POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

COMPUTER SCIENCE AND ENGINEERING
MASTER'S THESIS

QOS-AWARE RUN-TIME TASK ALLOCATION IN HETEROGENEOUS FOG ARCHITECTURE: A VIDEO SURVEILLANCE SCENARIO

Author:
dott. Filippo Sciamanna

Student ID (Matricola):
898515

Supervisor (Relatore):
Prof. William Fornaciari

Co-Supervisor (Correlatore):
Dott. Michele Zanella

A.Y. 2019/2020

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Professor William Fornaciari for his help and availability in the last crucial phases of our work and Michele Zanella for his patience and for the valuable advice given both during the development and during the writing of this thesis.

I am grateful to my parents Antonella and Marco, for giving me the opportunity to follow this path and for having been there every time it was needed. A sincere thanks also to my sister Elisa, always ready to make me laugh.

A heartfelt thanks to Alberto and Andrea. Without you, these years of study would not have been so pleasant. I must also thank all my dearest friends, your support has been essential.

Finally, I want to thank my Mimmi. You have always been there ready to listen to me in the most difficult moments. What is certain is that without having you by my side I would never have made it.

Sommario

Negli ultimi anni il numero di dispositivi IoT è cresciuto in maniera esponenziale, evidenziando le limitazioni dell'attuale infrastruttura Cloud, non progettata per gestire un così grande volume di dati né per supportare i requisiti delle applicazioni emergenti. In questo contesto, nuovi paradigmi di computazione hanno cominciato ad esplorare la possibilità di estendere i servizi offerti dal Cloud verso l'esterno della rete, utilizzando dispositivi geograficamente più vicini a dove i dati vengono generati. Ciò richiede l'esecuzione di calcoli aggiuntivi per distribuire il carico tra i dispositivi mobili e gli oggetti interconnessi, tenendo conto di eventuali vincoli sulle risorse. A questo proposito, diventa impellente la necessità di un nuovo modello di programmazione, nonché di nuovi approcci di gestione delle risorse e di distribuzione delle singole parti di un'applicazione, in grado di affrontare l'eterogeneità dei dispositivi coinvolti.

Questa tesi presenta l'analisi e l'implementazione di un'applicazione di videosorveglianza in uno scenario sperimentale di Fog computing. Sfruttando il modello di programmazione sviluppato all'interno del progetto europeo MANGO, l'applicazione è scomposta in moduli che possano essere distribuiti sui dispositivi disponibili in rete, tramite l'utilizzo del software BarbequeRTRM. A tal proposito, proponiamo due differenti politiche di distribuzione dei moduli con l'obiettivo di massimizzare le prestazioni dell'applicazione, considerando alcuni aspetti, quali carico e connettività, dei dispositivi disponibili al momento dell'esecuzione. Successivamente, una valutazione sperimentale eseguita su un sistema reale, fornirà diversi scenari di esecuzione mostrando come con le funzionalità offerte dall'adozione del framework e l'utilizzo delle politiche proposte si ottenga un miglioramento fino al 66% della latenza di elaborazione.

Abstract

Over the last years, the number of IoT devices has grown exponentially. This trend highlights the limitations of the current Cloud infrastructure, not designed neither to deal with the volume of data generated by IoT devices nor to support the real-time requirements of emerging applications. In this context, new computing paradigms like Fog and Edge computing began to explore the possibility of extending the services offered by the Cloud to be closer to the source of data. This requires additional computations to distribute the load between interconnected mobile devices and smart objects, taking into account their resource constraints. In this regard, it becomes compelling the need for a novel programming model, as well as resource management and tasks scheduling approaches able to deal with the heterogeneity of the involved devices.

This thesis presents the analysis and the implementation of a video surveillance use-case application in a Fog computing scenario. By leveraging the MANGO programming model, the application is decomposed into tasks that can be deployed on the available devices on the network, guided by the BarbequeRTRM resource manager. In this regard, we propose two different scheduling policies, LAVA and LAVAnet, to maximize application performance, considering run-time aspects, such as load and connectivity, of the time-varying available devices. Through an experimental evaluation performed on a real cluster equipped with heterogeneous embedded boards, we will provide different execution scenarios to show the functionality of the framework and the benefit of a distributed approach, leading up to an improvement of 66% on the frame processing latency.

CHAPTER 1

Introduction

1.1 Evolution of computing

At the beginning of 1960s computers were large and expensive so that very few organizations could afford them. Moreover, those machines had to run one program at a time, spending most of the time in waiting state. In order to solve this problem and make full use of the available resources, the first *time-sharing* mainframes was developed. This approach allowed a central computer to be shared by many users connected through terminals. The central processor was used by each process for a fixed period of time. When the time was over, the program has been interrupted and the next one could resume the execution. In the following years, one of the main goal of computer industries was to make computing power accessible to a large set of people. At the same time, various researches started to investigate the possibility to interconnect different machines and distribute the computation between them.

In this context, the first network protocols, like Token Ring, ARCNET and Ethernet, were born allowing universities and private companies to create local networks. Between the late 60s and early 70s in the USA, the ARPANET project began with the goal of create the first wide packet-switching network that would connect the various university networks with each other. The ARPANET used

nodes interface message processors (IMPs) connected by 56 Kb/s telephone lines to allow the communication between devices of the network. The first IMP was installed in University of California at Los Angeles(UCLA) in 1969 and, by the end of that year, another three nodes were installed in Augmentation Research Center at Stanford Research Institute, University of California at Santa Barbara (UCSB) and University of Utah School of Computing.

In the 80s, thanks to the advances in the production of electronic components, computers became small and inexpensive enough to be purchased by individuals, inaugurating the personal computer era and raising the interest in internetworking. In the late 90s, with the birth of the Internet and the possibility of having a unique global network, new computing paradigms has been adopted. At this regard, the following Sections will briefly introduce some key concepts.

1.2 Distributed systems

In the literature there have been several attempts to define a distributed system. Tanenbaum and Van Steen [6] have provided the following definition:

Definition 1.2.1. *A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.*

From that definition we can infer that the most important characteristic of a distributed system is transparency, both the users and applications should perceive the distributed as a unique system rather than as a collection of cooperating components. In addition to transparency, a well-designed distributed system has to meet the following requirements:

- *Accessibility*: resources must be accessed and shared easily and efficiently;
- *Openness*: components can be used by or integrated into other systems;
- *Scalability*: ability to adapt the size according to the number of resources or users, as well as in term of geographic and administrative span;
- *Reliability*: the system must be designed in such a way that it is available all the time even after some component fails.

1.2.1 Architecture and middleware

The architecture of a distributed systems is very complex due to the nature of the devices (called nodes) that compose them. Distributed systems may count up to millions of heterogeneous devices that could be geographically dispersed and be interconnected using different network technologies, both wired and wireless.

Moreover, distributed systems are often highly dynamic: the number of devices, the topology and performance of the underlying network can change over time.

Those architecture and nodes characteristics, combined with the transparency requirement, make complex the development of distributed applications capable of dealing with such heterogeneous systems. In order to simplify the development of distributed applications, the devices that are part of the distributed systems are often organized to have a separate layer of software, called middleware, that is logically placed on top of the respective operating systems. This layer enables interoperability between applications by supplying services to exchange data.

1.2.2 Types of distributed systems

Among the different topologies of distributed systems we will only focus on grid computing and pervasive systems since are related to our topic.

Grid computing systems, unlike other high performance distributed systems, integrate heterogeneous devices, typically servers, storage facilities and databases, from different organizations into a federation of systems creating a form of virtual organization. Given this characteristic, the main focus during the development of a grid system is to find the correct software architecture that is able to provide access to resources belonging to different domains and only for users of a specific virtual organization. An architecture that still form the basis of many grid computing systems consists of the following four layers [6]. At the bottom, we find the *fabric layer* that provides interfaces to local resources at a specific site, typically providing functions for inquiring resources states and capabilities. Above that, there are the *connectivity and resource layers*. The former consists of communication protocols for supporting secure communications between resources and between these and users, while the latter is responsible for managing a single resource. The next layer is the *collective layer* that manages the access to multiple resources. This layer typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources. Lastly, on top of the architecture, the *application layer* consists of the applications that works with distributed systems.

Unlike grid computing systems, which are characterized by nodes that are fixed and have a more or less permanent connection to a network, *pervasive systems*, instead, should encompass every device worldwide that has any type of resource, many of which are characterized by being small, battery-powered, mobile, and wireless connected. An important feature is that a pervasive system is intended to be part of the surrounding with limited administrative control by

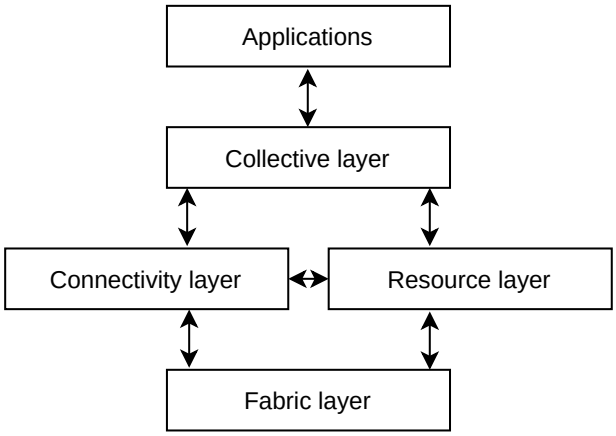

Figure 1.1: *Grid computing architecture*

human. Devices can be configured by their owners but they have to be able to automatically integrate with the environment in which they are located discovering services and reacting accordingly.

1.3 Cloud computing

As mentioned in Section 1.1, Cloud computing is a concept that has evolved through a number of phases that cover about fifty years. The first time that a concept amenable to Cloud computing came to light was in the 1961 when, during MIT centennial ceremony, John McCarthy said [7]:

"Computing may someday be organized as a public utility just as the telephone system is a public utility. Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system. Certain subscribers might offer service to other subscribers. The computer utility could become the basis of a new and important industry".

Only after decades, with the birth of the World Wide Web we finally had computers and network infrastructures able to realize "computing utility delivered as a public utility". From 2000, the growth of the Cloud services was exponential and a lot of companies started to provide any sort of service through the web, reaching in around ten years a market value of more than 200 billions USD.

In 2012 the National Institute of Standards and Technology (NIST) gave a formal definition of Cloud computing [8]. They defined it 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 manage-

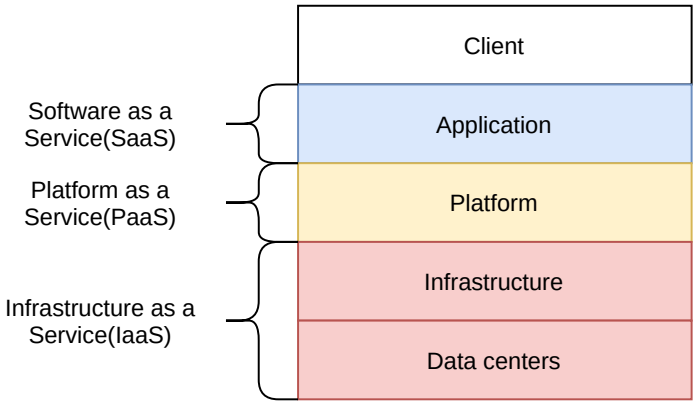

Figure 1.2: *Cloud Layers*

ment effort or service provider interaction.". We can clearly see that the definition provided by NIST is very similar to John McCarty's concept.

1.3.1 Cloud computing architecture

The standard Cloud computing architecture is formed by four layers: [9]:

- *Data centers layer*: this layer provides the hardware on which the Cloud runs. Modern data centers usually consist of thousands of inter-connected servers;
- *Infrastructure layer*: this layer is built on top of hardware layer and it has the task of virtualizing computing power, storage and network connectivity of the data centers, and offers it as provisioned services to consumers. Users can dynamically scale up and down these computing resources on demand;
- *Platform layer*: it provides a development platform with a set of services to assist application design, development, testing, deployment, monitoring and hosting on the Cloud. Google App Engine and Microsoft Azure are examples of this layer;
- *Application layer*: software is presented to the end users as services on demand, usually in a browser saving the users from the troubles of software deployment and maintenance. The software is often shared by multiple users and automatically updated from the Clouds;
- *Client Layer*: it consists of computer hardware or software used to the user to access a particular Cloud service through the internet network.

The dividing lines for the four layers are not distinctive. Components and features of one layer can also be considered to be in another layer. Moreover, even if the Figure 1.2 suggest a hierarchical relationship, there's no guarantee that the upper layer has to be built on top of its immediate lower layer. For example, a SaaS application can be built directly over IaaS, instead of PaaS.

1.3.2 Benefits and issues

Cloud computing has emerged as a computing infrastructure that enables rapid delivery of computing resources as a utility in a dynamic, scalable and virtualized manner. The advantages of Cloud computing over traditional computing are:

- *Easy Management*: users can access data, applications or any other services with the help of a browser regardless of the device used and the user's location;
- *Cost reduction*: owning a server to run application has direct and indirect costs regardless of the workload. Instead, Cloud-based solutions allow to scale the resources available according to various needs;
- *Green Computing*: harmful emissions due to extensive use of systems in organizations, generated electronic wastes and energy consumption are the main disadvantages of the actual computing systems. This can be reduced to some extent by using Cloud computing services, leading to environment preserving and minimum e-waste generation.

Although the Cloud is widely recognized as a revolutionary IT concept with clear advantages, enterprises can expect to face many trade-offs when they move IT into the Cloud. Main problems are:

- *Security*: in the Cloud, data is stored with a third-party provider and accessed over the internet. This means that visibility and control over data is limited;
- *Interoperability*: in an on-premise model, enterprises control their infrastructure and platforms at any time. In the Cloud, they're locked into a provider and no longer control their own IT. Moreover, due to different technologies adopted by Cloud providers, users can face severe constraints or impossibilities in moving their data from one Cloud provider to another;
- *Performance instability*: Cloud services can suffered from variations in performance and availability due to loads;

- *Latency and network limits*: As applications make ever-more intense use of large volumes of data, data transfer poses an increasing bottleneck preventing to exploit the Cloud services for applications that need to meet strict time requirements.

1.4 Internet of Things

The advancements of micro-electro-mechanical-systems (MEMS) technology, wireless communications, and digital electronics has resulted in the development of miniature devices having the ability to sense, compute, and communicate wirelessly in short distances, opening the doors to the "Internet of Things". Since the birth of the internet there were a lot of attempts to connect any sort of object. In 1999 Kevin Ashton, executive director of the MIT Auto-IDCentre, coined the term Internet of things (IoT) [10]. His thought was driven by the idea that all the data available on the Internet was captured and created by human directly. However, humans limitations in the capability of capturing data, could be overcome by computers that are able to automatically gather data from things and process them, reducing errors, time and costs. Starting from this concept, IoT has been defined as a network of physical objects (i.e., devices of all type and sizes) all connected and communicating between each other, sharing information in order to achieve given objectives.

1.4.1 IoT devices and embedded systems

IoT devices are heterogeneous embedded systems equipped with sensors and network interfaces.

Definition 1.4.1. *An embedded system is the support structure for the functioning of applications with a high level of interaction with the surrounding environment and which behaviour depends on data or signal that comes from the outer world.*

Unlike to general purpose computing systems that are designed to run lot of different applications, embedded systems are dedicated to the execution of a specific task or class of tasks. Consequently, leveraging in-depth knowledge of the application, the hardware can be strongly optimized correctly sizing the resources [11]. Due to the application dependency it is very difficult to find a standard architecture, moreover non-functional requirements (strict time-to-market and production volume for example) can lead to very different solutions even starting from the same functional requirements. Although it is not possible to search for a common architecture, some common features can be highlighted [12]:

- *Physical characteristics*: the physical footprint of the system is often crucial, especially for devices that do not have a fixed location;
- *Energy consumption*: more systems are becoming portable so it is essential that the systems are designed taking into account energy efficiency both as regards hardware and software development;
- *Code size*: embedded systems has often their software stored in the same chip or board. This constraint is reflected, especially for cost and reasons footprint, on the size of the code that must be as contained as possible;
- *Performance*: performance is not a generic goal but it depends on the application. In general, the key aspects are: the reaction time to a event and the event management time by executing the associated code. These constraints will result in architectural solutions aimed at meeting these requirements while maintaining a limited cost;
- *Real-time requirements*: unlike general purpose solutions, embedded systems are often real-time systems. This means that the whole system, or parts of it, has to start or complete some of the operations within a fixed and precise time. Based on the consequences of non-compliance with the constraints, we identify *soft-real time*, where the failure to satisfy time constraints leads to a degradation of system performance, and *hard real-time* systems, where the failure to satisfy time constraints leads to catastrophic effects on the system and/or on the surrounding environment;
- *Reliability*: an accurate analysis of potential failures must be conducted. Reliability can be a constraint in itself;
- *Safety*: it indicates a measure of the possibility that in the event of a fault, the system does not cause serious consequences for people or things with which it interacts with;
- *Security*: is the capacity of a system to protect and verify information authenticity;
- *Price*: it is a determining factor especially for the large volume productions;
- *Flexibility and time-to-market*: the methodologies and technologies chosen for the project must be chosen so allow you to get to the product within strict time, so as to seize the maximum market opportunities.

1.4.2 IoT architecture

There is no single consensus on architecture for IoT and during the years, due to enhancement in the field and the arise of new challenges multiple architectures have been proposed [13] and shown in Figure 1.3.

The first proposed IoT architecture consisted of three layers [14, 15]:

- *Perception* layer: it is responsible for perceiving the physical properties of things using sensors. Data collected by sensors is pre-processed in order to be transmitted;
- *Network layer*: it is responsible for processing the received data from the perception layer and transmitting data to the application layer using various network technologies;
- *Application layer*: it uses the processed data to deliver application specific services to the user.

This architecture defines the main idea of the Internet of Things but is not sufficient to detail finer aspects of IoT. For this reasons four-layers [16] and five-layers [17] architectures were proposed. The four-layers architectures adds a support layers between network and application layers. This layer consists of services and actions related to the control, security and management of the application. Differently, the five-layers architectures maintains only perception and application layers and introduces the processing, transport and business layers. The perception and application layers maintain the same roles of the previous architectures. The transport layer has the task of transferring the sensor data from the perception layer to the processing layer and vice versa using wired and/or wireless networks. The processing layer stores, analyzes and processes the huge amounts of data that comes from the underlying layer. Lastly, the business layer manages and controls applications, business and profits model of IoT.

1.5 Fog computing

Although the success of IoT has witnessed the possibilities and potentials of that technology, we have to face some issues. In fact, it is estimated that there will be more than 65 billion IoT devices by the year 2025. The problem with this growth is that IoT devices are able to generate enormous amount of data that are mainly transfer to Cloud for processing, risking to saturate the capability of the current network infrastructure. Another problem is that current IoT infrastructure, relying on internet connection, is not suitable for real-time applications. As

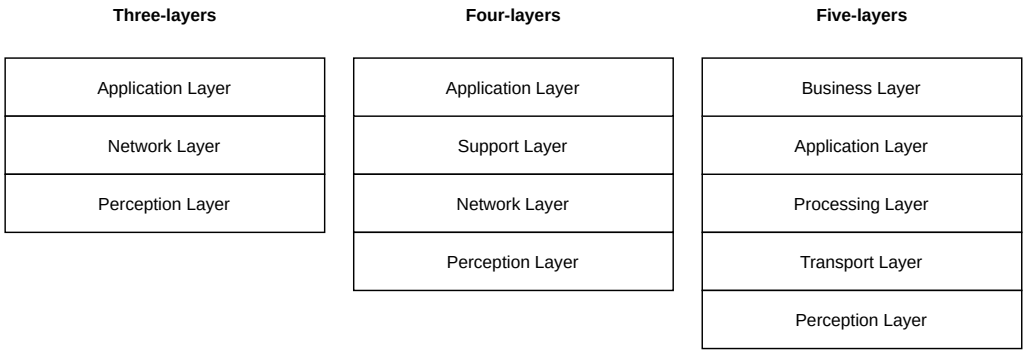

Figure 1.3: *IoT layers*

a solution to these problems, in 2015 Cisco proposed the concept of *Fog computing*, which consists in an additional layer between the IoT/Edge frontier of the network and the remote Cloud. Its objectives is to bring Cloud services closer to the IoT devices that generate the data. Thus, decongesting the network and decreasing latency due to the fact that Fog devices locally perform all or part of the pre-processing workload and send only the useful data to Cloud services for further processing. This computing model has the following advantages:

- *Low latency*: the close location of Fog nodes to the source of data and the processing capabilities of Fog nodes enables low latency and opens the doors to real time applications;
- *Avoid network saturation*: performing data processing and data storing at node level significantly reduces data movement across the Internet due to the fact that only useful data is forwarded to the Cloud.
- *Security*: Fog nodes provide access control policy, encryption, integrity check and isolation measures;
- *Low energy consumption*: using short range communication mode and optimal energy management policies, Fog nodes are able to reduce their power consumption leading to a decrease in cost.

1.5.1 Architecture of Fog computing

The reference model of Fog computing architecture is a significant research topic and many architectures have been proposed through the years mostly derived from a three-layer structure. In particular, we consider a multi-dimensional architecture, based on the OpenFog reference architecture [18], that has been presented by Zanella et al. [19]. According to this work, it is possible to consider two dimensions on the resource space.

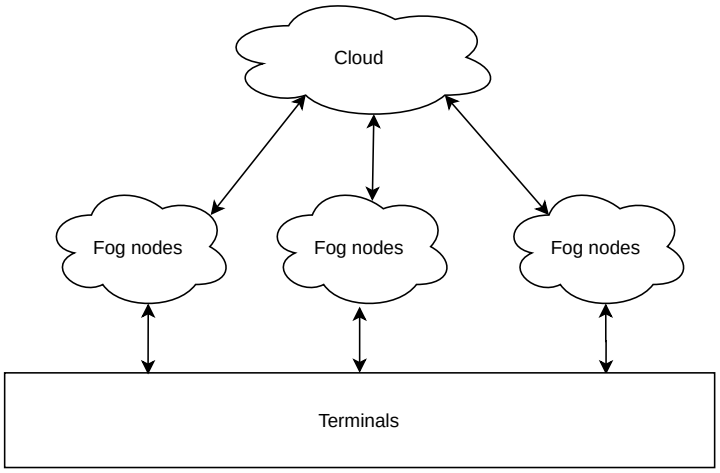

Figure 1.4: *Fog hierarchical architecture*

Through the *vertical* dimension it is possible to interconnect devices with high different paradigms, performance, energy efficiency characteristics and geographical proximity. At the bottom, the *Edge layer* consists of various IoT devices geographically distributed. These devices are responsible for sensing data from the environment and transmitting the data gathered to the upper layers. In the middle, the *Fog layer* is composed of a large number of devices spanning from gateways to high-end embedded, desktops, micro-server or even mobile devices. On top of that, there is the *Cloud layer*, which consists of multiple high-performance servers and storage devices and it provides various application services

The *horizontal* dimension, instead, introduces the possibility to distribute and balance the workloads on nearby or federated devices in the same layers in order increase scalability and redundancy.

