POLITECNICO DI MILANO
Department of Electronics, Informatics and Bioengineering
M.Sc. programme in Computer Science and Engineering

POLITECNICO
MILANO 1863

Enabling Flexibility of Data-Intensive Applications on Container-Based Systems with Node-RED in Fog Environments

Advisor:
Ing. Pierluigi Plebani

Master Thesis of:
Nenad Petrovic
ID 854768

Academic Year 2016-2017

Summary

Fog Computing extends the Cloud Computing to the Edge of the network, closer to the things that produce and act on data including different types of devices, ranging from personal computers, laptops, workstations to IoT devices, wearable gadgets and sensors.

Having the right data at the right place without breaking legal issues and policies related to privacy is crucial in data-intensive applications, especially in Fog computing where data could be produced and consumed at both Edge and Cloud. In general, there are two approaches to tackle this problem – move the data or move the computation tasks. In this case, we focus on the latter – the computation task movement.

The goal of this thesis is to explore state-of-the-art concepts and utilize the state-of-the-art technologies for container-based virtualization and development of data-intensive applications in order to design and implement a framework which gives ability to move computation tasks between Cloud and Edge in case of mixed architecture execution environment – the so-called Fog Computing, as a way to deal with issues of privacy and latency in data-intensive applications and taking into account the heterogeneity of devices executing the tasks.

As a result of the research carried out during the thesis, a framework for task movement in Fog environment has been developed, using Node-RED development tool, Docker containers as task abstraction and exploiting the capabilities of Docker Swarm container management system to allocate tasks.

The main advantage of the solution that it provides flexibility of data-intensive applications, without distracting application developers from focusing on the business logic. It is completely transparent to them where is the task executed – whether it is a personal computer, workstation, virtual machine in Cloud or IoT device. Thus, it is possible to keep the same structure of the application even after the changes of legal regulations, privacy/security policies and network condition.

Keywords: Data-intensive applications, Internet of Things, Cloud Computing, Fog Computing, Raspberry Pi, Container-based virtualization, Docker containers, Docker Swarm, Kubernetes, Node-RED

Sommario

Fog Computing estende il Cloud Computing verso la frontiera della rete (i.e., Edge), vicino a dispositivi che producono e agiscono sui dati, tra cui: personal computer, workstations, dispositivi IoT, wearables, sensori e altro.

Avere il dato rilevante nel posto giusto e senza violare vincoli di privacy è cruciale soprattutto per le applicazioni data-intensive, specialmente quando si considera il Fog Computing dove i dati possono essere prodotti e consumati sia nell'Edge che nel Cloud. In generale, vi sono due approcci per affrontate questo problema: spostare i dati o spostare i task che operano sui dati. Questo lavoro si concentra sul secondo caso: il task movement.

Scopo della tesi è di esplorare lo stato dell'arte, sia in termini di concetti che di tecnologie, relativo alla virtualizzazione dei container e allo sviluppo di applicazioni data-intensive col fine di progettare e implementare un framework in grado di migrare i computation task tra Cloud e Edge, considerando anche architetture elaborative eterogenee.

Tutto ciò ha prodotto un framework per il task movement in ambiente Fog che utilizza Node-RED per lo sviluppo di applicazioni data-intensive, Docker container per l'esecuzione dei task, e Docker Swarm per il deployment dinamico dei task.

Il vantaggio principale della soluzione proposta è la flessibilità fornita alle applicazioni di tipo data-intensive, lasciando allo sviluppatore il solo compito di definire la logica applicativa senza preoccuparsi di dove il task verrà eseguito: su un personal computer, una workstation, o una Virtual Machine su Cloud. Questo permette di riorganizzare in modo trasparente l'applicazione per soddisfare requisiti, quali quelli di privacy, che potrebbero limitare il data movement.

Parole chiave: Data-intensive applications, Internet of Things, Cloud Computing, Fog Computing, Raspberry Pi, Container-based virtualization, Docker containers, Docker Swarm, Kubernetes, Node-RED

Contents

1 Introduction ... 1
1.1 Data-intensive Applications ... 1
1.2 Motivation ... 3
1.3 Research goals ... 5
1.4 Thesis structure ... 5
2 Fog Computing ... 7
2.1 Edge overview ... 10
2.2 Cloud overview ... 12
3 Container-Based Virtualization ... 14
3.1 Concept ... 14
3.2 Technology ... 18
3.2.1 Docker ... 19
3.2.2 Container management systems ... 30
4 Data-Intensive Applications Development in Node-RED ... 48
5 Framework for Computation Task Movement in Fog Environment ... 54
5.1 Framework overview ... 54
5.2 Framework database design ... 57
5.2.1 Nodes ... 58
5.2.2 Tasks ... 59
5.2.3 Mappings ... 60
5.3 Task development environment ... 61
5.3.1 Template-based task creation mechanism ... 62
5.3.2 Task allocation (binding) node generation ... 65
5.3.3 Task execution proxy generation ... 67
5.3.4 Task creation user interface ... 73
5.4 Application development environment ... 75
5.5 Deployment environment ... 78
5.5.1 Node Selector ... 83
5.5.2 Service creator ... 87
5.5.3 Task allocator ... 92
5.6 Execution environment ... 94

5.6.1 Task execution proxy ... 95
5.7 Prerequisites and limitations ... 98
5.7.1 Prerequisites ... 98
5.7.2 Constraints ... 101
5.7.3 Hardware and software specification ... 102
6 Case study ... 103
6.1 Healthcare scenario example ... 103
6.2 Application overview and architecture ... 107
6.3 Application interface ... 111
6.4 Node-RED application implementation ... 116
6.5 Experiments ... 119
7 Conclusion and Future Work ... 120
References ... 122

