth of client-to-MEC and client-to-cloud, respectively, was 53.3Mb/s and 2.44Mb/s. In the case of small-size image of 1 MB for the described virtualized functions, the migration time is already non-negligible for the 2-layer cloud solution (i.e., 3.04s), while still acceptable for the proposed 3-layer MEC-based approach (i.e., 0.3s); in the case of VM images or overlays of tens of MB, mobile services for many application domains are unusable without the MEC-oriented approach.

The experimental evaluations show the effectiveness of the described proposal in relation to the targeted MEC technical challenges and highlight the deep gap of performance achievable with/out exploiting the intermediate MEC layer. Although the live migration procedure proposal introduces non-negligible latency, mobility prediction has demonstrated to have the potential to anticipate migration requests sufficiently to have no service interruption. In this way, it is possible to enable a wide range of mobile services to offload application-level functions on the MEC layer and developers are encouraged to adapt application structure to the opportunity of runtime migration of tasks for remote execution, also in case of hostile environment. In particular, the relevance of such an approach for different families of mobile services: high mobility services that do not suffer from possible latency associated with frequent migrations; low-latency services that benefit from both nearby MEC interactions and predictive migration; file-exchange services that can avoid upload-ing/downloading files to the cloud, thus benefitting from usability also in case of no global connectivity and reducing slowdowns.

4.3 Automated Offloading for Computation/Energy-usage Optimizations

This solution presents an innovative task selection algorithm able to identify the most suitable Android application methods to be dynamically offloaded, on either remote cloud resources or edge nodes with virtualization capabilities, thus providing a crucial support element to leverage the adoption of mobile offloading techniques for apps of industrial interest. In particular, the proposed solution implement and evaluate an offloading tool that can autonomously scan a generic Android mobile application, without a-priori and application-specific knowledge, to autonomously create a new version of the application, which is functionally equivalent, but with the capability to offload computations to a remote server; the algorithm analyzes and dynamically partitions the application into offloadable and non-offloadable methods; then, it is able to prioritize and select the most suitable tasks to be offloaded. It is a fine-grained offloading decision with a granularity at the method level. It is achieved this by a careful automated analysis to detect which methods can be executed on the server consistently and by marginally modifying those methods code to run them remotely in a completely transparent way. The result is then integrated into a new Android application that runs consistently some methods locally on the mobile device and others remotely on the server. To achieve this goal, an existing prototype from [129] has been improved and extended in two main directions:

- **Autonomous method selection.** The tool includes an algorithm to autonomously select the methods suitable to be executed both locally and on the remote server, through a complete scan of the: i) APK file; ii) the classes developed by the programmers; iii) the methods for each class. The selected methods are then sorted in terms of the number of offloadable method calls.

- **Translate methods and execution on a remote server.** The extension has added the ability to offload every suitable method whatever: i) its input parameters, by extending the methods translation code generation to support complex objects including lists and arraylists; ii) its state, being either static or non-static method; iii) the static or no-static variables used internally in its body with the related object management.

4.3.1 Autonomous Methods Selection

The autonomous selection algorithm takes any Android application as input (only APK package, no need of source-code) and detects the included methods that are suitable to be dynamically offloaded. The proposed solution is designed to be general-purpose and independent from application type, domain, size, and internal structure. In particular, the algorithm starts by scanning the whole application structure, by applying both incremental checks on each single class/method and dependency checks among different classes/methods. More specifically, the selection algorithm starts by retrieving the Android manifest file of the considered application, as shown in Figure 4.7.

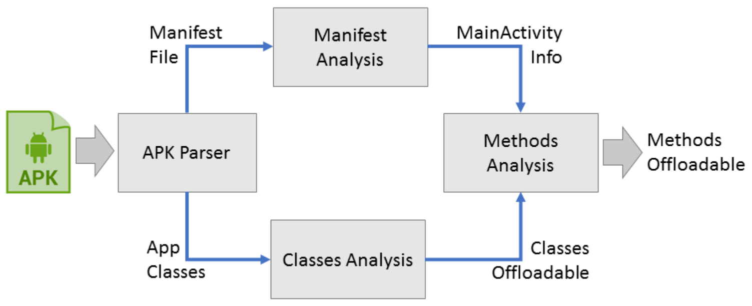


Figure 4.7: Tasks Selection Algorithm Architecture

The Android manifest is a file, located in the root directory of every valid Android application; among the others, it defines the requirements to run the corresponding code in the Android platform and is used to identify the MainActivity of the application. Indeed, as shown in Figure 4.7, during the method analysis, the analyzer iteratively checks multiple requirements to analyze first classes and then methods to discard from the list of offloadable candidates by using a predefined set of checks. The class analysis is the initial component to detect which methods to offload. The purpose is to determine the non-offladable classes present into an application, and obviously to discard their methods during the following analysis. The method analysis scans each method by using multiple checks and then integrates the method requirements with the information previously obtained about MainActivity and the other application classes from the manifest file. About the considered criteria to take the offloadability decision, the purpose is to minimize the needed computation while achieving an overall accurate result anyway. Table 4.11 concisely lists the criteria that the algorithm considers for both classes and methods.

The internal usage check determines whether a class or method is a compiler-generated one created by the Java compiler or used to manage the application lifecycle. For example, anonymous inner classes or <clinit> methods that are static initialization blocks

Table 4.11: Checks Automatically Performed by the Proposed Algorithm

Type of Checks	Classes	Methods
Internal Usage	✓	✓
Class Offloadability	✓	✓
Internal Objects Calls	✗	✓
Internal Methods Calls	✗	✓

for the class, as well as static field initializations, are not offloaded. The class offloadability checks determine if a class is offloadable by understanding whether the class extends a non-offloadable superclass or one of its method belongs to classes already classified as non-offloadable. The internal objects and method call checks are performed only at the method level in order to detect if a method contains a call towards non-offloadable classes/objects/methods, e.g., Android native objects or methods.

It is worth noting that the algorithm classifies a device-dependent class that includes objects impossible to be available/migrated to remote cloud/edge nodes as non-offloadable, e.g., native Android libraries or non-serializable Java objects (i.e., Threads). All methods that are included into a non-offloadable class are classified as non-offloadable. Vice versa, an offloadable class is a class that successively passes all the class checks and whose methods will be considered as possible offloading candidates (after method-level checks). In fact, only after a method passes all the incremental checks in the table, it is marked as offloadable.

The implemented task selection algorithm is composed of many components for the methods checking analysis, which will be explained in the following, as shown in Figure 4.8.

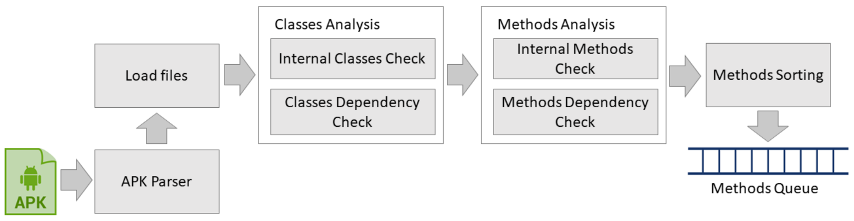


Figure 4.8: Methods Selection Algorithm

APK Parser

The APK parser retrieves and scan the Android manifest file from an APK file provided. The Android manifest analysis provides essential information about the application. The manifest file contains information about the internal application structure, in particular it lists packages in the application and the permissions the application must have in order to access protected parts of the API. It also contains information about the application components that compose include activities, services, broadcast receivers and content providers; for each component, it indicates the class that implements that component, it publishes its capabilities, e.g., which activity is the main activity, and how it can be launched.

For each application, it retrieves the Android manifest file and parses configuration XML entries for Android activity until the MainActivity, which is identified with the Intent filter sets to android.intent.action.MAIN. The algorithm uses the MainActivity information to detect part of the internal methods for the further methods analysis, as explained in the following section.

Load Files

The solution leverages on the Soot framework [147] to retrieve, parse and manipulate class files methods. Soot provides a set of Java APIs to modify the bytecode from APK file. Jimple [148] is an intermediate representation of Java bytecode used to optimize modifications of existing java byte code.

A list of keywords has been defined in relation to the Android functionalities contained into the native Android libraries that represent the components that cannot be offloaded. All these keywords are stored into configuration files, loaded at the startup, that are used, during the following analysis, to detect the Android-dependent components and, thus, to discard the non-offloadable methods. The keywords are divided into some text files in relation to the type of usage every keyword is associated to:

- Application lifecycle management. Mmethod calls to Android activity classes that defines device specific behavior and state changes, e.g. creating, stopping, resuming, or destroying a device specific activity.

- Application GUI management. Keywords that contain all the methods to manage a View, which is an object that draws something on the screen that the user can interact with, or a ViewGroup, a set of View objects hold together with a layout interface, with functions of display notifications, communication of device status, and device navigation.

- Application events management. Keywords that contain the methods to allow the interactions of the users on the mobile device via touch gestures, e.g. onEventListener, onClick, etc.

- Application input/output management. Keywords that contain all the methods used to manage input/output built-in components that allow the device to interact and communicate with the external environment, e.g. all the categories of sensors (motion, environmental, position), camera, and so on.

Classes Analysis

The classes analysis phase aims to minimize the number of offloadable classes that belong to the package path among those loaded by Soot framework. The classes analysis aim to detect which classes cannot be offloaded and which can to be scanned in the next phases to detect the specific methods to offload. The classes analyzer uses a set of tests to detect which classes are not suitable to be offloaded that can be resumed as follows:

- Internal Class Test. It checks whether a class is internal and, thus non-offloadable. In particular, the internal classes are the anonymous inner classes, because they do not contain the associated class name and, thus, the name the Java compiler gives them cannot be associated to a specific class name in the code. The anonymous inner classes name ends with the suffix $+number and the overall name is in the form class_name+$+number. Note that the proposed algorithm, in general, allows to offload inner classes, which are identified with the name class_name+$+inner_class_name.
- Android Class Test. The Android package contains classes that may contain device specific information. The component inspects which classes belong to the Android path. This step is not mandatory from a functional point of view, because all the Android classes can be detected from the next methods analysis, but it allows to discard a high number of classes and, thus, to alleviate the computation required and speed up the system performance.
- Superclass Test. For each non-internal class, the algorithm checks if it extends an already defined non-offloadable classes. These classes are not allowed to be offloaded because a child class is very likely to call or depends on the constructors/methods of the parent class. In fact, due to the inheritance and polymorphism concepts, the methods are platform-dependent because they will use non-offloadable methods from the superclass. For example, if there is the MyThread class that extends the platform-dependent Thread class, it will use the Thread management functionalities.
- Dependency Test. It checks the dependencies of each offloadable class checking if a class extends a non-offloadable superclass. In this case, the child subclass inherits all the public/protected variables and methods of the parent class and is very

likely to use them, thus, will be likely called a non-offloadable object or method during subclass lifecycle. Since the number of classes to check may be high for large-size applications, the algorithm optimizes the computation by limiting at each iteration the number of classes to check to the only ones strictly necessary: the offloadable and non-offloadable methods found during the previous iteration. The check finishes when there is no new non-offloadable classes are found during an iteration. In this way, the number of classes to check is always relatively small and can be processed in a short amount of time, i.e. negligible into the overall algorithm time. The pseudo code of the classes dependency check is the following

```
do {
    offloadClasses_new = {}
    notOffloadClasses_new = {}
    for (clazz : offloadClasses) {
         if (notOffloadClasses.contains(clazz.superclass)) {
            notOffloadClasses_new.add(clazz);
          } else {
            offloadClasses_new.add(clazz);
          }
    }
    notOffloadClasses = notOffloadClasses_new;
    offloadClasses = offloadClasses_new;
} while (!notOffloadClasses_new.isEmpty);
```

The classes analysis, apart for the functional perspective, is important also because it allows to restrict the amount of classes to scan before to check their methods, highly decreasing the overall computational time required, particularly when it deals with a large-size Android application with thousands of methods.

Methods Analysis

The methods analysis follows the class analysis, parsing each method body of the offloadable classes in a fine-grained way. The algorithm used to check offloadability is mainly composed of two parts; a main control steps that checks the suitability of each method to be offloaded and the dependency checks that analyze the dependencies among methods. The main control steps are the following:

- Internal Method test. It first determines whether the methods are internal and device-dependent by using three criteria:
  - Static initializer. The method represents a static initializer that is used to initialize the class object itself. The method is not offloaded because it is a static method added by the Java compiler and called by JVM after class loading. The static initializer is indicated with the <clinit> method name.
  - MainActivity usage. The method contains the MainActivity among its input parameters. The method is used

internally by the Android compiler to manage the MainActivity class or to use the context of the MainActivity for the application startup.

- Class test. If the method belongs to a class marked as not-offloadable, the methods in the class cannot be offloaded because it is likely to access not-offloadable objects, methods or variables of the class that belongs to non-offloadable.

- Objects Calls test. It checks if the methods contains platform-dependent calls towards objects that cannot be offloadaded. These classes marked as non-offloadable during classes analysis.

- Keywords test. It check if the methods contains platform-dependent methods calls towards methods that cannot be offloadaded. These methods are in the blacklist keywords that are loaded previously from the configuration files.

During the dependency check phase, the algorithm checks which methods are called during the execution of each inspected method. It is possible to determine, for each method, its dependencies methods and discard it if at least one is non-offloadable because, in this case, it will be invoked during the method execution. To this purpose, it checks the body of each method to see if there are any calls to the, already found, non-offloadable methods.

To scan dependencies of methods and minimize the procedure, the algorithm adopt the creation to build a directed graph where each node is a method and each edge is a method call directed from the caller method (called as parent node in the following) towards the callee method (called as child node in the following). Each node contains: the signature of the method, to identify unequivocally one method among the others; offloadable variable, that indicates if the method is offloadable, non-offloadable or temporally unknown; visited variable, that indicates if the dependency check has already been executed for that node; parents list that contains the list of the caller methods. The algorithm used to create, modify and parse the graph is shown below:

- **Step 1.** It parses each method found in the offloadable classes to build a directed graph. In each method invocation from the method body, an edge from child to parent node is created and the visited flag is set for both nodes to false initially. At this step the graph is created.

- **Step 2.** The graph is parsed to detect which methods depend on non-offloadable method. Starting from the non-offloadable classes list found during the methods checks, and iterating through parents of non-offloadable method until the root, it is possible to recursively set as non-offloadable all the branch of the graph from the method to the root of the graph, excluding graph nodes without scanning all the nodes. Every time a node is scanned, the visited attribute is also set to true.

- **Step 3.** All the methods that are marked as offloadable from the graph are retrieved.

The first step is executed during the methods checks. It aims to create a directed graph, by expanding it with a new method at each iteration. The second step is recursive and takes place after the graph is fully created to scan the graph to detect the methods dependencies. The last step results a list of offloadable methods after the method call dependency test.

The same dependency check can be performed through an iterative approach where, for each non-offloadable method, the internal offloadable methods calls are checked. Since the target application may be large-sized, the iterative approach needs to be optimized in a similar way to what done for the classes dependency check, as shown in the pseudo-code, through incremental iterations that consider a limited number of methods each iteration. Unfortunately, due to the great amount of methods to scan, even with this optimization, the iterative approach is not feasible to check methods dependency within a reasonable amount of time. The graph approach explained is the best way to perform the methods dependencies check, and also allows to retrieve the list of offloadable methods within some seconds in relation to the application size, that is a quite impressive result considering the typical huge amount of methods to scan per application.

The computation time and resources needed to perform the methods analysis may be extremely high for both control steps and dependencies check. The time complexity for the control steps is $\Theta(n * m)$, where n is the number of methods and m the number of classes, because the agorithm scans both classes and their methods. The time complexity for the dependencies check with the graph approach is $\Theta(n)$, where n is limited to the number of non-offloadable methods. The proposed graph approach, using a directed graph and applying the graph properties to analyze the methods dependencies, decreases complexity. In case of a large number of methods, it is very inefficient and resource consuming to check dependency in the iterative manner. In fact, the alternative iterative approach complexity is $\Theta(n^2 * \log n)$, where n is the total number of methods. The iterative approach is composed of the complexity to parse all the methods presents in all the classes and the complexity to check inside each methods the presence of calls towards non-offloadable methods.

Methods Sorting

One of the main purpose of this solution is to create a tasks decision algorithm that can work with mobile computation offloading mechanisms or code optimizations, thus it must be general enough to cope as many of their requirements as possible. For example, some offloading mechanisms or code optimizations change the code of the application analyzed, i.e. ULOOF [129, 151].

When the code optimization is applied without considering method dependency, the translation of a method may result a number of issues. When it translates a method, it inspects the instances used in the method and the other method being called from the method. When the other methods being called is already translated, it may contain additional instances only related to enable offloading. When this happens, those additional instances are also inspected and causes offloading to malfunction. To avoid this, the algorithm needs to sort the list of offloading methods in terms of the method call invocation. A priority queue is introduces, where the methods are added with a priority coefficient (Wm) related to the maximum weight among the internal offloadable methods ones (Wi), as explained in 4.2

$$Wm = \begin{cases} 1, & if\ no\ internal\ method \\ max(W_i) + 1, & otherwise \end{cases} \tag{4.2}$$

then using the following algorithm:

- **Step 1 (priority 1)**: Search the methods that do not contain any calls to methods declared offloadable in their body, and put the method in the priority queue.
- **Step 2 (priority 2)**: Search methods that contains calls to offloadable method with at maximum Wm=1 in their body and put the method in the priority queue.
- **Step N (priority N)**: Search the methods that contains calls to offloadable method with at maximum Wm=(N-1) in their body and insert in the priority queue.

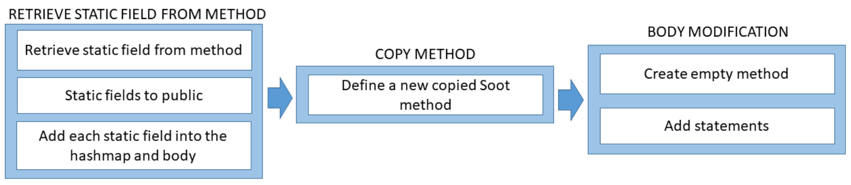
This results a queue with methods sorted in a decreasing order in terms of their weight, where the weight represents the total number of offloading methods being called by that method and by the methods being called from that method.