1.5.2 Fog nodes characteristics

The Fog nodes are the core components of the Fog computing architecture. Fog nodes, as seen for embedded systems and IoT devices, can be devices of any sort, both physical and virtual, and have different capabilities in terms of processing power and storage. As mentioned before, Fog nodes process and store data generated by sensors and Edge devices. To deploy a given Fog computing capability, Fog nodes operate in a centralized or decentralized manner. In fact, they can work as stand-alone nodes that communicate among them to deliver the service or can be federated to form clusters that provide horizontal scalability over disperse locations. Fog nodes are strongly characterized by heterogeneity

in terms of both connectivity and resources within the Fog level and compared to Cloud and IoT devices. Regarding nodes communications, inside and between the levels, nodes use various technologies, both wired and wireless. Fog nodes are connected with end devices and users mainly by a wireless connection such as 4G/5G, Bluetooth, or WiFi. Otherwise, Fog nodes can also be connected with the Cloud using a classic wired connection to exploit Cloud services.

For what concerns resources, we can identify three types of heterogeneity:

1. *Inter-level heterogeinity*: we can have computing devices with very different capabilities in different levels;
2. *Intra-level heterogeinity*: we can have computing devices with very different capabilities at the same level;
3. *Intra-node heterogenity*: we can have different type of resources available in the same node.

1.5.3 Challenges

As any other new technology, Fog computing, in addition to the aforementioned advantages, brings some problems that must be addressed and solved in order to take full advantage from this new computing paradigm. The first aspect to consider is that Fog computing devices may face serious security problems due to the fact that these devices, being geographically dispersed, could be deployed in places out of strict surveillance becoming potentially vulnerable to any sort of malicious attacks. Solutions to similar problems for Cloud computing have been found, but they don't necessarily work for Fog computing.

Although the security aspect is very important, the biggest challenge of Fog computing is the *control and management* of such a distributed and heterogeneous system. In particular, in order to fully exploit the workload distribution features and the heterogeneity advantages of the aforementioned architecture, the first step is the development of a *suitable programming models* that allow the decomposition of applications into execution units that can exchange data among them. Secondly, there is the need of a *run-time management system* that is able to: (a) efficiently discover and organize nodes and (b) map application execution units to the available resources in order to meet the application requirements. At this regard, *proper task mapping and resource allocation strategies* have to consider optimizations at both distributed system (e.g., performance, balancing, energy consumption, reliability...) and device level (e.g., load, utilization, energy budgeting, temperature...). At experimental stand point, there are many synthetic benchmarks and simulation software. However, in the research field

there is an high demand also for *real-world use-case application* that can be used in experimental settings, as well as *accessible and physical hardware test-bed* to perform measurements, improving simulation models.

1.5.4 Thesis structure and contribution

The work described in this thesis has the purpose of presenting the design and implementation of a real-world multi-task application and its porting to a full-stack Fog management and deployment framework. The focus of our work is to be able to distribute the workload of the application according to the characteristics and availability of heterogeneous devices and to the application and tasks requirements. For these purposes, we will leverage the *BarMan* framework and propose two different task-allocation policies.

The thesis is organized as follows. In Chapter 2 the state-of-the-art related to Fog programming frameworks and the approaches to dealing with the resource management problem are presented. The Chapter 3 presents the *BarMan* framework, used in our work to develop a Fog distributed application. Then, in Chapter 4 we analyze a Fog computing use-case and then present its implementation. In Chapter 5.1, we present two tasks allocation policies, *LAVA* and *LAVAnet*, for low-latency applications. Then, in Chapter 6 we will analyze the performances of the application and the allocation policies on a real test-bed Fog cluster made with heterogeneous embedded boards. Finally, Chapter 7 concludes the thesis and highlights the future improvements.

CHAPTER 2

State of the Art

This chapter introduces the state of the art for related to the main challenges in Fog computing field mentioned in Chapter 1: suitable programming models and resource management. In Section 2.1, we analyze some of the programming and management frameworks that has been proposed, considering both academic and commercial solutions. Instead, in Section 2.2 we present the main problems in the applications and resource management fields and some of the approaches used to deal with them.

2.1 Programming and Management Frameworks

As stated in Chapter 1, one of the biggest challenge in Fog computing is the development of a suitable programming model that allows to fully exploit the overall architecture. In the literature, research focused on the architectural aspects of Fog systems [20]. However, not many solutions presents complete and available frameworks to implement such architectures. In this Section, we will analyze some of the state-of-the-art solutions basing on the following characteristics, as reported in Table 2.1:

- *Scalability*: handling a large number of applications and IoT devices at the Fog layer while satisfying each application requirements needs to scale ser-

vices elastically. Thus, Fog managers should provide modules to support scalability;

- *Device mobility support*: Fog services may be hosted on mobile devices that are connected to the network through unreliable wireless connection and often they could change their points of attachment to the network. Therefore, mobility support is critical and management systems have to provide modules to solve this issue;
- *Resource heterogeneity support*: Fog devices are highly heterogeneous, a programming framework should provide tools for designing services and tasks according to the different capacities of the devices. Moreover, it needs to be able to decide which application components should be deployed on the different nodes transparently with respect to the user;
- *Providing energy awareness*: Fog services may be hosted on battery-powered mobile devices, it is essential that a management framework offers the possibility to manage applications taking care of power consumption;
- *Quality-of-Services management*: each application has QoS requirements, therefore Fog programming frameworks should provide mechanisms to support that;
- *Openness*: Fog programming framework should be standardized and open in order to guarantee the interoperability among different systems.

2.1.1 MobileFog

MobileFog was developed by Hong et al. [21] in 2013 with the purpose of providing an high–level programming model that could simplify the development of Fog applications and allow dynamic scaling of the resources allocated to the applications. In this work, the authors consider the network as formed by heterogeneous physical devices, called Fog computing nodes, placed at different layers of the network hierarchy and associated with a certain geophysical location. Moreover, they assume that the Fog system provides a programming interface to create and terminate computing instances, each one characterized by certain system resource capabilities that could be used to execute application code.

In the proposed programming model, application is formed by *MobileFog processes* that are mapped onto distributed computing instances in the Fog and Cloud. Each process runs application code and performs specific tasks with respect to its geographical location and level in the network hierarchy and can communicate with other processes running on different devices using both point-to-

point and hierarchical communication API. Moreover, each process has a unique identifier called *appkey* that can be used by developer to manage the application through the management interfaces.

To exploit Fog devices hetereogeneity, MobileFog allows to run the same application code on various devices available in the system. Moreover, it is possible to dynamically scale resources depending on application's workload. At this regard, users have to provide a scaling policy that consists of a set of metrics (e.g., CPU and RAM utilization) and conditions that triggers the dynamic scaling.

The approach presented has some drawbacks. First of all, due to the lack of a QoS management module, the framework does not allows to define and monitor application requirements and, thus, configure the application's parameters accordingly. This, together with the limited metrics that can be monitored in the scaling policies, does not allow the creation of energy-aware applications. Finally, *MobileFog*, is not open source nor standardized, so it has a limited interoperability with devices from different systems.

2.1.2 Distributed DataFlow

Giang et al. [22] proposed the Distributed Data Flow (DDF), a distributed version of the well-known *Dataflow* [23].

Dataflow is a well-known programming paradigm born for parallel execution of tasks using multiprocessors, in which, unlike traditional procedural programming that focuses on commands, the focus is the movement of data and programs are seen as a directed graph where each node can have inputs, outputs and runs as an independent processing units, as shown in Figure 2.1. With DDF the Dataflow program is deployed on multiple physical devices. Each device executes one or more node in the flow, forming sub-flows able to communicate with each other.

According to the authors, this approach offers some advantages. First of all, it raise the abstraction level of the underlying IoT hardware simplifying the application development. Secondly, it allows IoT applications to exploit Fog vertical heterogeneity by differentiating devices on their computing resource, while constraining the deployment of nodes in the flow considering only devices with appropriate resources. Moreover, it is possible to strategically deploy nodes of the flow to Edge servers and physical devices in order to meet the QoS requirements of the applications. Moreover, by replicating a node in the flow on more than one physical device, the system is even able to handle the movement of devices through the network at the cost of using extra resources. Furthermore, due to the fact that a DDF applications does not rely on a centralised management system, scaling is supported. Finally, DDF is open and standardized thus it

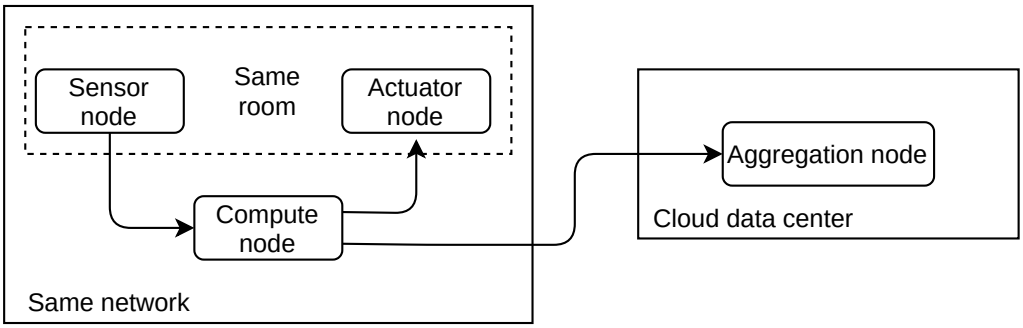

Figure 2.1: *Sample DDF application*

guarantees the interoperability among different Fog devices.

However, this solution does not take into account energy management and power issues of Fog devices, limiting the possibility to develop energy-aware applications. Moreover, it does not consider the possibility to configure at run-time application's specific parameters.

2.1.3 FogFlow

In 2018 Cheng et al. [24] proposed a new computing framework called *FogFlow* with the aim of creating a standard-based programming model that would simplify the development of applications for IoT smart city platforms over Fog and would increase interoperability between heterogeneous devices.

The FogFlow framework extends the Dataflow programming model with declarative hints with Next Generation Service Interface (NGSI). IoT services are represented by a service topology which consists of multiple operators, that receives certain input streams, performs data processing, and then publishes the generated results as output streams. The framework operates on Fog heterogeneous resources with three logically separated divisions: service management, data processing and context management.

The *service management* division, typically deployed in the Cloud, includes task designer, topology master, and docker image repository. The task designer provides the interfaces to manage all deployed IoT services, as well as to design and submit new services. A Docker image repository manages the docker images of all dockerized operators. While the topology master (TM) is responsible for service orchestration, i.e. it translates a service requirement and the processing topology into a concrete task deployment plan.

The *data processing* division is formed by set of workers that perform data processing tasks assigned by the topology master. Each worker is associated with a computation resource in the cloud or on an edge node and can launch multiple

tasks. The number of supported tasks is limited by the computation capability of the compute node. The internal communication between TM and the workers rely on message queuing protocol-based message bus to achieve high throughput and low latency.

Finally, the *context management* division is usually deployed in the Cloud and includes a set of IoT Brokers, a centralized IoT Discovery and a Federated Broker. These modules decide the data flow across tasks using NGSI and manage the system contextual data, such as the available resources of devices.

The proposed framework supports heterogeneity and QoS through the processing requirements submitted to the topology master. Openness is guaranteed by the use of the open programming model Dataflow and the NGSI standard. Moreover, the introduction of a distributed context management approach allows to support scalability. The only flaws of the proposed solution are the lack of modules to support device mobility and the impossibility to create energy-aware strategies. Furthermore, the framework does not allow to configure application's parameters at run-time.

2.1.4 Commercial Solutions

Some companies have independently propose their Fog computing platforms extending Cloud platform to support edge services. The first company to work in this area was Cisco. After they proposed the Fog computing concept in 2012, they started to work on their framework and release some products, such as Cisco Fog Data Services, Cisco Fog Director and Cisco Fog Fabric, that allow to partially exploit Fog capabilities using only Cisco devices.

In 2017 Amazon released *AWS IoT Greengrass* to extend Cloud capabilities to local devices. Their goals were: (a)enabling devices to collect and analyze data closer to the source information and (b) bringing a development technology that can be used both on Cloud and Iot devices. With Greengrass, users can develop the code in the Cloud and deploy it directly to the IoT devices using AWS Lambda. The most important module of their framework is AWS IoT Greengrass Core that, acting as a communication hub between the Cloud and devices that run Amazon FreeRTOS or AWS IoT Device SDK, enables local execution of AWS Lambda code, messaging, data caching, and security. Despite the easy of use of their solution, it guarantees only support for scalability and QoS management.

Lastly, Microsoft proposed *Azure IoT Edge* a complete edge management service built on Azure IoT Hub. Their solution enables the deployment of services on IoT Edge devices through standard containers. *Azure IoT Edge* is made

	Scalability	Device Mobility	Resource Heterogeneity	Energy Awareness	QoS Management	Openness	App run-time configuration
MobileFog	✓	✓	✓				
Distribuited Dataflow	✓	✓	✓		✓	✓	
FogFlow	✓		✓		✓	✓	
AWS Greengrass	✓				✓		
Azure IoT Edge	✓				✓	✓	
Our Solution	✓	✓	✓	✓	✓	✓	✓

Table 2.1: *Summary of Fog management frameworks*

up of three components: IoT Edge Modules, IoT Edge runtime and a Cloud-based interface. The IoT Edge modules are units of execution, implemented as Docker compatible containers, that run Azure services, users' code or third-party services. Multiple modules can be configured to communicate with each other, creating a pipeline of data processing. The IoT Edge runtime is located on the IoT Edge device and performs management and communication operations. Finally, the Cloud-based interface allows the user to monitor and manage all the IoT Edge devices remotely. Azure IoT Edge is open and guarantees support for scalability and QoS management.

2.2 Applications and Resources Management

Due to the dynamicity, heterogeneity and uncertainty of Fog environment, it is essential to have a robust resource management system. As pointed out by M. Ghobaei-Arani et al. [25], the resource management issue cannot be considered as a single problem but as formed by multiple sub-problems. In our work we will split the resource management approaches into the following categories:

- Application placement
- Resource scheduling
- Resource provisioning
- Resource allocation
- Task offloading

We will analyze each of the aforementioned problems and present some of the approaches proposed in the literature.

2.2.1 Application placement

The application placement problem consists in defining an optimal placement plan between IoT services and Fog nodes to maximize Fog resource utilization

while meeting application requirements. A correct deployment is crucial because a not optimal placement plan could result in non-compliance with the application requirements, or cause malfunctions to other Fog services. For example, an incoherent application deployment solution for a big volume of data may cause network congestion and affect hard real-time services.

Normally, application placement involves one or more management nodes that, after searching for all the resources available that satisfy application requirements, try to optimize the application placement. The approaches to solve this problem can be divided into three categories: centralized, decentralized and hierarchical.

The *centralized* approach requires the existence of a central node whose job is to make global optimization decisions on the placement of applications and services based on the available resources. To do that, the central node needs information from all the entities available in the Fog layer. This approach, having one node responsible for decisions, is more vulnerable to failures than the others. Moreover, due to the need of receiving data from Fog nodes, the network overhead becomes problematic as the number of fog nodes increases, resulting in a limit on scalability.

The *decentralized* approach, instead of using one central node to take global optimization decision, make use of multiple nodes each of which takes local optimization decisions. This approach adds communication overhead between management cores but it solves the scalability and reliability problems of the centralized one. Finally, the *hierarchical* solution combines the previous two approaches in order to take their advantage. In addition to the architectural differences, the solutions proposed in the literature differ from each other in the metrics taken into consideration while searching for the optimal placement. Most of the studies consider the delay network metric while the energy consumption metric is less considered.

Skarlat, Nardelli, Schulte et al. [26] proposed a hierarchical approach based on a genetic algorithm to find a suitable service placement solution into Fog landscape. Their approach take into account Fog devices capabilities, cost of the solution and latency. Brogi et al. [27], instead, proposed a centralized approach QoS-aware deployment approach able to find eligible deployments of IoT applications over a Fog infrastructure in a context-aware manner considering latency and bandwidth usage during optimization. Both the approaches could be not optimal if Fog devices are battery powered or in scenarios where there are constrains on device's power/energy availability. At this regard, Taneja et al. [28] proposed a hierarchical heuristic-based approach that uses a low computational

complexity algorithm that considers as performance metrics network usage, application latency and energy consumption (allowing energy-aware placement). Finally, Selimi et al. [29] proposed a decentralized approach based on a fast service placement heuristic algorithm, called BAST, for community network micro-cloud infrastructures. Their approach allocates services taking into account the bandwidth of the network and the node availability executing the algorithm every single time a new service deployment is about to be made. However, latency is not evaluated during the placement evaluation, potentially leading to a not optimal placement for real-time Fog application.

2.2.2 Resource scheduling

Services requested from IoT devices could be divided into a set of tasks and served by several Fog nodes. The resource scheduling problem determines an optimal assignment of different tasks to be executed on the Fog nodes in order to meet the IoT application requirements. This problem is an NP-hard optimization problem. Generally, resource scheduling is solved by using meta-heuristic algorithms to find feasible and near-optimal solutions in linear time. Between the various approaches presented in the literature, we can identify three main class: *static*, *dynamic* and *hybrid* scheduling.

In *static* scheduling approaches, tasks arrive concurrently at the Fog nodes and scheduling decisions are made before tasks are submitted. So, they need to know in advance all the necessary information about received demands and available resources before scheduling. Clearly, in heterogeneous and dynamic systems like Fog, it is not always possible to have all necessary knowledge before resource scheduling. Thus, static approaches can be limited in achieving optimal scheduling and in supporting scalability.

In *dynamic* scheduling approaches the arrival times of the tasks are not known before submission so task scheduling is done when they arrive in the system.

Hybrid scheduling approaches mix both dynamic and static criteria to cover different types of applications.

Most of the research studies proposed dynamic approaches that aim to minimize latency and response time while increasing dynamic efficiency. For example, Sun et al. [30] proposed a two-level dynamic resource scheduling algorithm. Their approach perform the resource scheduling a first time among various Fog clusters to determine the cluster designated to perform the task when it arrives and a second time among Fog nodes in the same cluster to specify the node on which execute the task. The scheduling is made in order to reduce the service latency and improve the stability of the task execution. Deng et al. [31] proposed a

dynamic approximate solution considering the energy consumption-delay trade-off into Fog computing, the energy consumption-delay trade-off into Cloud computing and the minimization of communication delay. Rodopoulus et al. [32] and Massari et al. [33], within the *HARPA* project, proposed the HARPA-OS, a software run-time resource manage able to manage the system resource allocation while taking into account both the status of the system resources and the application requirements. A peculiarity of the proposed system is that the resource management is done combining both pro-active and reactive strategies in order to always allocate the minimum amount of resources that satisfy the application requirements. The allocation is performed collecting the deviation in percentage between the current monitored performance and the expected one [33] for each application. Then, the system estimates the desired performance value and searches for the allocation that allows to obtain the closest performance value possible.

Due to the fact that scalability is a very important factor in the resource management, few static solutions were proposed. Pham et al. [34] presented a two-phases static heuristic-based algorithm with the aim of providing a good tradeoff between the makespan and the cost of task execution. The first phase of the algorithm consist of determining the task priority and the second phase of selecting the most appropriate node based on the *Earliest Start Time* and *Earliest Finish Time*. This approach is computational expensive and limits system scalability.

2.2.3 Resource provisioning

IoT services workload changes over time. This may result in over-provisioning or under-provisioning problems. In under-provisioning problem the resources allocated for a IoT service are less than the ones required by the service to meet its performance requirement. Conversely, in over-provisioning problem the resources allocated for a IoT service are more than the ones needed. While the second problem has the only drawback of incurring into an unnecessary cost, the first problem could lead to application or system level malfunctions. Therefore, it is important to dynamically provide the appropriate number of Fog nodes required to handle the application's workload in order to reduce system cost while meeting the QoS constraints. To solve this problems we need to use dynamic resource provisioning to scale-in/out Fog nodes according to the incoming demand. The resource provisioning approaches autonomously manage resources allocation and release in order to satisfy the IoT devices resource demand and to avoid resource wastage. The resource provisioning approaches can be classified into three policy: reactive, proactive, and hybrid.

The reactive policy does not perform any kind of prediction on system state, just respond only when workload change has already happened.

The proactive policy utilizes prediction techniques, such as time series and neural network, to estimate future demands of IoT applications and react with enough anticipation.

In hybrid policies both policy are used. Typically, the reactive policy is utilized for scaling out decisions and the proactive policy for scaling in decisions.

The various resource provisioning approaches presented in the literature do not consider all the QoS parameters simultaneously. Some approaches focus on latency, delay and cost while others focus on throughput, CPU utilization and energy consumption. Dos Santos et al. [35] proposed a resource provisioning approach based on Integer Linear Programming to optimize multiple objectives, such as latency, service migrations and energy consumption. Instead, Zanni et al. [36] proposed a dynamic scaling solution using geometric monitoring to reduce latency.

2.2.4 Resource allocation

The resource allocation problem consist of efficiently allocate a set of geographically distributed and heterogeneous Fog nodes to execute IoT services with different QoS requirements. Resource allocation approaches can be divided into two classes: auction-based and optimization-based.

The *auction-based* resource allocation methods are market-driven approaches where IoT devices applied their needs for Fog nodes with bids, then, according to a specified auction mechanism, nodes are assigned to the best bidder. These methods are evaluated using formal Petri nets, approximate algorithm, and game theory to decrease the cost and time factors based on user requests. For example, Jiao et al. [37] proposed an auction-based model using an approximate algorithm for cost-effective resource allocation to allow the service providers to make practical and efficient computing resource trading mechanisms on the blockchain network. Moreover, Nguyen et al. [38] proposed an approach based on the market equilibrium technique for resource allocation.

In *optimization-based* methods, the resource allocation is modelled as a double matching problem where Cloud servers and Fog nodes are coupled to IoT devices while Fog nodes and IoT devices are coupled to Cloud servers. Anglano et al. [39] presented an efficient profit maximization method using an approximation algorithm. The aim of the approach is to increase the profit of edge providers despite of the workload fluctuations while the applications requirements are satisfied.

2.2.5 Task offloading

IoT and mobile devices are often resource-limited and it could become necessary to outsource some of their resource-intensive tasks to the Fog or Cloud for improving their performance and/or saving battery [40–42]. When the other entity has executed the tasks, it has to return the result to the original IoT device. This process is called task offloading and it can occur for different reasons such as load balancing, data management, energy efficiency and so on. The task offloading approaches based on the number of offloading destinations can be divided into two main categories:*single* and *multiple*.

In *single-type* offloading approaches, computation tasks can be offloaded to only one Fog node for sequential processing. In *multi-type* offloading approaches instead, the tasks can be offloaded to more than one destination. This type of approaches is more suitable for IoT applications with repetitive processing and parallelized parts.

Most of the research studies proposed approaches that investigate the application energy consumption and latency. For example, Tran et al. [43] proposed a service that allows to offload computing intensive task to Fog nodes with free resources, naming it "Offload as a Service" (OaaS). The algorithm is able to compute the energy utilized by each Fog node and select one of them to offload computing dynamically.