Abbreviations and Acronyms

API	Application Programming Interface
ARM	Advanced RISC Machine
DB	Database
DIA	Data-Intensive Application
DNS	Domain Name System
ECG/EKG	Electrocardiography Electrocardiogram
CLI	Command-line Interface
GPIO	General-purpose input/output
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
HTML	Hypertext Markup Language
IaaS	Infrastructure-as-a-Service
I/O	Input/Output
IoT	Internet of Things
JS	JavaScript
k8s	Kubernetes
iSCSI	Internet Small Computer System Interface
JSON	JavaScript Object Notation
LXC	Linux Containers
NFS	Network File System
NoSQL	Not only SQL Non-relational Non-SQL
OS	Operating System
PaaS	Platform-as-a-Service
PHP	Hypertext Preprocessor (formerly: Personal Home Page)
REST	Representational State Transfer
QEMU	Quick Emulator
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RPi	Raspberry Pi
SaaS	Software-as-a-Service
SQL	Structured Query Language

USB	Universal Serial Bus
VM	Virtual Machine
YAML	Yet Another Markup Language

1 Introduction

1.1 Data-intensive Applications

Data-intensive computing is defined as collecting, managing, analyzing and understanding data at volumes and rates that push the frontier of current technologies[1].

Many applications today are data-intensive. Raw CPU power is rarely a limiting factor for this kind of applications – bigger problems are usually amount of data, complexity of data, the speed at which it is changing and the fact that data generated at one location often needs to be sent and processed somewhere else.

Increasingly many applications now have such demanding or wide-ranging requirements that a single tool can no longer meet all of its data processing and storage needs. Instead, the work is broken down into tasks that can be performed efficiently on a single tool, and those different tools are stitched together using application code.

Taking this into account, a new concept of tools was born to satisfy these new needs – mashup tools. They allow visual, interactive modeling way of connecting the data flow between various tasks in order to achieve the common goal. Typical example of mashup tool is Node-RED development environment, based on NodeJS.

Data-intensive application is typically built from standard building blocks which provide commonly needed functionality (tasks) [1][2].

These tasks include different operations over the data [1]:

- acquiring the data (from different sensors, devices, social networks, various types of datastores – relational databases, NoSQL databases)
- storing the data (into different types of datastores)
- processing the data (both locally or using external Cloud services)
    - batch processing, where we periodically process a large amount of accumulated data

[1] Definition by Pacific Northwest National Labs

    - stream processing, where we have a continuous computation that happens as data is flowing through the system
- managing the data
- filtering the data (looking for instances satisfying a given condition)
- viewing the results

The relation between Internet of Things (IoT) and data-intensive application concepts is of utmost importance, as IoT devices play an important role in such applications. The basic idea of IoT vision is the pervasive presence of a variety of things or objects – such as RFID tags, sensors, actuators, mobile phones, single-board computers, embedded and wearable devices, etc. producing a data and able to interact with each other in order to reach common goals. IoT-based systems are growing in functionality, size and management complexity and systematic software-engineering approach towards building IoT systems is becoming more and more relevant [2].

There are many factors that may influence the design of a data system, including the skills and experience of the people involved, legacy system dependencies, the time scale for delivery, organization's tolerance of different kinds of risk and regulatory constraints. Those factors depend very much on the situation.

Three concerns that are important in most software systems, including the data-intensive applications are [1]:

- *Reliability:* The system should continue to work correctly (performing the correct function at the desired level of performance) even in face of adversity (hardware or software faults, and ven human errors).

- *Scalability:* As the system grows (in data volume, traffic, complexity), we should have reasonable strategy of dealing with it.

- *Maintainability:* Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should be able to work on it productively.

Keeping this in mind, data-intensive applications and the underlying infrastructure and architecture should be constructed in a way to enable flexibility and adaptivity to environment conditions and changes, while not preventing us from focusing on the main goals of our applications and business logic.

1.2 Motivation

Countless number of networked automated devices with ability to connect, communicate, collect information and share it via Internet are present in our environment, being developed and their number is still increasing every day.

Data is being generated at ever-increasing rates in all domains where information technology is used. This is caused, not only by the increasing number of devices pervasively generating data, but due two other factors [1][2]:

- high-resolution instruments and sensors (such as scanners, cameras, gene sequencers, microscopes, etc.)
- fast, but affordable processing power (which leads to higher resolution models and multi-disciplinary optimizations, taking into account various types of data at the same time)

Therefore, data-intensive applications require larger data volumes and devote most of their processing time to I/O and manipulation of data. Amount of data is, generally too big to fit in memory. Also, it is often necessary to transfer large amount of data from one location to another as it might happen that one application is gathering data, while another one is processing it. Having the data at the right place at the right time is crucial in this kind of applications. Thus, a proper design and execution of data-intensive applications are more and more important aspects.

There are two major issues becoming more and more relevant, when it comes to computing IoT-generated data [3]:

- *latency*
    - the processing time of time-critical IoT applications can be limited by the network delay for offloading data to Cloud

  - uploading data from a large number of IoT data-generating devices could induce network congestion, producing even further delay

- *privacy/security:* in more sensitive domains, like healthcare and clinical systems, there are privacy and security issues related to data movement, constrained by legal regulations which prevent certain type of data from uploading to Cloud, so some of the data has to stay inside the within the devices which are property of the institution using it, inducing that some of the processing has to be done locally.