4.3.2 Method Translation and Optimization

The method translation and optimization refers to the modification of the methods selected as offloadable, after the application of the autonomous selection algorithm. Those methods are modified adding hooks into the code to let them to be executed on the mobile devices as well as on the remote server.

The post-compiler can optimize every method detected from the previous method selection algorithm into a Jimple pseudo-code. The optimized methods can be executed on a remote server at runtime regardless of the type of parameters passed as input arguments and whether the method is static or non-static. The framework from [129] was able to only offload methods with primitive parameters. The extension allows methods with the ability to offload also complex type of parameters by modifying the assignment portion of the Jimple code. The extension allows methods with complex parameters to be optimized for offloading by serializing the complex objects and send them along with the offloading request to the server.

The method translation algorithm in the post-compiler and the remote execution platform is further extended to synchronize the object instances between the devices involved in the offloading. While the previous method translation algorithm could offload only static methods that can only access static variables, the introduced optimization fully exploits object management, allowing to offload non-static methods that can access any variable both primitives and complex. When the method is executed remotely, the application: i) sends the variables to the remote server; ii) updates the values of those variables in the server-side; iii) invokes and executes offloaded method; iv) sends the values of the variables back to the client; v) update the client variables. Figure 4.9 resumes the main operations applied on a method suitable to offloaded.


RETRIEVE STATIC FIELD FROM METHOD
Retrieve static field from method
Static fields to public
Add each static field into the hashmap and body
COPY METHOD
Define a new copied Soot method
BODY MODIFICATION
Create empty method
Add statements
Figure 4.9: Methods Modification Procedure

The post-compiler: i) retrieves the method body and for each line it looks for assignments or invoke-statements and static fields; ii) modifies the modifiers of the method; iii) inserts each static field into the hashmap and into the body; iv) defines a new Soot method called offloadcopy_<method_name> with the same name, parameterTypes, returnType, modifiers, class, etc.; v) removes all method lines not related to the parameters passed, i.e. the body will be composed of variable declaration and parameters; vi) adds the statements to call the framework that allows to run the method remotely.

4.3.3 Experimental Evaluations

The selection algorithm has been tested using a wide range of different Android applications. The tests and the experimental evaluations have been performed on the top 250 most popular Android applications from the Google Play marketplace. After that, two popular applications have been analyzed more in detail to show the amount of computation it is possible to save. In particular, in terms of cyclomatic complexity metric, a measure of the maximum number of linearly independent paths in a program control graph, used to measure the amount of decision logic in a single software module. Note that the cyclomatic complexity does not captures all aspects of software complexity, but rather can serve as a useful engineering approximation to retrieve the complexity of an application [149].

Cyvis [150] is the tool to evaluate the real computation complexity. Cyvis is a free software metrics collection, analysis and visualization tool for Java based software. It parses a Java class file and calculate cyclomatic complexity of each method in the class file. The Cyvis tool has been extended by adding the retrieval and evaluation of inner classes, which are not supported in the latest existent version.

For sake of completeness, note that the tasks selection algorithm has been fully tested, also combined with an external computation offloading mechanism (i.e. ULOOF). The addition of a tasks selection algorithm that complements ULOOF creates a complete offloading platform that autonomously creates new Android applications with the ability to scan dynamically those applications code, modify them and offload some methods on a remote server, if necessary, without any human intervention or specific configurations. Additional information related to the interactions with ULOOF framework are available at https://uloof.lip6.fr.

Top 250 Most Downloaded Apk

The tests refer to the 250 free most downloaded applications of the US Android marketplace, at the 1st February 2017. Those applications compose an appropriate and statistical-relevant basis, with a proper range of heterogeneous and large-size applications, to analyze the results obtains because they include vary various applications in terms of types, domains, internal structure and so on.

The measurements are divided in classes and methods analysis, and follow the analyzer process. The proposed tasks selection algorithm has been applied to analyze those applications and measures how many classes are present into the whole applications and, then, how many classes are offloadable, classifying the classes in relation to the reason why they are considered non-offloadable. Figure 4.10 resumes the obtained experimental results using a Cumulative Distribution Function (CDF), very useful to show the comparison among different amount of classes classification.

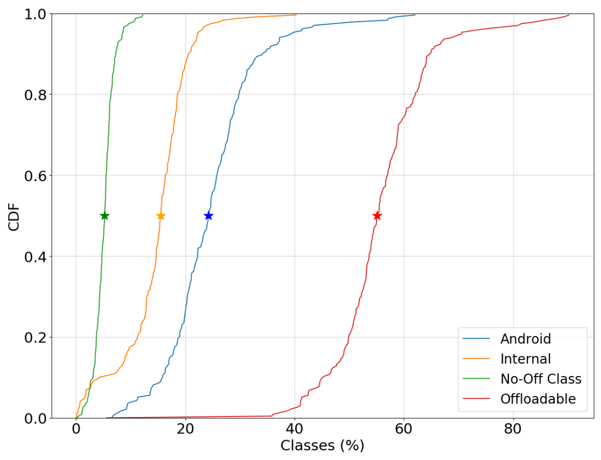


Figure 4.10: CDF on Classes Analysis

For each class, the algorithm indicates how many methods are considered offloadable and how many methods are discarded at each step, following the classification described previously. Figure 4.11 shows the finer-grained results obtained with the analysis on the top 25 application due to space limitation. Extending the analysis to the

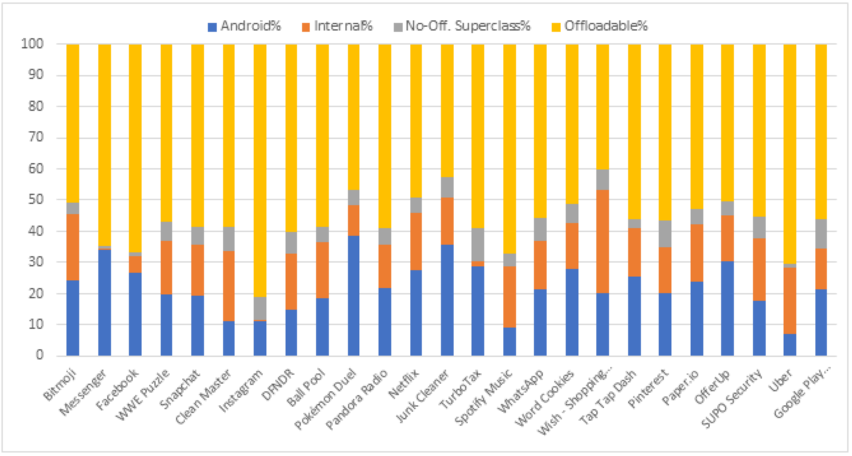


Figure 4.11: Classes Analysis

methods, the algorithm classifies them according to the classification described previously. Figure 4.12 resumes the obtained experimental results using the CDF distribution. Figure 4.13, similarly to the classes analysis, shows the finer-grained results obtained with the methods analysis on the top 25 application. The average percentage of methods suitable to offload on the top 250 application considered is 25.51%. Note that, without considering the methods including into the Android packages but only the feasible methods present into the aplication, the average percentage of offloading methods is 59.25%. In fact, a significant amount of the application overall methods is related

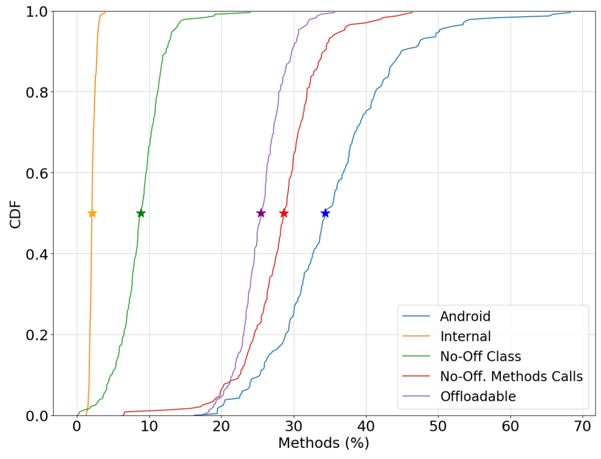


Figure 4.12: CDF on Methods Analysis

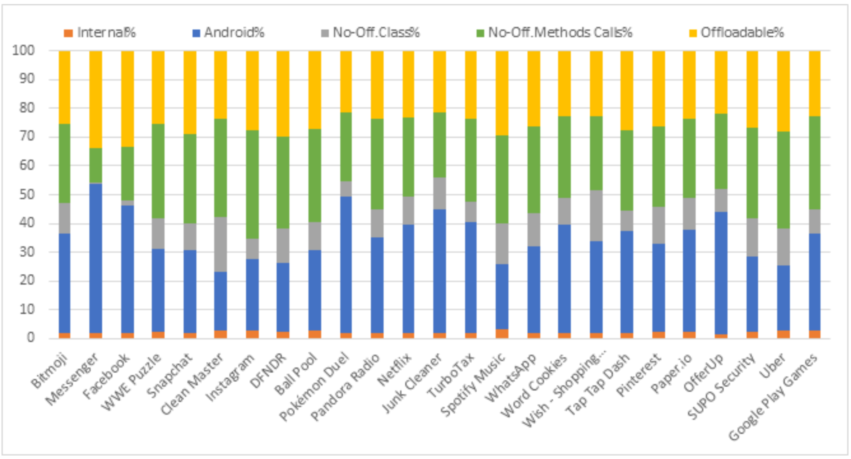


Figure 4.13: Methods Analysis

to Android classes that contain obviously mobile device-dependent methods and it is not possible to offload them to the edge platform that should use only a general-purpose JVM.

The results achieved are extremely relevant in terms of amount of methods to offload towards the remote server and, in particular, in terms of resource consumption savings. Some work in the literature highlight significant computation time and energy savings, with just few methods offloaded. For instance, [1] achieves savings in the order of 30%, just offloading a single method into a limited-size application. In the case described, it is not possible to run the same tests on the considered applications because due to unability to access the source codes, but time and energy savings results are easily predictable and the offloading of a great amount of methods may lead to a very high application optimization with an extremely significant reduction in both computation time and energy consumption.

The next paragraphs extend the results obtained, by providing a cyclomatic complexity analysis for a more complete estimation of the average saving achieved on the top 250 most downloaded applications and, more specifically, on Twitter and Uber usecases.

Cyclomatic Complexity

To evaluate the tangible savings introduced by the proposed algorithm on real-world applications, the cyclomatic complexity is evaluated, though Cyvis tool, on the top 250 most downloaded applications and, successively, on Twitter and Uber applications. These two applications are examined because they are very popular and medium-size applications among the ones considered.

Initially, the APK files is translated into jar archives, readable by Cyvis, with Dex2jar tool [152] that decompiles classes.dex into a jar file. The cyclomatic complexity analysis has been performed with Cyvis that provides: a tree structure, that shows all the packages in the project with the associated classes and interfaces it contains; metrics panel, that contains the metrics at different grained level, e.g. project view, package view, classes view with the list of methods and the cyclomatic complexity and the instruction count for each method. Extracting the Cyvis complexity analysis evaluation and combining it with the offloading and non-offloading methods found, it is possible to evaluate the cyclomatic complexity associated to each group of methods. Figure 4.14 resumes the CDF related to the number of offloading and non-offloading methods in the top 250 most downloaded applications. The average complexity suitable to be offloaded is about 20%

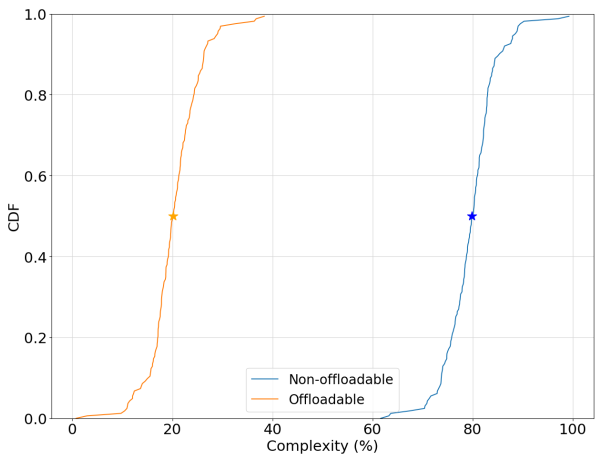


Figure 4.14: CDF on Applications Complexity

of the total application complexity. Note, as detailed better on Twitter and Uber usecases analysis, that a relevant part of the overall application complexity is intrinsically non-offloadable, i.e. Android libraries, and it is possible to act only on a portion of the total application complexity. As shown in Figure 4.13, the Android-related methods are, in average, the 35% of the total application methods. Thus, the cyclomatic complexity has been analyzed on the considered applications and Table 4.13 and 4.15 resume the results, respectively, on Uber and Twitter applications.

Table 4.13: Complexity Analysis in Uber App

Type	Complexity
Overall	55840
Non-Offloadable	37646
Android-related	3603
Offloadable	34.83%

Table 4.15: Complexity Analysis in Twitter App

Type	Complexity
Overall	104287
Non-Offloadable	79192
Android-related	36131
Offloadable	36.82%

where: overall, the total cyclomatic complexity of the application; overall non-offloadable, the sum of the complexity of all the non-offloading methods; Android-related, the sum of the complexity associated to the Android methods; offloadable complexity, the percentage of the complexity associated to the offloadable methods.

The offloadable complexity reported into Table 4.13 and 4.15 is the complexity percentage related to the available complexity suitable to be offloaded, without considering the Android-related complexity. It is easy to note that the Android-related complexity can be also more than one third, e.g. in the case of Twitter usecase, and the possible amount of computation is to search in the remaining application complexity.

The cyclomatic complexity percentage is almost proportional to the number of methods previously detected as offloadable, meaning that the method complexity is quite balanced between offloadable and non-offloadable methods. Thus, an offloading decision evaluation, roughly based on the number of offloadable methods, can give valuable hints about the amount of complexity moved towards the remote server.

As shown in the experimental evaluation, the analysis described provides the list of methods suitable to be offloaded, that are managed at runtime by the offloading mechanism which will decide to offload them or not, along with the ability to associate the complexity to each method to identify the heaviest methods to offload among all. In this way, it is possible to integrate the methods complexity functionality to strengthen the overall offloading framework, composed of tasks selection algorithm and offloading execution mechanism, by using the complexity parameter that can extend the calculation to decide to execute locally or offload a method.

Task Selection Performance

To measure the performance of the proposed task selection algorithm, for each app, it is necessary to compare the total number of methods found and the time needed by the algorithm to scan the app and to apply the proposed heuristics. The results are shown in Figure 4.15.

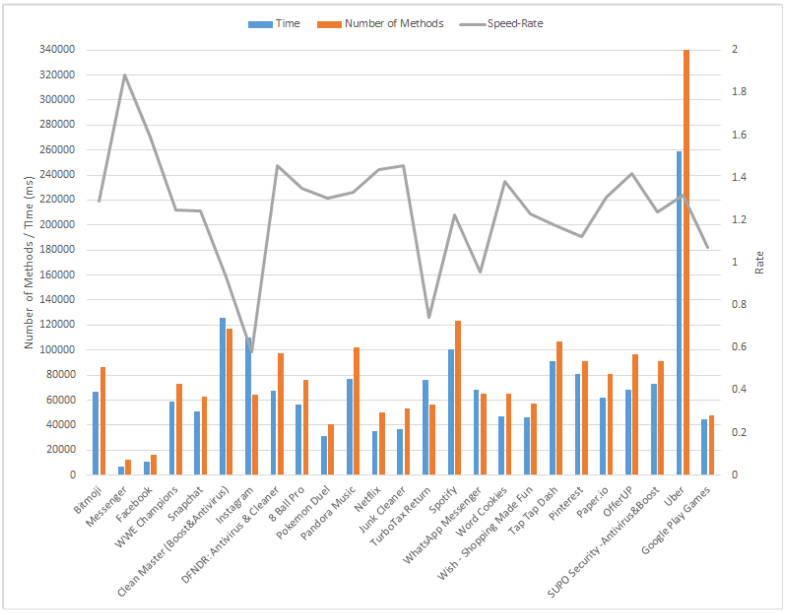


Figure 4.15: Task Selection Performance

The total time needed to scan an app is the sum of the times needed for classes and methods. Due to the limited number of classes usually present into an app if compared with the number of methods, the latency associated with classes has demonstrated to be almost negligible (always less than 5% of total time) and the overall performance is mainly determined by the time needed to scan methods. Note that the algorithm exhibits very good performance and applicability to real-world scenarios, with its ability to scan the vast majority of existing top-25 apps in less than 1 minute. In addition, Figure 10 shows the speed-rate of the applications considered, defined as the number of methods evaluated per ms. The speed-rate ranges from 0.6 to 1.9, with an average rate of 1.3 methods scanned per ms, which underlines the feasibility of the solution proposed application as well as its effectiveness also in large-scale applications.

5 | Scalability, Elasticity, and Federation for the Cloud Computing Middleware

This chapter describes two relevant solutions based on the usage of cloud computing to provide mobile services to mobile devices. It is worth noting that, the proposed solutions cannot be used instead of fog or edge computing solutions but are complementary to them and help to cover a broader range of applications scenarios. In fact, the following solutions can be used in a valuable way in case of powerful mobile devices, that do not have limited resources available and can work in a quite autonomous way, that connect with the mobile services in a more loosely-coupled way. On the contrary, their application is unfeasible, for instance, for IoT devices that send almost all the data sensed from the environment to a remote server and exchange with it a high amount of data. They cannot really deal with strict requirements of IoT, in particular in sensible environments about latency, location-awareness or privacy.

In particular, the two proposed solutions significantly improve the scalability and interoperability requirements that are two of the main challenges when dealing with a great amount of different mobile devices. Scalability is necessary because pervasive environments are usually characterized by a higly variable number of requests to serve that must not affect the overal system performance and, thus, the system should adapt to elaborate different amount of requests and data. Interoperability is also challenging when very different amount of mobile devices, that gathered various data, are present in the environment and the second solution proposal can be very relevant to address it because it proposes to abstract data by decoupling the collected data and the referring model that manages information.