2.3 Our solution

Basing on the aforementioned works, our solution involves the use of the Barman framework to analyze and implement a video surveillance use-case application in a Fog computing scenario. A key feature of the proposed use-case real-world application is the possibility of being executed on *real hardware* for experimental purposes, being easy to inspect and measure, and provides heterogeneity of the tasks' workload performed: i.e., execution of Neural networks, image processing, simple mathematical calculation, GUI management.

As summarized in Table 2.1, our work enables the exploitation of the framework to pursue the following goals:

- having a *resource-aware task-based programming model* that allows us to easily divide applications into tasks and that is deeply integrated with a *complex and complete resource manager*;

- making the *application run-time configurable* bu the resource manager based on the available and assigned resources;

- *deploying the application's module* to different devices of the system transparently with respect to the user;
- managing a *distributed system* that is variable both in the number of the devices and in their availability over time, thus supporting scalability and mobility;
- performing *low-level management of computational resources*, including heterogeneous ones;
- considering *QoS requirements* of an application and also allowing to develop dedicated *allocation strategies* on the metrics of interest to the type of applications, addressing both QoS-Management and openness, as well as enabling the possibility to consider future energy-aware tasks allocation.

In the implementation of the use-case application we will consider a centralized execution of the resource manager. This choice is based on two reasons. First of all, we want to start from the exploration of a single aspect, in order to investigate the limits and benefits of such approach. Secondly, the hardware characteristics of Fog nodes do not always allow to run a resource manager on multiple nodes. Despite our decision, in case of multiple nodes compatible with the resource manager, it is possible to have multiple instances of the resource manager collaborating in the resource allocation decision.

Regarding the allocation strategies, we will present an initial strategy that takes into consideration the QoS of the application with information at run-time such as the load of the device and its connectivity.

The structure of the framework, the process of the design and implementation of both the application, the allocation policies and the analysis of the execution results on real hardware will be presented in detail in the following chapters.

CHAPTER 3

The BarMan Framework Overview

In this chapter we will introduce the framework used in this thesis to develop a Fog distributed application. Starting from Section 3.1, we will briefly present an overview of the MANGO project. Through Sections 3.2 and 3.3 we will analyze the main components of the framework: the *BarbequeRTRM* and the `libmango`. Finally, in Section 3.4 we will cover the *BeeR* extension.

3.1 The MANGO Project

High-Performance Computing (HPC) is quickly evolving and new application requirements, such as power efficiency and QoS support, are emerging. At this regard, the MANGO project [44] aims to extend the traditional optimization space to *power*, *performance* and *predictability* in order to keep up with the aforementioned requirements. The project investigates the architectural implications of the emerging constraints and aims at the definition of a new generation of high-performance, power-efficient, deeply heterogeneous architectures that are able to comply with QoS. To achieve resource efficiency, when dealing with QoS-sensitive applications, resources cannot be managed by the application developer. In fact, this would lead to a limitation on the ability of the system to balance resource usage. To this purpose, MANGO provides a hierarchical re-

source management strategy made of a global resource manager (GRM) and a local resource manager (LRM). The former is in charge of workload balancing and thermal control of the system, while the latter, in charge of the allocation of node resources, allowing multiple applications to share resources located on a single node. To achieve those results a software stack including a new programming model for heterogeneous multi-processor systems and a run-time resource management solution were developed. Finally, the heterogeneous context of the MANGO project leverages on a multiprocessor CPU-based system to which are interconnected boards containing a set of different accelerators and memory nodes. In the following sections we will present the different modules of the overall framework, since they have been extended to operate also in a distributed Fog context, laying a base for our work.

3.2 The BarbequeRTRM

The *Barbeque Run-Time Resource Manager* (BarbequeRTRM) is a modular and extensible run-time resource manager developed by Barbeque Open Source Project team [45] at *Politecnico di Milano*. The aim of this framework is to transparently manage the allocation of computing resources to multiple concurrent applications while taking care of both applications QoS requirements and dynamic resource availability. Moreover, the framework has been designed to be highly modular and portable. Currently, the *BarbequeRTRM* supports several types of hardware platforms (e.g., HPC, embedded, mobile...), both homogeneous and heterogeneous architecture, thanks to the *Platform Proxy* modules which handles the communication with the platform. Finally, BarbequeRTRM offers a distributed and hierarchical control scheme, where each controller is in charge of a specific subsystem. This approach allows to scale better with system complexity.

The principal feature of the framework is the support for application that could reconfigure themselves during the execution. To accomplish this, the applications must integrate an execution model, as detailed in Section 3.2.1, provided by the framework library and that defines a finite set of possible run-time configurations [46]. In particular, two different levels of reconfiguration and configuration information have been identified.

The first level identifies the *Application Working Modes* (AWMs) which define the resource requirements for the achievement of a certain QoS level. Usually, AWMs are provided by the developers through a file, known as *Recipe*, that contains a finite set of configurations that has been identified as optimal at design-time. Thus, AWMs are later assigned to the application by the resource

manager following a selected policy. Over the years, different policies have been proposed. For example:

- *Tempura* [32]: resources assignment aims to minimize power consumption and avoid thermal hotspots;
- *Contrex* [47]: resources assignment is focus on finding optimal configurations that minimize cost, size, weight and power consumption of the system without compromising its safety or overall performance;
- *PerdeTemp* [33]: as well as Tempura, the resource management has the aim of optimizing the resource usages according to the real requirements, saving power consumption and limiting the occurrence of thermal hotspots. Thus, reducing of the aging effects affecting the computing resources and increasing the system reliability;
- *Manga* [48]: the resource assignment is done with the purpose of finding the best allocation of application modules on deeply heterogeneous resources in order to meet application QoS in HPC scenario.

The second level is represented by the set of tunable parameters called *Operating Points*(OPs). These parameters are chosen by the application itself, depending on the assigned AWM, in order to meet the QoS expected by the end user. Normally a single AWM can support multiple OPs.

3.2.1 Abstract Execution Model

Application reconfigurability requires that the execution can be controlled by the resource manager. In particular, it must be able to start, stop, suspend or reconfigure the application.

To accomplish that, the *Application Run-Time Library* (RTlib) provides the *Abstract Execution Model* (AEM), a callback-based API similar to the Android programming model. Each application must integrate the AEM deriving a class from the base class `BbqueEXC` and implementing some callbacks functions. The main ones are the following:

- `OnSetup`: this method is called by the base class after the constructor and should contain application initialization code;
- `OnConfigure`: this method is called every time the BarbequeRTRM assigns a new AWM and should contain the code related to application reconfiguration;

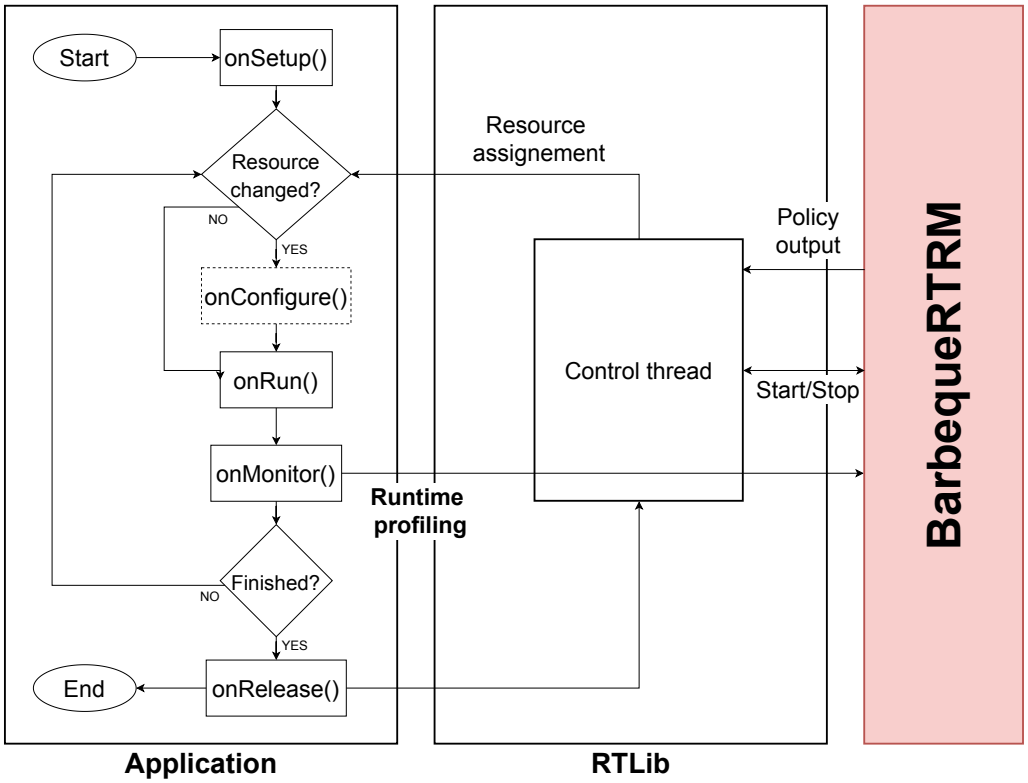

Figure 3.1: *The RTLib Abstract Execution Model*

- `OnRun`: it is the main method of the class and contains all the code to execute the application computational task;
- `OnMonitor`: this method is called after the `onRun` to check if the level of QoS or performance has been achieved. If there are no more data to be processed the `onRelease` method is called, otherwise application calls `onConfigure` or `onRun` depending on whether resources have changed or not;
- `OnRelease`: once the application has terminated, this method is called in order to clear memory allocations, data structures or references.

At run-time, the instance of the class created will spawn a control thread which will call the methods, as shown in Figure 3.1.

3.3 The `libmango` Programming Model

One of the goal of MANGO project is to provide a resource-aware parallel programming model for heterogeneous architectures that allows programmers to easily develop application compatible with different types of nodes. Moreover,

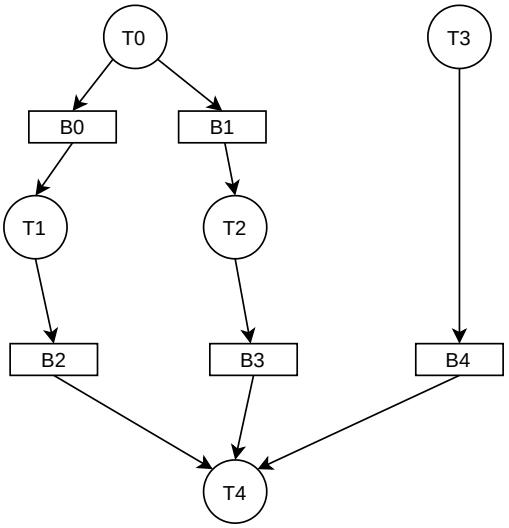

Figure 3.2: *Example of the Task Graph of a complex application*

since applications may be developed by domain experts with limited knowledge of parallel computing, the programming model needs to be simple and hide all the complexities of shared memory management and resources management. To this intent, the `libmango` library has been proposed to achieve the aforementioned challenges [49].

MANGO applications are made up of different tasks, called *kernels*, compiled to be executed on heterogeneous nodes that are characterized by different architecture and specifics. This tasks can exchange data between each others through shared memory. To represent the application and all the dependencies between the tasks, `libmango` uses *Task graphs*, that are directed acyclic graphs where the vertices are the tasks while the directed edges indicates the execution dependencies between operations. In Figure 3.2 we can see an example of an application formed by 5 tasks. Each task is a separate unit of work which is able to pass the result of its execution to one or more dependent tasks. In the figure we represent with rectangles the buffers through which the data exchange takes place.

The creation of kernel and the synchronization is obtained through two abstraction layers: one placed at the host-side and one at the device-side. We will present the two layers while showing two samples from the *Mangolibs* repository: GIF_ANIMATION for the host-side layer and GIF_FIFO for the device-side one.

3.3.1 Host-side layer

The *host-side low-level runtime* (HLR) provides an interface to access the functionality of the accelerators from code running on a *general purpose node* (GN), normally CPU-based. Moreover, the HLR provides functionalities and data structures that can be used to represent and manipulate kernels. The layer is in charge of initialize all the service to allow kernel communication and deployment. Programming wise, the initialization routine is provided by the `BBQContext` that instantiates the Application controller and initializes the communication with the HN library. Once initialized the context, it is possible to create for each application task a `KernelFunction` object that allows to load an executable for a specific target architecture, as shown in Listing 3.1.

Listing 3.1: *Initialization*

```
1  mango_rt = new mango::BBQContext("gif_animation", "gif_animation");
2  auto kf_scale = new mango::KernelFunction();
3  auto kf_copy = new mango::KernelFunction();
4   ...
5  #ifdef GNEMU
6       kf_scale->load( scale_kernel_file_path ,mango::UnitType::GN, mango::FileType::BINARY);
7       kf_copy->load(copy_kernel_file_path , mango::UnitType::GN, mango::FileType::BINARY);
8  #else
9       kf_scale->load( scale_kernel_file_path ,mango::UnitType::PEAK, mango::FileType::BINARY);
10      kf_copy->load(copy_kernel_file_path , mango::UnitType::PEAK, mango::FileType::BINARY);
11 #endif
```

After having initialized the `BBQContext` and the KernelFunction objects, the Task Graph of the application needs to be created, as shown in Listing 3.2. In order to do that, all the buffers and kernels should be registered through the context. Once registered all these elements, it is possible to generate the Task Graph of the application and pass it to the resource allocation command, which is the novelty core of the library. In fact, when the `resource_allocation` function returns, the Task Graph contains all the information about the resource allocation without the intervention of the developer. At this point the application is almost ready to start.

Listing 3.2: *Kernels and buffers registration*

```
1  auto kscale = mango_rt->register_kernel(KSCALE, kf_scale, {B1}, {B2});
2  auto kcopy = mango_rt->register_kernel(KCOPY, kf_copy, {B2}, {B3});
3   ...
4  auto b1 = mango_rt->register_buffer(B1, SX*SY*3*sizeof(Byte), {HOST}, {KSCALE});
5  auto b2 = mango_rt->register_buffer(B2, SX*2*SY*2*3*sizeof(Byte), {KSCALE}, {KCOPY,
       KSMOOTH});
```

```
6  auto b3 = mango_rt->register_buffer(B3, SX*2*SY*2*3*sizeof(Byte), {KCOPY, KSMOOTH},
       {HOST});
7
8  tg = new mango::TaskGraph({kscale, kcopy, ksmooth}, { b1, b2, b3 });
9  mango_rt->resource_allocation(*tg);
```

As last step before launching the application, the list of the kernel arguments needs to be defined and the input buffers initializated. Once done that, the `start_kernel` method triggers the execution of the task and it returns a completion event that can be used to wait for the end of kernel execution (see Listing 3.3).

Listing 3.3: *Initialization*

```
1  argB1 = new mango::BufferArg(b1);
2  argB2 = new mango::BufferArg(b2);
3  argB3 = new mango::BufferArg(b3);
4  argSX = new mango::ScalarArg<int>(SX);
5  argSY = new mango::ScalarArg<int>(SY);
6  argSX2 = new mango::ScalarArg<int>(SX*2);
7  argSY2 = new mango::ScalarArg<int>(SY*2);
8
9  auto argsKSCALE = new mango::KernelArguments({argB2, argB1, argSX, argSY }, kscale);
10 auto argsKCOPY = new mango::KernelArguments({argB3, argB2, argSX2, argSY2}, kcopy);
11 auto argsKSMOOTH = new mango::KernelArguments({argB3, argB2, argSX2, argSY2}, ksmooth);
12
13 b1->write(in);
14
15 auto e1=mango_rt->start_kernel(kscale, *argsKSCALE);
16 e1->wait();
17
18 auto e2=mango_rt->start_kernel(kcopy, *argsKCOPY);
19 e2->wait();
20
21 auto e3=mango_rt->start_kernel(ksmooth, *argsKSMOOTH);
22 e3->wait();
```

3.3.2 Device-side layer

At device (e.g., accelerators...) stand point, the *device-side low-level runtime support* (DLR) provides synchronization mechanisms and allows the device to perform memory mapping of buffers that are allocated in the shared memory generating virtual addresses [49]. In particular, the synchronization is obtained through the functions `mango_wait` and `mango_write_synchronization`. The former allows the kernel to wait for a specific event value while the latter

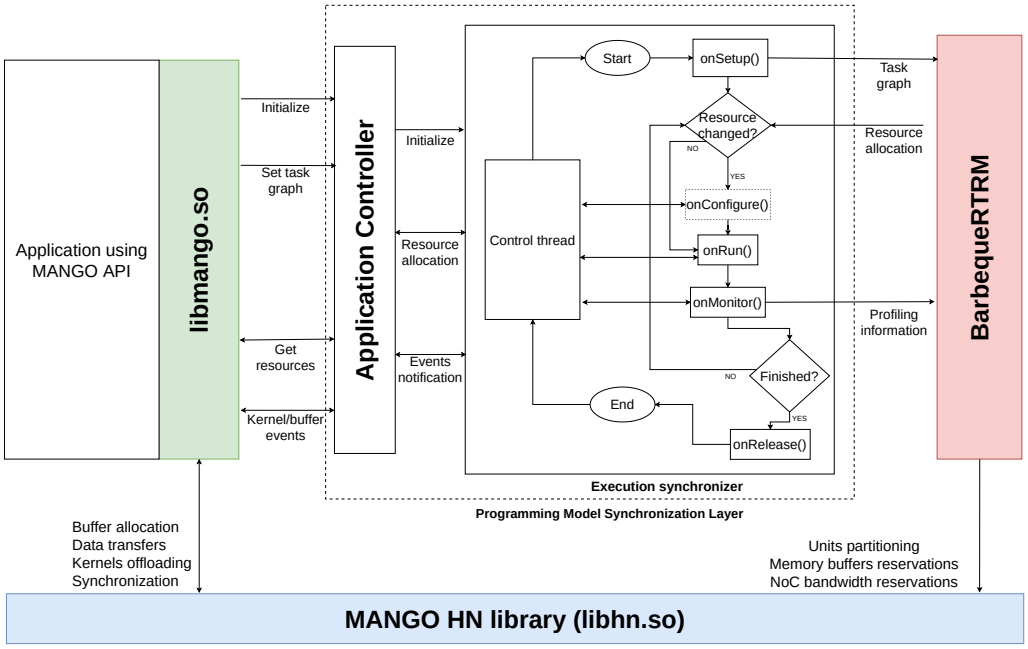

Figure 3.3: *`libmango` integration with BarbequeRTRM*

allows to write to an event a value. In the Listing 3.4, the `kernel_function` waits that input buffer is ready to be read and output buffer ready to be written, makes the computation and notify the host that the buffers contain the result of the kernel execution.

Listing 3.4: *Kernel function and synchronization*

```
1
2  void kernel_function(uint8_t *out, uint8_t *in, int X, int Y, mango_event_t e1, mango_event_t
       e2){
3      for(int i=0; i<4; i++) {
4          mango_wait(&e1, READ);
5          mango_wait(&e2, WRITE);
6          scale_frame(out, in, X, Y);
7          mango_write_synchronization(&e1, WRITE);
8          mango_write_synchronization(&e2, READ);
9      }
10 }
11
12 void scale_frame(uint8_t *out, uint8_t *in, int X, int Y){
13     // perform computation
14 }
```

3.3.3 Integration with BarbequeRTRM

In order to integrate `libmango` programming model with the BarbequeRTRM daemon, the *Programming Model Synchronization Layer* (PMSL) was built. This layer provides functionalities to synchronize the execution of the application, and its tasks, with the management actions of the BarbequeRTRM daemon. Moreover, it provides an abstraction of the resource assignment problem.

As shown in Figure 3.3, the MANGO application is interfaced with the PSML through the *Application Controller*. This controller, initialized by the `BBQContext` class, is in charge of providing a bridge between the two frameworks. During initialization of the application controller, an instance of the *Execution Synchronizer* class is created providing an execution context for the AWM. The execution synchronizer will wait on the `onSetup` until the initialization is completed and the MANGO application has provided the task graph description, so that the PMSL can forward it to the BarbequeRTRM daemon to make the resource allocation. After that, the daemon executes the allocation policy and then returns the task graph object, enriched with all information needed for allocation, to the PMSL. The PSML forwards the task graph generated to the application and the execution synchronizer enters the `onConfigure` phase waiting for kernel execution. Once received the mapping information, the application can initialize buffers, load the kernels and it is able to start the kernels with the `start_kernel` method. The `onConfigure` function starts a thread for each kernel executed to monitor its execution time and throughput, then the execution synchronizer enters the execution phase with the `onRun` function. The `onMonitor` function will collect data retrieved from the control thread spawned in the `onConfigure` and send it to the resource manager for profiling purposes. Finally, the `onRelease` function, when all kernels terminate, releases all the assigned resources.

3.4 The BeeR Framework

The Beer Framework [50] extends the `libmango` capabilities to allow distributed embedded devices, connected together through the network, to participate in the execution of an application. The framework allows a client application to choose where to execute his tasks both leveraging the resources of the nodes distributed across the network or launching tasks on heterogeneous nodes connected through PCI express link. In order to do that, a simple client-server architecture (Figure 3.4) is adopted. One of the strengths of BeeR is that the server does not need to link the MANGO library, thus, the user does not need to

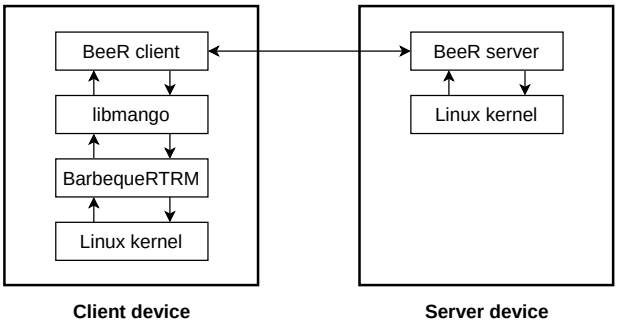

Figure 3.4: *Beer architecture*

build the MANGO framework for each device architecture. This allows to save time but also simplifies portability because MANGO library may not be easy to port to different architectures due to the long list of dependencies.

3.4.1 Framework architecture

Client and server are connected via TCP and exchange messages through a simple protocol. The client sends a message to the server and receives a response containing the result of the operation and optional messages regarding the execution of the command.

The BeeR server is the component in charge of managing the execution of a kernel on a device and it is executed as a daemon on the device itself. As soon as the server receive a new connection, it instantiates a new `Task` object that will manage all the incoming request assigning a separated thread for each connection. Inside the thread, the kernel execution is handled by the `Subprocess` class. This object takes the binary sent by the client and runs the binary code on a separate process. Moreover, as we seen in the previous section, the MANGO programming model exchanges data between different kernels running on different nodes through buffers allocated on a shared memory. The BeeR server instantiate a `Buffer` object for each kernel buffer. This class manages a shared memory reference for a specific buffer, identified by an id and a size, accessing memory through the POSIX shared memory API. The data is saved using C++ `string` class in order to be easily serialized and transferred using BeeR messages. About the client, `libmango` was extended with a new `Device` class that represents a remote instance of the BeeR daemon and provides functions to communicate with remote kernel. The remote device object is initialized by providing the host and port of the remote endpoint. Finally, a dedicated DLR library has been developed in order to implement base synchronization mechanism between tasks.