To tackle these issues, there are two possibilities: to move the data or to move the computation. In this thesis, we focus on the latter approach.

The goal is to move tasks closer to the data generators and consumers. First, processing the data at the Edge can significantly reduce the network delay. Considering the second issue, it could be solved in the same way – by processing sensitive data at the Edge, legal and privacy issues related to data migration to the Cloud are avoided.

But, we should also keep in mind that in one system we could have different types of devices (even within the Edge). Therefore, another challenge is dealing with device heterogeneity [4], making possible data and computation task migration between traditional computers (x86/x64-based) and IoT devices (single-board computers, wearable and smart devices, which are in most cases ARM-based).

Thus, the challenge is to make flexible architecture and execution environment for the computation, which would allow us to move computation tasks od data-intensive applications between Cloud and Edge, according to the constraints imposed by previously mentioned issues – network delay and privacy, taking into account the heterogeneity of devices, but still letting the application developer focus only on the business aspects of the application [3], without taking care of location where is the task executed and spending as least time as possible on the infrastructure configuration-related problems.

For this reason, we focus our work on Fog Computing, that extends the Cloud Computing to the Edge of the network, closer to the things that produce and act on data including different types of devices, ranging from personal computers, laptops, workstations to IoT devices, wearable gadgets and sensors.

1.3 Research goals

The goal of this thesis is to explore state-of-the-art concepts and utilize the state-of-the-art technologies in order to develop a framework which gives ability to move computation tasks between Cloud and Edge in case of mixed architecture execution environment, as a way to deal with issues of privacy and latency in data-intensive applications.

In order to achieve this, a set of mechanisms needs to be designed and implemented covering various aspects of previously stated goals.

First, it is necessary to find an appropriate solution for moving tasks in Fog Environment, taking into account that they can be on different locations (Cloud or Edge) and have different computing architecture (heterogeneity).

Furthermore, it is necessary to provide a mechanism for selection of device where is the task going to be executed, according to the given criteria, by taking in the account the current state of the execution enviroment. In many cases, there are many devices which are candidates to execute the same task, but we want to select the most appropriate one at that moment, without breaking security/privacy policies and introducing larger network delay.

Finally, we should make our solution as convenient as possible for the application developers. They should not be distracted by the previously mentioned mechanisms from focusing on business aspects. It means that they should not take care where is the task really executed. For them it should be completely transparent and the original business logic of the application should remain unaffected by the fact that it is executed in Cloud or Edge; laptop, traditional personal computer, workstation or a single-board device.

1.4 Thesis structure

The first chapter introduces the concept of data-intensive applications, main issues, motivation of the thesis, research goals and thesis structure.

The second chapter presents state-of-the-art concept of Fog Computing and explaining its purpose, context, benefits and correlation with existing IoT and Cloud infrastructures.

The third chapter presents state-of-the-art concept of container-based virtualization and relevant technologies. The first part is focusing on theoretical aspects of this concept and comparison with hypervisor-based virtualization approach. The second part is focusing on relevant technologies, introducing Docker containers and two container management systems – Docker Swarm and Kubernetes. The final part of this chapter highlights pros and cons of the considered container-management system solutions in order to justify the choice of Docker Swarm.

The fourth chapter is providing an overview of Node-RED mashup tool for development of data-intensive applications and mentions the main features considered relevant for the implementation of our solution.

The fifth chapter presents the solution that was implemented as a result of the research carried out during the thesis. First, it provides high-level overview of the framework environment and then explains the implementation details for each of the components.

The sixth chapter presents the application implemented as a proof-of-concept based on a healthcare domain scenario in order to explain the capabilities of the implemented framework. The developed application is used for computation task movement experiments between different devices.

The final chapter concludes this document providing an overview of what has been done and discusses what are the features that are missing and which aspects should be further considered and extended in future and how.

2 Fog Computing

Fog Computing, oftenly also referred to as Edge Computing, is an emerging paradigm aiming to extend Cloud computing capabilities to fully exploit the potential of the Edge of the network where traditional devices, as well as new generations of devices – such as smart, wearable gadgets and mobile devices (the so-called "IoT devices") are considered [4]. It involves the research and application of enabling technologies that allow computation to be performed at the network Edge so that computing happens near data sources [8].

Moving all the computing tasks to the Cloud has been a traditional way to process data for years. The main reason for this is much higher processing capability of the devices in the Cloud, compared to the devices within the network Edge – which mostly consists of low-voltage, single-board computers and devices with relatively low processing capability. This is what the traditional Cloud computing consists of.

However, despite the rapid evolution of the data-processing speed, the bandwidth of the networks that carry data to and from the Cloud has not increased appreciably. Thus, with Edge devices generating more data – the network is becoming Cloud computing's bottleneck. In these cases, the processing time of time-critical applications could be limited by the network delay. Also, when we have a large number of devices sensing the information, uploading the data generated by them would induce additional network congestion, increasing this way the network delay, even further. In order to tackle network delay issues, we move the data processing closer to the data generators and consumers [9].

The concept of enabling a computer to sense information without any human intervention was applied to many other fields – such as healthcare, home technology, environmental engineering and transportation. Because of this, the second major issue appears. Keeping the privacy and dealing with security of the information in more sensitive domains is crucial, in most cases regulated by law, and dramatically restricts freedom of data movement. Due to laws and regulations, in some cases the data is not allowed to leave the boundaries of the institution which is storing the data. Thus, in these cases, the data needs to be processed within the Edge, instead of being uploaded over vulnerable network to the Cloud [3][4].