5.1 Related Work

5.1.1 Virtual Machine Management and Migration

The management and orchestration of cloud-based resources and services at different levels and with different purposes has been fairly investigated by researchers in recent years, as well as the VM management within the Openstack platform, and many works in the literature have tried to address it.

OpenStack Heat [153] is a deployment technology based on templates, that manages VMs creations and the entire lifecycle of infrastructure and applications within OpenStack clouds, through building stacks which are sets of virtual resources. Nirmata [154] is designed for microservices, providing seamless service discovery, registration, load-balancing and customizable routing for microservices, where an application is composed of multiple services and provides an innovative cloud service that makes it easy to automate the entire DevOps lifecycle. Hurtle [155] is an orchestration framework that allows to automate the life-cycle management of the service, from deployment of cloud resources all the way to configuration and runtime management. It ensures the creation and management of not only the foundational resources required to operate the target service logic but also the so-called external requirements, i.e., the external service resources, possibly not networking-related, needed to compose the final service. Some proposals are based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) [156], a middle-level language for the specification of the topology and orchestration of services in the form of a service template, that enables interoperability of application descriptions and cloud services infrastructures, the relationships between parts of the service, and the operational behavior of these services, e.g., deploy, patch, shutdown, independently from the service provider. [157] presents how the portable and standardized management of cloud services is enabled through the TOSCA. Cloudify [158] is an open source TOSCA-based cloud orchestration framework that allows to model applications and services and automate their entire life cycle, including deployment on any cloud or data center environment, monitoring all aspects of the deployed application, detecting issues and failure, manually or automatically remediating them and handle ongoing maintenance tasks.

Several migration and replication mechanisms have been proposed to provide high availability inside virtualized environments and overcome migration issues. In fact, VMs migration may have several impacts on network and service performance, as described in [159], related to: resource consumption either at the source or at the destination hosts; network congestion, when massive migrations are triggered; long provisioning time may prevent cloud providers from allocating new hosts resources; service disruption time to ensure a seamless and efficient live migration but, as underlined in [160], inevitable completely due the impossibility to stop a running service, even for a very short time, without impacting the service. To alleviate these impacts various VMs management plans based on live VMs migrations have been widely studied and explored in the literature. Pre-copy is the most common approach used for live migration and has been successfully applied and optimized into the most successfully commercially available hypervisors, e.g. Xen, KVM, VirtualBox, VMware, and so forth [161]. [159, 160] adopts a typical pre-copy strategy as a base, which combines a push phase, during which the VM memory is transferred in subsequent rounds while the VM is still running, and a stop-and-copy phase, during which the VM is stopped and just a residual part of the data is transferred.

Many works study the performance variations in relation to the migration efficiency of the cloud data center, with different purposes. S-CORE [162] is a scalable live VM migration scheme, based on a distributed migration solution with multiple distinct policies, to dynamically reallocate VMs in order to minimize communications cost in a cloud data center. They also underline the importance of measurement-based, network-aware VM migration providers to significantly increase the infrastructures capacity also in highly varying traffic contexts. [163] attempts to enhance data center performance and scalability, minimizing the overall network cost, with the introduction of an algorithm to improve VM placement management. [164] migrates a group of VMs in relation to some network indicators (i.e. cost of migration, available bandwidth), detecting the most overloaded links in the network to alleviate network congestion. [165] propose a hierarchical placement approach to address VMs placement problem for large problem sizes on IaaS cloud. Sandpiper [166] automates resource allocation and migration of virtual servers in a data center to avoid machine overload by relocating VMs with the Xens migration mechanisms. They adopt black-box and gray-box strategies for VMs provisioning in large data centers to automate the monitoring tasks of system resource usage, hotspot detection, allocating resources and initiating any necessary migrations to alleviate congestions. Finally, multiple VMs live migration is an increasingly relevant topic due to applications complexity growth and big size, thus, requires more efforts to improve and optimized it since it has not been extensively studied in the literature [160].

Regarding the state migration, several migration techniques have been proposed in the literature to move the entire databases between two nodes, with the purpose to minimize service interruption and unavailability. Albatross [167] is a technique for live migration in multitenant databases in a shared storage architecture that initially creates a snapshot of the database on the destination host and then uses several iterations copying the state incrementally to minimize the unavailability window.

Zephyr [168] is a technique to efficiently migrate a live database in a shared nothing transactional database architecture. In Zephyr, contrarily to Albatross, the disks are locally attached to every node, hence, to minimize the unavailability, the persistent image is also migrated introducing a synchronized dual mode that allows both the source and destination to simultaneously execute transactions for the tenant: the destination starts serving new transactions while the source completes the active transactions. During migration, requests at the destination force a pull on the data page from the source and any transaction at the source accessing a page that has been migrated to the destination must restart at the destination. Although Zephyr does not require the nodes to be taken off-line at any point, it does require that indexes are frozen during migration.

Slacker [169] is an end-to-end database migration system that optimizes the impact of migration changing the throttling rate that pages are migrated from the source to destination. Slacker uses recovery mechanisms to stream updates from the source to the destination. It minimizes the migration process performance impact on both the migrating and destination tenants by leveraging migration slack, resources that can be used for migration without excessively impacting performance latency. To avoid straining the other tenants at migrating nodes, a PID (Proportional-Integral-Derivative) controller monitors average transaction latency to adjust throttling the network connection used to stream the updates, by the ability to automatically detect and exploit the available migration slack of computing resources in real time according to the dynamics of the executed workloads on both of the source and destination servers. ProRea [170] represents a live database migration approach that combines proactive and reactive measures, in order to reduce page faults and improve buffer pool handling compared to purely reactive approaches. To prepare migration, the source sets up local data structures and migration infrastructure and sends an initial message to the destination to create an empty database and sets up its local migration infrastructure. ProRea proactively migrates hot pages, i.e. pages that have been recently accessed which are in the buffer pool of the RDBMS instance, thus, new transactions start at the destination and pull pages on-demand. Successively, to finish migration, the source additionally pushes pages which have not been transferred during previous phase or as response of a pull request from the destination. Finally, after the destination owns all pages, migration cleans up used resources and completes by a handshake between the source and the destination.

Dolly [171] uses a live migration technique for virtualized database servers exploring virtual machine (VM) cloning techniques to spawn database replicas and address the provisioning shared-nothing replicated databases in the cloud. Dolly creates VM snapshots and clone VMs to replicate database state and start new replicas. It clones the entire virtual machine of an existing replica, comprehensive of the operating environment, the database engine with all its configuration and data. The cloned virtual machine is started on a new host, resulting in a new replica, which then synchronizes state with other replicas prior to processing application requests. Since, usually, creating a new database replica is a time consuming process which increases proportionally with the size of the replicated database, Dolly incorporates a model to estimate the latency to create a new database replica based on the snapshot size of the virtual machine and the database re-synchronization latency and uses this model to trigger the replication process well in advance of its necessity to occur according to the anticipated workload increase [166].

5.1.2 Semantic Web for Data Federation

Federation of semantic data and navigation via SPARQL queries is still at an early stage and requires users to explicitly express the distributed nodes upon which to perform semantic queries and subsequent result aggregations, therefore negating the intrinsic benefits of adopting a semantic approach to distributed data aggregation and reasoning [172, 173]. Other architectural approaches, alternatives to the one described in the following, have been proposed in the past: plugin, endpoint extension.

Plugin. Some Semantic platforms and SPARQL implementations typically allow developers to extend platform features via a plugin module. Data federation may be realized as a dedicated plugin that transparently handles data federation across nodes, and overcomes current SPARQL limitations. This approach poses non-trivial technology issues: plugin implementation strictly depends on the actual semantic platform, therefore allowing to federate only nodes that rely on the same semantic platform; only a subset of currently available, production-grade semantic platforms support a plugin model.

These limitations adversely impact all KPIs, thus making this option viable only for controlled environments where the semantic platform is shared across the federation and supports a plugin model.

Endpoint extension. This solution extends the internal logic of a SPARQL endpoint, from the classic logic to the execution of the query for extracting data to a model for the interception of the query, the query rewriting through the wired application logic, by inserting the statement required to query all the nodes in the federation, and finally performing on the various endpoints, returning the result in ac-

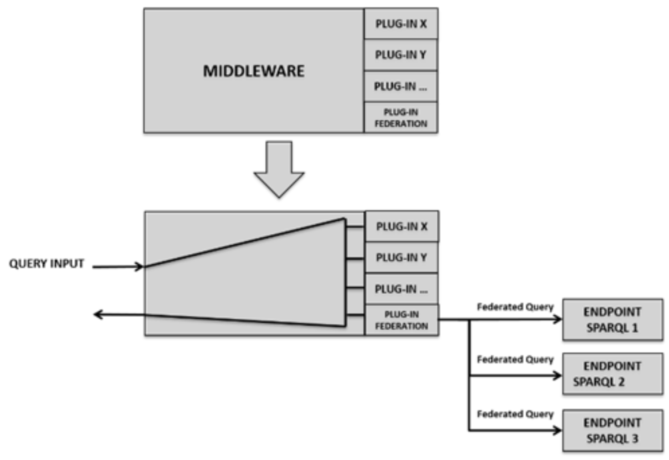


Figure 5.1: Plugin Alternative

cordance with the provisions of the specific SERVICE clause SPARQL 1.1.

This solution extends the internal logic of a SPARQL endpoint, from the classic logic to the execution of the query for extracting data to a model for the interception of the query, the query rewriting through the wired application logic, by inserting the statement required to query all the nodes in the federation, and finally performing on the various endpoints, returning the result in accordance with the provisions of the specific SERVICE clause SPARQL 1.1.

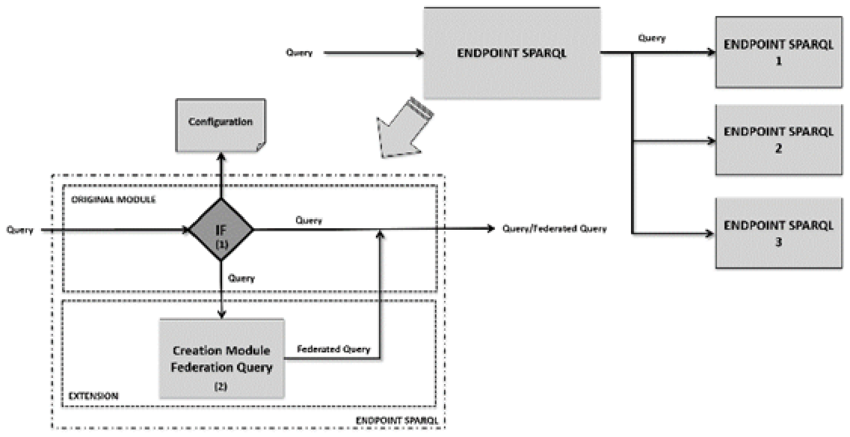


Figure 5.2: Endpoint extension alternative

This solution allows to provide input to any type of SPARQL query, with the only constraint of not being able to use the SERVICE clause, both because this is used as the main construct for the manipulation, either because it would make the handling and the highly complex of final query. This aspect is important as it does not allow the use of all the constructs that conform to the standard SPARQL 1.1, giving the value added to the product that more to the solution. Another aspect to consider is the difficulty in handling the query itself, which could lead to non-trivial query implementation and poor performance. This approach is extremely platform-dependent from both a technical point of view (need to modify existing SPARQL endpoint source code), and a management one (modifying source code from other vendors and distributing it may pose legal and organizational challenges in terms of distribution process and code ownership), therefore achieving a low score on each KPI.

5.2 Elastic Provisioning of Mobile Services in the Cloud

Mobile Cloud Networking (MCN) is a large-size EU project that involves several leading companies, research centers, and universities, that aimed at exploring a very large-scale coverage of a wide range of on-demand telco services in an efficient way [175]. In the following, to facilitate the full understanding of the cloud solution proposal an overview of the general main project architecture developed is presented, with some architecture hints, by focusing specifically on the services and functionalities central to the present proposal. For a broader description and additional technical details about the general MCN architecture, refer to [174].

The overall MCN project goal is to provide innovative and effective solutions for enabling dynamic network function and self-adaptation to mobility with the exploitation and extension of cloud computing techniques in order to ease the deployment and operations of future mobile telco services through self-management, self-maintenance, on premise design and operations control functions. In particular, the project requirements have been typical key requirements of highly dynamic and distributed system (i.e., mobility, scalability, etc.) have been carefully explored through the exploitation of cloud computing cutting-edge technologies, and a smart on-demand deployment and distribution of mobile network functions, providing mobile services independent from physical location.

The main focus of this solution is to significantly improve system performance and scalability, key requirements for highly dynamic cloud solution, by introducing the service state migration of mobile services provided by the MCN system. The service state migration allows to move all the VMs that compose a service, along with all the data collected and associated to the service, in order to reactivate the service on the destination location with the same state conditions of the origin service. In fact, in nowadays cloud services a typical single VM migration is not enough because there are several different components and functionalities that cooperate together to create a unique behavior, i.e. multiple tightly related VMs that compose multi-tiers applications with specific tasks for each VM. The goal to preserve high workload conditions purposes can be achieved by an efficient VMs management and orchestration plan, based on VMs migrations, and a migration of the whole service state. VMs migration must assure, in a completely automatic way, a limited total migration time and, more important, a minimal downtime that is the real cause of service unavailability, with a complete transparency towards end-users that should not observe any changes, like the VM did not change location. On the other side, the state migration consists of transferring all the service history that is composed on the data collected during its execution and is achieved by moving the whole database towards the newly created service.

The main strengths of the proposed solution are: migration schema definition to minimize unavailability; solution conceived and applied in large-size, complex and real-world application; proactive behavior that allows to monitor the system and act before congestions occur; fully open-source platform and tools usage. i) First, a general migration schema has been designed to provide a minimal disruption to end-user service availability, keeping the state of the origin running service on the destination service after the migration, and able to manage applications based on multiple correlated VMs that must maintain reciprocal connectivity after the migration. ii) The solution integrates with a real production system that provides ubiquitous cloud services and is characterized by large-size usage application. MCN is deeply analyzed in order to outline and enhance an adaptive mobile-cloud orchestration mechanism to deliver services and a dynamic management implementation of the services state migration. iii) The present solution works in a proactive way, preemptively migrating service state from the congested host by self-initiated VMs migration, before services congestion can affect all the system, triggering VM migrations when a maximum amount of resource is used. Pre-congestion indicators are used at runtime to guarantee high-performance, low failures and scalability through VMs migration towards a less loaded destination host and automatic reconfiguration of VMs on the target host. In addition, the solution is based on gray-box techniques [161], that allows to operate properly even with a minimal priori knowledge of the system. iv) Finally, this solution is completely based on open-source platforms, e.g. Openstack, Zabbix, and independent from the underlying hardware/host. Openstack allows to deploy VM instances on-the-fly to handle different system tasks when required and the usage of VMs to implement services is a key enabler for dynamic services management because decouples service instances from the hardware and isolates specific functionalities into different VMs, allowing to flexibly deploy any application independently from the target host and able to handle different workload variations.

5.2.1 MCN Background and Architecture

The main MCN objectives are to develop a novel mobile architecture and technologies to create a fully cloud-based system and to extend cloud computing, beyond datacenters to the edge of the network, towards mobile end-users. In fact, cloud networking is explored as a mechanism to support on-demand and elastic provisioning of mobile services, implementing a platform to process and storage data near the end-points in order to enhance performance and deliver services in an elastic and dynamic way. The MCN architecture is very modular and the key concept is to combine different services to create other more complex end-to-end (E2E) services. The MCN Service Management Framework enables and affords the means to compose and orchestrate the MCN operations across multiple domains and service types, creating the E2E composition. MCN supports and enables developers to build upon MCN services so they can compose and orchestrate their own services delivering much needed additional value and new revenue streams. These dependencies are executed upon specifically by the Service Orchestrator (SO) Resolver component. Figure 5.3 illustrates the MCN architecture portion related to the services used. The MCN architecture is based on some key components used by all

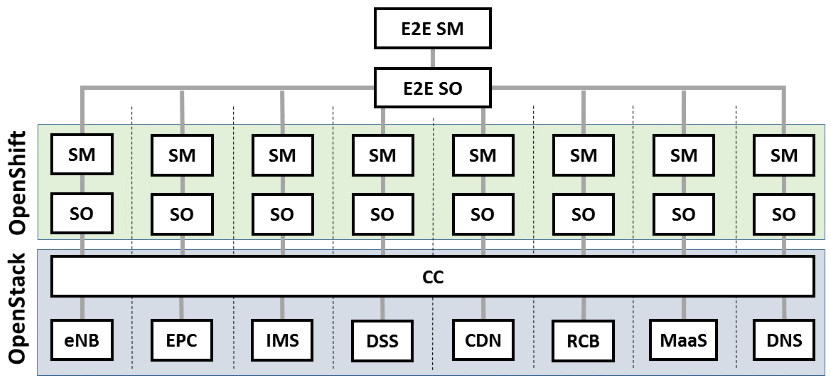


Figure 5.3: MCN Architecture

the services of the architecture. The Service Manager (SM) is the service provider that exposes an external interface to the Enterprise End-user (EEU) and is responsible for managing SOs to request the creation of services instances. It adheres and implements the MCN lifecycle. The SM programmatic interface (northbound interface, NBI) is designed so it can provide either a CLI and/or a UI. Through the NBI, the SM gives the EEU or SO, both classed as tenant, capabilities to create, list, detail, update and delete (EEU) tenant service instance(s). Its Service Catalogue contains a list of the available services offered by the provider. Its Service Repository is the component that provides the functionality to access the Service Catalogue. The Cloud Controller (CC) supports the SO requirements and service life-cycle management, providing the management interfaces used by SM and SO, abstracting from specific technologies that are used in the technical reference implementation. The SO creates, configures, orchestrates and manages every service instance in order to access functionality provided by the specific service. The SO Management (SOM) component has the task of receiving requests from the NBI and overseeing, initially, the deployment and provisioning of the service instance. Once the instantiation of a service is complete, the SOM component can oversee tasks related to runtime of the service instance and also disposal of the service instance. The SO is a self-contained tenant process that runs within a container, managed by the CC. Its primary responsibility is to manage, according to the MCN lifecycle, resources and external services required to deliver the tenants service instance.