CHAPTER 4

The Use-Case Application

This chapter presents the design and implementation of the use-case Fog application, which leverages the BarMan framework described in the previous chapter and the *OpenCV* library. Section 4.1 presents the use-case application. In Section 4.2, we present the application structure and the workflow followed during the development. Then, in Section 4.3 we analyze in depth the MANGO kernels developed, presenting some code snippet of the most important parts. Finally, in Section 4.4 we present how the kernels share data between each others.

4.1 Smart Surveillance use case

We decided to realize an application that could fully exploit the potentials of the BarMan framework, starting from one of the Fog simulated use cases proposed by Gupta et al. in the *iFogSim* presentation paper [51]. In their work, they have analyzed two possible application scenarios: a human-vs-human online game and an intelligent surveillance system through distributed camera networks. These examples aim at showing the potential benefits introduced by a distributed and cooperative infrastructure in real-world scenarios.

The human-vs-human game runs as a smartphone application and involves augmented brain-computer interaction. To this purpose, each player needs to

wear a EEG headset that is connected to its smartphone. The application performs real-time processing of the EEG signals sensed and calculates the brain state of the user. On the smartphone display, the game shows all the players on a ring surrounding a target object. Each player can exert an attractive force onto the target in proportion to his level of concentration. To win the game, a player should try to pull the target toward himself.

Instead, the intelligent surveillance system aims at coordinating multiple cameras to supervise a given area. Furthermore, the system alerts the user in case of events, which may demand attention. When the smart camera detects motion in its view, it starts sending a video stream to the application. The application locates the moving object in the video stream and initiates the tracking. Tracking of moving objects is done by constantly tuning the Pan-Tilt-Zoom (PTZ) parameters of the cameras to obtain the best view of all the tracked objects. Moreover, in the event of detection of an event of interest, the application notifies the system user and sends captured video streams to him.

While the former scenario has the only requirement of real-time processing, a situation similar to the latter use case presents multiple requirements and allows us to fully exploit the capabilities of our framework such as:

- *Low-latency computation*: for an effective surveillance parameters of the application need to be tuned in real-time. This requires low-latency computation on the captured image;
- *High bandwidth usage*: security cameras continuously send captured video frames for processing. It is necessary to handle such a large amount of data basing on the network capability of the devices;
- *Heavy processing*: being image analysis a computationally intensive it's important to correctly choose where deploy the intensive tasks.

4.2 Application model and development workflow

Gupta et al. in their work identified the surveillance system as formed by the following modules, as schematized in Figure 4.1:

- *Motion detection*: this module is embedded inside the smart cameras used in the case study. It reads the raw video streams captured by the camera to find motion of an object. If motion is detected, the video stream is forwarded to the object detection module;
- *Object detection*: this module receives video streams in which motion has been detected. The module extracts the moving object from the video

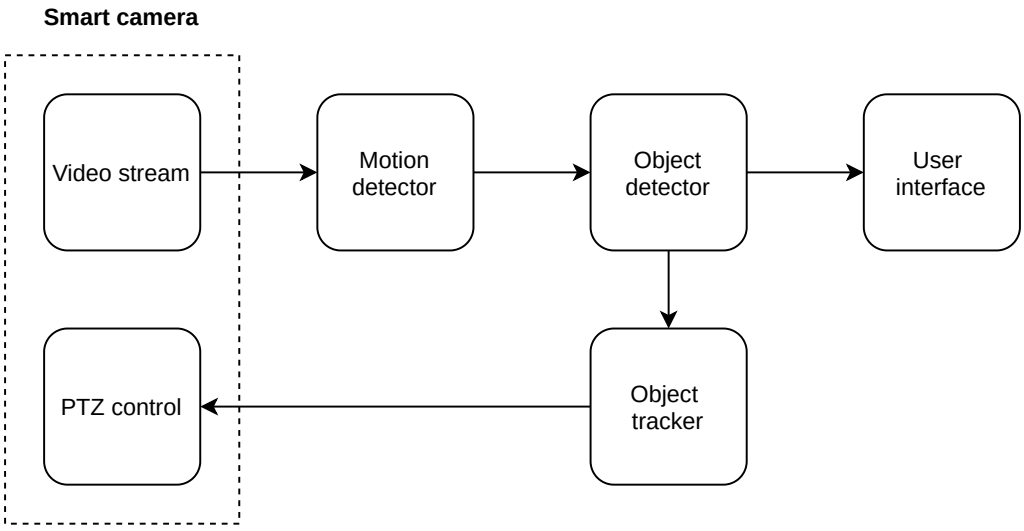

Figure 4.1: *Original smart surveillance application*

streams and compares them with previously discovered objects. In case the detected object has not been in the area before, tracking is activated and object coordinates are calculated;

- *Object tracker*: this module receives the last calculated coordinates of the currently tracked objects and computes an optimal PTZ configuration of all the cameras in order to capture the objects in the most effective manner. Once the computation is completed, PTZ informations are send to PTZ module;

- *PTZ control*: this module runs on each smart camera and adjusts the physical camera to conform to the optimal PTZ parameters;

- *User interface*: the application presents a user interface by showing the video streams containing each tracked object.

Starting from this application model, we have decided to tailor the use case to our needs and make some changes. First of all, we have considered the surveillance system as formed by only one camera. Secondly, we have also removed the PTZ control module and made some modifications on modules behaviour. Lastly, we have removed the constraint on where the module has to physically be placed. With that in mind, after having developed a multi-threaded version to test the feasibility of the application, we develop a task-based application using *OpenCV* library for all the image processing tasks. The resulting application is formed by the following kernels as shown in Figure 4.2:

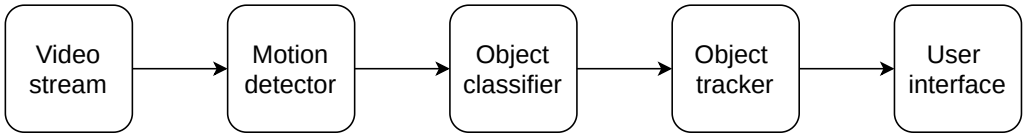

Figure 4.2: *Sequential smart surveillance application*

- *Motion kernel*: this module reads the video streams to find motion of an object. The video stream and the bounding boxes of object found in motion are forwarded to the *Classifier kernel*;

- *Classifier kernel*: this module receives video streams and motion information and performs object classification over the current frame. After the classification task is completed, motion information are compared with the results of classification in order to determine which objects are in motion. After that, bounding boxes and class information of each object are sent to the *Tracker kernel*.

- *Tracker kernel*: this module is in charge of tracking the moving object in the scene. It receives a series of object detections from the *Classifier kernel* and use that to track the objects;

- *GUI kernel*: the application presents a simple user interface where the video stream is shown and the tracked objects are enclosed in rectangles showing their id and class.

During the development, we noticed that most of the time needed to complete a computation on a single frame was spent in the *Object classifier* kernel. For this reason we started to think how to speed up the process. In the implementation, the *Motion detection* kernel generates bounding boxes around the areas of the image that are in motion and the *Object classifier* kernel uses these information only to check which objects of the ones detected and classified are in motion. Analyzing the surveillance problem, we can notice that obtaining a fast detection of something that is moving is more important than knowing what the moving object is. Moreover, the information about the type of an object is not relevant in the tracking problem. For that reasons, as summarized in Figure 4.3, we decided to develop a second version, which we will refer to as *parallel version*, where the kernels were modified as follows:

- *Motion kernel*: this module works as in the sequential version but the result of the computation is forwarded to the *Tracker kernel*;

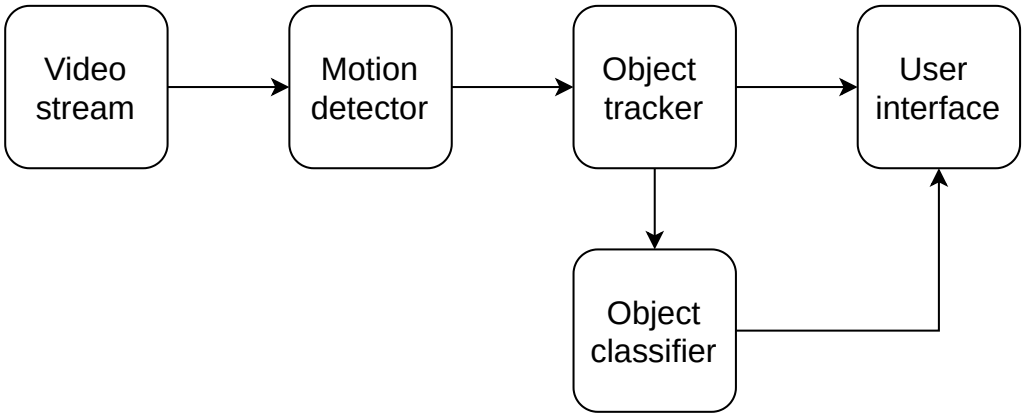

Figure 4.3: *Parallel smart surveillance application*

- *Tracker kernel*: it receives a series of bounding boxes from the *Motion kernel* and use that to track the objects. If the *Classifier kernel* is available, the *Tracker kernel* sends to it the current frame and the objects that are being tracked with their id number. Then, also in case the *Classifier kernel* is busy, the current frame and the objects tracked are send to the *GUI kernel*;
- *Classifier kernel*: this module receives current frame and objects information and performs object classification over the current frame. After the classification task is completed, motion information are confronted with the tracked object in order to generate the pair $< track_{id}, object_{class} >$. Once done that, the pairs are sent to the *GUI kernel*;
- *GUI kernel*: the application presents a simple user interface where the video stream is shown and the tracked objects are enclosed in rectangles and their id and class are shown. The bounding boxes and ids are received from the *Tracker kernel* while the class information are received, once available, from the *Classifier kernel*.

4.3 Application kernels

4.3.1 Motion Kernel

The *Motion kernel* is in charge of reading the video stream and detect motion, then it has to initialize the video stream reader and the background segmentation model. To do that, we create an instance of `cv::VideoCapture` class passing the address of the video source, then we initialize a background subtractor model. In our case we choose to use the *Counting BackgroundSubtractor* due

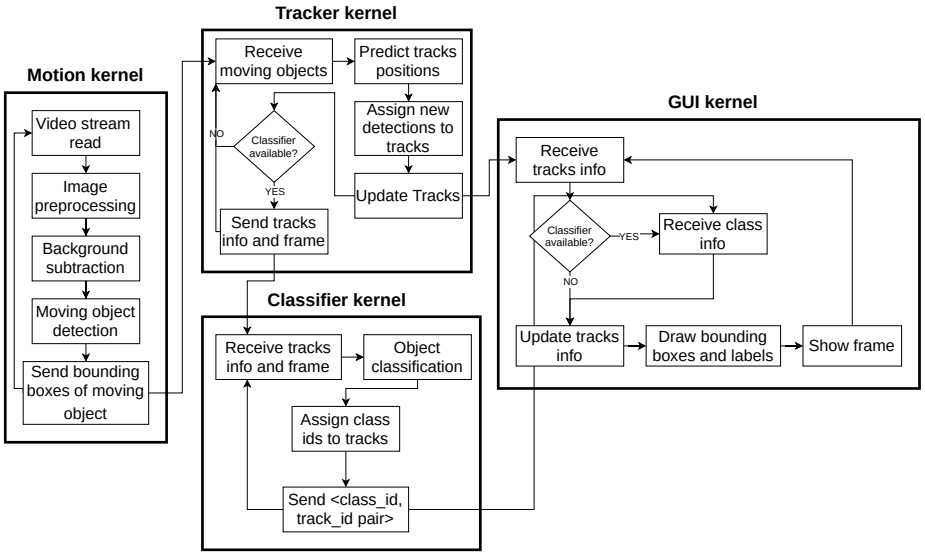

Figure 4.4: *Detailed operation of parallel version*

to its performance with low spec hardware. Once the initialization is over, the kernel is ready to process the frame of the video source, as reported in Listing 4.1.

Listing 4.1: *Image pre-processing*

```
1 cv::VideoCapture video_in(video_stream);
2  float  fps=video_in.get(cv::CAP_PROP_FPS);
3 cv::Ptr<cv::BackgroundSubtractor>
      p_back_sub=cv::bgsegm::createBackgroundSubtractorCNT(fps*1,true,fps*60);
```

While in the sequential implementation the accuracy of this step is not really important, in the parallel one we need that the kernel has to be able to correctly identify objects that are moving, rejecting all the noise determined by light condition and wind. For those reasons, it is important to adequately pre-process the images before operating any motion detection techniques.

First of all, we resize the image in order to both reduce the duration and size of image transfers and the time needed to operate over the image. Then, we apply a bilateral filter, using `cv::bilateralFilter` function (details in Appendix A.3.1), to the image in order to remove noise and enhance object border. Once done that, we obtain a version of the image where all the object contours are very sharp while the content is blurred, as we can see in Figure 4.5. Then we use that image to update our background model through the `apply` method, obtaining a mask of the foreground. All the steps are reported in Listing 4.2.

Resized image | Blurred image

Figure 4.5: *Image preprocessing*

Listing 4.2: *Image pre-processing*

```
1 cv::resize(frame, frame_resized, cv::Size(416,416));
2 cv::bilateralFilter(frame_resized,frame_smoothed,9,200,200,cv::BORDER_DEFAULT);
3 p_back_sub->apply(frame_smoothed, mask);
```

Once obtained the foreground mask, we have to determine the bounding boxes of the object detected as in motion. To do that, we need to firstly remove any possible noise from the mask created, then find the contours of the objects and finally transform those contours into bounding boxes. The noise is removed through some morphological operations like erosion and closure (for details see Appendix A.3.2). An example of the obtainable results can be seen in Figure 4.6. On the obtained mask we apply the function `cv::findContours` in order to get sets of points that identifies the contours of the objects present in the mask. Then we transform the sets of points into bounding boxes, discarding all the rectangles that have area lower than the limit set by the user. At the end of these operations, the bounding boxes and the resized frame are ready to be sent to the next kernel (see Listing 4.3).

Listing 4.3: *Generating rectangles around moving objects*

```
1 cv::erode(mask,mask,cv::Mat());
2 cv::morphologyEx(mask,mask,cv::MORPH_CLOSE,cv::Mat());
3 std::vector<std::vector<cv::Point>> contours;
4 std::vector<cv::Vec4i> hierarchy;
5
6 cv::findContours(mask, contours, hierarchy, cv::RETR_EXTERNAL,
      cv::CHAIN_APPROX_SIMPLE);
```

Figure 4.6: *Background subtraction and mask denoising*

```
7
8
9  // build rectangles around moving objects
10 std :: vector<std :: vector<cv :: Point> > contours_poly( contours . size () );
11 std :: vector<cv :: Rect> filtered_rects ;
12 for( size_t i = 0; i < contours . size (); i++ ){
13     cv :: approxPolyDP( contours[ i ], contours_poly[ i ], 7, true );
14     cv :: Rect bb=boundingRect( contours_poly[ i ] );
15     if (bb. area ()>min_area)
16         filtered_rects .push_back(bb);
17 }
```

4.3.2 Classifier Kernel

The *Classifier kernel* is in charge of taking the current frame and classify all the object contained in it. The kernel behavior differs, based on the version of the application, only in how the results of the neural network are treated and in what the kernel receive/send and from/to who.

During kernel initialization, both version loads a pre-trained neural network using the OpenCV `dnn` module(for details see Appendix A), where the weights and configuration files are provided by the user. We decided to make the application compatible with *YOLO* (Appendix B) and *MobileNetV2*(Appendix C) with the aim of allowing to load different networks depending on the performances desired and the hardware limitations. Moreover being the OpenCV `Net` object wrapped inside our abstract class `NeuralNetwork`, it is possible to easily add support to various neural networks, even custom ones, without touching the kernel code.

Once initialized the net object, in the sequential version of the application, the kernel waits for frame and bounding boxes of moving objects from the *Motion*

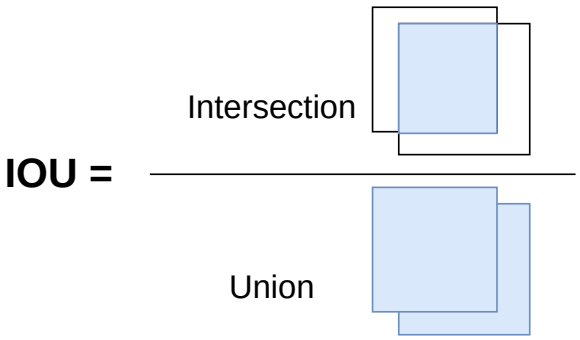

Figure 4.7: *Intersect over Union (IoU) is a metric that allows us to evaluate how similar two bounding boxes are*

kernel. Otherwise, in the parallel version it waits for the frame and pair formed by track id and the bounding box of the object tracked from *Tracker kernel*. In both cases, then it processes the frame with the neural network. Once the frame is passed through the neural network the results are interpreted depending on the output format of the neural network, in order to obtain a sequence of pairs $< bounding\_box, class\_id >$. After that, the sequential version simply iterates over the results calculating the Intersection-Over-Union (IOU) with each bounding box of moving objects. If the IOU is greater of a threshold decided by the user, the object is marked as in motion and forwarded to the *Tracker kernel*. Instead, the parallel version takes the results and the bounding boxes received by the *Tracker kernel* and applies to them the *Hungarian algorithm* [52], also known as *Kuhn–Munkres algorithm*, in order to create the pairs $< track\_id, class\_id >$. Those pairs are then forwarded to the *GUI kernel*.

4.3.3 Tracker Kernel

The *Tracker kernel* is in charge of tracking the objects that are in motion using a tracking-by-detection approach inspired to the SORT algorithm [53]. The tracking is based on the assumption that if we apply an object detector to a high frame-rate video stream, detections of the same object in subsequent frames overlap almost perfectly. So, to track an object we just have to solve the assignment problem between old detections and new detections, updating the position of the tracked objects accordingly. We can see an example in Figure 4.8, in frame *(a)* we are tracking three objects, each one characterized by a bounding box and an id. As soon as we process a new frame, new detections, identified in frame *(b)* by red rectangles, are generated. We can see that there are detections that almost overlap perfectly with the objects that we are tracking: those new detections are most likely the new position of the tracked objects. The detections that does not

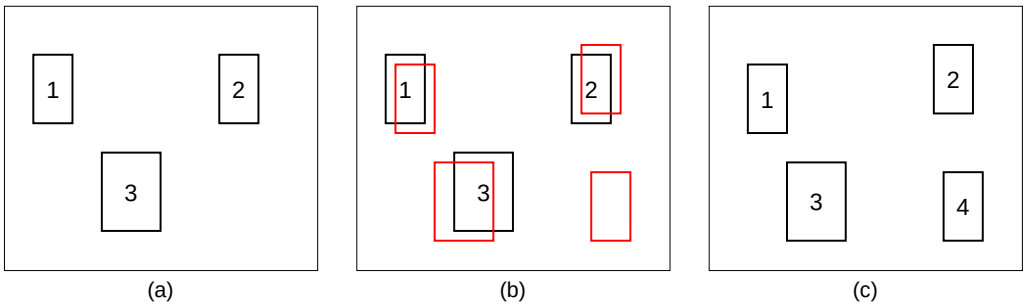

Figure 4.8: *Multi-Object tracking example*

overlap with any of the already tracked objects are most likely new objects, so we have to assign to each of them an id and start to follow it. As result of those two steps, in *(c)* we update the positions for objects 1, 2 and 3 and start to track, assigning id 4, the unpaired detection. In our solution, we associate a Kalman filter to each tracked object in order to further increase the IOU value between new detection and old position. Morover, the presence of the Kalman filter helps us to manage the situation in which the object is still in the scene but the object detector fails to detect it.

Considering the kernel operations, for each frame we predict the position of the already tracked objects, if any, using the Kalman filter. Then, we apply the Hungarian algorithm to make the associations between the object tracked and the new detection, generated by *Motion kernel* in parallel version and by *Classifier kernel* in the sequential one, using IOU as a cost function.

Once the assignment is completed we update the tracks positions for each new detection that has been paired. For each unpaired new detection, we create a new track object with a new id and we initialize a new Kernel filter (Listing 4.4). Then, for each unpaired track we update it with its Kalman filter prediction. If the track has not been updated for 10 times in a row, the track is deleted and the object is no more tracked.

Listing 4.4: *Tracks update*

```
1  Result result =makeBoundingBoxesPair(tracks, inputRects ,minIOU);
2  for(auto pair: result . getPairs ()){
3      tracks [pair . first ]. update(inputRects [pair .second]);
4  }
5
6  for(auto i: result .getUnPairedObjects ()){
7      tracks .push_back(Tracker(inputRects [i ]));
8  }
9  for(auto i: result .getUnPairedTracks ()){
10     if (! tracks [i ]. update ())
```

Figure 4.9: *Frame that is displayed to the user*

```
11          tracks . erase ( tracks . begin ()+i ) ;
12  }
```

Once the track update process is terminated, in both the application versions the kernel sends the results to the *GUI kernel*. In the parallel version, the kernel checks if the *Classifier kernel* is available and in that case it forwards the current frame and objects positions and ids to it, in order to classify the objects that are being tracked.

4.3.4 GUI Kernel

The *GUI Kernel* simply receives the information from the previous kernels and shows the result of the whole application computation to the user, as reported in Listing 4.5. The information are shown to the users in a very basic way. The current frame is displayed in a window, rectangles of different colors are drawn around the object and the pair $< objectid, objecttype >$ is shown in the upper side of the rectangle. In the parallel version it is possible that an object it is currently tracked but not classified yet, in this case a question mark is shown near the object id. All those elements are displayed using the methods offered by *OpenCV* library as can be seen in the following piece of code. This module can be adapted to run on different devices, smartphones included. In Figure 4.9 we can see an example of the image shown to the user.

Listing 4.5: *GUI drawing parallel version*

```
for(auto t : tracks){
    int track_id=t.id;
    cv::Rect boundingBox=t.rect;

    std::string label;
    if(class_ids.find(track_id) == class_ids.end()){
        class_ids.insert(std::pair<int,std::string>(track_id,"?"));
        label=std::to_string(track_id)+" ?";
    }else{
        label=std::to_string(track_id)+" "+class_ids.at(track_id);
    }

    cv::Scalar color=colors[t.id%20];

    cv::rectangle(frame,boundingBox,color,2);
    cv::putText(frame, label,
                cv::Point(boundingBox.x,boundingBox.y+15,
                cv::FONT_HERSHEY_SIMPLEX,1,cv::Scalar(0,0,255));
}
cv::imshow("Out",frame);
```

4.4 Kernels communications

The communications between kernels are managed by the *BeeR* framework. As seen in Section 3.4, once the kernel is loaded on a device, the host node and the device can communicate through BeeR-messages. For each pair of task that need to communicate, the host will spawn a thread that will manage the communication, exploiting BeeR buffers and the possibility to wait and trigger state change on events. In Listings 4.6 and 4.7, we can see an example of synchronization and data exchange between the *Motion kernel* and *Tracker kernel*.