In Fog Computing the Cloud collects data from existing databases, sensors, mobile phones and other devices as it has been done traditionally. But, the main difference between the traditional Cloud paradigm and fog computing is, that the device itself acts both as data consumer and data producer. Thus, requests between devices and Cloud are bidirectional instead of just one direction (from device to Cloud), as in the past - some of the devices are also able to perform even more demanding computing tasks: data processing, caching, device management, and privacy protection, in order to reduce traffic from devices to the Cloud [8]. For this reason a Fog Environment should build a continuum between the Cloud and Edge.

Considering the fog architecture, we can identify, in general, two types of resources. In what follows their overview and main characteristics are given [3][10]:

- *Edge nodes*
    - The resources which physically belong to the certain organization, including computers, laptops, single-board computers, networking hardware and different types of sensors and devices within its boundaries, residing between data sources and Cloud-based datacenter.
    - Receive feeds from IoT devices using any protocol, in real time.
    - Run IoT-enabled applications for real-time control and analytics
    - Typically provide transient storage, often few hours.
    - Send periodic data summaries to the Cloud.
- *Cloud nodes*
    - The resources which are rented from various service providers, mainly used for Cloud data storage (database servers), virtual machine or data analysis and processing tasks (as described in *Cloud overview sub-section*).
    - Receive and aggregate data summaries from many Edge nodes.

- Perform analysis on the IoT data and data from other sources to obtain business insight.

- Can send new application rules to Edge nodes based on these insights.

Developers either port or write IoT applications to support execution on the network Edge. The Edge nodes ingest the data from IoT devices, and then direct different types of data to the optimal place for analysis, taking into account network congestion, execution time requirements, privacy and security issues.

There are many scenarios where the coordination between Edge and Cloud is utilized. For example, the most time-sensitive data is analyzed on the Edge node closest to the things generating the data. Data that can wait seconds or minutes for action is passed to aggregation node for analysis and action. Data that is less time-sensitive is sent to the Cloud for historical analysis, big data analytics and long-term storage.
Data sensitive to privacy issues and whose movement is constrained by legal regulations is also analyzed and processed within the Edge of the network.

Main advantages obtained using Fog architectures are [10]:

- *Lower operating expense:* Conserve network bandwidth by processing selected data locally instead of sending it to the Cloud for analysis

- *Better security:* Protect your fog nodes using the same policy, controls, and procedures you use in other parts of your IT environment. Use the same physical security and cybersecurity solutions.

- *Privacy control:* Analyze sensitive data locally instead of sending it to the Cloud for analysis.

- *Moving data to the best place for processing:* Which place is best depends partly on how quickly a decision is needed. Extremely time-sensitive decisions should be made closer to the things producing and acting on the data. In contrast, big data analytics on historical data needs the computing and storage resources of the Cloud.

In recent years, smartphones and single-board computers have become powerful enough to handle various tasks, as their specifications are quite close to personal computers which were widely used less than a decade ago. Therefore, Fog Computing is becoming more and more relevant topic, as the growing processing capability of Edge devices is providing new horizonts to exploit the Fog-based architectures in various domains and contexts.

The concept of Fog Computing is relevant to this thesis, as we want to enable the flexibility of data-intensive applications by providing the ability to move tasks between Cloud and Edge, according to the constraints imposed by legal regulations on the first place, taking into account the security, privacy, network-related issues and the device heterogeneity.

In the rest of this chapter, an overview of Edge and Cloud Computing devices is given.

2.1 Edge overview

Within the network Edge, we can have different types of devices – from traditional personal computers, laptops, workstations to IoT devices. IoT devices are becoming the main focus of the Edge Computing due to their increasing power in recent years.

The Internet of Things (IoT) is the emerging network infrastructure that is gaining considerable attention in modern telecommunications and information technology. This novel paradigm consists of physical objects (or things) which are able to interact with each other and with users, thus providing communication and exchanging data over Internet. The rapid growth of wireless technologies and the Internet in recent years have made IoT an integral part of scientific research [5].

The main idea behind this concept is to make communication even more pervasive and agile, without any requiring human-to-human or human-to-computer interaction. As a result of this phenomenon, the objects of everyday life, such as vehicles, buildings, home appliances, jewelry, wearable watches and other objects are becoming equipped with micro-controllers, sensors, displays and electronic chips. These chips also contain distinctive identifiers and have the capability to send/receive specialized signals.

Moreover, by enabling network access, the objects can be used for the development of society, causing a whole new set of applications across many domains and areas, such as traffic management, home automation, healthcare, industrial automation and many others.

In general, the goal of IoT is to connect objects anywhere, anytime with any network in order to provide ease of access and high availability.

The term Internet of Things is composed of two words. The first word focuses on more on the network-oriented domain, while the second word gives the idea that any generic object from our environment could be integrated with one or more networks, to make a network of connected things, in order to achieve some goal.

In recent years, the computers used for IoT applications started becoming more and more powerful. One of the most widely used devices for this purpose is a credit card-sized, single-board computer named Raspberry Pi. The latest revision of this device is Raspberry Pi 3B[2], released in February 2016. It contains 1.2 GHz quad-core ARM Cortex-A53 processor and 1GB of RAM, which is quite impressive for such a small device. It also has Bluetooth and WIFI support, USB ports, Ethernet port and GPIO pins for communication and attaching devices providing new features, like displays, proximity sensors, cameras and many others. The device is originally designed to run Linux distribution for ARM architecture devices called Raspbian, but there are more and more operating systems becoming available. Among them – there is even a version of Windows for IoT devices and other already famous Linux distributions – Cent OS, Ubuntu and many others.