The proposed solution is based on two MCN services: Monitoring as a service (MaaS) and Rating, Charging and Billing as a service (RCBaaS). MaaS addresses the design, implementation and test of monitoring mechanisms, from the low-level resources to the high-level services, across the four different domains: radio access network, mobile core network, cloud data center and applications. MaaS is considered as a full-stack monitoring system equipped with the capabilities to provide monitor and metering functionalities in a large scope of telecommunication systems. Service stability of MaaS has been achieved by making use of solid and established Zabbix open-source project [176]. Zabbix is a software toolkit that provides an effective, scalable and reliable monitoring, with a wide range of monitoring performance indicators and metrics, of a distributed infrastructure using Zabbix agents which may be used to collect data locally on behalf of a centralized Zabbix server and report the data to the server. It provides agents for a wide range of operating systems and supports both active and passive checks to monitor data and CRUD operations via JSON-RPC based API interface. MaaS retrieves information in polling mode using the Zabbix APIs and exchanging data with Advanced Message Queuing Protocol (AMQP) protocol based on the publish/subscribe model. MaaS provides an interface to retrieve at runtime the monitoring information from the agents related to the single services to monitor that allows services to dynamically subscribe and retrieve asynchronous data from the IaaS. Each monitored service who wish to use MaaS functionalities implements the interaction interface with the MaaS service and have to integrate the configuration of their Zabbix agents per node to be monitored in their provisioning or deployment. The wrapper objects support methods for retrieval of metrics without being locked-in to a specific realization of the MaaS. Differently to MaaS, the RCBaaS is a monitoring service that collects information for accounting and billing purposes. RCBaaS is employed in MCN as a support service that takes as input the service consumption metrics, processes them, calculates the price to be charged to the user, and generates the invoice for payment. It allows to charge both the end user (EU) and the EEU or service operator itself that operates a service in a cloud, as-a service way. It also takes into account information about anomalous events (e.g. service failures, consumptions exceeding a given threshold, etc.) in order to correctly enforce different types of charging models. RCBaaS interacts with the other MCN services with the Rabbit Message Queue (RabbitMQ) used to collect asynchronous messages generated from different entities. RCBaaS, when instantiated, subscribes to MaaS to receive monitoring data through the RabbitMQ server that receives the information sent by the RabbitMQ client, dynamically instantiated on MaaS to mediate the interaction towards RCBaaS, which is triggered by Zabbix every time MaaS detects the condition inserted in the subscription. The RabbitMQ client receives as input the relevant monitoring values from Zabbix, translates them in the RCB format and delivers the messages to the RCB RabbitMQ.

5.2.2 Service Instance Migration Design

The proposed solution adopts the service instance migration in order to be able to migrate on-the-fly the whole state of the service. The state migration operations are key for high-performance and reliable systems. Various services performance objectives, e.g. scalability, responsiveness, may be significantly improved by optimizing the VMs placements, and thus resource allocations, among hosts avoiding bottlenecks due to congestion links. In terms of reliability, fault tolerance is a major concern to guarantee availability and reliability, in particular with critical services. State migration reliefs also fault-tolerance requirement minimizing failures impact on services executions by moving the services when failures occur or when are likely. A typical service state migration process is composed on multiple steps depicts in Figure 5.4. The physical host is continuously monitored, for its whole life-cycle, by a monitoring system that is able to activate a trigger when the host manifests signs of congestions because it struggles to serve all the incoming requests. When the trigger is activated the service state migration starts and the service orchestrator, though a placement decisioning activity, must find the most suitable place where is possible to move the running service, selecting a destination host. An empty copy of the same service will be deployed and provisioned on the new and less loaded location, binding the new service with the same references earlier attached to the origin service. Successively, the core migration phase is executed, pushing all the state service data towards the destination service in order to replicate exactly the same situation of the origin host. Finally, after all the service state data are moved to the target destination to recreate

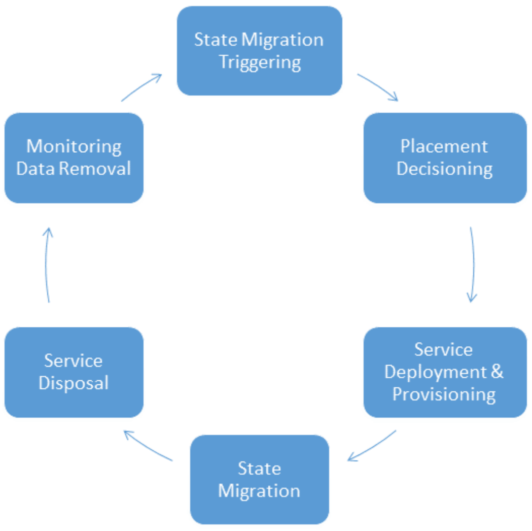


Figure 5.4: High-Level Vision of the Overall Service State Migration Process

the original state, the old service will be unbound and disposed in order to release resources no longer used and to relief the overloaded computation. Before to restart to monitor the newly created service, the monitoring system collected data is reset in order to refresh it deleting old measurements related to the previous service location and no more valid.

The overall goal of the proposed system is to realize a coordinated set of mechanisms that allow moving all internal state of a service instance to another instance created from scratch at runtime, and with minimal service instance interruption. For the current proposal, the design and implementation, apart for placement decision, include all the orchestration and migration activities outlined previously, commonly used during service state migration, using a state migration phase based on a pre-copy [177] approach, that pushes most of the data to destination host before stopping VM at the origin host and move it to the destination, rather than post-copy [161], that pulls most of the data from source host after resuming VM at the destination host.

Initially, the service is continually monitored using MaaS functionality via integration with Zabbix in the testbed environment, as the main potential trigger for migrations. Zabbix is an open-source software toolkit that provides an effective, scalable and reliable monitoring, with a wide range of monitoring performance indicators and metrics, of a distributed infrastructure using Zabbix agents which may be used to collect data locally on behalf of a centralized Zabbix server and report the data to the server. Based on the collected data, Zabbix allows to activate triggers that sends alerts or automate unsupervised actions to automatically resolve issues. In this solution, in order to manage the values gathered by monitoring system performing further analysis when overloaded data are retrieved, a component between

Zabbix and the trigger is added, which implements black-box and gray-box techniques in order to be completely agnostic from the application and thus be able to make decisions by observing each VM from outside and without any knowledge of the service provided. The Grey-based model has been selected as the basis of the implementation in order to make it acting as a predictive model able to analyse the usage evolution and smooth peaks and fluctuations, by amplifying at the same time the monitored resource metric growth [178]. In fact, Grey Model is recognized in the related literature as a simple baseline solution for a multi-parametric systems control that acts as a low-pass filter to detect usage resources in order to trigger events related to resource scarcity or specific service placement problems: it tends to offer high hit rate and good overall performance, also when the model information is partial or incomplete [179]. In addition, Grey model is particularly suitable because it allows the adoption in contexts with relatively limited data collected with only few discrete data needed to analyze an unknown system, allowing to forecast next values also when the decision-makers only own a limited set of historical data. In this solution, this behavior allows to begin the hosts observation just after the VMs are created, as soon as a couple of data are retrieved. The one-step ahead prediction derivation, via the Grey Model approach, for the utilization percentage of each monitored resource (Ures) is:

$$Ures_{(t+1)} = f_{GM}(Ures_t, Ures_{(t-1)}, , Ures_{(t-k)}) \tag{5.1}$$

where $f_{GM()}$ is the Grey Model function and k is the number of historical value considered for the prediction, thus, influencing also the model accuracy.

Hence, all the one-step ahead predictions 5.1 are combined into a polynomial linear formulation with all the P parameters considered both the host resources, comprehensive of CPU, memory, network and disk operations, and application requirements present in the system SLA criteria, comprehensive of round trip time (RTT) and throughput in order to define the overall resource usage (Uhost) into the equations:

$$\begin{aligned} Uhost_{(t+1)} = x_1 * Ucpu_{(t+1)} + x_2 * Umem_{(t+1)} + \\ x_3 * Unet_{(t+1)} + x_4 * Udisk_{(t+1)} + \\ x_5 * Urtt_{(t+1)} + x_6 * Uthr_{(t+1)} \end{aligned} \tag{5.2}$$

where: $Uhost_{(t+1)}$ is the overall resource usage on the host one-step ahead; $Ucpu_{(t+1)}$ is the CPU usage one-step ahead; $Umem_{(t+1)}$ is the memory usage one-step ahead; $Unet_{(t+1)}$ is the network bandwidth usage one-step ahead; $Udisk_{t+1)}$ is the read/write disk operations usage one-step ahead; $Urtt_{(t+1)}$ is the deviation from the minimal RTT value one-step ahead; $Uthr_{(t+1)}$ is the deviation from the maximum throughput value one-step ahead; x_i is the weight for each resource and it is used to stress or weaken the relevance of specific resources to be able to adjust the monitoring system in relation to the importance of the single resource to monitor:

$$\sum_{i=1}^{P} x_i = 1 \tag{5.3}$$

Finally, the threshold T is defined to be compared with the $Uhost_{(t+1)}$ to detect if the host is overloaded and, hence, in order to activate the trigger and start the migration procedure. In particular, in relation to the $Uhost_{(t+1)}$ it is possible to detect either to move just a single VM instance to the target hosts or if it is needed to migrate more VM instances due to high forthcoming congestion:

$$(Uhost_{(t+1)} - T) \propto N_{VM} \tag{5.4}$$

where N_{VM} is the number of VM instances to migrate.

Note that the decision of the target place where to migrate the targeted service instance is out of the scope of the described implementation of the service instance migration because it might depend on the internal management goal of the MCN CC. Given a target destination place, the state migration procedure starts. Assuming service state migration time to be split into epochs and considering every epoch to a particular service state, it is possible to define the following epochs:

- E1: the interval to re-create the same environment onto the new selected host.
- E2: the interval to perform the first data migration phase.
- E3: the interval the perform the residual second data migration phase.
- E4: the interval to release the unused resources.

According with the epochs definition, the algorithm used for the service state migration realization is:

- Step 1 (E1): create the new state skeleton, composed of M_{VM} (independent from N_{VM}) VMs instance, by the SO to prepare the target place to receive the state to be moved.
  - 1.1 VM instances creation and startup.
  - 1.2 VMs configuration to recreate the same environment.

- Step 2 (E2): freeze the database, with the associated service state, and migrate all the service state towards the newly created VMs, while the old VMs continues to collect data and be able to serve requests. Since the migration is a modification of the pre-copy based approach, this is the initial push phase where all the data associated to the database state, for the captured time instant coincident with the start of the E2 epoch, on the origin service are moved on the pre-created destination service.
    - 2.1 push all the data collected in VMs (at the time E1 starts).
    - 2.2 service reference changing to retrieve the service from the new VMs.
    - 2.3 delete the data moved to be aware of the data collected during Step 2 and still not moved.
- Step 3 (E3): is the stop-and-copy phase and pushes the residual data collected during Step 2 to the new VMs in order to update the service state with the latest information.
- Step 4 (E4): dispose the old overloaded VMs to release the associated resources and lighten the workload on the overloaded host.

This general-purpose procedure has been designed in order to allow: to minimize, towards a very negligible, data losses and service unavailability time, independently from the amount of data and time of the migration; to integrate different algorithms and techniques to manage the specific temporal epoch in a more adaptive and distributed way among different service instances, i.e., introducing priority settings with high-priority sessions, or some fault-resilience requirements. In the specific use case, in order to do not further overcomplicate the already complex MCN project and, for seek of simplicity, the requests are managed only on one instance (M_{VM}=N_{VM}=1) without a distributed management, but providing the design opportunity to easily extend the current implementation in future applications. When the push phase is terminated, the service is active on the new VMs and the newly created instance is the target service instance that serves all the incoming requests from clients: the service is restored on the new host and the origin service instance does not receive more incoming data. As soon as the origin service instance is no longer used, the stop-and-copy phase is performed, pushing all the residual data inserted on the old instance previously and belonging to the temporal epoch, towards the destination service instance. In this way, it is possible to timely run out all the old epochs data in order to converge and realign the service state to complete the state migration transaction. After the whole state has been moved, the implemented solution deletes its stack with the old service instance to release migration-related internal resources and relief computation. Finally, before to restore the monitoring component, the buffered monitoring values read from Zabbix are reset to enable new future triggering actions without old spurious data. Note that the service to migrate in MCN project is RCBaaS and during the whole service instance migration process, the service continues to be running and available without performance degradation and with a complete transparency for both system operations and end-users.

By delving into a more detailed description, Figure 5.5 illustrates the service state migration process steps. Apart from the starting

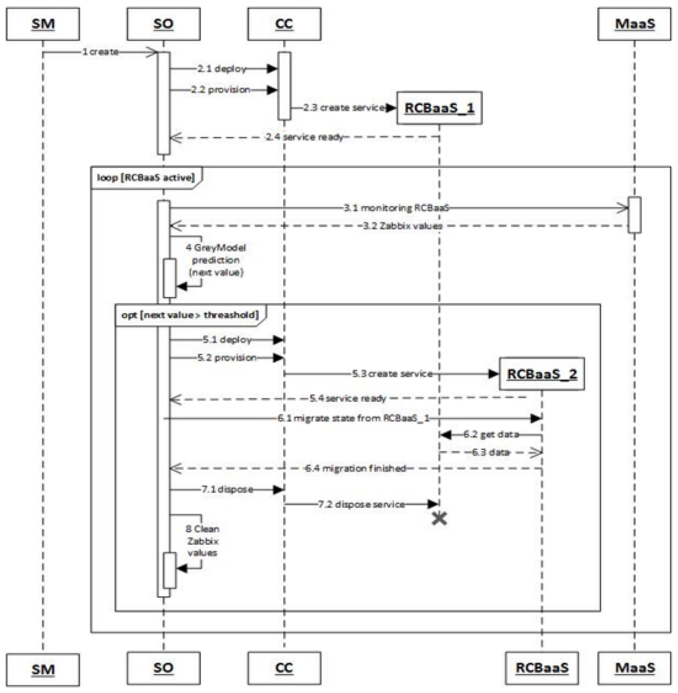


Figure 5.5: State Migration Process

SM/SO typical activities to deploy and provision a new RCBaaS service (1) (2), the first step regarding the state migration is the triggering of the whole migration. The real resource usage values provided by Zabbix (3.1) (3.2) are stored into a sliding window array with a fixed buffer length that can be easily configured programmatically. Of course, the buffering of a time series of monitoring data samples enables algorithms for resource usage analysis and prediction. In this case, as an external triggering decision algorithm, a lightweight first-order Grey Model filtering module (4) has been provided. When the migration trigger has been activated, the steps already explained in Figure 2 start: prepare the target place to receive data (5); data migration towards the target VMs (6); stack disposals (7); reset the buffered monitoring (8). Finally, the system returns to store monitoring data from the hosts and restart the loop.

5.2.3 Implementation Insights

The implementation of the described solution is composed of three macro-steps. Starting from the RCBaaS division into smaller and more specific VMs instances in order to monitor more efficiently and accurately the effective resource consumption, the definition of the monitoring to keep track of the effective resource consumption and finally the service state migration to move data on the fly during service usage are detailed.

Preliminary Work on Service Separation

RCBaaS is composed by two main components that interact very frequently: Cyclops [180] and InfluxDB [181]. Cyclops is the core component of RCBaaS, that contains all the logic inside the service for accounting and billing purposes. Cyclops is divided into three micro-services: user data records (udr) collects the usage data from a source, e.g. OpenStack, CloudStack, SaaS, PaaS, etc. and stores it in the database; rating and charging (rc) uses the usage data records generated by the udr to calculate, in relation to the cloud resource rate, the charge data records; billing interfaces with the rc to generate an invoice. InfluxDB is an open-source time-series database, particularly suitable to keep trace of large amount of data from sensor data, applications metric and real-time analytics, thus it is the backend of RCBaaS as the service monitoring metrics repository to keep the all history of the service measurements.

To apply the service state migration procedure and mechanism described above in a practical valuable case, the first focus is about the RCBaaS monitoring service, which is treated as a monolithic VM for the sake of maximum separation. As a first step to enable the state transfer migration process, it is necessary to split it into a couple of disjoint and only dynamically bound VMs: RCBaaS-VM that perform the core operation of the monolithic service and contains only Cyclops component; InfluxDB-VM that contains the backend component where data are stored. The two VMs have been uncoupled in order to be able to act on the state skeleton (that is the original component that actually stores the service state), in this case the InfluxDB instance, without requiring any change on the RCBaaS, thus actually behaving as a stateless component in this uncoupled version.