When the *Motion kernel* has finished to process the frame, it starts to wait for *Host* on a shared event. When the *Host* is ready, it writes the state *READY* on the shared events and starts to wait for the data. Then, the *Motion kernel* writes on the buffer the frame and waits for the reception confirmation from the *Host* and so on. Supporting the *Beer* framework only fixed size buffers, the exchange of objects like dynamic containers is performed sending the total number of the objects and then sending one object at a time.

Once the communication between the *Motion kernel* and the *Host* is completed, the *Host* starts to forward the data received to the *Tracker kernel* using the same communication pattern.

Listing 4.6: *Kernel side communication*

```
// Motion kernel ready, wait for Host
mango_wait(&img_out_e,READY);

// Send frame to Host and wait for reception confirmation
memcpy(image_out,frame_resized.data,3*416*416);
mango_write_synchronization(&img_out_e,FRAME_SENT);
mango_wait(&img_out_e,FRAME_RECEIVED);

// Send number of detections that will be sent to Host
*num_objs= filtered_rects.size();
mango_write_synchronization(&nob_e,ARRAY_LENGHT);

for(size_t i = 0; i < filtered_rects.size(); i++){
    // Send one detection to Host and wait for reception confirmation
    memcpy(obj_out,&filtered_rects[i], sizeof(cv::Rect));
    mango_write_synchronization(&obj_e,OBJ_SENT);
    mango_wait(&obj_e,OBJ_RECEIVED);
}
```

Listing 4.7: *Host side communication*

```
// Host ready to receive frame
mat_out_event->write(READY);

// Receive frame from Motion kernel and send reception confirmation
mat_out_event->wait_state(FRAME_SENT);
(mat_out_buffer->read(mat))->wait();
mat_in_event->wait_state(FRAME_RECEIVED);

// Receive number of detections that will be sent by Motion kernel
num_out_event->wait_state(ARRAY_LENGHT);
(num_out_buffer->read(&num_rect))->wait();

for(int i = 0; i < num_rect; i++){
    // Receive one detection and send reception confirmation
    rect_out_event->wait_state(OBJ_SENT);
    (rect_out_buffer->read(&obj))->wait();
    rect_out_event->write(OBJ_RECEIVED);
}
```

CHAPTER 5

Tasks Allocation Policies

This chapter presents two variants of the *LAtency Versus Accuracy* (LAVA) policy. While the first one is a basic implementation that considers a wired network scenario, the second implementation is enhanced taking into account the possibility of wireless links between the devices.

5.1 The LAVA Policy

This Section presents the design and implementation of the *LAtency Versus Accuracy* (LAVA) policy. In the following parts of the section the task scheduling problem for our application is analyzed and formally defined. Then, in Section 5.3 we present the *LAVA* policy to solve the previously defined problem.

5.1.1 Problem statement

Analyzing the parallel version of our application, we can identify three main QoS metrics: detection accuracy, detection latency and tracking latency.

The nature of the application's structure makes complex to find an optimization problem that could take into account all the metrics at once. This reason combined with the fact that detection accuracy and latency are exclusively bounded to the DNN chosen, lead us to split the tasks scheduling problem in two

steps. Firstly, we choose a DNN that meets and balance the accuracy and latency requirements. Then, we solve the task allocation problem aiming to minimum latency. In the following subsections we will formally define the two problems and explain our design choices.

DNN choice

As aforementioned, the first problem that we need to solve is the choice of a DNN that meets the QoS desired by the user. We decided to characterize each DNN with two parameters: a_n, $\beta_{n,r}$. The former indicates the accuracy of the DNN *n* registered on the COCO dataset, while the latter is used as latency indicator of the DNN *n* when executed on a resource type *r*. This parameter is defined as $1 - \frac{Latency_{n,r}}{\max_{x \in \mathbb{N}} Latency_{x,r}}$ for each resource type available on the devices. For simplicity, we assumed that the value $\beta_{n,r}$ does not depend on the device used for the latencies measurements.

Depending on the usage scenario, it may be more important to achieve minimal latency rather than high accuracy in classifying objects in the video stream. For that reason we introduce the weight γ (defined in 5.2), whose value is between 0 and 1, that allows to balance the latency and the accuracy of the DNN depending on the requirements.

All the parameters are then inserted in the objective function 5.1 where y_n is the optimization variable that indicates if the $n-$DNN is selected. The function takes into account both DNN mean latency and accuracy in order to select the best DNNs available depending on the desired behaviour expressed by the value of γ. In particular, with $\gamma = 0$ we will obtain the DNN with maximum accuracy, while with $\gamma = 1$ we will obtain the once with minimum latency.

We can see that in our objective function we have considered only the parameter $\beta_{n,CPU}$. This choice was driven by the fact that we wanted to consider the worst possible scenario regarding latency.

$$\text{minimize} \quad \sum_{n\in\mathbb{N}}(1-(\gamma\times\beta_{n,CPU}+(1-\gamma)\times a_n))\times y_n \tag{5.1}$$

$$\text{subject to} \quad \sum_{n\in\mathbb{N}} y_n = 1$$

$$y_n \in \{0,1\} \qquad \forall n \in \mathbb{N}$$

$$\gamma \in [0,1] \tag{5.2}$$

$$a_n \in [0,1] \qquad \forall n \in \mathbb{N} \tag{5.3}$$

$$\beta_{n,r} \in [0,1] \qquad \forall n \in \mathbb{N}, \forall r \in \mathbb{R} \tag{5.4}$$

$\mathbb{N} =$ Set of DNNs available

$\mathbb{R} = \{CPU, ACC\}$

Tasks allocation

Once we have identified the best DNN $\hat{n}$, we can find the tasks placement that guarantees minimum application's latency. First of all we introduced some parameters to characterized each task and device. As far as concerns tasks, we introduced the followings:

- in_t^i: assumes value 1 if task *t* needs input of type $i \in \mathbb{I}$, 0 otherwise;
- out_t^o: assumes value 1 if task *t* needs output of type $o \in \mathbb{O}$, 0 otherwise;
- s_t: assumes value 1 if task *t* can be accelerated, 0 otherwise;
- $c_{t,d,r}^{req}$: quantity of CPU's resources needed by task *t* when executed on device *d* using resource type *r*. The parameter concerns exclusively requirements on CPU resources due to two main reasons. Firstly, we assumed that the accelerator present on a device can only accelerate one task at a time, then representing requirements on ACC type resources would be redundant due to the presence of s_t. Secondly, in our tests we found that even if a task is executed on an accelerator, some of the workload may still be supported by the CPU;
- n_t: assumes value 1 if task *t* needs to use a DNN, 0 otherwise.
- $e_{t,d,r}^{\alpha_d}$: is the latency of task *t* on device *d* with load α_d using resource type *r*. The latencies are computed at design time, while the α_d term is chosen at run-time based on the actual load of the device.

Instead, each devices is characterized by:

- in'^{i}_{d}: assumes value 1 if device *d* has the possibility to provide input of type $i \in \mathbb{I}$ to tasks, 0 otherwise;
- out'^{o}_{d}: assumes value 1 if device *d* has the possibility to provide output of type $o \in \mathbb{O}$ to tasks, 0 otherwise;
- s'_d: assumes value 1 if device *d* has an accelerator available, 0 otherwise;
- c_d^{total}: total number of cores of resource of CPU type on device *d*;
- c_d^{free}: number of free cores of resource of CPU type on device *d*;

Then, we introduced $L_{t,d,r}$(5.6) that is the estimated latency for task *t* on device *d* considering resource type *r* and the load to which the device *d* is subjected. The load on device *d* is computed as shown in equation 5.7 considering the ratio between the free resources of type CPU and the total number of them. Finally, we introduced the variable $x_{t,d,r}$ that represents if the task *t* is executed using resource type *r* on device *d*. The cost function to be minimized (5.5) is simply the sum of all the $L_{t,d,r}$.

$$\text{minimize} \quad \sum_{t\in\mathbb{T}}\sum_{d\in\mathbb{D}}\sum_{r\in\mathbb{R}} L_{t,d,r} \times x_{t,d,r} \tag{5.5}$$

$$\text{subject to} \quad \sum_{d\in\mathbb{D}}\sum_{r\in\mathbb{R}} x_{t,d,r} = 1 \qquad \forall t \in \mathbb{T}$$

$$c_d^{free} \leq c_d^{total} - \sum_{t\in\mathbb{T}}\sum_{r\in\mathbb{R}}(x_{t,d,r} \times c_{t,d,r}^{req}) \qquad \forall d \in \mathbb{D}$$

$$x_{t,d,r} \times in_t^i \leq x_{t,d,r} \times in_d'^i \qquad \forall t \in \mathbb{T}, \forall d \in \mathbb{D}, \forall r \in \mathbb{R}, \forall i \in \mathbb{I}$$

$$x_{t,d,r} \times out_t^i \leq x_{t,d,r} \times out_d'^i \qquad \forall t \in \mathbb{T}, \forall d \in \mathbb{D}, \forall r \in \mathbb{R}, \forall i \in \mathbb{I}$$

$$x_{t,d,ACC} \times s_t \times s'_d = x_{t,d,ACC} \qquad \forall t \in \mathbb{T}, \forall d \in \mathbb{D}$$

$$\sum_{t\in\mathbb{T}} x_{t,d,ACC} \leq s'_d \qquad \forall d \in \mathbb{D}$$

$$L_{t,d,r} = e_{t,d,r}^{\alpha_d} \times (1 - n_t \times \beta_{\hat{n},r}) \tag{5.6}$$

$$\alpha_d = ceiling((1 - \frac{c_d^{free}}{c_d^{total}}) \times 100) \qquad \forall d \in \mathbb{D}$$

(5.7)

$$c_d^{free} \geq 0 \qquad \forall d \in \mathbb{D}, \forall r \in \mathbb{R}$$

$$x_{t,d,r} \in \{0,1\} \qquad \forall t \in \mathbb{T}, \forall d \in \mathbb{D}, \forall r \in \mathbb{R}$$

$$n_t \in \{0,1\} \qquad \forall t \in \mathbb{T}$$

$$v_t \in \{0,1\} \qquad \forall t \in \mathbb{T}$$

$$v'_d \in \{0,1\} \qquad \forall d \in \mathbb{D}$$

$$g_t \in \{0,1\} \qquad \forall t \in \mathbb{T}$$

$$g'_d \in \{0,1\} \qquad \forall d \in \mathbb{D}$$

$$s_t \in \{0,1\} \qquad \forall t \in \mathbb{D}$$

$$s'_d \in \{0,1\} \qquad \forall d \in \mathbb{D}$$

$$\alpha_d \in \{0,10,25,50,75,100\} \qquad \forall d \in \mathbb{D}$$

$\mathbb{T}$ = Tasks of the application

$\mathbb{D}$ = Devices available

$\mathbb{I}$ = Types of input

$\mathbb{O}$ = Types of output

$\mathbb{R} = \{CPU, ACC\}$

5.2 The LAVAnet Policy

The total execution time of a task is composed of its execution, reception of the input data and transmission of the result. While the time needed for communication can be neglected in homogeneous wired networks, the presence of both wireless and wired connections with different bandwidths makes it necessary to estimate the input/output transfer times of tasks during scheduling to achieve latency minimization. For that reason we proposed a second version of the policy that can manage the presence of heterogeneous connections directly attached to the host node.

5.2.1 Network characterization

As explained in Section 4.4, devices communicates between each other through the node, called host, where the *BarbequeRTRM* instance is running. Due to the fact that applications may needs to transfer relatively important amount of data while achieving low latency, we consider only wireless connections where the devices and the host are directly connected. The overhead of multi-hop communication between the host and a wireless device could not be compatible with strict requirements on latency while dealing with large ammounts of data. At this regard, Chanho Jin et al. [54] investigated the possibility of real-time data transmission of high-definition images for wireless medical, comparing the throughput performance of Wi-Fi Direct to the conventional WLAN architecture. Their work showed that data-rate transmission of WLAN architecture was not reliable due to the presence of multiple devices that could generate network load. Instead, a dedicated solution as the Wi-Fi Direct was able to meet the latency requirements of the real-time transmission.

To take into account the possibility of wireless links between the devices, the host node can measure, for each wireless device, the available bandwidth of the connection using network analysis tools like *Iperf*[1]. Those measures are repeated at regular time intervals and the estimates are updated using an exponential average formula. In particular, for each wireless device we have:

$$B^{up}_{d,n+1} = \alpha \times BW^{up}_{d,n} + (1-\alpha) \times B^{up}_{d,n}$$

$$B^{down}_{d,n+1} = \alpha \times BW^{down}_{d,n} + (1-\alpha) \times B^{down}_{d,n}$$

Where $BW^{up}_{d,n}$ and $BW^{down}_{d,n}$ are the measured bandwidth of the link between the host and device *d* at time *n* in both directions, while $B^{up}_{d,n+1}$, $B^{down}_{d,n+1}$ and $B^{up}_{d,n}$,

[1] https://iperf.fr/

$B_{d,n}^{down}$ are respectively bandwidth estimates at time n and $n+1$. The weight $\alpha \in [0,1]$ controls the balancing between recent and past history in our prediction. For what concerns wired connection delay, we decided to not consider them because they are negligible if compared to the delays of wireless links.

5.2.2 Problem definition

Given the considerations of the previous paragraphs, the optimization problem presented in Section 5.1.1 needs some adjustments. First of all, we need to add the following parameters:

- $data_t^{in}$: maximum quantity of input data that the task *t* receives at each iteration;
- $data_t^{out}$: maximum quantity of output data that the task *t* sends at each iteration;
- w_d: 1 if the device is wirelessly attached to the host node, 0 otherwise;
- B_d^{up}: bandwidth estimation of the communication link at schedule time between host node and the device *d*;
- B_d^{down}: bandwidth estimation of the communication link at schedule time between the device *d* host node.

The parameters B_d^{up}, B_d^{down} can be updated regularly by *BarbequeRTRM* as explained in Section 5.2.1.

Then, we need to modify the task latency function 5.6, obtaining the equation 5.8.

$$L_{t,d,r} = e_{t,d,r}^{\alpha_d} \times (1 - n_t \times \beta_{\hat{n},r}) + w_d \times [(\sum_{t' \in \mathbb{T}} \sum_{r \in \mathbb{R}} x_{t',d,r}) \times (\frac{data_t^{in}}{B_d^{up}} + \frac{data_t^{out}}{B_d^{down}})] \quad (5.8)$$

5.3 Proposed solutions

We proposed a greedy policy to solve both the problems formally defined in the previous section. The *LAtency Versus Accuracy* (LAVA) policy takes a list of application tasks, a list of devices available, a list of DNNs and a QoS weight and returns a sub-optimal solution. In Algorithm 1 we can see the pseudocode of the LAVA base version policy. First of all, we select the optimal DNN based on γ value and 5.1 cost function. Then, from line 10, we iterate over tasks list that has been ordered in descending order depending on the value $t.res \times (1 + t.screen + t.video)$. For each task we search for the device and resource that will minimize latency, also taking into account the slowdown that the task could lead to tasks, if any, that have already been scheduled on that device. We decided to not discard devices on which the available resources are not enough to satisfy task requirements. Doing so, we will be able to schedule all the tasks. However, in the search for the best placement for the task, the fulfillment of its requirements on resources is taken into account in order to make the best choice possible. Once we have found an assignment for each task, the algorithm ends and the scheduling informations are returned. In Algorithm 2, we can see the pseudocode of the LAVAnet policy that differs from the wired version only for the device cost solution.

Algorithm 1 Choice of the best task placement depending on the resources available

LAVA $(Tasks, Devices, Nets, \gamma)$

inputs : $Tasks$ list of application tasks; $Devices$ list of available devices; $Nets$ non-empty lists of DNNs; γ QoS weight;

`/* Find best DNN according to requirements */`

$min\_cost \leftarrow 1$
$\hat{n} \leftarrow 0$
foreach $net \in Nets$ **do**
  $dnn\_cost \leftarrow 1 - (\gamma \times net.latency\_multiplier + (1 - \gamma) \times net.accuracy)$
  if $dnn\_cost \leq min\_cost$ **then**
    $min\_cost \leftarrow dnn\_cost$
    $\hat{n} \leftarrow net$
  end
end
`/* After dnn choice we search for best tasks assignments */`
$schedule \leftarrow \{\}$
`/* Sort tasks in descending order according to` $t.res \times (1 + t.screen + t.video)$ `*/`
$sort\_tasks(task)$
foreach $t \in Tasks$ **do**
  $assignment \leftarrow \{\}$
  $min\_d\_cost \leftarrow -1$
  foreach $d \in Devices$ **do**
    if $d.is\_compatible(task)$ **then**
      foreach $r \in device.resources$ **do**
        if $r.available$ **then**
          $r\_cost \leftarrow t.latency(d, r, \hat{n})$
          if $(min\_d\_cost \leq 0)$***or***$(r\_cost \leq min\_d\_cost)$ **then**
            $min\_d\_cost \leftarrow r\_cost$
            $assignment \leftarrow create\_assignment(t, d, r, t.req(r.type))$;
          end
        end
      end
    end
  end
  if $min\_d\_cost < 0$ **then**
    return $ERROR$
  else
    $allocate(assignment)$
    $schedule.push(assignment)$
  end
end
return $schedule$

Algorithm 2 Choice of the best task placement depending on the resources available

LAVAnet $(Tasks, Devices, Nets, \gamma)$

inputs : $Tasks$ list of application tasks; $Devices$ list of available devices; $Nets$ non-empty lists of DNNs; γ QoS weight;

`/* Find best DNN according to requirements */`

$min\_cost \leftarrow 1$
$\hat{n} \leftarrow 0$
foreach $net \in Nets$ **do**
  $dnn\_cost \leftarrow 1 - (\gamma \times net.latency\_multiplier + (1 - \gamma) \times net.accuracy)$
  if $dnn\_cost \leq min\_cost$ **then**
    $min\_cost \leftarrow dnn\_cost$
    $\hat{n} \leftarrow net$
  end
end
`/* After dnn choice we search for best tasks assignments */`
$schedule \leftarrow \{\}$
`/* Sort tasks in descending order according to` $t.res \times (1 + t.screen + t.video)$ `*/`
$sort\_tasks(task)$
foreach $t \in Tasks$ **do**
  $assignment \leftarrow \{\}$
  $min\_d\_cost \leftarrow -1$
  foreach $d \in Devices$ **do**
    if $d.is\_compatible(task)$ **then**
      foreach $r \in device.resources$ **do**
        if $r.available$ **then**
          $r\_cost \leftarrow t.latency(d, r, \hat{n})$
          if $d.is_wireless()$ **then**
            $r\_cost \leftarrow r\_cost + d.link_delay(t.data_up, t.data_down)$
          end
          if $(min\_d\_cost \leq 0)$ ***or*** $(r\_cost \leq min\_d\_cost)$ **then**
            $min\_d\_cost \leftarrow r\_cost$
            $assignment \leftarrow create\_assignment(t, d, r, t.req(r.type))$
          end
        end
      end
    end
  end
  if $min\_d\_cost < 0$ **then**
    return $ERROR$
  else
    $allocate(assignment)$
    $schedule.push(assignment)$
  end
end
return $schedule$

5.3.1 Tasks and devices characterization

To use *LAVA* policy with our application we need to compute and set the value of the parameters defined in Section 5.1.1 for tasks and devices. Fist of all, we define the sets:

- $\mathbb{T} = \{Motion, Classifier, Tracker, GUI\}$;
- $\mathbb{N} = \{YOLO, MobileNetV2\}$;
- $\mathbb{D} = \{Odroid, Jetson, Freescale1, Freescale2\}$;
- $\mathbb{I} = \{VIDEO\}$;
- $\mathbb{O} = \{SCREEN\}$.

Then, we define DNN parameters:

- $a_{YOLO} = 0.55$;
- $a_{MobileNetV2} = 0.22$;
- $\beta_{YOLO,CPU} = 0$ and $\beta_{YOLO,ACC} = 0$;
- $\beta_{MobileNetV2,CPU} = 0.9$ and $\beta_{MobileNetV2,ACC} = 0.65$;

Instead, as far as concerns tasks we have:

- $in_{Motion}^{VIDEO} = 1$, $in_t^{VIDEO} = 0$ for all other tasks $t \in \mathbb{T}$;
- $out_{GUI}^{SCREEN} = 1$, $out_t^{SCREEN} = 0$ for all other tasks $t \in \mathbb{T}$;
- $s_{Classifier} = 1$, $s_{Motion} = 1$ and $s_t = 0$ for all other tasks $t \in \mathbb{T}$;
- $n_{Classifier} = 1$, $n_t = 0$ for all other tasks $t \in \mathbb{T}$;
- $data_t^{in}$ and $data_t^{out}$ are shown for all tasks as in Table 6.3. The explanation of how we obtained those values is reported in Section 6.2.3;
- $e_{t,d}^{\alpha_d}$ are computed for all tasks on all devices and resources as reported in Table 5.3.
- $c_{t,d,r}^{req}$ are computed from the results in Table 5.3 as the quantity of resources needed by a task *t* on device *d* and resource type *r*, in order to have an estimated latency that is less the 1.5 times the estimated latency when the device *d* load is 0% with a minimum assignable value of 50. The values are shown in Table 5.4.

Finally, as far as concerns devices we have:

Task	v_t	g_t	s_t	n_t
Motion	1	0	1	0
Classifier	0	0	1	1
Tracker	0	0	0	0
GUI	0	1	0	0

Table 5.1: *Values of the tasks parameters*

Device	v'_d	g'_d	s'_d	$c^{total}_{d,CPU}$
Freescale1	1	1	0	400
Freescale2	1	0	0	400
Odroid	0	1	0	400
Jetson	0	0	1	400

Table 5.2: *Values of the devices parameters*

- $in'^{VIDEO}_d = 1$, for $d \in \{Freescale1, Freescale2\}$, $in'^{VIDEO}_d = 0$ for all other devices $d \in \mathbb{D}$;
- $out'^{SCREEN}_d = 1$, for $d \in \{Freescale1, Odroid\}$, $out'^{SCREEN}_d = 0$ for all other devices $d \in \mathbb{D}$;
- $s_{Jetson} = 1$, $s_d = 0$ for all other devices $d \in \mathbb{D}$;
- $c^{total}_{d,CPU} = 400$ for all the devices $d \in \mathbb{D}$.

Odroid	0%	10%	25%	50%	75%	100%
Motion kernel	54.04	54.66	56.33	62.90	88.80	189.781
Classifier kernel	2022.62	2292.45	3034.47	3984.49	7764.74	9687.35
Tracker kernel	0.13	0.1229	0.10	0.10	0.10	0.17
GUI kernel	5.19	5.14	4.96	6.2604	15.09	23.74
Jetson	0%	10%	25%	50%	75%	100%
Motion kernel (CPU)	19.18	22.50	30.61	36.54	41.20	42.141
Motion kernel (GPU)	10.01	10.70	10.80	12.4	17.8	23
Classifier kernel (CPU)	2932.62	3310.68	4130.79	5001.77	5933.91	5770.17
Classifier kernel (GPU)	287	215.22	208	208	210	226.80
Tracker kernel	0.14	0.14	0.14	0.14	0.14	0.16
GUI kernel	6.75	6.78	6.85	7.23	7.79	16.08
Freescale	0%	10%	25%	50%	75%	100%
Motion kernel	210.18	222.95	242.29	347.76	678.24	846.09
Classifier kernel	38881.97	44136.19	54469.3	79937.4	166112	184479
Tracker kernel	0.47	0.47	0.47	0.49	0.51	0.91
GUI kernel	24.05	24.47	25.23	27.12	36.66	50.31

Table 5.3: $e_{t,d}^{\alpha_d}$ *for each board and task (in* ms*)*

Device	Motion kernel	Classifier kernel	Tracker kernel	GUI kernel
Freescale, CPU	250	250	50	150
Jetson, CPU	350	250	50	50
Jetson, GPU	150	50	–	–
Odroid, CPU	150	350	50	150

Table 5.4: *Values of resource requirements for each task and device*

CHAPTER 6

Experimental Results

In this section, we will present the experiments carried out on our applications. First of all, in Section 6.1, we introduce the Fog Nodes cluster used during experimental evaluation. Then, in Section 6.2 we present the tests' organization and we discuss the results.