The fact of having such power embodied in credit card-sized devices, has inspired both engineers and hobbyists around the world to experiment and find new fields and areas where they can be used. Therefore, many technologies, platforms, applications, libraries and compilers for Linux were ported to ARM. This heavily affected the use – in increasing number of cases where they started to get roles in many new tasks. So, it is becoming possible to use Raspberry Pi as a database server (MySQL, MongoDB), PHP server even inside the Docker container (as Docker Engine was ported to ARM versions of Linux).

---

[2] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

For this thesis, Raspberry Pi model 3B has been used as a representative of IoT devices, in order to demonstrate the capabilities of the developed framework, by moving tasks and processing to ARM-based IoT devices. Hypriot[3] OS 1.4.0 Linux distribution was used, as this operating system comes with Docker container engine support preinstalled.

2.2 Cloud overview

Cloud computing is considered as a concept which provides resources and services through the use of Internet and these services are distributed in Cloud. The word "Cloud" is a metaphor, which in fact describes a web where computation is pre-installed. All the applications, operating systems, storage and many other components reside in the web and are also shared with multiple users.

The Cloud Computing is defined as a collection of inter-connected distributed and parallel virtualized computers which are dynamically provisioned and presented as one or more unified system based on service level agreements established through negotiations between the service provider and the consumer [6]. In a broader sense, the Cloud is considered as a data center in which computer nodes are virtualized as Virtual Machines (VMs).

The main reason for using Cloud computing is to solve large-scale problems by properly utilizing the resources and services distributed in the Cloud. Users do not know the location of resources, as they are transparent and hidden; resources can be shared with multiple users, and are easy to access at any time.

Since Cloud computing is growing rapidly, it is further divided into three service models for utilizing the overall computing power [7]:

- *Infrastructure-as-a-Service (IaaS):* It is the simplest service of Cloud computing, which basically allows us to access the computing resources remotely. It provides processing, block storage and network resources for computing large problems. Amazon Elastic Compute Cloud (EC2) is the most common example of IaaS.

[3] https://blog.hypriot.com/

- *Platform-as-a-Service (PaaS):* The service offers a development environment and has some sets of application interfaces, which are used to develop and run software programs. It provides database, web server, operating system, run-time environment, storage and many other resources as a computing platform. Microsoft Azure and Google App Engine are two key examples of PaaS.

- *Software-as-a-Service (SaaS):* This is the latest service model which enables users to access databases and application software. End users can easily consume this software, since it is deployed over the Internet. It is considered as pay-per-use service and also, sometimes – requires a subscription fee. Google Docs is a typical example of document editor offered in form of SaaS, while Mlab[4] and MySQL Free Hosting[5] are examples of SaaS-based databases, and were used during the framework development in this thesis for experiments and proof of concept.

Understanding the concept and the existing categories of Cloud Computing is necessary for this thesis, as the framework needs to handle the movement and execution of data-intensive application tasks in Cloud.

---

[4] https://mlab.com/

[5] https://www.freemysqlhosting.net/

3 Container-Based Virtualization

3.1 Concept

Considering the current state-of-the-art virtualization techniques, we can notice that another concept co-exists beside the traditional virtual machine-based virtualization in this domain – the container-based virtualization concept.

Containers could be described as a lightweight virtualization approach that creates virtual environment at a software level inside the host machine, also known as operating system-level virtualization. It removes the overhead of using hypervisors by creating virtual machines in the form of containers (act as guest systems) thereby sharing the resources of the underlying host operating system. It provides different levels of abstraction in which a kernel is shared between containers and more than one process can run inside each container. This way, the whole system can become more resource-efficient as there is no additional layer of hypervisor, and thus no full operating system which can occupy a lot of storage space for each virtual machine [11][12][13].

Unlike virtual machines, where we run a workload – the complete operating system which can run many applications, on the other side - containers are used for running applications. We do not create a general-purpose container and then install the applications how we do in a virtual machine. In this case, we create a container that is specifically set up for a particular application or microservice. An application might easily use a dozen of containers to provide the microservices it is composed of. A container does not have its own copy of the operating system the way a virtual machine does – it has an instance of the same operating system host is running. That is why containers can be created so quickly, compared to virtual machines. Virtual machines start via a full boot-up process. To the container, the disk looks like a brand-new copy of the OS that is just booted, but, in reality, we have the namespace isolation that gives each container its own, virtual namespace with access to files, network ports, and its own list of running processes. This makes containers very efficient for deploying similar applications that can use the same kernel code – providing almost native performance without additional hypervisor overhead [11][12].

Using container-based virtualization, we can create a huge number of virtual instances on a single host machine, as the kernel and other libraries are shared

between those instances. This causes the overall system to require less disk storage as compared to hypervisor-based solutions [12].

Figure 3.1 presents the architecture of hypervisor-based virtualization systems (Type 1 and Type 2 hypervisors), while Figure 3.2 displays the case of container-based virtualization systems.