The service have been separated through the Openstack Heat, the main orchestration program inside Openstack, that implements an orchestration engine that allows to launch multiple composite cloud applications based on a text file, i.e. the Heat template file. The Heat template is opportunely configured to create two different VMs for Cyclops and InfluxDB when lunched and to invocate two script files located on the Cyclops-VM after the creation to automatically configure the IP address of the InfluxDB-VM to send data to store. The following code outlines the script to configure the IP address into Cyclops-VM, editing three configuration files, one for each Cyclops micro-service.

```
ip=$1
echo "-> Cyclops-udr configuration file"
python string_substitution.py ~/configuration.txt InfluxDBURL= http://$ip:8086
echo "-> Cyclops-rc configuration file"
python string_substitution.py ~/configuration.txt InfluxDBURL= http://$ip:8086
echo "-> Cyclops-billing configuration file"
python string_substitution_js.py ~/config.js url: '
"http://'$myip':8086/db/udr_service",' ' "http://'$ip':8086/db/grafana",'0
```

Monitoring System

The monitoring system is mainly based on two main components: MaaS that uses Zabbix to retrieve resource information from the physical hosts; the Grey Model that predicts the next values 1-step ahead. MaaS runs a Zabbix server that communicates with distributed monitoring agents instantiated during the services provisioning and aggregates the resource information retrieved from them. This monitoring agent is designed to collect networking statistics, processing and normalizing the raw monitoring data retrieved and exposes them communicating with the Zabbix server provided by MaaS. Every deployed service, that needs to integrate MaaS in their service for resource monitoring purpose, requires the installation and configuration of a Zabbix agent, as shown in the following code, through the heat template file when the VM is created, that actively monitor resources by the interaction with the Zabbix server on MaaS.

```
apt-get install -y zabbix-agent
sed -i -e 's/ServerActive=127.0.0.1/ServerActive=160.85.4.28:10051/g'
-e 's/Server=127.0.0.1/Server= 160.85.4.28/g'
-e 's/Hostname=Zabbix server/#Hostname=/g' /etc/zabbix/- zabbix_agentd.conf
service zabbix-agent restart
```

Successively, the SO implementation contains the list of which monitoring information to retrieve, apply the Grey Model and define the parameters to activate the trigger that starts the full state migration. The next script shows a snippet of the SO implementation for the monitoring part. The cpu load is checked, as the value to monitor to evaluate if the host is overloaded. The cpu load considers the queue length of processes the are waiting to be processed and it is a common and widespread parameter to detect accurately the host workload. The cpu load threshold is 10 for the InfluxDB-VM and, given that the VM has 2VCPU, it means the trigger is activated when at least 5 processes per single core are waiting to be processed. The cpu load value considered is returned by the Grey Model considering the last five values read from MaaS stored into a sliding window array. At startup, there are three minimum reading before to invoke the Grey Model, in order to avoid false positives and thus to avoid a trigger activation caused by few anomalous readings. Finally, the monitoring values from MaaS are retrieving every 1 minute, that is the sensitivity of Zabbix and the minimal time interval to retrieve data, in order to have a relatively fine-grained periodicity to balance overhead and responsiveness.

```
class MyList(list):
    def append(self, item):
        list.append(self, item)
        if len(self) > myparameters.WINDOW_SIZE: self[:1]=[]
class SOD(service_orchestrator.Decision, threading.Thread):
    def getGreyModelValues(gateway, composedList):
        values = []
        for list_py in composedList:
            list_java = ListConverter().convert(list_py, gateway._gateway_client)
            nextValue = gateway.entry_point.nextValue(list_java)
            values.append(float("{0:.4f}".format(nextValue)))
            return values
    def monitoring(self):
        self.monitor = RCBaaSMonitor(myparameters.MAAS_DEFAULT_IP)
        self.hosts_cpu_load = []
        self.hosts_cpu_load.append(MyList())
        metrics = self.monitor.get(myparameters.ZABBIX_INFLUXDB)
        self.hosts_cpu_load[0].append(metrics[0])
        if len(self.hosts_cpu_load[0]) >= myparameters.ZABBIX_MIN_READING:
            cpu_load_GM = getGreyModelValues(self.gateway, self.hosts_cpu_load)
        if cpu_load_GM > myparameters.TRIGGER_VALUE:
            print "Trigger activated. I'm going to move the VM state."
            ...
        time.sleep(myparameters.ZABBIX_UPDATE_TIME)
```

Service Instance Migration Implementation

The service instance migration step, it consists of two phases in this RCBaaS case: i) a complete InfluxDB dump from the old to the new instance; ii) storage and migration of new data inserted on the old database instance while the migration is occurring and after the dump operation. About the database dump, it is the core operation of the state migration and all the data of the old InfluxDB instance are moved to the new instance. To reduce the duration of this phase, all the data of the old InfluxDB are compressed into an archive that is moved to the new InfluxDB VM, where they are extracted and located into the target InfluxDB data folders.

As shown later, the dump operation could have a non-negligible duration, also tens of seconds, in relation to the amount of database instances and records to transfer, due to both external operations (e.g., data compression, movement, extraction, and database restart) and internal database operation to synchronize to the new status. In order to minimize the database unavailability time and, thus, to preserve overall service continuity, the second phase is performed as follows. As soon as the old data have been copied into the archive, all dumped data in the old databases are dropped to be sure that every data successively inserted has not been transferred during migration and, as a consequence, to relieve further the old database performance. When the database dump has completed and the target InfluxDB has been configured and made available, all the new data at the old instance are selected and moved to the target InfluxDB, merging with the data already migrated during the dump. By delving into some finer implementation details, this mechanism has required to save these during-migration entries a JSON file, and then to convert them through a Python script into a LineProtocol format file used by InfluxDB to insert data on-the-fly; this copy of the new data to the target InfluxDB instance completes the data migration step

Figure 5.6 graphically summarizes all the operations performed during the data migration process, by distinguishing the actions executed on the old and those run on the new InfluxDB VM instance. In particular, the first three blocks from the left refers to the database dump operation (phase i) and the last two blocks to the storage of new data inserted during the migration process (phase ii).

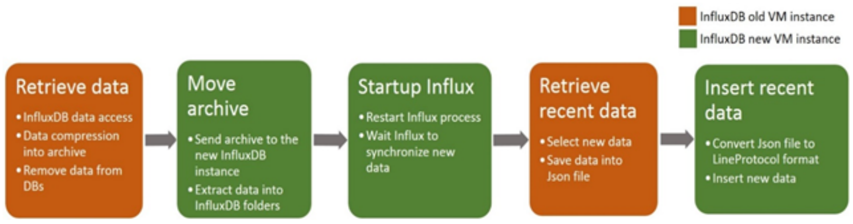


Figure 5.6: Essential Steps of the Proposed Data Migration Process

The following code shows the code used inside the SO component to invocate the migration script and to move data between the two instances. A third-party Python library is used that, with a SSH connection, allows to send a command to a remote host. In this way, it is possible to access the new instance as a typical SSH communication and execute, from the SO implementation, the script already prepared on the newly created InfluxDB-VM instance passing as a parameter the IP address of the old instance to migrate. The code is within a loop block because the creation of a new VM may take a few seconds and it tries to connect with the VM created until it is not fully started.

```
while True:
    try:
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        key = BUNDLE_DIR + myparameters.MIGRATION_KEY
        ssh.connect(self.so_e.influxdb_ip, username=myparameters.MIGRATION_USERNAME, key_filename=key)
        command1='cd '+myparameters.MIGRATION_SCRIPT_FOLDER+';
        command2=' bash '+myparameters.MIGRATION_SCRIPT_NAME+' '+self.so_e.influxdb_ip_old
        command=command1+command2
        stdin, stdout, stderr = ssh.exec_command(command)
        ssh.close()
        break
    except paramiko.ssh_exception.NoValidConnectionsError:
        print "VM not ready"
        time.sleep(2)
```

Experimental Evaluation

To better understand and give a hint of the complete MCN test environment and the overall complexity of MCN deployment, the E2E MCN deployment scenario is introduced without the description of all the services involved for seek of briefness and pertinence with this proposal. The integrated E2E MCN deployment scenario, shown in Figure 5.7, creates a very heterogeneous and broad-based cloud platform composed of a high number of services and components, that have been evaluated following several different tests activities that takes into consideration the wide scope and objectives of MCN project. Only to mention the most relevant prove of concepts adopted on the overall project, are pointed out: Digital Signage System (DSS) to evaluate the over-the-top applications for playing content through digital signature services; IP Multimedia System (IMS) to evaluate the support of video and voice for mobile users; Follow-Me-Cloud (FMC) to better evaluate and demonstrate the capabilities of mobility prediction and dynamic content placement, particularly important to show how information-centric networking technologies can be fostered by prediction and how the FMC concept is supported. In this

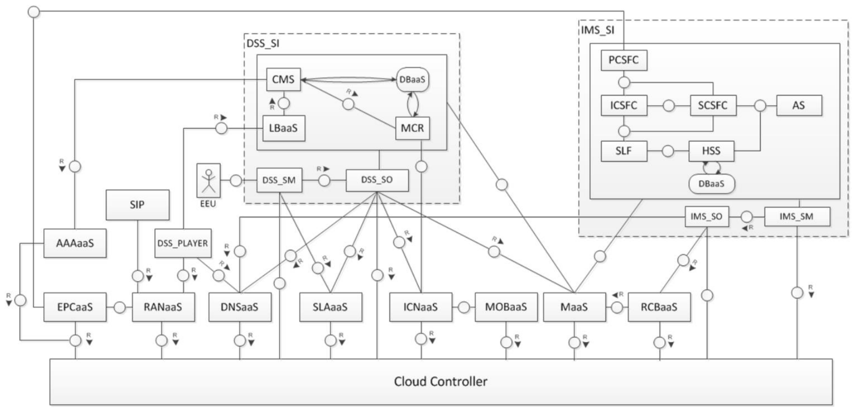


Figure 5.7: MCN E2E Deployment Scenario

very wide and complex scenario, the relevance of each component greatly varies in relation to the specific use case considered and in the following only the RCBaaS service will be detailed. RCBaaS is the component where the state migration management solution has been added, thus, only the experimental results, that strictly highlight the isolated behavior of the state migration functionality among all the functionalities present in MCN, is shown.

Several tests have been performed on all the steps and phases discussed in the previous sections, deploying stacks on Bart OpenStack platform and using RegionOne as the default region. Openstack Bart is a testbed provided by MCN consortium that runs the basic OpenStack services, based on Kilo version. In particular, for the sake of performance evaluation, the RCBaaS utilization has been stressed with a wide range of different workloads in order to observe the performance of the newly introduced function and the perceived limited unavailability time that are able to provide notwithstanding dynamic state migration and synchronization. All the performance tests reported in the following refer to the average values measured across multiple runs, anyway observing an overall low variance (<5%).

Table 5.1 shows the performance related to: service initialization, Zabbix monitoring, the Grey Model usage, the new target stack creation, and the RCBaaS-VM creation. Note that only the monitoring performance, that in this case are negligible, may potentially cause performance issues because they are repeated continuously during the service life-cycle. The other operations reported are only performed at startup, thus they do not introduce any latency during system operations at runtime.

Table 5.1: VM Operations

Operation	Time (s)
MaaS monitoring setup (Zabbix)	9.5
Setup GreyModel Java	3.5
Read Zabbix values	0.5
Calculate Grey Model value	<0.1
Create a new stack	45
Cyclops-UDR deploy	25-26
Cyclops-RC deploy	15
Cyclops-Billing deploy	30-32
Total time to setup the VM	90-100

Figure 5.8 shows the performance of the data migration for different amounts of data to migrate. The overall performance varies slightly from test to test mainly in relation to the network conditions and the load on the physical host where the VM is running, thus, the average values measured on multiple runs are reported. The overall latency is divided into several times that allow to distinguish the duration of the different phases; in particular, the main times measured and defined are as follows:

- Tvmconn: time the SO takes to connect to the InfluxDB-VM, or in other words, the latency time between when the trigger becomes active and the data migration starts.
- Tcompress_move: time to compress data into a tar.gz archive and move to the new VM instance.
- Tdelete: time to delete all the data from the old InfluxDB-VM instance, directly proportional to the number of database to delete.

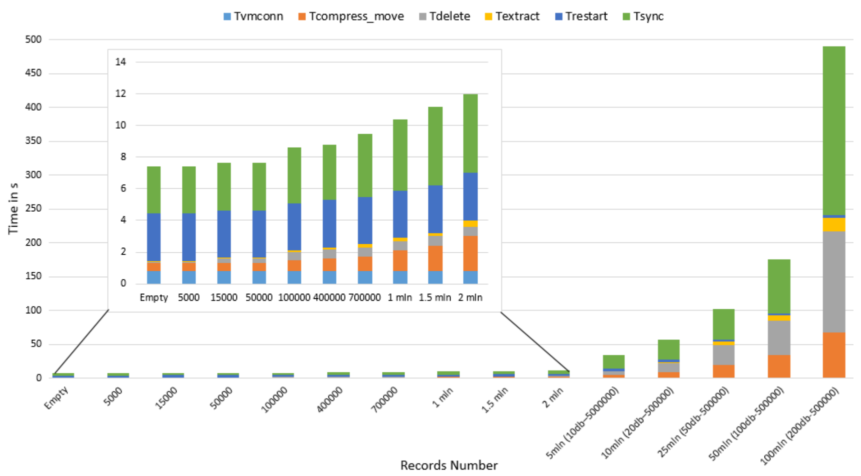


Figure 5.8: Performance Evaluation of RCBaaS State Migration

- Textract: time to extract the archive into the InfluxDB folders of the new VM instance.
- Trestart: time to restart the InfluxDB service in order to get the update about the new data.
- Tsync: time used by Influx process for internal synchronization after the dump.

Other time latencies are related to the storage and insert of the new data during the migration that is the time necessary to: get all measurements, retrieve data inserted into a Json file, convert the Json file into the LineProtocol format and insert the data into the databases. These latencies are not reported in the chart in Figure 5.8 because the amount of data insert during the migration is assumed limited and, thus, the associated time is negligible (in the order of 0.1-0.2s to move a dozen of records).

It is important to stress that during the overall state migration procedure the database unavailability, considering the latest assumption that ignores the new data time retrieval, is limited to the process related to the measurements deletions (Tdelete), proportional to the number of databases presents but always very low and, for typical execution and average migration, below 1s, guarantying relatively negligible unavailability, and thus proving the effectiveness of the proposed state migration function and its wide applicability to stateful services service state migration.

Summing up, depending on the dimension of the state to be migrated, the overall service migration process time can go from 112 seconds for up to two millions of records (namely, 100 seconds to setup the target VM and 12 seconds for data migration) to 590 seconds (for 100 millions of records). In any case, this solution is able to achieve a fully scalable behavior with good overall performance, mainly limited by the InfluxDB internal operations (Tsync), that are the real bottleneck of the solution even if they do not affect the unavailability time but only the duration time of the migration.

Simulation Results

Simulations have been performed, by using the CloudSim [182] framework, in order to evaluate the proposed solution and to show the technical feasibility of the solution under different and realistic load conditions for next-generation and cloudified 5G services. CloudSim is an extensible and widely adopted simulation toolkit that enables modelling and simulation of cloud computing environments, supporting modelling and creation of infrastructures and application environments for single and distributed multiple clouds. In particular, the RCBaaS service infrastructure has been mapped into the simulator by creating:

- two data centers, used to migrate the VMs
- one host per data center, with 2048MB of ram, and 250GB of storage each
- two VMs for each host, with 512MB of ram, 100GB of storage and 1 cpu each
- one process per VM, which represent Cyclops and InfluxDB components

The proposed hybrid solution has been compared with two baseline more traditional migration approaches: reactive-only and proactive-only. The reactive approach moves all data at once when the host is already overloaded, thus, it is characterized by minimal migration time because it does not include data reconciliation, but also may cause significant data loss in case of high amount of data received during the migration. The proactive approach, instead, moves the data in advance before the service migration takes place and then reconciliates the new inserted data after the service migration, thus providing minimal data loss but higher migration time if data variability is high. Figure 5.9 shows the results related to the time required to complete the migration with the proactive approach (light colors) and the data loss during the migration with the reactive approach (dark colors). The migration time of reactive approach and the data loss in proactive approach are not showed in Figure 5.9 because they are essentially constant to the data variability increase and equal to the best cases of the other baseline approach.

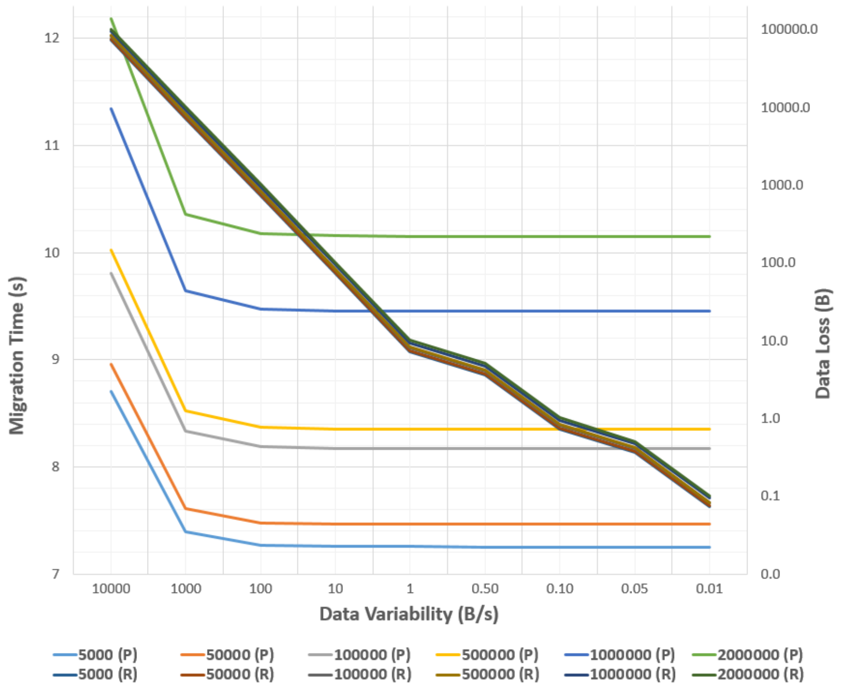


Figure 5.9: Migration Time and Data Loss

Let us note that the results showed in Figure 5.8 have been obtained by considering an average data variability, i.e. 1B/s. The hybrid solution proposed works in a more proactive behavior to approximate the migration time of the reactive approach, keeping at the same time a negligible data loss. In fact, as shown in Figure 5.9: until 1kB/s, apart from time-constraint scenarios, the proactive approach is able to grant low migration time and data loss. Above 1kB/s data variation rate the migration time variation between the two approaches becomes relevant and a hybrid solution should consider which approach to adopt in relation to the application scenario.

Results in Figure 5.9 highlight the importance of a system to adapt in a dynamic way to data variability, as the strategy adopted in this solution that consider the delta number of records inserted into the database between two consecutive time intervals. By focusing on the adjustment of the reactive or proactive behavior at runtime, it is possible to optimize the tradeoff between migration time and data loss in relation to the purpose of the application.

Figure 5.10 shows the simulation results about the data center workload, which analyze how it changes in relation to the number of users and the amount of data stored into the database, and evaluate under which circumstances the data center is overloaded and the migration is likely to happen. By considering the solution design and the multiple test performed, the data center workload is defined as highly related to the percentage of database storage used on the overall host storage and slightly related to the number of users that use the service.