6.1 Experimental Setup: the SmokyGrill

The experimental evaluation has been performed on the SmokyGrill, a real cluster equipped with different embedded boards, in order to reproduce an heterogeneous distributed hardware setup, as shown in Figure 6.1 and Figure 6.2. First of all, we deployed and run the BarbequeRTRM on a *Banana Pi BPI-R*1, that acts as the starting node of the test application. It is equipped with a 1.0 GHz A20 ARM Cortex A7 dual-core processor and 1 GB DDR3 of RAM. Then, we deployed the BeeR daemons on the interconnected boards to distribute the application's tasks. In particular, we used the following devices:

A) two NXP Freescale i.MX6Q SABRE development boards, featuring an ARM 32 bit Cortex A9 1.2 GHz quad-core CPU and 1 GB DDR3 SDRAM; the default `cpu_freq` governor is set to `performance` with boosting disabled;

Figure 6.1: *Picture of the SmokyGrill Fog cluster*

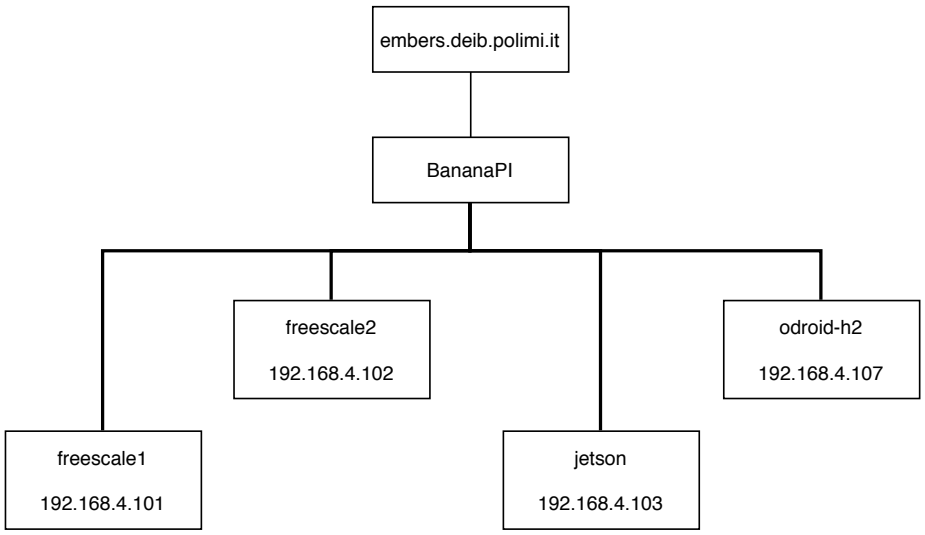

Figure 6.2: *Scheme of the connected boards of the SmokyGrill cluster*

B) one Jetson TX2 module with a 2.0 GHz quad-core ARM 64 bit Cortex A57 and 8GB LPDDR4 memory; it is also equipped with 256 CUDA cores; the default `cpu_freq` governor is set to `schedutil`;

C) one x86 Odroid-H2, equipped with an Intel 2.5 Ghz quad-core processor and 4GB of dual-channel DDR4 memory; the default `cpu_freq` governor is set to `powersave` with Intel boost enabled.

We decided to maintain the default CPU governors to simulate a real world scenario in which different boards can have proper power settings.

From the networking point of view, the boards has been interconnected with a Gigabit Ethernet switch.

6.2 Experimental Evaluation

The experimental evaluation of our application consisted of three main steps. At first, we tested the application on a desktop computer exploiting `libmango` emulation mode with the purpose of measuring application latency and under-

Figure 6.3: *Some frames from the video used*

standing how to improve the performances. Then, after some optimizations on the application workflow, we analyzed the execution latency of the kernels on the three cluster boards. Finally, we executed the distributed application leveraging *BarbequeRTRM* and *Beer* frameworks, testing the policy on multiple configurations. In this regard, we compared the performance improvements with respect to the monolithic execution considering only a single low-end node. For all the tests, we used a short video taken from the *VIRAT* dataset [55], simulating the stream coming from a surveillance camera. The video used, recorded at 30 FPS and 1920x1080 pixels resolution, was taken by a security camera placed above a parking lot. In the segment used in our tests, that counts a total number of 506 frames, we can see two small groups of standing people in the bottom left corner of the image. As soon as a car enters the scene from the top right corner, two people break away from one group and start walking towards the approaching car. After some seconds, one person of the other group starts to follow the two walking people. The video ends with the car stopping. In Figure 6.3, we can see some frames taken from the video.

6.2.1 Local execution and application speedup

As aforementioned, we fed our application with the test video. We executed the application 10 times using `libmango` emulation mode, annotating for each

	Motion Kernel	Classifier Kernel	Tracker Kernel	GUI Kernel	Application
Mean latency sequential version(ms)	13.91	381.47	0.05	3.88	384.12
Mean latency parallel version(ms)	15.05	655.89	0.07	2.52	22.41

Table 6.1: *Mean latencies in* `libmango` *emulation mode*

iteration the mean processing times of both single kernels and the whole application. To analyze the worst case, we decided to use `YOLO` as the back-end neural network since it is the computationally heaviest network supported by our application.

As anticipated in Section 4.2, the first version of the application was very slow and it was only able to process 2.6 FPS by mean. As we can see from the results reported in Table 6.1, the application latency is dominated by the *Classifier Kernel*. Thus, we decided to detach the *Classifier Kernel* from the main computation flow to reduce the application execution time, as explained in Sections 4.2 and 4.3. This new approach produced better results for what concerns application latency, allowing us to reduce it by more than 20 times, as we can see from the comparison in Figure 6.4. Regarding the kernel-level performance, as we can notice in Figure 6.5, the kernels' execution times are almost the same in the two versions except for the *Classifier Kernel* whose latency has almost doubled in parallel version. This increment is due to the fact that *OpenCV* by default automatically parallelizes the neural network execution according to the number of CPU's threads. Considering the local execution, this behavior could reduce latency in the sequential version, where the *Classifier Kernel* is performing its task when all the other processes are waiting. However, it could be counterproductive in the parallel one where all the kernels are performing their tasks at the same time. However, this latency difference between the two version is due only to the execution of all the kernels on the same machine. Moreover, *OpenCV* offers the function `cv::setNumThreads(int nthreads)` in order to manually limit the maximum number of threads that can be spawned by library functions. Future versions of the application could expose this function to allow the resource manager to configure threads available to the tasks scheduled on the same device.

Regarding the functional behaviour, this optimization have two main drawbacks. Although both versions perform multiple object tracking using the tracking-by-detection approach, the detections used by the optimized version are produced in the *Motion Kernel* using a background subtractor and detect portions of pixels of the scene that are in motion. On the contrary, the detections produced by the neural network in *Classifier Kernel* are the objects recognized by the classifier. Consequently, the first method could be sensible to lighting noise and lead

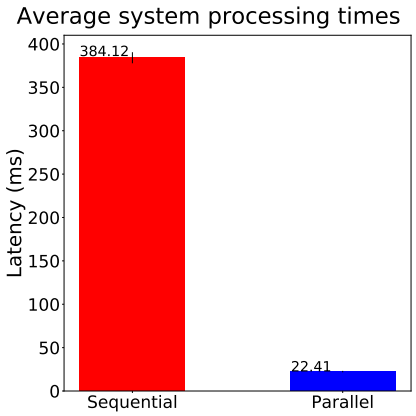

Figure 6.4: *Application latency of both sequential and parallel implementations using* `libmango`

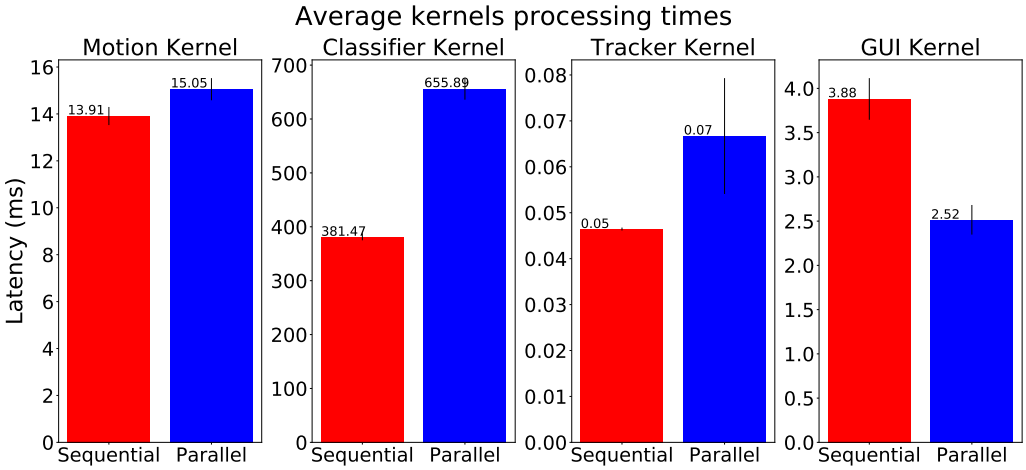

Figure 6.5: *Kernels latency of both sequential and parallel implementations using* `libmango`

to poor performance in scenarios where there are a lot of overlapped moving objects, which would be seen as a single object. However, thanks to the image pre-processing steps presented in Section 4.3, the empirical results are however acceptable. Moreover, due to the fact that the *Classifier Kernel* does not process all the frames, the object classification results are available only seconds after the object detection. This could be a problem in some very small environments where it is important to immediately identify moving objects.

6.2.2 Execution on the SmokyGrill

In the next step, we focused on profiling the execution times of the kernels on the cluster's boards processing the same video used in the `libmango` experiment.

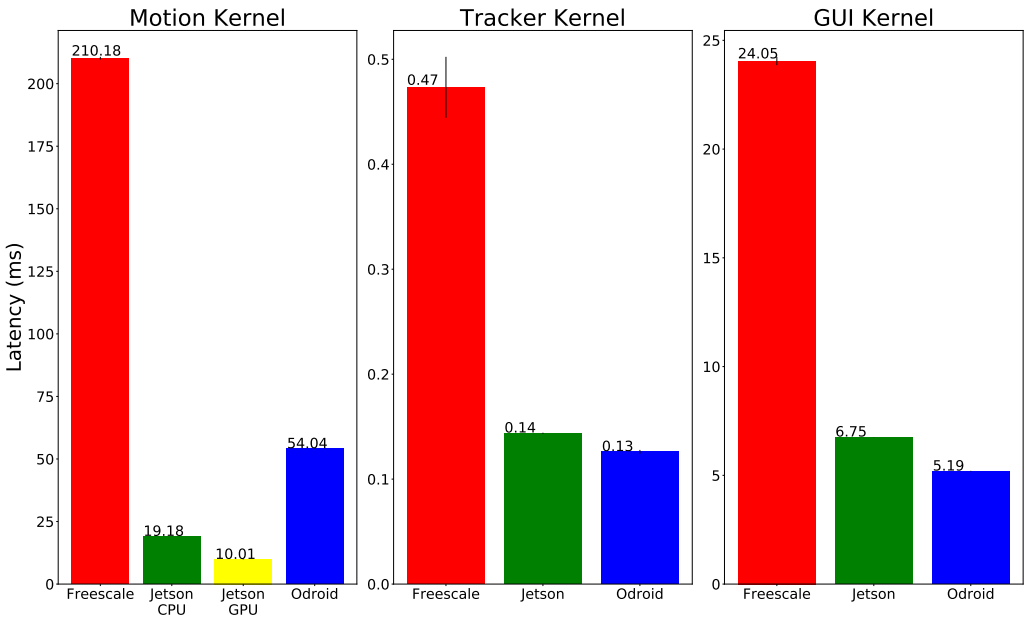

Figure 6.6: *Latency of single kernels on SmokyGrill boards*

In Figure 6.6, we can see the results of the execution of the *Motion kernel*, *Tracker kernel* and *GUI kernel*. The Freescale board, due to its low computing resources, registered the worst execution times in all the kernels. Instead, Odroid board showed the best performances in *Tracker* and *GUI* kernels. Finally, Jetson registered the best performance in the execution of *Motion kernel*, either by using only the CPU or by taking advantage of GPU acceleration, and slightly worse performance than Odroid in the *Tracker* and *GUI* kernels.

For what concerns *Classifier kernel*, as explained in Section 4.3.2, we decided to make the application compatible with *YOLO* and *MobileNetV2*. The choice of these two particular neural networks was dictated by the fact that we want to have the possibility, considering the available resources, to launch the kernel with networks that are more or less computationally demanding. In particular, although *YOLO* is more precise than *MobileNet*, we chose the *MobileNet* because is the state-of-the-art for mobile object classification and require less resources. Indeed, the former DNN obtains a mean Average Precision(mAP) of 0.55 on COCO Dataset while the latter scores 0.22 on the same dataset.

We decided to run some tests to evaluate the latency of the two neural networks on the three cluster boards obtaining the results showed in Figure 6.7. As we expected, *MobileNetV2* obtained better results in terms of latency than *YOLO* on all boards. However, the mean number of objects detected, over the 7 actually present in the scene, was 1.18 by the *MobileNetV2* and 4.33 by *YOLO*.

For the Jetson board, we evaluated both the utilization of the CPU and GPU

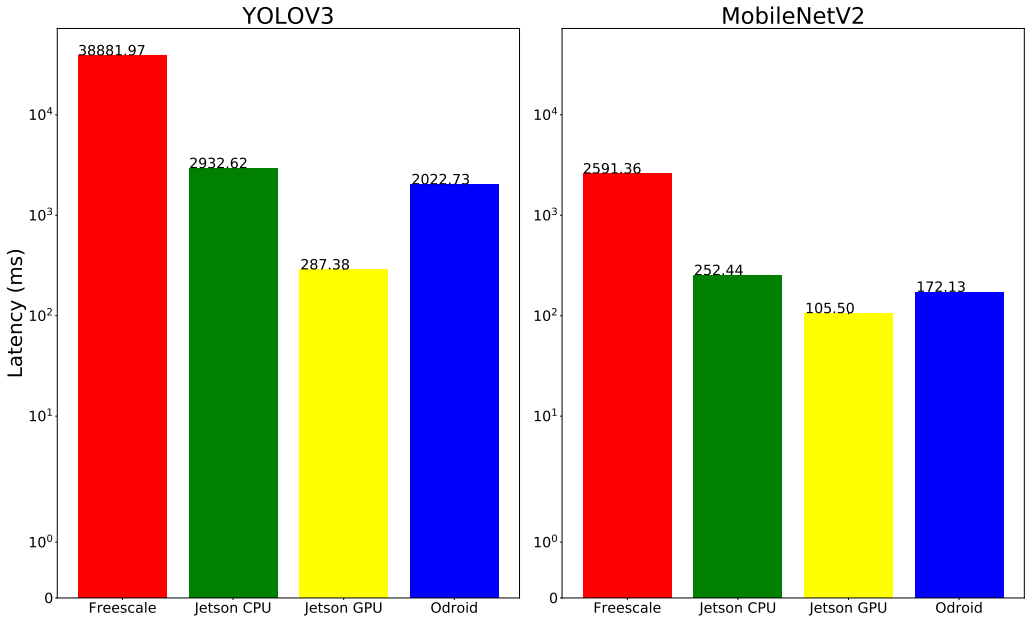

Figure 6.7: *Latency of YOLO V3 and Mobilenet V2 on SmokyGrill boards (Classifier kernel)*

separately. At this regarding, the GPU accelerated execution on Jetson of both DNN registered the best latencies, recording a 10 times reduction for *YOLO* and 2 for *MobileNetV2* with respect to the CPU execution. Moreover, we can see that the Freescale board, due to its limited resources, performed poorly with both the DNN while the Odroid board showed similar performances to the ones of the Jetson when not accelerated.

After the collection and analysis of the *Classifier kernel* performances, we ran some more test to see how much the concurrent execution of the *Classifier Kernel* with another kernel would affect its performance on the Jetson and Odroid boards (the Freescale boards were excluded due to the poor performances detected in the previous tests). For this purpose, in order to see how sensible is the *Classifier Kernel* to the presence of another kernel, we ran as second kernel the *Tracker kernel*, that is the least resource demanding between the other kernels. This choice was made in order to put us in the best case scenario and see the minimum slowdown expected when running multiple tasks on a single board. As we can see from Figure 6.8, despite the modest resource usage of the *Tracker kernel*, the execution time of the *Classifier kernel* got worse as happened in the local tests. In particular, the execution times of both the boards are slower of about 50% with respect to the single kernel execution. This shows the importance of considering the device's current load during tasks allocation and expected latency computation. Thus, as previously reported in Chapter 5.1, we measured

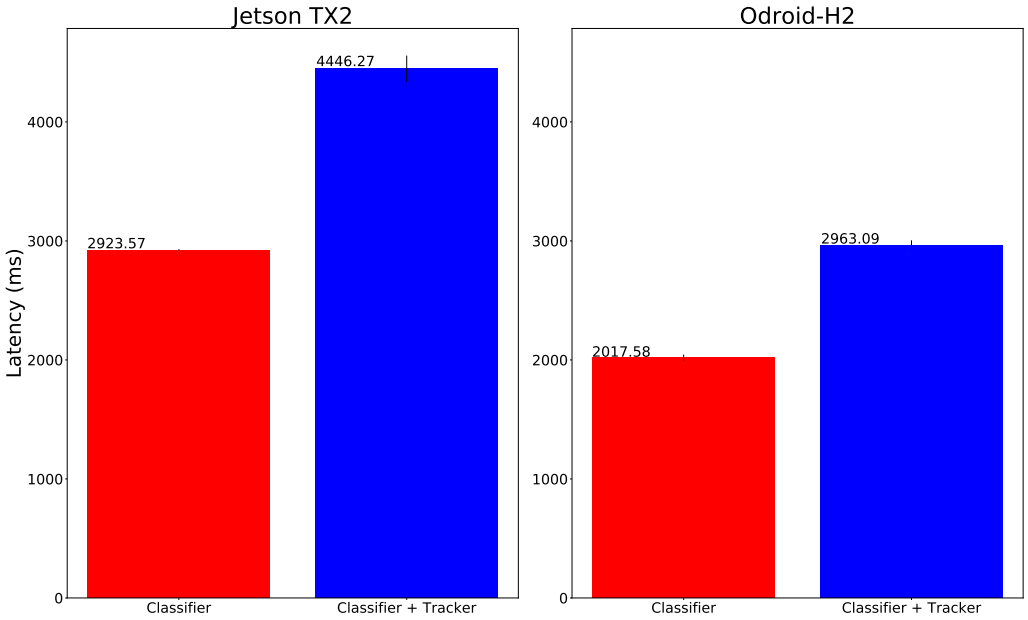

Figure 6.8: *Latency of Classifier kernel and Tracker kernel on single SmokyGrill boards*

the latencies of the tasks executed on the various boards at different CPU load levels. The results are visible in Table 5.3.

6.2.3 Network and Framework overheads

In order to evaluate the LAVAnet policy, we need to know the maximum amount of data that the kernels exchange between each others. The size of the exchanged data, except for the classifier kernel, depends on the contribution of three terms. The first term is fixed and it depends on the size of the frame used by the application, while the second term depends on the maximum amount of objects present and tracked on the frame and the data structures that contain them. Finally, the third term has a fixed size and consist of an integer number representing the number of the objects present and tracked on the frame. This last term is required by the BarMan framework in order to allow sending multiple objects. Therefore once annotated the size of all the data structures, we run the application to determine the maximum quantity of tracking related objects exchanged for frame processed, obtaining the results reported in Table 6.2. After that, we used the data collected to determine $data_t^{in}$ and $data_t^{out}$ parameters for each task *t* as reported in Table 6.3.

Sender	Receiver	Data sent	Max
Motion kernel	Tracker kernel	$frame + detections \times 16B + int\_size$	$519168B + 5 \times 16B + 4B = 519252B$
Tracker kernel	Classifier kernel	$frame + tracks\_info \times 28B + int\_size$	$519168B + 7 \times 28B + 4B = 519368B$
Tracker kernel	GUI kernel	$frame + tracks\_info \times 20B + int\_size$	$519168B + 8 \times 20B + 4B = 519332B$
Classifier kernel	GUI kernel	$class\_info \times 8B + int\_size$	$5 \times 5B + 4B = 44B$

Table 6.2: *Amount of data exchanged by the kernels for each frame*

	Motion	Classifier	Tracker	GUI
$data_t^{in}$	0	519368	519252	519376
$data_t^{out}$	519252	44	1038700	0

Table 6.3: *Max input/output data expressed in Bytes*

6.2.4 Policies execution

After the presentation of the *LAVA* policies in Chapter 5.1, we decided to test the task allocation in multiple scenarios with different device load levels and availability. In detail, for each combinations we annotated the type of resources, their initial load level and the presence of accelerators. Moreover, the devices has the input and output capabilities described in Section 5.3.

Preliminary considerations

As mentioned in Sections 4.3.2 and 6.2.2, our application supports two DNNs: *YOLO* and *MobileNet*. The former is more precise than the latter at the expense of higher detection latency. Consequently, using *MobileNet* will minimize the detection latency while using *YOLO* will maximize the detection accuracy. As explained in the Chapter 5.1, the choice of the DNN is governed by the weight $\gamma \in [0, 1]$. A value of $\gamma = 1$ selects the net that will minimize the latency and a value of $\gamma = 0$ the net that will maximize the detection accuracy, intermediate values will choose the DNN with the desired trade-off between accuracy and latency. Having only two DNNs, we decided to use only integer values of the γ weight in the policy evaluation, given the fact that using fractional values would have been meaningless in a binary decision problem.

For what concerns devices availability, the choice of those devices configurations aims at covering the following three categories of scenarios, that are characterized by:

- the availability of multiple low-end and multiple high-end devices with accelerators;
- the availability of multiple low-end and some high-end devices with possibly accelerators;
- the availability of only multiple low-end devices.

Devices	
$D0$	$Freescale1$
$D1$	$Freescale2$
$D2$	$Jetson$
$D3$	$Odroid$

Tasks	
$T0$	$Motion$
$T1$	$Classifier$
$T2$	$Tracker$
$T3$	GUI

DNNs	
$N0$	$YOLOv3$
$N1$	$MobilenetV2$

Table 6.4: *Values of the devices,tasks and DNNs parameters*

Table 6.4 shows the different values of devices, tasks and DNNS name convention used in the results visualization.