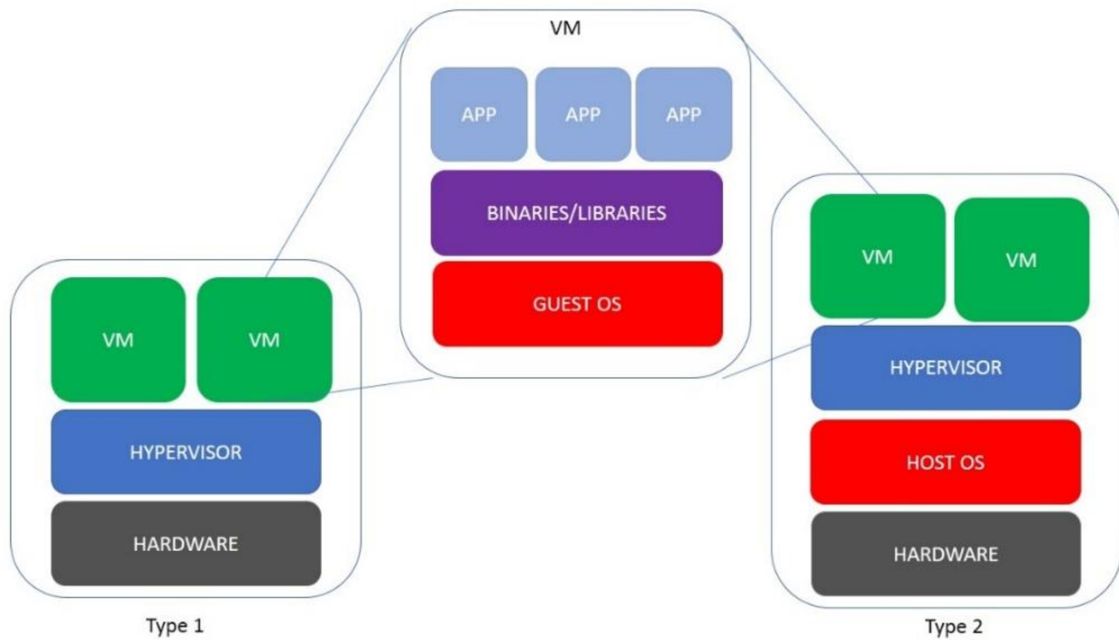

Figure 3.1 – Hypervisor-based virtualization architecture

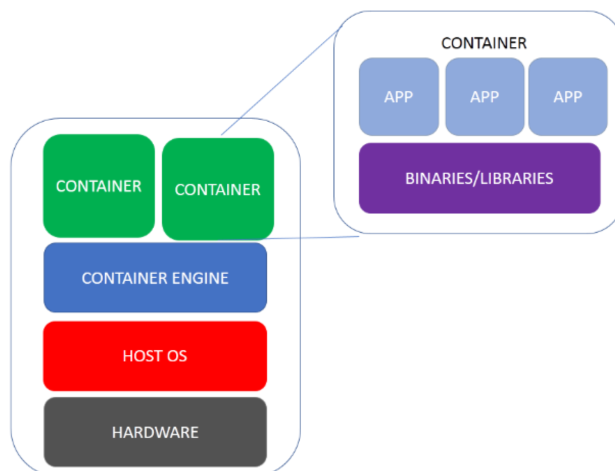

Figure 3.2 – Container-based virtualization architecture

But, in comparison to traditional virtual machines, there are some drawbacks : we cannot mix and match operating systems with containers the same way we can with virtual machines. For example, we can run a Linux virtual machine on Windows using Hyper-V, or Windows virtual machine on a Linux host, but if we create a Linux container – it is not going to run on Windows, unless we set up a Linux virtual machine with container engine running in which we would have to put the container [14].

Also, the security issues should be taken into account. A container is not a security boundary in the same way that a virtual machine is. Containers are independent views of an OS, all sharing the same kernel. This practically means that if the process in a container is made to hack the kernel, it can theoretically affect the while system. However, kernels usually include configurable security-oriented modules [14].

To sum up - although being considered as a new, lightweight way to perform virtualization, containers are not able to fully replace virtual machines. Compared to virtual machines, they do not have the hypervisor layer that usually provides a complete abstraction of the underlying architecture, enabling execution of any operating system within each virtual machine. Because of this, the way we work with them and think about them is different.

While virtual machines are platforms for isolating entire operating systems; containers are for isolating user spaces – they isolate applications and all the associated files and services but not the whole OS. With containers, we can have a single OS hosting multiple applications – where each of them thinks it has its own OS – but it actually does not.

In what follows, the comparison of hypervisor and container-enabled kernel is summarized [11[[12][13](Table 3.1).

Parameters	Hypervisors	Containers
Operating System (OS)	Runs full operating systems on top of a hardware (Type-1 hypervisors) or a host OS (Type-2 hypervisors) and kernel is assigned individually to	Each container runs on top of a host operating system with kernel and other hardware resources shared

	each VM with full hardware resources	between those and guest operating systems
Startup time	It takes few minutes to boot virtual machine	Containers can boot up in few seconds depending upon the system specifications
Storage	Hypervisor-based virtualization takes much more storage because whole system components have to be installed and run including kernel	The storage is less as compared to VM because OS is shared and so is the kernel
Performance	There is an additional layer of hypervisor software which degrades performance as it takes time to translate machine instructions from guest OS to host OS	Only the container engine is needed between the host OS and containers, which is much lighter than full OS, so the containers provide almost native performance
Flexibility	Designed for generality, can theoretically emulate any architecture.	Designed for minimalism and speed.

Table 3.1 – Hypervisor and container-based virtualization comparison

But, here comes one more question: "Can virtual machines and containers co-exist?" The answer is - yes. At the most basic level, virtual machines are a place for running container engines. It can be any virtual machine – either in Cloud or virtual machine deployed on a local computer. Also, running our application inside a set of containers does not preclude it from talking to services running in a virtual machine. For example, our container-based application needs to interact with a database that resides in virtual machine. Provided that the right networking is in place, our application can interact with database seamlessly.