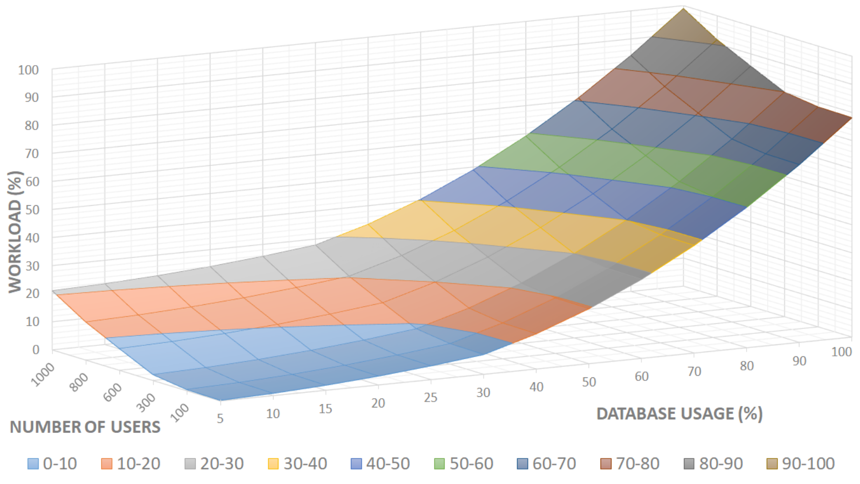


Figure 5.10: Performance Evaluation of RCBaaS State Migration

5.3 Federation Model to Support Semantic Queries

Information systems have long become valuable assets for companies and, along with Data Governance, are key disciplines that drive methodologies and techniques for managing crucial IT assets. IT Governance has become one of the most relevant decision-making processes, and supports the dynamic adaptation to ever-changing business/market needs and goal achievement. In addition, Data federation, the ability to seamlessly and transparently integrate data stored at different places, plays a crucial role in Data Governance and is usually seen as a specialization of Data integration systems, where data storage abandons the idea of single centralized physical endpoints, and leverage fully integrated, fully distributed models. Data Federation is crucial in letting organizations easily and effectively shaping and supporting information flow across organization branches.

The governance of large distributed organizations poses non trivial challenges in discovering, aggregating, and managing data in heterogeneous forms and from different and distributed sources, both within and outside the organization boundaries. That produces governance models where data integration and organization knowledge is typically limited and with no flexibility. A typical example would be the case of the integration of enterprise resource planning (ERP) systems data with both traditional customer relationship management (CRM)/marketing analysis and with novel social media analysis systems to proactively influence enterprise production and operations, e.g., supply chain and stock management tuning as well as workforce optimization. Semantic web [183] principles, methodologies, and techniques have long proven to be crucial in modeling and managing complex relationships between entities/data and to support interoperability, reasoning and inference/automation in coarse-grained, semi-structured, and heterogeneous data environments.

A key element of the Semantic Web is W3C SPARQL [184, 185] protocol, which is a query language for Resource Description Framework (RDF), and has long set itself as the de facto standard to perform semantic queries on content exposed on the Web in order to extract and manipulate information from distributed data sources on the web. RDF describes the concepts and relationships about them through the introduction of triples (subject-predicate-object); triples that have some elements in common become parts of a knowledge graph. SPARQL helps navigating such knowledge graphs and searching for sub-graphs corresponding to the user's request. Semantic Web and SPARQL query language form a promising platform to support Data Governance and federation needs, and allow to build a connected network of information [186].

This solution proposes a novel, semantic-based approach to overcome organization governance complexity and to mitigate/hide heterogeneity and distribution of data sources: this work describes a reference architecture model that relies on a federation of SPARQL endpoints, as well as implementation guidelines and real use-case scenarios to prove the viability of the current proposal.

This approach goes further than traditional governance models (where data harvesting and integration are expensive, difficult, and often limited to specific domains/areas), and fosters a much broader-scoped, horizontal, and continuous integration of data: the federated approach promotes a higher level of data normalization and integration which proposes a more proactive governance model where data analysis may be used not only to lead retrospective analysis, but also to proactively support upcoming strategic decisions.

5.3.1 The SPARQL Federation Model

The present work aims at defining a viable and effective model and implementation to adopt Semantic Web methodologies and the SPARQL implementation to overcome typical issues such as heterogeneity and distribution of data sources. Key principles in defining this solution are:

- Openness. Data federation and integration in large organizations typically means integrating data sources from heterogeneous (both custom and third party) systems; avoiding vendor lock-in and preserving openness and portability is key in defining a sustainable, long-term data federation strategy for any organization.

- Lightweightness. The proposed solution should pose limited to no overhead on running systems, so as to minimize the impact of Data Federation on large organizations with complex, highly distributed data sources.
- Autonomy and ease of use. The proposed solution should be able to cope with uncertainty, and to autonomously discover relationships among federated data even in case or partial a priori data model and network knowledge.

Semantic Web standards and the SPARQL query language have long proven to be key in enabling open, autonomous, machine-driven content matching and reasoning knowledge management infrastructures. However, SPARQL support for data federation via the SERVICE construct - (e.g. distributed data source integration and query) is still at an early stage and proves poorly suitable for large, real-world scenarios. Current SPARQL data federation limitations mainly relate to the fact that designing federated queries requires complete, a priori knowledge of data models and actual data across all data sources involved; this clearly becomes a relevant limitation in large complex scenarios where data sources are heterogeneous and can change frequently and at different paces from each other.

Architecture

The plugin and endpoints extension architectures described previously, have been analyzed and evaluated to realize a fully scalable and extensible SPARQL endpoint federation. A web service federation architecture has been introduced and checked against some Key Performance Indicators (KPIs) to compare their properties and ease the choice with the existent ones.

The Federation Web Service solution is based on creating a Web Service which realizes all the functions related to federation. The Web Service approach makes this solution portable to any semantic platform implementation, thus fostering openness and interoperability. Furthermore, to facilitate the management of large semantic data networks and the definition and navigation of network topology, a specific Network Federation Ontology is developed. The Federation Web Service relies on such an ontology to transparently determine which nodes and endpoints should be involved, thus facilitating the definition of federated semantic queries. The logical flow of the Federation Web Service is as follows:

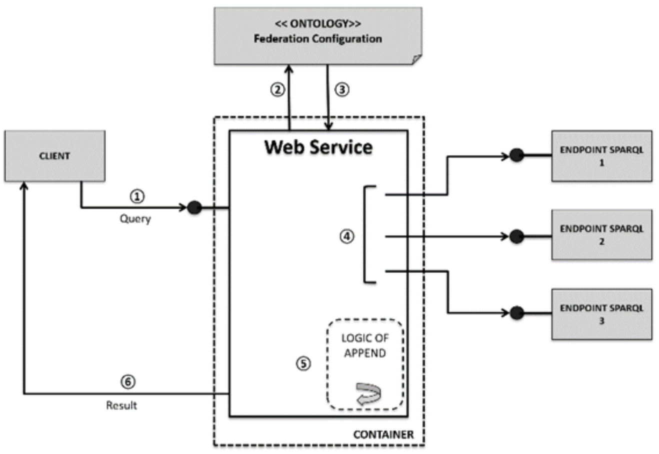


Figure 5.11: Federation Web Service Alternative

- Federation resolution. Thanks to the Federation Ontology, the Federation Web Service determines the actual endpoints involved in the federation.
- Query execution. The Federation Web Service performs queries on all individual nodes involved in the federated query.
- Result aggregation. The Federation Web Service aggregates the results obtained from single nodes and returns them to the caller. The aggregation techniques can be managed through the typical constructs of the SPARQL language, such as the UNION clause.

KPI evaluation and choice

Several qualitative Key Performance Indicators (KPIs) have been used to help to organize and compare benefits and shortcomings of these alternatives, and to ultimately facilitate the choice of the proposed solution, as shown in Table 5.3.

Table 5.3: Alternatives Comparison

KPI	**Plugin Development**	**Endpoint Extension**	**Federation WS**
Ease of Development	LOW - Platform-dependent	LOW - Platorm-dependent	HIGH
Integration	LOW - Platform-dependent	LOW - Requires new endpoint version	HIGH - No vendor lock-in
Maintenance	LOW - Platform-dependent	LOW - Platform-dependent	HIGH
Evolution	LOW - Platform-dependent	LOW - Platform-dependent	HIGH - No vendor lock-in

Where: ease of development refers to the limited development efforts required from the SPARQL federation, no matter the specific Semantic framework implementation; integration refers to the ability of the SPARQL federation solution to easily integrate with other framework features, with little to no added effort; maintenance because the SPARQL federation solution should be conceived so as to limit development effort in cases of bug fixes or feature adaptation; evolution refers to the ability of the solution to easily be extended in order to cope with new requirements.

In particular, openness and portability of the Federation Web Service model grant this solution high KPI values, from both a development (ease) and management (integration, maintenance, evolution) point of view.

5.3.2 Implementation Details

The implementation of federation consists of two main elements: the Federation Ontology, and the Federation Web Service.

Federation Ontology

Enterprise contexts stress the need to manage large amounts of data that have characteristics such as heterogeneity and distribution of the data model. However, aggregation of heterogeneous data is crucial for business processes, such as Decision Making, Data Quality, and Data Management. This aggregation supports the concept of federation, which provides a federative part on which to build the federation itself. The implementation has gone in this direction, defining a uniform data model and to be shared between all members of the federation, in order to eliminate the problems described above. The proposed ontology is composed of three main elements: the concept of federation; the concept of the federation member; the *member_of relationship* that represents the relationship between the member of the federation and the federation itself. The ontology described above enables the management of the federation independently of semantic platform and can be deployed and queried to any SPARQL endpoint. From an implementation standpoint, the ontology is realized according to the RDF schema modeling.

Federation Web Service

The Federation Web Service manages semantic data query and aggregation via multiple steps. i) Clients can query the Web Service with a standard SPARQL query, and express a specific federation (in our examples, we modeled disparate data sets from Italian universities). ii) The Web Service inspects the federation to determine which endpoints should be queried. iii) The Web Service performs the semantic query on each such node. iv) The Web Service aggregates results from the above queries and returns them back to the caller. Result aggregation is a particularly crucial task, and the Federated Web Services support three main strategies:

- APPEND strategy. Results coming from different endpoints are simply stacked on top of each other, and sent back to the client once all replies are received. This strategy is implemented via the APPEND operator and does not handle result triple duplications.
- INTERSECTION strategy. This approach returns all the triples that are common to each involved endpoint.
- UNION strategy. Results from endpoints are combined into a set of unique triples, hence performing duplicate detection and removal.

The Federation Web Service clearly decouples the query definition, execution, and aggregation from the definition and management of the data set distribution. Even though the proposed solution may conceptually handle highly distributed data sets, some management concerns may arise: endpoint time-out, how to handle alive nodes that fail to reply to queries in a given amount of time (e.g., due to temporary network issues); ignore/retry strategies should be carefully planned to reach a viable balance between result completeness (e.g. retrying queries in order to orvercome the temporary network outage), and total response time; endpoint unavailability, endpoints may become unavailable for larger amounts of time, e.g. due to severe server/system failures.

5.3.3 Usecase

A proof of concept Federated Education Portal has been realized. It provides a synergic academic offering that federates education and administrative data sources from Italian academic institutions together with citizenship distribution data sources from Italian municipalities in order to realize a more integrated education offering.

A traditional Semantic approach allows to decouple actual data sources and their implementations, hence allowing to reuse the same SPARQL query over any academic and municipality data source, no matter the real data source implementation (would it be an ERP system backed by a traditional RDBMS, or a legacy mainframe system).

However, a SPARQL-only approach for the Federated Education Portal would require to: i) have a priori knowledge of all municipalities and universities involved in the overall integrated offering; ii) explicitly/manually perform the same SPARQL query on academic data sources to retrieve students distribution; iii) explicitly/manually perform the same SPARQL query on municipality data sources to retrieve citizen distribution; iv) explicitly map and combine both semantic result sets into a cohesive data set that highlights gaps in actual course offering with respect to real citizen distribution.

This process is obviously largely inefficient and poorly extensible: the proposed Federated Education Portal should be extended any time new data sources get added, in order to query new endpoints and combine results with old ones.

The proposed approach leverages on a Federation Ontology implementation, with an instance of the proposed Federation Web Service, that maps Italian municipalities and relevant information about citizen geographic distribution, as well as academic institutions and relevant information about courses and student distribution. Students and Citizens are linked via the semantic notion of person (via the foaf:Person ontology) and both municipalities and universities have a set of SPARQL endpoints, on top of Virtuoso Semantic middleware.

In this scenario, the proposed Federated Education Portal performs a single query to retrieve geographic distribution of both persons involved in academic courses (students), and persons (citizens) from municipalities. The Federation Web Service identifies all involved academic and municipality data sources and transparently query each one of them and takes care of combining results (e.g., via an INTERSECTION strategy).

The proposed approach dramatically facilitated the realization of the Federated Education Portal, with no a priori data source knowledge should be hard-coded into the Federated Education Portal, hence resulting in a more open and flexible solution. Since a single query can be executed both on academia and on municipality endpoints, it is possible to decrease the development efforts of the overall solution.

6 Conclusions

This chapter summarizes the contributions of the present thesis work, by reporting the most relevant findings and results related to the support of mobile devices and meet their requirements in a pervasive environment to build large-scale and industrially-feasible applications. The chapter also points out the main future research directions for further extensions of the described work.

6.1 Major Contributions

This thesis work outlines and clarifies the motivations that lead to the introduction of an intermediate middleware between mobile devices and cloud computing, the mobile devices requirements and their potentialities and relevance for future applications. Several relevant real-world systems are presented, describing their open technical challenges and highlighting the importance for the mobile devices to have powerful virtualized resources available, leveraging their utilization in real world contexts.

The main middleware solutions have been described in order to support and meet mobile devices requirements. In particular, it clarifies the topics of fog computing, edge computing and other middleware solutions, analyzing them in detail, describing their characteristics at the state of the art and motivations of their usage, and their relevance for future mobile devices systems. An original taxonomy is proposed to structure the identified features into different layers, with the three layers architecture, and distribute the features among components at different architectural levels in order to highlight correlations among the components of the middleware solution and its features. The most common categories of middleware solutions are defined, specifying the related application suitability and drawbacks for each category. In addition, due to the general confusion in the related literature where terms are usually used in a interchangeable and not proper way, a very relevant contribution of this work is to clarify each single solutions terms with the novelties brought by the related approach, by analyzing the main original elements and outlining the differences of one approach compared to the others.

The initial theoretical analysis has been expanded by proposing implemented solutions that, in different ways and following different approaches, aim to enable 3-layers infrastructure and enable the applications potentialities described in Chapter 1. In particular, impactful extensions of industrially accepted, widely used and state-of-the-art M2M protocols, technologies and framework, for fog computing solutions, and new wide-scale and novel industrial deployments have been explained, in order to meet many support requirements, such as lightweightness, scalability, security, flexibility, and so on. In fact, some gateway-oriented solutions for enhanced IoT scalability and IoT federation have already been published in the recent literature in the field, in particular, within large-scale deployment scenarios, with focus on how to effectively and efficiently achieve scalability. Even if they explored relevant solution guidelines and were someway inspiring for the community of researchers in the field, they only partially solved some aspects related to hard technical challenges thus leaving still open space for additional, especially industry-mature, solutions. In fact, with no solution yet based on a gateway-oriented architecture where gateways jointly exploit MQTT and CoAP to achieve highly scalable IoT device management, through dynamic hierarchical tree organizations, and relevantly extending promising frameworks for fog gateway. Similarly, chapter 3 describes the enrichment of the fog intermediate layer by giving the opportunity of exploiting container-based virtualization on top of IoT gateways, with full infrastructure support that includes download, update, and management of virtualized images based on industry-mature Docker. This is one of the first cases of implementation and experimentation of virtualization techniques over fog nodes, in particular while working with IoT gateways with very limited resources, such as RaspberryPi nodes. Container-based solutions for the fog offer the well-known advantages of abstracting implementation details, more easily achieving interoperability and portability between possibly heterogeneous fog nodes, etc., similarly to the adoption of virtual machines, but with lightweight migrations and better performance. Fog solution containerization can significantly leverage the introduction and diffusion of fog computing techniques in mature deployment scenarios, by allowing more automated and easier integration and installation, in particular over large-scale and industrial execution environments.

Successively, original solutions, adopting edge computing approach, have been proposed to overcome the challenges of limited-resource mobile devices in hostile environments. Elijah-based solution aims to efficiently design and implement a support platform where highly demanding computation tasks on mobile devices can be delegated to the MEC layer that executes the tasks and returns the related results. Service continuity is preserved, also in case of end users mobility and in hostile environment, without impacting device workload, thanks to proper virtualized function migration between MEC nodes. This is the first implemented MEC platform that extends the widely accepted Elijah platform with transparent hot migration of edge functions via proactive handoff mechanisms by complementing typical reactive migration with predictive handoff mechanisms that move a specific virtualized function into the most suitable MEC node, based on the consideration of multiple aspects, such as network statistics, availability, and recoverability.

In addition, Uloof is a lightweight and efficient framework for mobile computation offloading equipped with a smart decision engine that minimize remote execution overhead, while not requiring any modification in the underlying device operation system that leverages on MEC nodes to reduce execution time and energy consumption of the mobile devices computation. Many works, in the past, have tried to address computation offloading in different ways focusing on offloading execution, rather than selecting the tasks to offload, thus, they generally require the developers to add annotations to indicate which portion of application to offload, modifying the application code manually. In addition, they usually use cloud computing in order to be able to fully scan the applications. The proposed solution provides a fully autonomous code selection mechanism, at method level granularity, able to analyze every kind of Android application dynamically and detecting the list of methods suitable to be offloaded. It is based on a decoupled and lightweight solution that has no platform constraints and can be used in many remote servers, aiming to offload code at the method level that can run on every server equipped with a JVM without the same hardware/OS of the mobile device and, thus suitable also for less-powerful platforms, i.e. MEC nodes.

Finally, more cloud-based solutions have been explored, highlighting solutions suitable for more powerful devices. MCN extends the several migration and replication mechanisms proposed in the past, also at the database side, by including resource monitoring techniques based not only on reactive monitoring, but also applying predictive models, thus, triggering for support when resource usage exceeds a pre-defined threshold. It offers a more precise migration timing of the database, acting before congestion happens and avoiding slowing down the overall migration process.