The LAVA policy execution

Considering the base version of the LAVA policy, we selected for the first scenario's category the configurations $S0$ and $S1$. As shown in Table 6.5, those represents the best possible scenarios where computational heavy tasks can be executed on high-end devices. In those two configurations, all the boards are available and with low load levels, moreover in $S0$ the Jetson board has the GPU accelerator available. From the scheduling results in Table 6.6, we can see that in both the scenarios, the policy scheduled the *Motion* task on one Frescale board (due to the Camera constraints), the *Classifier* task on the Jetson (leveraging the accelerator in $S0$) and the remaining two tasks on the Odroid board.

The second category includes the configurations $S2$, $S3$ and $S4$ and represents more realistic scenarios where multiple low-end devices are available while the high-end devices may be unavailable. The execution of the policy produced the same schedule for $S2$ and $S4$ scenarios due to the similar configuration. Having the policy the latency minimization as goal, only the *Motion* task is scheduled on a Freescale board while all the other tasks are placed on the more computationally powerful Odroid. Instead, in scenario $S4$, we can see that due to the lack of input and output capabilities of the Jetson board, the *GUI* task and *Motion* tasks were scheduled on the Freescale while the *Classifier* and *Tracker* on the Jetson.

Finally, the last category includes the scenarios where only low-end devices are available, like $S5$. In those scenarios is important that the policy is able to evenly distribute the load across the devices. As we can see in the results, the policy distributed the tasks between the two Freescale boards.

The LAVAnet policy execution

Considering the extended version of the LAVA policy, we decided to test the task allocation adding the possibility of also having wireless connections. We selected, among the various scenarios, the results of those reported in Table 6.7,

	γ	$D0$	$D1$	$D2$	$D3$
$S0$	0	$\{0\%, CPU\}$	$\{0\%, CPU\}$	$\{0\%, CPU\}, \{GPU\}$	$\{0\%, CPU\}$
$S1$	0	$\{50\%, CPU\}$	$\{25\%, CPU\}$	$\{25\%, CPU\}$	$\{50\%, CPU\}$
$S2$	1	$\{25\%, CPU\}$	$\{10\%, CPU\}$	N/A	$\{25\%, CPU\}$
$S3$	0	$\{50\%, CPU\}$	N/A	$\{10\%, CPU\}, \{GPU\}$	N/A
$S4$	1	$\{50\%, CPU\}$	$\{25\%, CPU\}$	N/A	$\{25\%, CPU\}$
$S5$	1	$\{10\%, CPU\}$	$\{50\%, CPU\}$	N/A	N/A

Table 6.5: *LAVA Scenarios description*

	DNN	$D0$	$D1$	$D2$	$D3$	Pipeline Latency(ms)	Classification Latency(ms)
$S0$	$N0$	{}	$\{T0, 0\}$	$\{T1, 1\}$	$\{T2, 0\}, \{T3, 0\}$	215.44	287.00
$S1$	$N0$	{}	$\{T0, 0\}$	$\{T1, 0\}$	$\{T2, 0\}, \{T3, 0\}$	228.31	3310.68
$S2$	$N1$	{}	$\{T0, 0\}$	N/A	$\{T1, 0\}, \{T2, 0\}$ $\{T3, 0\}$	246.86	247.89
$S3$	$N0$	$\{T0, 0\}, \{T3, 0\}$	N/A	$\{T1, 1\}, \{T2, 0\}$	N/A	922.38	215.22
$S4$	$N1$	{}	$\{T0, 0\}$	N/A	$\{T1, 0\}, \{T2, 0\}$ $\{T3, 0\}$	246.86	247.89
$S5$	$N1$	$\{T0, 0\}, \{T3, 0\}$	$\{T1, 0\}, \{T2, 0\}$	N/A	N/A	248.77	4413.62

Table 6.6: *LAVA Scheduling results*

annotating the type of resources and their initial load level and the presence of accelerators. Moreover, the devices, as well as the input and output capabilities described in Section 5.3, can be connected to the host through a wired connection or through $802.11n$ or $802.11ac$ interface. Given the fact that the cluster boards do not have wireless capabilities yet, we simulated a symmetric wireless connection assuming as maximum bandwidth, for each type of interface, the one used by Kaewkiriya [56], so a total bandwidth of 145 Mbits for $802.11n$ interface and 870 Mbits for $802.11ac$ interface.

As we can see in Table 6.7, the devices availability configurations considered for the *LAVAnet* policy are similar to the ones used for the base version but with the addition of the interfaces used by the devices to connect to the host. The choice of the connections aims at covering both scenarios where connections are only wireless and where connections are both wired and wireless. For the first category we have configurations $S0$, $S1$ and $S5$, in which all the devices are connected through a wireless connection but with different device availability and load levels. In particular, configurations $S0$ and $S2$ consider scenarios where there are both high-end and low-end devices, while $S5$ consider the situation where only low-end devices are available. Instead, the remaining configurations shows heterogeneous scenarios for what concerns both processing capabilities and network connections.

Considering the task scheduling results in Table 6.8, we can see that we obtained results similar to the base version except for the configuration $S1$. In this configuration, we can see that task $T0$ was placed on device $D0$, although $D1$

	γ	$D0$	$D1$	$D2$	$D3$
$S0$	0	$\{0\%, CPU\}$ $802.11n$	$\{0\%, CPU\}$ $802.11n$	$\{0\%, CPU\}, \{GPU\}$ $802.11ac$	$\{0\%, CPU\}$ $802.11ac$
$S1$	0	$\{75\%, CPU\}$ $802.11ac$	$\{50\%, CPU\}$ $802.11n$	$\{10\%, CPU\}$ $802.11n$	$\{50\%, CPU\}$ $802.11n$
$S2$	1	$\{10\%, CPU\}$ $802.11n$	$\{10\%, CPU\}$ $Wired$	N/A	$\{10\%, CPU\}$ $Wired$
$S3$	0	$\{10\%, CPU\}$ $802.11ac$	N/A	$\{10\%, CPU\}, \{GPU\}$ $Wired$	N/A
$S4$	1	$\{10\%, CPU\}$ $Wired$	N/A	$\{75\%, CPU\}, \{GPU\}$ $802.11ac$	N/A
$S5$	1	$\{10\%, CPU\}$ $802.11ac$	$\{25\%, CPU\}$ $802.11n$	N/A	N/A

Table 6.7: *LAVAnet Scenarios description*

	DNN	$D0$	$D1$	$D2$	$D3$	Pipeline Latency(ms)	Classification Latency(ms)
$S0$	$N0$	$\{\}$	$\{T0, 0\}$	$\{T1, 1\}$	$\{T2, 0\}, \{T3, 0\}$	215.68	287.07
$S1$	$N0$	$\{T0, 0\}$	$\{\}$	$\{T1, 0\}$	$\{T2, 0\}, \{T3, 0\}$	228.57	3310.78
$S2$	$N1$	$\{\}$	$\{T0, 0\}$	N/A	$\{T1, 0\}, \{T2, 0\}$ $\{T3, 0\}$	246.86	247.89
$S3$	$N0$	$\{T0, 0\}, \{T3, 0\}$	N/A	$\{T1, 1\}, \{T2, 0\}$	N/A	247.71	215.22
$S4$	$N1$	$\{T0, 0\}$,$\{T3, 0\}$	N/A	$\{T1, 1\}, \{T2, 0\}$	N/A	247.58	79.55
$S5$	$N1$	$\{T0, 0\}, \{T2, 0\}$ $\{T3, 0\}$	$\{T1, 0\}$	N/A	N/A	300.82	4413.72

Table 6.8: *LAVAnet Scheduling results*

has more resources available, thanks to the faster wireless connection. In a similar situation, the basic version, would have chosen device $D0$ as we can see from $S1$ configuration in Table 6.6. Instead, similar results of the other configurations could depend on the difference in terms of computing power between the boards or due to the lack of computing power of the Freescale boards, that makes scheduling preferable on more performing devices, even if those devices are connected wireless, as we can see in $S4$.

Monolithic vs Distributed comparison

Finally, to show the benefits of the distributed application execution, we measured the frame processing time of the multi-threaded entire application executed only on the Freescale board. Measurements have been taken considering the best scenarios possible, where the board does not have any external load and DNN *MobileNetV2* selected.

In Figure 6.9, we can see a comparison between the obtained results and some distributed configurations analyzed during policies evaluation. As we can see, distributed execution can reduce the times compared to running all modules in parallel on a single board, unless the boards present have few resources available or can not fulfill task requirements, as in the configuration $S3$. Instead,

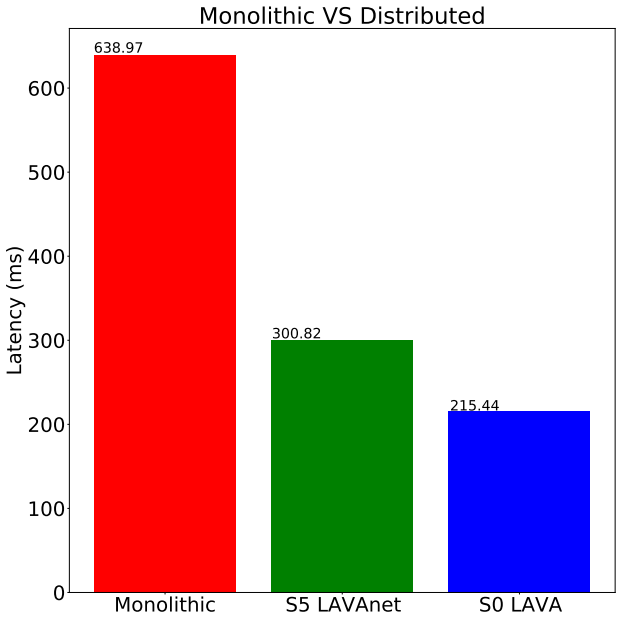

Figure 6.9: *Comparison of pipeline execution times*

in configurations with multiple boards available, such as $S5$ (LAVAnet) and $S0$ (LAVA), our approach can improve the execution time by 53% and 66% respectively.

CHAPTER 7

Conclusions and Future Works

In this chapter we discuss some final points about the work done and how our solution could be improved. In Section 7.1, we present some general considerations about the results obtained. Finally, Section 7.2 describes some improvements that could be the subject of future works.

7.1 Conclusions

In this work, we explored the design and implementation of a video surveillance use-case application and the relative allocation policies in a Fog computing scenario, using the *BarMan* framework.

We evaluated the implementation of the application both by exploiting the emulation capabilities of the *BarMan* framework and through the SmokyGrill cluster, analyzing the performance differences of the single tasks executed on the various devices.

Regarding the proposed policies, we investigated the results of their execution in multiple scenarios characterized by available devices and differences in their initial load. The results showed that the policies are able to correctly find the best task placement considering the constraints of the application and the resources availability.

Finally, we evaluated the differences between the monolithic and the distributed execution of the application. The results obtained showed that the distributed execution can improve the execution times even by 66%, when multiple devices are available and application requirements do not force tasks to run on devices with high initial load percentages.

7.2 Future Works

Fog environment is characterized by the presence of multiple heterogeneous devices, some of which may be battery-powered. Right now, our proposed policies search for the tasks scheduling that would minimize latency without considering that some devices could have limitation on the power consumption. Future implementations of those policies could address the problem through the introduction of models able to estimates the energy-related cost of the task-device mappings, leveraging tasks and devices frequency profiles. Moreover, tasks and devices profiles could be used to estimate the execution time of the various tasks for each device, thus eliminating the need for them to be calculated at design-time.

Another problem that needs to be addressed is the run-time application reconfigurability. In the current state, we have to restart the application each time that we need to reschedule the tasks on the available devices, losing the current objects tracking state. In future works, the application task graph needs to be dynamic, thus allowing the resource manager to reschedule the tasks to meet the specified QoS levels without any discontinuity of application operation. Finally, it would be interesting to investigate the impact of tuning application's specific parameters according to the status and the constraints of the devices.

APPENDIX A

OpenCV Library

A.1 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library officially launched by Intel Research in 1999. The first public alpha was released in 2000 during the IEEE Conference on Computer Vision and Pattern Recognition and the first version was released in 2006. Nowadays, the OpenCV library contains more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS [57]. Since OpenCV 2.0, the library is divided into several modules dedicated to a specific task or set of problems. In the following sections we will briefly discuss the modules used in this thesis.

A.2 Core module

The core module contains all the basic data structures and basic functions used by all the other modules in the library.

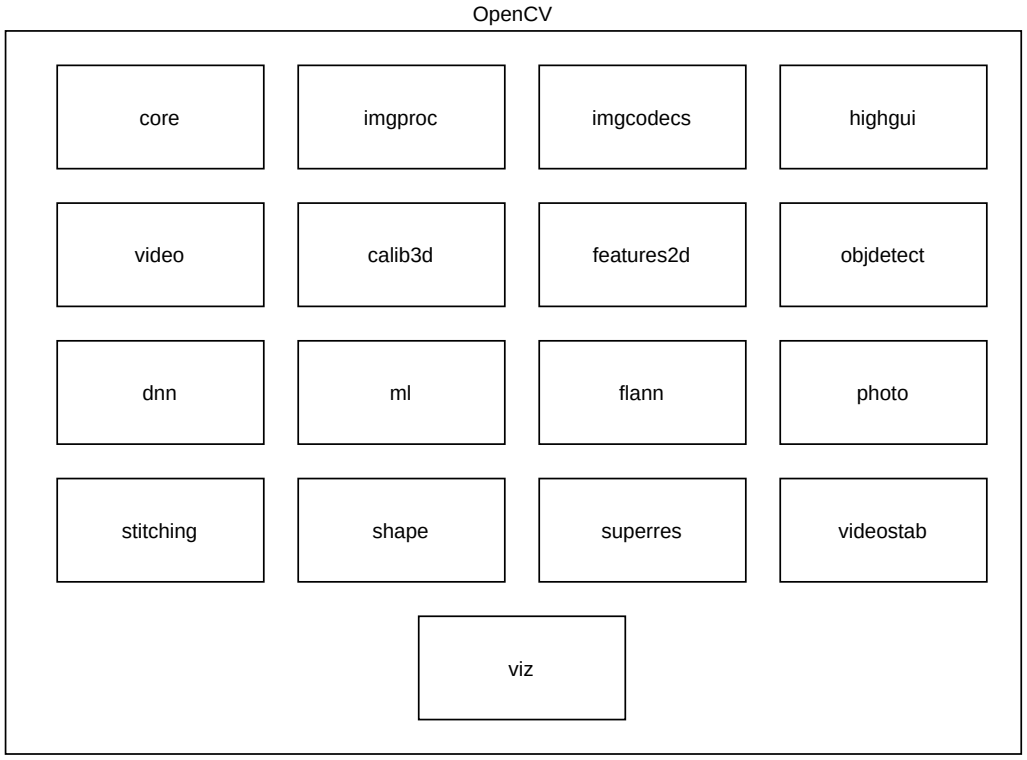

Figure A.1: *OpenCV modules*

A.2.1 `Mat`

The fundamental structure in computer vision is images. which are a two dimensional view of a three-dimensional world. A digital image is a numeric representation of a 2-D image as a finite set of digital values, which are called pixels. In turn, pixels consist of numeric values that represents measurements of the light intensity for the considered wavelength or for its range depending on the color space used. To store images inside computers memory, OpenCV provides the `Mat` class, an n-dimensional numerical single-channel or multi-channel array. The class is basically divided into two parts: matrix header (that contains information about the matrix such as size, storing method and pointer to matrix content) and the actual matrix content. The matrix header has a constant size while the matrix dimension may vary from image to image. To avoid of making unnecessary copies of potentially large images OpenCV uses a reference counting system. Each `Mat` object has its own header, when a matrix is shared between two `Mat` objects their matrix field simply point to the same address. Moreover, the copy operators will only copy the headers and the matrix pointer, not the data itself.

A.3 `imgproc` module

The `imgproc` module contains image-processing functions such as image filtering, geometrical image transformations, color space conversion and histograms.

A.3.1 Image filtering

The goal of using filters is to modify or enhance image properties and/or extract valuable information from the pictures, such as edges, corners, and blobs. Filtering of an image is accomplished through an operation called convolution. Convolution is a neighborhood operation in which each output pixel is the weighted sum of neighboring input pixels. The matrix of weights is called kernel, also known as the *filter*. Characteristics of the kernel will determine the type of filtering.

Image blurring

Image blurring is achieved by convolving the image with a low-pass filter kernel. It is useful for removing noise. Blurring actually removes high frequency content from the image resulting in edges being blurred when this filter is applied. OpenCV provides mainly four types of blurring techniques: average filtering, gaussian filtering, median filtering and bilateral filtering.

Average filtering: this is done by convolving the image with a normalized box filter. It simply takes the average of all the pixels under kernel area and replaces the central element with this average;

Gaussian filtering: differently from a box filter, which consists of equal filter coefficients, this type uses a Gaussian kernel. It results to be highly effective in removing Gaussian noise from the input image;

Median filtering: it computes the median of all the pixels under the kernel window and the central pixel is replaced with this median value. This is highly effective in removing the so called "salt-and-pepper" noise. The kernel size must be a positive odd integer;

Bilateral filtering: all the aforementioned types of filter have the tendency to blur edges. Instead, bilateral filter combines a Gaussian filter in the space domain, with one Gaussian filter component which is a function of pixel intensity differences. The Gaussian function of space makes sure that only the pixels that are "spatial neighbors" are considered for filtering. Instead, the Gaussian component applied in the intensity domain ensures that only those pixels with intensities similar to that of the central pixel are included. As a result, this method preserves edges while blurring the remaining parts of the image.

A.3.2 Morphological operations

Morphology is a broad set of image processing operations based on shapes. In a morphological operation, each pixel in the image is adjusted based on the value of other pixels in its neighborhood. [58] This behaviour is obtained convolving a structuring element, that is a matrix that identifies the pixel in the image being processed and defines its neighborhood. The choice of the structuring element and its anchor pixel determines the result of the transformation. *Dilation* and *erosion* are the two basic operators in the area of morphological transformations and, together with closing operation, they are the transformations used inside the motion tracking kernel.

Dilation

The main effect of a dilation operation on a binary image is to expand the boundary regions of the foreground object. Given an image and a kernel, the latter is scanned through the image and the maximal pixel value overlapped by the kernel is computed. Then, the image pixel in the anchor point position is replaced with that maximal value. The operation is synthesizable with the following formula:

$$\texttt{dst}(x,y) = \max_{(x',y'):\,\texttt{element}(x',y')\neq 0} \texttt{src}(x+x', y+y')$$

Erosion

The basic idea of this operation is like soil erosion: it erodes away the boundaries of foreground object. Like dilation, erosion operation is obtained scanning the kernel over the image, with the difference that in this case the operator takes into account the minimal pixel value overlapped by the kernel. The operation is synthesizable with the following formula:

$$\texttt{dst}(x,y) = \min_{(x',y'):\,\texttt{element}(x',y')\neq 0} \texttt{src}(x+x', y+y')$$

Closing

The two aforementioned basic morphological operations can be combined to obtain different results. One possibility is to use dilation followed by erosion, operation known as closing, with the result of removing holes or gaps present in the object while keeping unchanged the initial object size. The operation is synthesizable with the following formula:

$$dst = \texttt{close}(src, element) = \texttt{erode}(\texttt{dilate}(src, element))$$

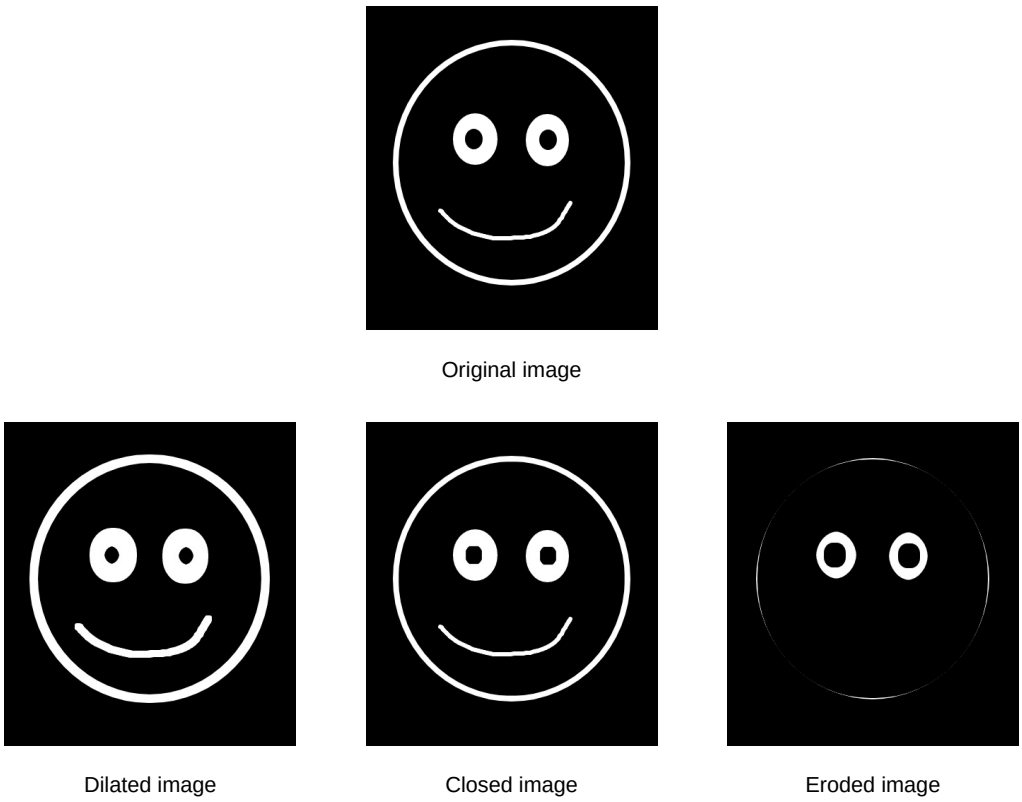

Figure A.2: *Results of dilation, erosion and closing on a binary image*

A.4 video module

Video module contains all the necessary tools for video-analysis including background subtraction, motion estimation and object-tracking algorithms.

A.4.1 Background subtraction

Background subtraction is a common technique for generating a foreground mask by using the images gathered by a static camera. This method calculates the foreground mask by performing a subtraction between the current frame and a background model. The background model, once initialized, is updated at each new frame in order to adapt to any possible changes in the scene. OpenCV library offers various types of background subtractor methods.