For this thesis, the virtualization techniques compatible with IoT devices (more concretely, Raspberry Pi) were considered. When it comes to virtual machines,

we can conclude that they are too heavy task to be handled by these low-power devices.

Considering the concept of container-based lightweight virtualization, it seems that it is much more IoT-device friendly. But, we know that their CPUs are ARM-based, so it was necessary to find out if there are ARM architecture ports of necessary components (container engine and container images).

Fortunately, there is a fully-functional Docker container engine port for Linux-based ARM operating systems compatible with Raspberry Pi. Therefore, container-based virtualization approach based on Docker Engine was selected for this thesis.

The goal of using containers is to pack inside them the components necessary to run various tasks of data-intensive applications and easily move them between devices within Edge and Cloud in mixed architecture environment.

3.2 Technology

The goal of this section is to present state-of-the-art technologies for container-based virtualization which were used and considered relevant for the framework development.

The first part introduces Docker technology based on Linux containers and Docker Engine architecture. Several important aspects of Docker containers are considered separately – data in Docker containers, building Docker images and image portability.

The second part presents and compares two container management systems for clusters running container-based applications – Docker Swarm and Kubernetes, from the perspective of their architecture and offered features.

The third part presents Node-RED mashup tool which was actually used for the framework implementation. Also, in the end, the key features which were relevant for the framework and which actually were the reason for decision to use Node-RED as a development environment are discussed.

3.2.1 Docker

Linux Containers

Docker[6] is an open-source, container-based platform for developers and system administrators to develop, ship and run applications. Docker adopts the concept of simple containers which are used to ship goods and items from one place to another, but instead of shipping goods, it ships software. It is mainly created in Go programming language, and originally, on top of libvirt and LXC – later replaced by unifying library - libcontainer which was also written in Go, built on cgroup and namespacing capabilities of the Linux kernel. Docker containers belong to the category of Linux containers [15].

Linux container (LXC)[7] is an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single control host. They are implemented using two important kernel features: control groups (*cgroups*)[8] and *namespaces*[9], which are basically used to provide isolation within containers and between the host system and containers. In what follows, these two kernel features are going to be described.

Control groups, also known as cgroups, are one of the main features of kernel which allow users to allocate or limit resources among groups of processes. These resources include certain features such as CPU time, system memory, disk bandwidth, network bandwidth and monitoring. The cgroups features provide efficient resource allocation between containers and the applications running inside them. For example, if an application requires two processes with different resource usage, then that application is divided into two groups. Each group has a separate profile based on cgroups resources and the process run inside their own dedicated group. This, in turn provides isolation between processes and there is no interfering between groups [15].

Namespaces are the second important kernel feature which provides per-process isolation. This feature ensures that each process has a dedicated namespace in

[6] https://docs.docker.com/engine/docker-overview/
[7] https://en.wikipedia.org/wiki/LXC
[8] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html
[9] https://lwn.net/Articles/531114/

which it can run freely, without interfering with other processes running inside different namespaces. There are currently six namespaces inside Linux, that are described as follows [16]:

- `mnt:` Deals with filesystem mount points. This feature allows processes to have their own distinct filesystem layout, providing this way isolation on set of filesystems.

- `uts:` Isolates two system identifiers called "nodename" and "domainname". This feature allows each container to have their own hostname different from other containers.

- `pid:`This feature allows processes to have their own identifiers (IDs). These IDs are different for every process within the same namespace, but can be same in different namespace.

- `user:` User namespace are used to achieve isolation based on user and group IDs and allows this way processes to have different user IDs.

- `net:` Network namespaces isolate network resources by allowing containers to have their own routing devices. Each network space contains separate routing tables, iptables, firewalls, network interface controllers and other network-related items.

- `ipc:` This feature isolates the inter-process communication resources, such as POSIX message queues, between multiple namespaces.

Docker Engine architecture and components

Docker is based on client-server model. Docker client sends requests to the Docker daemon, which will then create, launch, or distribute containers. Both components (client and daemon) run in most cases on the same host, but it is also possible to run daemon on some remote host and then connect it with the client. Both the daemon and client use sockets or RESTful APIs to communicate with each other. The overview of Docker architecture is shown in Figure 3.3. It is composed of the following core components [17][18]:

- *Docker daemon:* The main functionality of Docker daemon is to manage containers running on a host system. As can be seen from the architecture in Figure 3.3, daemon is running on the host machine and client is used to communicate with that daemon. Users cannot directly interact with the daemon. Both client and daemon use the same Docker binary, as they are located on the same host.
- *Docker client:* Docker client is a command line interface for users to communicate with Docker daemon. It takes commands from users and then interacts with the daemon.

- *Docker images:* Images are the main building block of Docker. They are also considered as a source code for running containers as Docker launches containers form these images.

- *Docker registries:* Registries are used to store images. They have two types: public and private. Public registries are often known as Docker Hub[10]. We need to create an account in order to access the hub and store our own images. It is considered as an extremely large storage system for uploading new images and downloading existing images which other people have created and uploaded. For example, we can easily find a web-server database image and then store it into Docker Hub. By using private registry. We can provide security to our images so we can only share them within the boundaries of our organization.

- *Containers:* Finally, container is the last in the series of Docker components. It was already stated that containers are created and deployed using images which also contain some applications and services, and one or more processes are also running inside a single container. In other words, we can say that images are considered as the building feature of Docker and containers are considered as the execution feature of Docker. Each container has a single software image running inside it and has the ability to perform sets of operations such as create, start, stop delete and move containers.