As a further extension, the federation of Sparql endpoints proposes a novel, semantic-based approach to overcome organization governance complexity and to mitigate or hide heterogeneity and distribution of data sources, to overcome traditional governance models, where data harvesting and integration are expensive, difficult, and often limited to specific domains or areas. The solution promotes a higher level of data normalization and integration which proposes a more proactive governance model where data analysis may be used not only to lead retrospective analysis, but also to proactively support upcoming strategic decisions.

6.2 Future Research Directions

This thesis has address many open points and challenges related to the introduction of a powerful computational paradigm able to face the massive adoption of mobile devices and their applications. Although, many solutions have explored several important research aspects on the topic, there are still some open points, both application-specific and general purpose across applications, that are not solved yet and require further work.

The primary research directions calling for significant and novel research work that can lead to valuable benefits in the field towards an efficient and industrially applicable solutions and should be explored in the near future:

- Actuation decision. A peculiar ability of fog and edge computing is to understand the border between situations when it is necessary an immediate actuation or when it is possible to send data the cloud for intense postponed analytics. Each application, in relation to the context where it is immersed, must define sets of actions with different priority and consequently different reactions, as well its desired level of fog-cloud interplay. A relevant extension may include the ability of the system to detect, in an (semi-)autonomous way, which actions must be performed on the middleware layer and which need to be sent to the cloud, similarly to what has been proposed in Uloof that autonomously choose which tasks to execute on the mobile devices and which on the remote server. Towards this realization, the tasks dispatcher should consider the functional requirements of the application, i.e. moving the fine-grained controls or high-priority tasks on the middleware layer, and the resources available required to perform the operations.

- Dynamic allocation and service provisioning. In this thesis some solutions focus on the movement of the applications or components of the infrastructure among edge nodes in order to maximize different requirements, i.e. performance, location-awareness, traffic minimization, and so on. Although these solutions are very effective in a general-purpose scenario, by using the nearest node available that usually find the optimal allocation decision, sometimes in busy contexts, where the closest node may already have a high usage percentage, may be more efficient to allocate the applications onto another node at an acceptable distance. In the literature many work already address this problem on the cloud but, very few work focus on middleware layer leaving the space for extensions of similar concepts in a fairly more sensitive and dynamic environment.

- Compliance with emerging standards. The solutions of this thesis work exploit and benefit by following the guidelines of emerging standards, such as the IEEE P1934, proposed by the OpenFog Consortium, and the ETSI MEC, proposed by the ETSI Multi-access Edge Computing (MEC) Industry Specification Group, in order to accelerate the creation and adoption of industry standards for fog and edge computing. In addition, all the proposed solutions lay on already widespread and promising protocols, frameworks, and tools. For future solutions, it is key to continue to follow the present and the future emerging and recognized standards in the field, in order to build solid and interoperable solutions.
- Merging with SDN solutions. Nowadays SDN is a popular technology that is gathering many efforts in order to improve network performance and monitoring, by facilitating network management and enabling programmatically efficient network configuration. A valuable extension of middleware solutions could be the introduction of SDN solutions, through the adoption of already widespread SDN controllers, e.g. ONOS, OpenDaylight, that orchestrate middleware nodes in the most efficient and effective way.
- Industrial applications. Many proposed solutions are specifically targeted for large-scale and industrial environments, by adopting framework, protocols, and tools able to handle real-world application scenarios. In addition, they have been fully tested also in large-scale, busy, and high workload contexts. These encouraging results already achieved call for further experimental work in the field, by experimenting those solutions in real industrial production sites with strict latency constraints and thousands of geographically distributed vending machines.

This thesis work has proposed original approaches to face the challenges related to the diffusion and widespread usage of a wide range of mobile devices on the topic, trying to introduce solutions that best fit the different application scenarios. The hope is that this work can be successfully extended and used as a base for further research activities to build novel solutions for future large scale applications that can face the increasing complexity and proliferation of mobile devices adoption.

Acronyms

AMQP	Advanced Message Queuing Protocol
BS	Base Station
CAM	Cooperative Awareness Messages
CC	Cloud Controller
CD	Continuous Delivery
CDF	Cumulative Distribution Function
CI	Continuous Integration
CLI	Command Line Interface
CoAP	Constrained Application Protocol
CPS	Cyber Physical System
CoRE	Constrained RESTful Environments
COW	Copy on Write
CRIU	Checkpoint/Restore in Userspace
CRM	Customer Relationship Management
CTH	CoAPTreeHandler
DENM	Decentralized Environmental Notification Messages
DS	DataService
DTLS	Datagram Transport Layer Security
E2E	End to End
EU	End User
EEU	Enterprise End-user
ERP	Enterprise Resource Planning
ETSI	Telecommunications Standards Institute
FMC	Follow-Me Cloud
IaaS	Infrastructure as a Service
ICN	Information Centric Network
IoT	Internet of Things
ISG	Industry Specification Group
ITG	Infrastructure Template Graph
LXC	LinuX Containers
M2M	Machine to Machine
MaaS	Monitoring as a Service
MCN	Mobile Cloud Networking
MQTT	Message Queue Telemetry Transport
MEC	Mobile Edge Computing

NAT	Network Address Translation
NFV	Network Function Virtualization
NIST	National Institute of Standards and Technology
NBI	Northbound Interface
NSD	Network Service Discovery
OBU	On Board Unit
OS	Object Serialization
OSGi	Open Service Gateway Initiative
P2P	Peer-to-Peer
PaaS	Platform as a Service
PID	Proportional-Integral-Derivative
QoE	Quality of Experience
RA	Requests Aggregation
RabbitMQ	Rabbit Message Queue
RAN	Radio Access Network
RD	Resource Directory
RDF	Resource Description Framework
RQ	Remote Query Resource
RP	Resource Directory Parsing
RSU	Road Side Unit
RTT	Round Trip Time
SaaS	Software as a Service
SCV	Smart Connected Vehicles
SDN	Software Defined Network
SLA	Service Level Agreement
SM	Service Manager
SO	Service Orchestrator
SR	String Refactoring
STG	Service Template Graph
UI	User Interface
UUID	Universally Unique Identifier
WSAN	Wireless Sensors and Actuators Networks
WSN	Wireless Sensors Networks
VM	Virtual Machine
VNF	Virtual Network Functions
VPN	Virtual Private Networking

Bibliography

[1] Cisco VNI Forecast : Cisco Visual Networking Index: Global Mobile data Traffic Forecast Update 20162021. Cisco Public Information. Available online at: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf, 2016.

[2] J. Manyika, et al., Disruptive technologies: Advances that will transform life, business, and the global economy, McKinsey Global Institute, 2013.

[3] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, E. Jansen, The Gator Tech Smart House: A Programmable Pervasive Space, IEEE Computer, Vol. 38, No. 3, 2005.

[4] X. Yu, A. Pan, L.-A. Tang, Z. Li, and J. Han, "Geo-Friends Recommendation in GPS-Based Cyber-Physical Social Network," Proceedings of the 2011 International Conference on Advances in Social Networks Analysis and Mining, 2011.

[5] A.E. Lee, "Cyber physical systems: Design challenges", 11th IEEE Int. Symposium on. Object oriented real-time distributed computing (ISORC), 2008.

[6] R.R. Rajkumar, et al. "Cyber-physical systems: the next computing revolution." Proceedings of the 47th Design Automation Conference. ACM, 2010.

[7] J. Lee, B. Bagheri, H.A. Kao, "A cyber-physical systems architecture for industry 4.0-based manufacturing systems." Manufacturing Letters 3, pp. 18-23, 2015.

[8] G. Cardone, et al., "Participact: A large-scale crowdsensing platform", IEEE Transactions on Emerging Topics in Computing, pp. 21-32, 2016.

[9] A.S. Massoud, B.F. Wollenberg, "Toward a smart grid: power delivery for the 21st century", IEEE power and energy magazine 3.5, pp. 34-41, 2005.

[10] K. Moslehi, R. Kumar, Smart Grid - A Reliability Perspective, Innovative Smart Grid Technologies (ISGT), 2010.

[11] A. Molderink, et al., "Management and control of domestic smart grid technology", IEEE transactions on Smart Grid 1.2, pp. 109-119, 2010.

[12] DOE - Buildings Energy Data Book, Department of Energy, 2009: http://buildingsdatabook.eren.doe.gov.

[13] J. Kleissl, Y. Agarwal, Cyber-Physical Energy Systems: Focus on Smart Buildings, Design Automation Conference (DAC), 2010 47th ACM/IEEE, 2010.

[14] V.L. Erickson, et al., ”OBSERVE: Occupancy-Based System for Efficient Reduction of HVAC Energy”, 2011.

[15] H. Chen, et al., ”The Design and Implementation of a Smart Building Control System”, 2009.

[16] UnitedNations, World Urbanization Prospects: The 2014 Revision, Department of Economic and Social Affairs, United Nations, 2014.

[17] S.A. Shaheen, R. Finson, Intelligent Transportation Systems, Encyclopedia of Energy, 487-496, 2004.

[18] G. Karagiannis, et al. "Vehicular networking: A survey and tutorial on requirements, architectures, challenges, standards and solutions", IEEE communications surveys & tutorials, 2011.

[19] B. Zhou, et al. "Adaptive traffic light control in wireless sensor network-based intelligent transportation system", IEEE 72nd.Vehicular technology conference fall (VTC 2010-Fall), 2010.

[20] REN21 Renewables 2012 Global Status Report. Available online at: http://www.ren21.net, 2012.

[21] L.Y. Pao, K.E. Johnson, "A tutorial on the dynamics and control of wind turbines and wind farms." American Control Conference IEEE ACC, 2009.

[22] F. Bonomi, et al., "Fog computing: A platform for internet of things and analytics", Big Data and Internet of Things: A Roadmap for Smart Environments. Springer International Publishing, pp. 169-186, 2014.

[23] S. Fadi, J. Ordieres, G. Miragliotta. "Smart factories in Industry 4.0: A review of the concept and of energy management approached in production based on the Internet of Things paradigm.", IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), 2014.

[24] D. Jeffrey, S. Ghemawat, "MapReduce: simplified data processing on large clusters", Communications of the ACM 51.1, pp. 107-113, 2008.

[25] K. Shvachko, et al. "The hadoop distributed file system.", IEEE 26th symposium on. Mass storage systems and technologies (MSST), 2010.

[26] Apache Spark. Available online at https://spark.apache.org.

[27] D. Laney, "3D data management: Controlling data volume, velocity and variety." META Group Research Note 6, 2001.

[28] A. Toninelli, et al., Supporting context awareness in smart environments: a scalable approach to information interoperability, ACM Int. Workshop on Middleware for Pervasive Mobile and Embedded Computing, 2009.

[29] P. Bellavista, et al., Integrated support for handoff management and context awareness in heterogeneous wireless networks, 3rd ACM Int. Workshop on Middleware for Pervasive and Ad-hoc Computing, 2005.

[30] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog Computing and Its Role in the Internet of Things, Proceedings of the first edition of the MCC workshop on Mobile Cloud Computing, 2012.

[31] P. Mell, T. Grance, "The NIST definition of cloud computing", 2011.

[32] P. Bellavista, A. Corradi, E. Magistretti, REDMAN: An optimistic replication middleware for read-only resources in dense MANETs, Elsevier Pervasive and Mobile Computing Journal, vol. 1, no. 3, pp. 279-310, 2005.

[33] L. Wang, Z. Wang, R. Yang, ”Intelligent Multiagent Control System for Energy and Comfort Management in Smart and Sustainable Buildings”, 2012.

[34] P. Zarko, A. Antonic, K. Pripuzic, Publish/Subscribe Middleware for Energy-Efficient Mobile Crowdsensing, ACM Conf. Pervasive and Ubiquitous Computing (UbiComp), 2013.

[35] N.D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, A.T. Campbell, A Survey of Mobile Phone Sensing, IEEE Communications Magazine, Vol. 48, No. 9, pp. 140-150, Sept. 2010.

[36] A. Faisal, D. Petriu, M. Woodside, Network Latency Impact on Performance of Software Deployed Across Multiple Clouds, Int. Conf. Center for Advanced Studies on Collaborative Research (CASCON), 2013.

[37] A. Botta, et al. "Integration of cloud computing and internet of things: a survey", Future Generation Computer Systems pp. 684-700, 2016.

[38] M.A. Razzaque, et al. "Middleware for internet of things: a survey." IEEE Internet of Things Journal, pp. 70-95, 2016.

[39] Ö. Yürür, et al. "Context-awareness for mobile sensing: A survey and future directions", IEEE Communications Surveys Tutorials, pp. 68-93, 2016.

[40] R. Petrolo, et al., "Integrating wireless sensor networks within a city cloud", 11th Annual IEEE International Conference on.Sensing, Communication, and Networking Workshops (SECON Workshops), 2014.

[41] L. Alonso, et al. "Middleware and communication technologies for structural health monitoring of critical infrastructures: A survey." Computer Standards Interfaces, pp. 83-100, 2018.

[42] G. Buttazzo, "Hard real-time computing systems: predictable scheduling algorithms and applications.” Springer Science & Business Media, vol. 24, 2011.

[43] W. Jansen, K. Scarfone, Guidelines on Cell phone and PDA security, NIST 800-124, 2008.

[44] OpenFog Consortium Architecture Working Group (2017) OpenFog Architecture Overview White Paper.

[45] L.M. Vaquero, L. Rodero-Merino, Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing, ACM SIGCOMM Computer Communication Review table of contents archive, vol. 44, issue 5, 2014.

[46] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, The Case for VM-Based Cloudlets in Mobile Computing, IEEE Pervasive Computing, vol. 8, no. 4, pp. 1423, Oct. 2009. [Online]. Available: http://ieeexplore.ieee.org/document/5280678/

[47] National Institute for Standard and Technology (NIST), Big Data Interoperability Framework: Volume 1, Definitions, NIST Special Publication 1500-1, 2015.

[48] O. Ferrer-Roca, R. Milito, Big and Small data: The Fog, International Conference on Identification, Information and Knowledge in the Internet of Things, 2014.

[49] Y. Mao, et al. "A survey on mobile edge computing: The communication perspective", IEEE Communications Surveys and Tutorials, 2017.

[50] X. Chen, et al. "Efficient multi-user computation offloading for mobile-edge cloud computing", IEEE/ACM Transactions on Networking, pp. 2795-2808, 2016.

[51] E. Cau, et al. "Efficient exploitation of mobile edge computing for virtualized 5G in EPC architectures", 4th IEEE International Conference on. IEEE Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2016.

[52] R. Bifulco, M. Brunner, R. Canonico, P, Hasselmeyer, F. Mir, Scalability of a mobile Cloud management system, MCC '12 Proceedings of the first edition of the MCC workshop on Mobile Cloud computing, pp. 17, 2012.

[53] Y.C. Hu, et al., Mobile Edge Computing A key technology towards 5G-First edition, ETSI White Paper, 2015.

[54] A. Ahmed, A. Ejaz, "A survey on mobile edge computing", IEEE 10th International Conference on Intelligent Systems and Control (ISCO), 2016.

[55] M. Firdhous, O. Ghazali, S. Hassan, Fog Computing: Will it be the Future of Cloud Computing?, Proceedings of the Third International Conference on Informatics Applications, Kuala Terengganu, Malaysia, 2014.

[56] R.A. Ali, et al. (2014) An Architecture-Based Approach for Compute-Intensive Pervasive Systems in Dynamic Environments, 5th ACM/SPEC International Conference on Performance Engineering ICPE.

[57] M. Satyanarayanan, et al. (2013) The role of Cloudlets in hostile environments, IEEE Pervasive Computing, vol. 12, no. 4, pp. 40-49.

[58] K. Kumar and Yung-Hsiang Lu, Cloud Computing for Mobile Users: Can Offloading Computation Save Energy? Computer, vol. 43, no. 4, pp. 5156, Apr. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5445167/

[59] A.V. Kempen, et al., MEC-ConPaaS: An experimental single-board based mobile edge cloud, IEEE Mobile Cloud Conference, 2017

[60] F. Bao, I.R. Chen, J. Guo, Scalable, adaptive and survivable trust management for community of interest based Internet of Things sys-tems, 11th IEEE Int. Symp. Autonomous Decentralized Systems (ISADS), pp. 1-7, 2013.

[61] I. Ishaq, J. Hoebeke, I. Moerman, P. Demeester, Internet of Things Virtual Networks: Bringing Network Virtualization to Resource-Constrained Devices, IEEE International Conference on Green Computing and Communications, pp. 293-300, 2012.

[62] A.Al-Fuqaha, et al., Towards Better Horizontal Integration among IoT Services, IEEE Communications Magazine, vol. 53, no. 9, pp. 72-79, 2015.

[63] S.M. Kim, H.S. Choi, W.S. Rhee, IoT Home Gateway for Auto-Configuration and Management of MQTT Devices, IEEE Conference on Wireless Sensors, 2015.

[64] L. Wu, Y. Xu, C. Xu, and F. Wang, Plug-configure-play service-oriented gateway-for fast and easy sensor network application development, SENSORNETS, pp. 53-58, 2013.

[65] J. Maenpaa, J. Jiménez Bolonio, S. Loreto, Using RELOAD and CoAP for Wide Area Sensor and Actuator Networking, Eurasip J. Wireless Communications and Networking, Vol. 121, pp. 1-22, 2012.

[66] M. Kovatsch, M. Lanter, Z. Shelby, Californium: Scalable Cloud Services for the Internet of Things with CoAP, IEEE 4th International Conference on the Internet of Things, 2014.

[67] H. Jo, H. Jin, Adaptive Periodic Communication over MQTT for Large-Scale Cyber-Physical Systems, IEEE 3rd Int. Conference on Cyber-Physical Systems, Networks, Applications, 2015.

[68] Y.T. Lee, et al., An Integrated Cloud-Based Smart Home Management System with Community Hierarchy, IEEE Transactions on Consumer Electronics, vol. 62, issue 1, 2016.

[69] S. Yi, et al., "Fog computing: Platform and applications." Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on. IEEE, 2015.