A.5 dnn module

Since OpenCV 3.1, a Deep Neural Network(DNN) module is included in the library. This module contains API to: (a) create new neural networks layers, (b)

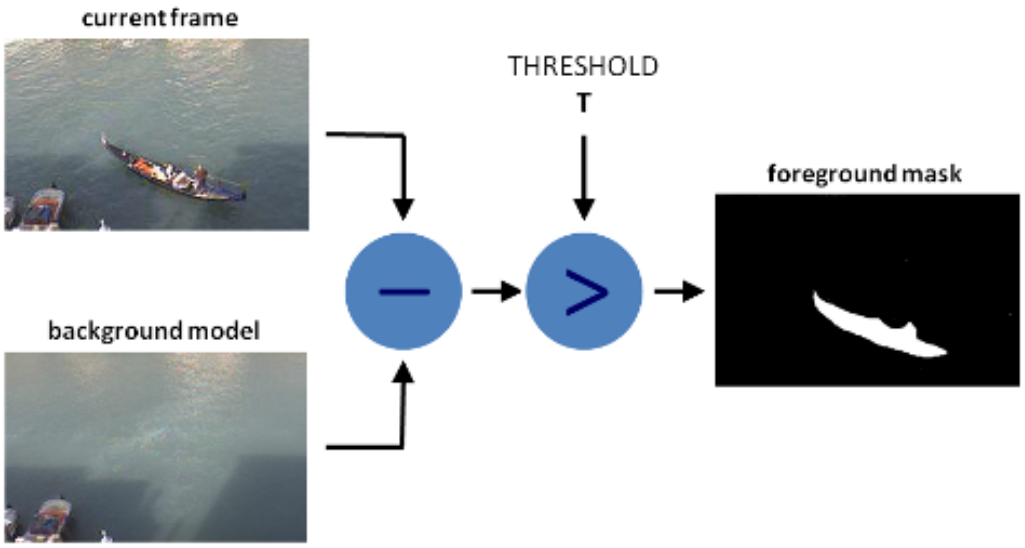

Figure A.3: *Background subtraction [1]*

construct and modify comprehensive neural networks from layers and (c) load serialized networks models from various frameworks (e.g. Caffe, TensorFlow, Torch and Darknet). The application described in this thesis makes use of only the last functionality, in fact, dnn module utilities are used to load a pre-trained network, those presented in Appendix B and C, and read the results of the net forward pass.

APPENDIX B

YOLO Neural Network

B.1 YOLO: you only look once

YOLO (You Only Look Once), is a neural network for object detection presented in 2015 [2]. The object detection task consists in determining the location on the image where certain objects are present, as well as classifying those objects. While this task is performed instantly and without too much effort by humans, it is not simple to obtain similar results using computers. The object detection problem can be solved using either machine learning-based approaches or deep learning-based approaches. Regarding machine learning approaches, firstly it is necessary to define image features, then using some technique to do the classification. Instead, deep learning techniques are able to perform detection without defining features and they are typically based on convolutional neural networks (CNN).

More recently, deep-learning based approaches, like recursive convolutionary neural network(R-CNN), use region proposal methods to first generate potential bounding boxes in an image and then run a classifier on these proposed boxes. After that, bounding boxes are refined eliminating duplicate detections and rescoring the boxes based on other objects in the scene. These approach is slow and hard to optimize because each individual component of the pipeline

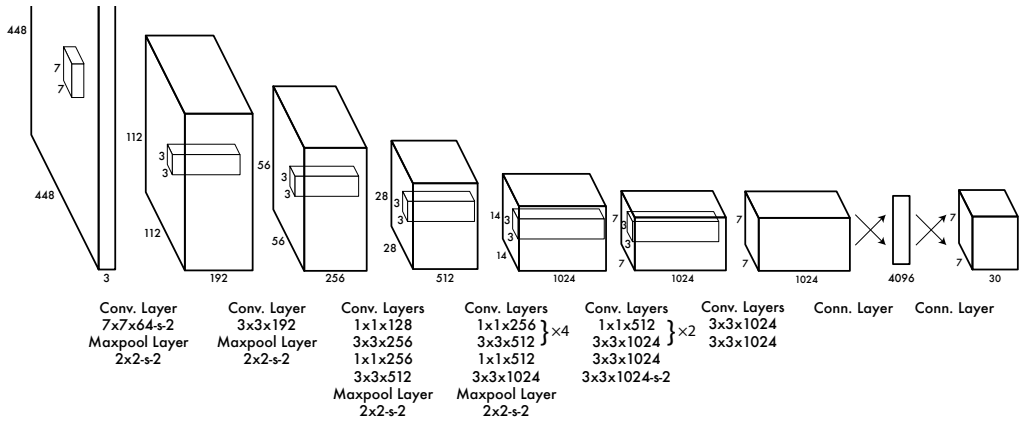

Figure B.1: *YOLO net architecture [2]*

must be trained separately.

Conversely, YOLO network reduces object detection to a single regression problem. It takes the image pixels and returns bounding boxes coordinates and class probabilities, by processing the image only once and using a single convolutional network. This unified model has the following benefits over traditional methods of object detection:

- *Velocity*: YOLO, considering the detection as a regression problem, is extremely fast. The first version of YOLO was able to process 45 frames per second on a Titan X GPU allowing process real-time video processing;

- *Global view*: unlike sliding window and region proposal-based techniques, YOLO takes into account the entire image during training and test time so it implicitly encodes contextual information about classes;

- *Generalization*; YOLO learns general representations of objects so it is less likely to make errors when applied to new domains or unexpected inputs.

B.2 Net structure and operation principles

The net architecture is inspired by the GoogLeNet [59] model for image classification. The first version of YOLO was formed by 24 convolutional layers followed by 2 fully connected layers.

As mentioned above, the object detection is performed through a single pass of the image inside the neural network. First of all, the input image is divided into a $S \times S$ grid of cells, if the center of an object falls into a grid cell, that cell is responsible for detecting that object. Each cell predicts B bounding boxes,

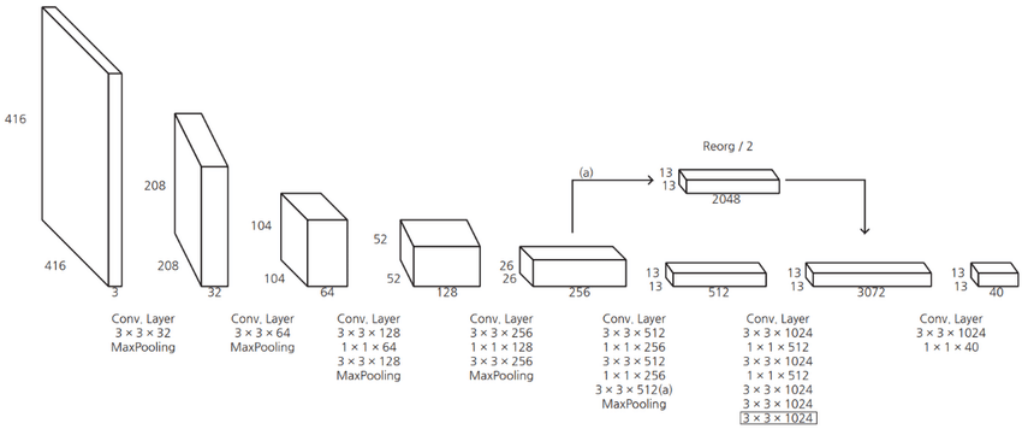

Figure B.2: *YOLOv2 net architecture [3]*

which consist of components: $(x, y, width, height, confidence)$. The (x, y) coordinates represent the center of the box relative to the grid cell location while the $width$ and $height$ are predicted relative to the whole image. Confidence, instead, is a score that reflects how confident is the model about the presence of an object inside of the considered bounding box. Moreover, each cell also compute C conditional class probabilities $P(Class_i|\text{Object confidence})$. All the predictions will be encoded as a $S \times S \times (B \times 5 + C)$ tensor. The first YOLO version was evaluated on PASCAL VOC [60] with $S = 7, B = 2, C = 20$.

It is worth to be mentioned that this approach has three main problems. First of all, the network can detect a maximum of $S \times S = 49$ objects due to the fact that each grid cell can predict only one object. This lead to difficulties in identifying small object or group of objects. Secondly, an object can be located in more than one grid so the model may detect the same object multiple times. Lastly, YOLO struggles to generalize to objects in new or unusual aspect ratios or configurations since it learns to predict bounding boxes from data.

While the second problem can be solved using non-max suppression to remove redundant results, the first and last problem cannot be faced without changing how the network works.

B.3 YOLOv2

Due to the aforementioned problems the second version of YOLO was released in 2016 [61] with the name of YOLO9000. The main upgrades consisted in:

- *High resolution classifier*: the original YOLO trains the classifier network at 224×224 and increases the resolution to 448×448 for detection. In YOLOv2 the classification network is trained with images at the full $448 \times$

448 resolution for 10 epochs on ImageNet [62];

- *Anchor boxes*: YOLO predicts the coordinates of bounding boxes directly using fully connected layers. In YOLOv2 the fully connected layers are removed and anchor boxes are used instead;
- *Multi-scale training*: training phase uses multiple image scales randomly choosed every 10 batches;
- *Fine-grained feature*: YOLOv2 predicts detections on a 13×13 feature map, helping localizing small objects while being efficient even for large objects;
- *New network architecture*: with YOLOv2 the previous network architecture is substituted by Darknet-19. The new network has 19 convolutional layers and 5 maxpooling layers.

This YOLO revision using multi-scale training, solves the object generalization problem and, including fine grained features, increases also average precision for small object.

B.4 YOLOv3

In 2018 the third version of YOLO was released [63] with the aim of increase accuracy. At this regard, the following changes have been made:

- *Improved bounding box prediction*: YOLOv3 uses independent logistic classifiers for each class instead of a regular softmax layer like in previous versions. This allows to multi-label classification;
- *New underlying network*: a new network called Darknet-53 is used for performing feature extraction is used. It is formed by 53 convolutional layers and is more accurate but slower than Darknet-19;
- *Improved abilities at different scales*: YOLOv3 predicts boxes at 3 different scales. The features are extracted from each scale by using a method similar to that of feature pyramid networks.

APPENDIX C

MobileNets

C.1 MobileNet

MobileNet is a class of efficient neural networks presented in 2017 by Howard et al. [4] and intended to be used for mobile and embedded vision applications. As outlined in Appendix B, the general trend has been to make deeper and more complicated networks in order to achieve higher accuracy at the cost of network size and speed. However, in many real world applications such as robotics and self-driving car the recognition tasks need to be carried out within reasonable times and often on a computationally limited device. For these reasons, the work of Howard primarily focuses on optimizing for latency but also yield small networks, allowing a model developer to choose a network that matches the resource restrictions of their application.

C.2 Net structure and operation principles

The *MobileNet* structure is built on depthwise separable convolutions except for the first layer which is a full convolution. All layers are followed by a batch-norm and ReLU nonlinearity with the exception of the final fully connected layer which has no nonlinearity and feeds into a softmax layer for classification, the full architecture can be seen in Figure C.1.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure C.1: *MobileNet architecture [4]*

The depthwise convolutions are made up of two layers: *depthwise convolutions* and *pointwise convolutions*. The *depthwise convolutions* are used to apply a single filter for each input channel (input depth) while the *pointwise convolution*, that is a simple 1×1 convolution, is then used to create a linear combination of the output of the depthwise layer. Depthwise convolution is extremely efficient and requires about 10 times less computation than standard convolutions at only a small reduction in accuracy [4].

Even if the base MobileNet architecture is already small and low latency, a specific use case or application may require the model to be smaller and faster. In order to comply with stricter requirements two parameters were introduced. The first parameter α is called *width multiplier* and it has the task of reducing a network uniformly at each layer, for a given layer and width multiplier α, the number of input channels M becomes αM and the number of output channels N becomes αN, reducing computational cost and the number of parameters by α. Width multiplier can be applied to any model structure to define a new untrained

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure C.2: *MobileNetV2 architecture [5] where t is the expansion factor, c the number of output channels, n the repeating number and s the stride*

smaller model with a reasonable accuracy, latency and size trade off. The second hyper-parameter introduced to reduce the computational cost of a neural network is a *resolution multiplier* ρ that is applied to the input image and subsequently the internal representation of every layer is reduced by the same multiplier with the effect of reducing computational cost by ρ^2. In practice, the ρ value is implicitly set through the input resolution.

C.3 MobileNetV2

MobileNetV2 [5] builds upon the ideas from *MobileNet*, using depthwise separable convolution as efficient building blocks. However, the new version introduces two new features to the architecture: linear bottlenecks between the layers and shortcut connections between the bottlenecks. Overall, the *MobileNetV2* models are faster for the same accuracy across the entire latency spectrum. In particular, the new models use half operations, need 30% fewer parameters and are about 40%faster on a Google Pixel phone than *MobileNetV1* models, all while achieving higher accuracy.

Bibliography

[1] OpenCV.org. How to Use Background Subtraction Methods. `https://docs.opencv.org/4.3.0/d1/dc5/tutorial_background_subtraction.html`. Accessed: 2020-05-25.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[3] S. Seong, J. Song, D. Yoon, J. Kim, and J. Choi. Determination of vehicle trajectory through optimization of vehicle bounding boxes using a convolutional neural network. *Sensors*, 19:4263, 09 2019.

[4] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun 2018.

[6] M. van Steen and A. S. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98(10):967–1009, Oct 2016.

[7] D. M. Ajay and E. Umamaheswari. An initiation for testing the security of a cloud service provider. In V. Vijayakumar and V. Neelanarayanan, editors, *Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC – 16')*, pages 33–41, Cham, 2016. Springer International Publishing.

[8] P. Mell and T. Grance. The nist definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), September 2011.

[9] Y. Jadeja and K. Modi. Cloud computing - concepts, architecture and challenges. In *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*, pages 877–880, 2012.

[10] P. Suresh, J. V. Daniel, V. Parthasarathy, and R. H. Aswathy. A state of the art review on the internet of things (iot) history, technology and fields of deployment. In *2014 International Conference on Science Engineering and Management Research (ICSEMR)*, pages 1–8, 2014.

[11] C. Brandolese and W. Fornaciari. *Sistemi Embedded*. Pearson Education, 2007.

[12] C. Brandolese and W. Fornaciari. Sistemi embedded caratteristiche, tecnologie e mercato. *Mondo Digitale*, 31(3):3–10, 2009.

[13] M. Burhan, R. A. Rehman, B. Kim, and B. Khan. Iot elements, layered architectures and security issues: A comprehensive survey. *Sensors*, 18, 08 2018.

[14] I. Mashal, O. Alsaryrah, T. Chung, C. Yang, W. Kuo, and D. Agrawal. Choices for interaction with things on internet and underlying issues. *Ad Hoc Networks*, 28, 01 2015.

[15] M. Yun and B. Yuxin. Research on the architecture and key technology of internet of things (iot) applied on smart grid. In *2010 International Conference on Advances in Energy Engineering*, pages 69–72, 2010.

[16] D. Darwish. Improved layered architecture for internet of things. *International Journal of Computing Academic Research (IJCAR)*, 4(4):214–223, 2015.

[17] R. Khan, S. U. Khan, R. Zaheer, and S. Khan. Future internet: The internet of things architecture, possible applications and key challenges. In *2012 10th International Conference on Frontiers of Information Technology*, pages 257–260, 2012.

[18] OpenFog Consortium Architecture Working Group et al. Openfog architecture overview. *White Paper OPFWP001*, 216:35, 2016.

[19] M. Zanella, G. Massari, A. Galimberti, and W. Fornaciari. Back to the future: Resource management in post-cloud solutions. In *Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications*, INTESA '18, page 33–38, 10 2018.

[20] D. Lan, A. Taherkordi, F. Eliassen, and G. Horn. A survey on fog programming: Concepts, state-of-the-art, and research challenges. In *Proceedings of the 2nd International Workshop on Distributed Fog Services Design*, DFSD '19, page 1–6, New York, NY, USA, 2019. Association for Computing Machinery.

[21] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, page 15–20, New York, NY, USA, 2013. Association for Computing Machinery.

[22] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung. Developing iot applications in the fog: A distributed dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 155–162, 2015.

[23] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[24] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE Internet of Things Journal*, 5(2):696–707, 2018.

[25] M. Ghobaei-Arani, A. Souri, and A. Rahmanian. Resource management approaches in fog computing: a comprehensive review. *Journal of Grid Computing*, 09 2019.

[26] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner. Optimized iot service placement in the fog. *Service Oriented Computing and Applications*, 11(4):427–443, Dec 2017.

[27] A. Brogi and S. Forti. : QoS-aware deployment of IoT appli-cations through the fog. *IEEE Internet Things J. 4(5)*, 4(5):1185–1192, 2017.

[28] M. Taneja and A. Davy. Resource aware placement of iot application modules in fog-cloud computing paradigm. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1222–1228, 2017.

[29] M. Selimi, L. Cerdà-Alabern, F. Freitag, L. Veiga, A. Sathiaseelan, and J. Crowcroft. A lightweight service placement approach for community network micro-clouds. *Journal of Grid Computing*, 17(1):169–189, Mar 2019.

[30] Y. Sun, F. Lin, and H. Xu. Multi-objective optimization of resource scheduling in fog computing using an improved nsga-ii. *Wirel. Pers. Commun.*, 102(2):1369–1385, September 2018.

[31] R. Deng, R. Lu, C. Lai, T. H. Luan, and H. Liang. Optimal workload allocation in fog-cloud computing toward balanced delay and power consumption. *IEEE Internet of Things Journal*, 3(6):1171–1181, 2016.

[32] D. Rodopoulos, S. Corbetta, G. Massari, S. Libutti, F. Catthoor, Y. Sazeides, C. Nicopoulos, A. Portero, E. Cappe, R. Vavrík, V. Vondrák, D. Soudris, F. Sassi, A. Fritsch, and W. Fornaciari. Harpa: Solutions for dependable performance under physically induced performance variability. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 270–277, 2015.

[33] G. Massari, S. Libutti, A. Portero, R. Vavrik, S. Kuchar, V. Vondrak, L. Borghese, and W. Fornaciari. Harnessing performance variability: A hpc-oriented application scenario. In *2015 Euromicro Conference on Digital System Design*, pages 111–116, 2015.

[34] X. Pham and E. Huh. Towards task scheduling in a cloud-fog computing system. *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–4, 2016.

[35] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. Resource provisioning for iot application services in smart cities. *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–9, 2017.

[36] A. Zanni, S. Forsström, U. Jennehag, and P. Bellavista. Elastic Provisioning of Internet of Things Services using Fog Computing: an Experience Report. In *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, page 17–22, 2018.

[37] Y. Jiao, P. Wang, D. Niyato, and K. Suankaewmanee. Auction mechanisms in cloud/fog computing resource allocation for public blockchain networks. *IEEE Transactions on Parallel and Distributed Systems*, 30:1975–1989, 2019.

[38] D. T. Nguyen, L. B. Le, and V. Bhargava. Price-based resource allocation for edge computing: A market equilibrium approach. *IEEE Transactions on Cloud Computing*, 2018.

[39] C. Anglano, M. Canonico, and M. Guazzone. Profit-aware resource management for edge computing systems. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, EdgeSys'18, page 25–30, New York, NY, USA, 2018. Association for Computing Machinery.

[40] G. Massari, M. Zanella, and W. Fornaciari. Towards distributed mobile computing. In *2016 Mobile System Technologies Workshop (MST)*, pages 29–35, 2016.

[41] M. Zanella. Energy-aware run-time management of distributed mobile devices. *Master Thesis, Politecnico di Milano*, 2017.

[42] M. Zanella, G. Massari, and W. Fornaciari. Enabling run-time managed distributed mobile computing. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, page 39–44, New York, NY, USA, 2018. Association for Computing Machinery.

[43] D. H. Tran, N. H. Tran, C. Pham, S. M. A. Kazmi, E. Huh, and C. S. Hong. Oaas: offload as a service in fog networks. *Computing*, 99(11):1081–1104, Nov 2017.

[44] J. Flich, G. Agosta, P. Ampletzer, D. A. Alonso, C. Brandolese, E. Cappe, A. Cilardo, L. Dragić, A. Dray, A. Duspara, W. Fornaciari, E. Fusella, M. Gagliardi, G. Guillaume, D. Hofman, Y. Hoornenborg, A. Iranfar, M. Kovač, S. Libutti, B. Maitre, J. M. Martínez, G. Massari, K. Meinds, H. Mlinarić, E. Papastefanakis, T. Picornell, I. Piljić, A. Pupykina, F. Reghenzani, I. Staub, R. Tornero, M. Zanella, M. Zapater, and D. Zoni. Exploring manycore architectures for next-generation hpc systems through the mango approach. *Microprocessors and Microsystems*, 61:154 – 170, 2018.

[45] BarbequeRTRM OpenSource Project. The BarbequeRTRM OpenSource Project. `https://bosp.deib.polimi.it/doku.php?id=start`. Accessed: 2020-07-01.

[46] P. Bellasi, G. Massari, and W. Fornaciari. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Trans. Embed. Comput. Syst.*, 14(2), March 2015.

[47] R. Görgen, K. Grüttner, F. Herrera, P. Peñil, J. Medina, E. Villar, G. Palermo, W. Fornaciari, C. Brandolese, D. Gadioli, S. Bocchio, L. Ceva, P. Azzoni, M. Poncino, S. Vinco, E. Macii, S. Cusenza, J. Favaro, R. Valencia, I. Sander, K. Rosvall, and D. Quaglia. Contrex: Design of embedded mixed-criticality control systems under consideration of extra-functional properties. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 286–293, 2016.

[48] G. Massari, A. Pupykina, G. Agosta, and W. Fornaciari. Predictive resource management for next-generation high-performance computing heterogeneous platforms. In Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 470–483, Cham, 2019. Springer International Publishing.

[49] G. Agosta, W. Fornaciari, G. Massari, A. Pupykina, F. Reghenzani, and M. Zanella. Managing heterogeneous resources in hpc systems. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, page 7–12, New York, NY, USA, 2018. Association for Computing Machinery.

[50] D. Iezzi. Beer: an unified programming approach for distributed embedded platforms. *Master Thesis, Politecnico di Milano*, 2018.

[51] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[52] H. W. Kuhn and B. Yaw. The hungarian method for the assignment problem. *Naval Res. Logist. Quart*, pages 83–97, 1955.

[53] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft. Simple online and realtime tracking. *2016 IEEE International Conference on Image Processing (ICIP)*, Sep 2016.

[54] C. Jin, J. Choi, W. Kang, and S. Yun. Wi-fi direct data transmission for wireless medical devices. In *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, pages 1–2, 2014.

[55] S. Oh, A. Hoogs, A. Perera, N. Cuntoor, C. Chen, J. T. Lee, S. Mukherjee, J. K. Aggarwal, H. Lee, L. Davis, E. Swears, X. Wang, Q. Ji, K. Reddy, M. Shah, C. Vondrick, H. Pirsiavash, D. Ramanan, J. Yuen, A. Torralba, B. Song, A. Fong, A. Roy-Chowdhury, and M. Desai. A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*, pages 3153–3160, 2011.

[56] T. Kaewkiriya. Performance comparison of wi-fi ieee 802.11ac and wi-fi ieee 802.11n. In *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*, pages 235–240, 2017.

[57] OpenCV.org. OpenCV about. `https://opencv.org/about/`. Accessed: 2020-05-11.

[58] mathworks.com. Morphological Operations- MATLAB & Simulink. `https://www.mathworks.com/help/images/morphological-filtering.html`. Accessed: 2020-05-13.

[59] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[60] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, January 2015.

[61] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[62] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[63] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.