[70] D. Bernstein, Containers and Cloud: From LXC to Docker to Kubernetes, IEEE Cloud Computing, Vol. 1, No. 3, pp. 81-84, 2014.

[71] C. Pahl, B. Lee, Containers and Clusters for Edge Cloud Architectures a Technology Review, 3rd Int. Conf. on Future Internet of Things and Cloud (FiCloud), 2015.

[72] A. Arif, G. Pierre, "Efficient Container Deployment in Edge Computing Platforms." RESCOM 2017, 2017.

[73] M.S. De Brito, et al., "Towards programmable fog nodes in smart factories.", IEEE International Workshops on Foundations and Applications of Self Systems, 2016.

[74] A. Van Kempen, et al., "MEC-ConPaaS: An experimental single-board based mobile edge cloud." IEEE Mobile Cloud Conference. 2017.

[75] MQTT - Message Queue Telemetry Transport. Available online at: http://mqtt.org.

[76] CoAP Constrained Application Protocol. Available online at: http://coap.technology.

[77] Z. Shelby, K. Hartke, C. Bormann, The Constrained Application Protocol (CoAP), 2014.

[78] K. Hartke, Observing Resources in CoAP, IETF Internet Draft, 2014. Available online at: https://tools.ietf.org/html/draft-ietf-core-observe-16.

[79] C. Bormann, Block-wise transfers in CoAP draft-ietf-core-block-17, IETF Internet Draft, 2015. Available online at: https://tools.ietf.org/html/draft-ietf-core-block-17.

[80] Z. Shelby, C. Bormann, CoRE Resource Directory draft-ietf-core-resource-directory-02, IETF Internet Draft, 2014. Available online at: https://tools.ietf.org/html/draft-ietf-core-resource-directory-02.

[81] Z. Shelby, Constrained RESTful Environments (CoRE) Link Format, IETF Internet Draft, 2012. Available online at: https://tools.ietf.org/html/rfc6690.

[82] E. Rescorla, N. Modadugu, Datagram Transport Layer Security Version 1.2, IETF Internet Draft, 2012 . Available online at: https://tools.ietf.org/html/rfc6347.

[83] Available online at: https://hc.apache.org/httpcomponents-core-ga/index.html.

[84] K. Boumillion, J. Levy, Guava: Google core libraries for Java 1.5+. Available online at: https://github.com/google/guava.

[85] L. Rodrigues, J. Guerreiro, N. Correia, RELOAD/CoAP Architecture with Resource Aggregation/Disaggregation Service, 2016.

[86] M. Nottingham, E. Hammer-Lahav, Defining Well-Known Uniform Resource Identifiers (URIs), IETF Internet Draft, 2010. Available online at: https://tools.ietf.org/html/rfc5785.

[87] M. Nottingham, Web Linking, IETF Internet Draft, 2010. Available online at: https://tools.ietf.org/html/rfc5988.

[88] Kura. Available online at: https://eclipse.org/kura.

[89] P. Bellavista, A. Corradi, C. Giannelli, "Mobile Proxies for Proactive Buffering in Wireless Internet Multimedia Streaming", 25th IEEE Int. Conf. Distributed Computing Systems Workshops, 2005.

[90] A. Toninelli, et al. "Supporting Context Awareness in Smart Environments: a Scalable Approach to Information Interoperability", ACM Int. Workshop Middleware for Pervasive Mobile and Embedded Computing, 2009.

[91] P. Bellavista, A. Corradi, C. Giannelli, "The Real Ad-hoc Multi-hop Peer-to-peer (RAMP) Middleware: an Easy-to-use Support for Spontaneous Networking", IEEE Symp. Computers and Communications (ISCC), 2010.

[92] Mosquitto. Available online at: http://mosquitto.org.

[93] Scandium. Available online at: https://github.com/eclipse/californium.scandium.

[94] Kryo. Available online at: https://github.com/EsotericSoftware/kryo.

[95] Paho. Available online at: https://eclipse.org/paho.

[96] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An Updated Performance Comparison of Virtual Machines and Linux Containers, Technical Report RC25482 (AUS1407-001), IBM Research Division, 2014.

[97] LXC - Linux Containers. Available online at: https://linuxcontainers.org.

[98] CRIU - Checkpoint/Restore in Userspace. Available at: http://www.criu.org.

[99] Docker. Available online at: https://www.docker.io.

[100] AUFS - Advanced multi layered unification filesystem. Available online at: http://aufs.sourceforge.net.

[101] D.Merkel, Docker: Lightweight Linux Containers for Consistent Development and Deployment, Linux Journal, Vol. 239, 2014.

[102] Device-mapper. Available online at: http://www.sourceware.org/dm.

[103] OverlayFS. Available online at: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documen

[104] P. Bellavista, A. Corradi, Alessandro Zanni, Integrating Mobile Internet of Things and Cloud Computing towards Scalability: Lessons Learned from Existing Fog Computing Architectures and Solutions, 3rd Int. IBM Cloud Academy Conf. (ICA CON), 2015.

[105] Docker Swarm. Available online at: https://www.docker.com/products/docker-swarm.

[106] Kubernetes. Available online at: http://kubernetes.io.

[107] Apache Mesos. Available online at: https://mesos.apache.org.

[108] J.S. Preden, et al. "The benefits of self-awareness and attention in fog and mist computing." Computer 48.7 (2015): 37-45.

[109] V. Beltran, J. Paradells. "Middleware-based solution to offer mobile presence services", Proceedings of the 1st int. conf. on MOBILe Wireless MiddleWARE, Operating Systems, and Applications, 2008.

[110] C. Wu, Ho. J, and M. Chen, "A Scalable Server Architecture for Mobile Presence Services in Social Network Applications", IEEE Transactions on Mobile Computing, vol. 12(2), pp. 386-398, 2011.

[111] MongoDB Available online at: https://www.mongodb.com.

[112] QEMU. Avaialble online at: http://www.qemu.org.

[113] S. Izchak, A. Schuster, D. Keren, "A geometric approach to monitoring threshold functions over distributed data streams." ACM Transactions on Database Systems (TODS) 32.4 (2007): 23.

[114] D. Keren, et al. "Geometric monitoring of heterogeneous streams." IEEE Transactions on Knowledge and Data Engineering 26.8 (2014): 1890-1903.

[115] D. Ongaro, J. Ousterhout, "Raft consensus algorithm", 2015.

[116] Docker compose, Available online at: https://docs.docker.com/compose.

[117] K. Ha, et al., "Adaptive VM Handoff Across Cloudlets", Technical Report CMU-CS-15-113, CMU School of Computer Science, 2015.

[118] E. Saurez, et al., Incremental Deployment and Migration of Geo-Distributed Situation Awareness Applications in the Fog, ACM Distributed and Event-based Systems Int. Conf., pp. 258-269, 2016.

[119] M. Felemban, S. Basalamah, A. Ghafoor, A Distributed Cloud Architecture for Mobile Multimedia Services, 2013.

[120] A. Ravi, S.K. Peddoju, Handoff Strategy for Improving Energy Efficiency and Cloud Service Availability for Mobile Devices, Wireless Personal Communications, vol. 81, issue 1, pp 101132, 2015.

[121] G. Lewis, S. Echeverría, S. Simanta, B. Bradshaw, J. Root, Tactical Cloudlets: Moving Cloud Computing to the Edge, IEEE Military Communications Conference, 2014.

[122] S. Simanta, et al., A Reference Architecture for Mobile Code Offload in Hostile Environments, Proc. of MobiCase: 4th International Conference on Mobile Computing, Applications and Services, 2012.

[123] M.G.R. Alam, et al., Multi-agent and Reinforcement Learning Based Code Offloading in Mobile Fog, International Conference on Information Networking (ICOIN), 2016.

[124] E. Cuervo et al., MAUI: Making Smartphones Last Longer with Code Offload, in MobiSys Conf., 2010.

[125] S. Kosta, et al., "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading", in IEEE Infocom, 2012.

[126] M.S. Gordon, et al., COMET: Code Offload by Migrating Execution Transparently, in OSDI, vol. 12, pp. 93-106, 2012.

[127] R. Kemp, et al., Cuckoo: A Computation Offloading Framework for Smartphones, in International Conference on Mobile Computing, Applications, and Services, pp. 59-79, 2010.

[128] C. Shi, et al., Cosmos: computation offloading as a service for mobile devices, in Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing, pp. 287-296, 2014.

[129] J.L.D. Neto, D.F. Macedo, J.M.S. Nogueira, Location aware decision engine to offload mobile computation to the cloud, IEEE NOMS, 2016.

[130] M. Satyanarayanan, et al., The Case for VM-Based Cloudlets in Mobile Computing, IEEE Pervasive Computing, vol. 8, no. 4, pp. 14-23, 2014.

[131] S. Yi, L. Cheng, L. Qun. "A survey of fog computing: concepts, applications and issues", in Proceedings of Workshop on Mobile Big Data, 2015.

[132] B.G. Chun, et al., "Clonecloud: elastic execution between mobile device and cloud", in Proceedings of the sixth conference on Computer systems, 2011.

[133] Openstack. Available online at: https://www.openstack.org.

[134] Elijah. Available online at: http://elijah.cs.cmu.edu.

[135] Elijah-provisioning. Available online at: https://github.com/cmusatyalab/elijah-provisioning.

[136] K. Ha, M. Satyanarayanan, Openstack++ for cloudlet deployment, School of Computer Science Carnegie Mellon University Pittsburgh, 2015.

[137] Avahi. Available online at: http://avahi.org.

[138] Zero Configuration Networking (Zeroconf). Available online at: http://www.zeroconf.org.

[139] F. Siddiqui, S. Zeadally, T. Kacem, S. Fowler, Zero Configuration Networking: Implementation, performance, and security, ComputersElectrical Engineering, Vol. 38, Issue 5, pp. 11291145, 2012.

[140] Apache JCLOUD. Available online at: https://jclouds.apache.org/start/compute.

[141] Network Time Protocol (NTP). Available online at: http://www.ntp.org.

[142] LibSVM. Available online at: http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[143] Network Service Discovery (NSD). Available online at: https://www.w3.org/TR/discovery-api.

[144] OpenCV (Open Source Computer Vision Library). Available online at: http://opencv.org/about.html.

[145] W. Hu, et al., Quantifying the Impact of Edge Computing on Mobile Applications, Proc. of SIGOPS Asia-Pacific Workshop on Systems, 2016.

[146] K. Ha, et al., Just-in-Time Provisioning for Cyber Foraging, Proc. of ACM Mobile systems, applications, and services, 2013.

[147] Soot Framework. Available online at: https://sable.github.io/soot.

[148] R. Vallée-Rai, L.J. Hendren, Jimple: Simplifying Java Bytecode for Analyses and Transformations, 1998.

[149] M. Shepperd, "A critique of cyclomatic complexity as a software metric", in Software Engineering Journal, pp. 30-36, 1988.

[150] CyVis - Software Complexity Visualiser. [Online]. Available: http://cyvis.sourceforge.net/

[151] ULOOF project (website): https://uloof.lip6.fr.

[152] Dex2jar. [Online]. Available: https://github.com/pxb1988/dex2jar.

[153] OpenStack Heat. Available online at: https://wiki.openstack.org/wiki/Heat.

[154] Nirmata. Available online at: http://nirmata.com.

[155] Hurtle. Available online at: http://hurtle.it.

[156] OASIS (Topology and Orchestration Specification for Cloud Applications). Available online at: http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/TOSCA-v1.0-csd03.html.

[157] T. Binz, G. Breiter, F. Leymann, T. Spatzier, Portable Cloud Services Using TOSCA, in Internet Computing, vol. 16, no. 3, pp. 80-85, 2012.

[158] Cloudify. Available online at: http://getcloudify.org.

[159] M. F. Bari, et al., "CQNCR: optimal VM migration planning in cloud data centers", in Proceedings of the IFIP Networking Conference, 2014.

[160] F. Callegati, W. Cerroni, Live Migration of Virtual Network Functions in Cloud-Based Edge Networks, in IEEE SDN for Future Networks and Services (SDN4FNS), pp. 1-6, 2013.

[161] M. R. Hines, U. Deshpande, K. Gopalan, Post-copy live migration of virtual machines, in ACM SIGOPS Operat Syst Rev 43(3): 1426, 2009.

[162] F. P. Tso, G. Hamilton, K. Oikonomou, D.P. Pezaros, Implementing Scalable, Network-Aware Virtual Machine Migration for Cloud Data Centers, in Proceedings of IEEE Sixth International Conference on Cloud Computing, 2013.

[163] X. Meng, V. Pappas, and L. Zhang, Improving the scalability of data center networks with traffic-aware virtual machine placement, in Proceedings IEEE INFOCOM, pp. 19, 2010.

[164] V. Mann, et al., Remedy: Network-aware steady state VM management for data centers, in IFIP'12 Proceedings of the 11th international IFIP TC 6 conference on Networking, vol. 7289, pp. 190204, 2012.

[165] D. Jayasinghe, et al., Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement, in IEEE International Conference. on Services Computing, pp. 7279, 2011.

[166] S. Sakr, Cloud-hosted databases: technologies, challenges and opportunities, in Cluster Computing, vol. 17, issue 2, pp. 487502, 2014.

[167] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration., in Proceedings of the VLDB Endowment, vol. 4, issue 8, pp. 494-505, 2011.

[168] A. J. Elmore, S. Das, D. Agrawal, A. El Abbadi, Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms, in Proceedings of the ACM SIGMOD International Conference on Management of data, pp. 301-312, 2011.

[169] S. K. Barker, Y. Chi, H. J. Moon, H. Hacigümüs, P. J. Shenoy. "Cut me some slack: latency-aware live migration for databases, in Proceedings of the 15th International Conference on Extending Database Technology, pp. 432-443, 2012.

[170] O. Schiller, N. Cipriani, and B. Mitschang, ProRea: Live Database Migration for Multi-Tenant RDBMS with Snapshot Isolation, in Proceedings of the 16th International Conference on Extending Database Technology, pp. 53-64, 2013.

[171] E. Cecchet, R. Singh, U. Sharma, P.J. Shenoy, Dolly: virtualization-driven database provisioning for the cloud, in Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE), 2011.

[172] N. A. Rakhmawati, M. Hausenblas, On the Impact of Data Distribution in Federated SPARQL Queries, Semantic Computing (ICSC), 2012 IEEE Sixth International Conference on, 2012.

[173] F. Alahmari, Evaluating SPARQL using query federation and link traversal, Digital Information Management (ICDIM), 2011 Sixth International Conference on, 2011.

[174] T. M. Bohnert, A. Edmonds, MCN D2.2: Overall Architecture Definition, Release 1, mobile-cloudnetworking.eu/site/index.php?process=download&id=124&code=93b79f8f5b99f67a6cdc28369c05b65f624cfee7, Oct 2013.

[175] Mobile Cloud Networking project. Available online at: http://www.mobile-cloud-networking.eu.

[176] "Zabbix: An Enterprise-Class Open Source Distributed Monitoring Solution," [Online]. Available: http://www.zabbix.com/. [Accessed January 2015].

[177] C. Clark, et al., Live Migration of Virtual Machines, Proceedings of 2nd USENIX Symposium on Networked Systems Design Implementation (NSDI), 2005.

[178] H. L. Wang, Grey Cloud Model and Its Application in Intelligent Decision Support System Supporting Complex Decision, International Colloquium on Computing, Communication, Control, and Management, vol. 1, pp. 542546, 2008.

[179] P. Bellavista, A. Corradi, C. Giannelli, Evaluating Filtering Strategies for Decentralized Handover Prediction in the Wireless Internet, Proceedings. 11th IEEE Symposium on Computers and Communications, 2006.

[180] Cyclops. Available online at: http://icclab.github.io/cyclops.

[181] InfluxDB. Available online at: https://influxdata.com.

[182] R.N Calheiros, et al., "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, in Software: Practice and expe-rience, pp. 23-50, 2011.

[183] W3C Recommendation: Semantic Web, W3C, 2015 http://www.w3.org/2001/sw/.

[184] W3C Recommendation: OWL Web Ontology Languange, W3C, 2004. Avaialable online at: http://www.w3.org/TR/owl-ref/.

[185] W3C Recommendation: SPARQL 1.1 Federated Query, W3C, 2013. Avaialable online at: http://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/

[186] A. Zimmermann, et al., Towards Service-Oriented Enterprise Architectures for Big Data Applications in the Cloud, 17th IEEE International Enterprise Distributed Object Computing Conference Workshops, 2013.

Acknowledgements

I would like to thank Prof. Paolo Bellavista and Prof. Antonio Corradi for the incredible competence, kindness and availability, that constituted an invaluable guidance during the whole Ph.D. studies. Thank you also for granting me great opportunities.

Sincere thanks to Prof. Stefano Secci, from UPMC-Sorbonne Universitès, and Prof. Mikael Gidlund, from Mid-Sweden University, for the opportunity to work with their research groups, respectively, the Laboratoire d'Informatique de Paris6 (LIP6)-Phare group and the Sensible Things that Communicate (STC) research group. It was a pleasure working with their teams. Among all the nice and talented people I met during the experiences abroad, in particular, thanks to Se-young, from LIP6-Phare, for the kindness and the knowledge shown during the valuable work and long meetings together, and to Ulf and Stefan, from STC, for the support during my staying in Sweden.

Thanks to all the DISI research group at University of Bologna: Luca, Rebecca, Carlo, Alessandro, Jacopo, Riccardo, Michele, Domenico, Giuseppe, Carlo, Marco everyone very nice, helpful in every occasion and extremely competent. Thanks to my officemates of the CVLab for the great coffee breaks and the pleasant leisure time together.

A special acknowledge goes to my old and closest friends from Modena, that always bring me happiness and fun every time I meet them.

Many thanks to my parents and my brother Davide that have supported me in many ways, not only for the Ph.D., but during all my studies and working experiences.

Finally, the biggest thank to Alice, the person who share with me every single moment: happiness, joy, but also taught or stressful periods. Thank you for the comprehension during the many sleepless nights, working weekends, long distance periods and for being an incredible and reliable support that help me to overcome all the difficulties.

fin.