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

As it is visible in Figure 3.3, the scenario of usage is the following – the user types a CLI command which is accepted by Docker client and forwarded to the daemon. After that, the command is expressed in terms of Docker API calls in order to run the container from image. Docker Hub is used as a registry and images are obtained from this location if they are not locally available.

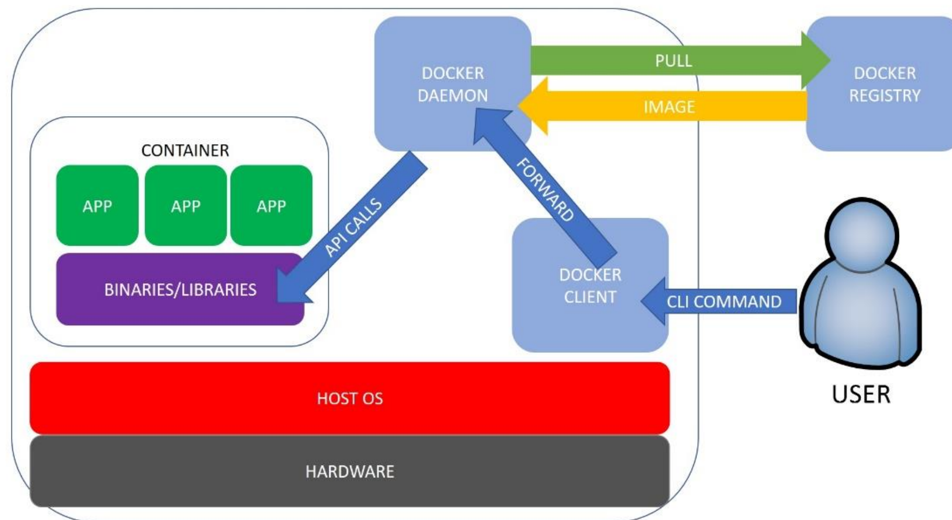

Figure 3.3 – Architectural diagram of Docker

Docker Images

In Docker, images are to containers what classes are to instances in OOP - their archetypes.

An image consists of:

- File system, which is isolated
- Process-related meta info (the default process to execute when a container is created from the image, or its command-line arguments)
- Network-related meta info – containing the ports to be exposed
- File system meta info -in terms of data volumes

Every container must be instantiated form an image. In other words, it encloses a program within the image's file system.

Building Docker images

We can build our own images from scratch, update the existing images or download images created by other people. Hence, they are portable and easy to create, share and store.

New images are, in most cases created from existing, following a single-inheritance taxonomy of arbitrary depth. All images are usually created from the images of well-known Linux distributions, such as Ubuntu, Debian or Fedora.

Also, the images could be created from scratch but it is less common. Starting from an existing image is easy and requires no significant overhead, thanks to the file system employed by Docker images, called the Union File system[11].

Docker images are created using multiple layers of instructions. For example, the instruction could be to run some specific command, add a file into a directory, setting the environment variable or exposing a port for communication. These layers are combined together with the help of Union File System to make a single usable image layer. Union File System helps combining the files and directories of separate file systems into single coherent file system. Due to its layered architecture of images, Docker technology powered container-based virtualization is considered as a light-way virtualization as compared to hypervisor. As an example, if we update an image or if there is a single parameter change, then the additional layer is built which is then combined to the existing layers of image. Therefore, there is no need replace/rebuild the image entirely as done in traditional virtual machines and the process of image distribution is done by just updating and distributing the existing images with changes being added. In what follows, an example of building an image from existing base image is provided [19].

For example, let us assume that user wants to create a container-based application running specific PHP scripts. First, the user locates the base image which corresponds to PHP server. Then, the user has to specify which are specific scripts to be executed (.php files) and include the host directory which is containing them inside image specification. In order to specify the previously described parameters, it is needed to create a DockerFile. Listing 3.1 provides an example of Docker file.

[11] https://en.wikipedia.org/wiki/UnionFS

```
FROM vicviper/apache-php
COPY src/ /var/www/html/
```

Listing 3.1 – Example of DockerFile for building Docker image whose container running a PHP server

With keyword FROM we include an image available locally or present on Docker Hub repository which serves as a base image for building a new one, while keyword COPY is used for copying the source directory of the host to the destination directory of the container. In our case, it means that directory named "src" actually contains the scripts which we want to include inside the PHP server. Therefore, we copy the contents of this directory from host to the default PHP server directory inside the container.

Once the DockerFile is created the following command is executed in order to build an image:

```
docker build -f DockerFile .
```

Image portability

The next step is to explore between which devices and under which constraints is possible migration of Docker containers – which are conditions to be fulfilled. This is relevant for the thesis, as we need to implement a mechanism which would allow us to move tasks from one device to another, taking into account their heterogeneity. Therefore, it is necessary to understand what are the limitations and constraints of Docker container portability.

Generally, these cases of Docker container migration are going to be considered:

- between machines running different OSes on same architecture
- between machines based on different architectures

Before that, an overview of currently available versions of Docker Engine is going to be given.

Originally, Docker was based to run natively on Linux kernel, so it was possible to move containers without any problems between x86/x64-based Linux computers. Later, the Docker Toolbox came out – for both Windows and Mac OS. But, Docker Toolbox is not running on native kernels of the previously mentioned operating systems. Actually, Docker Toolbox was based on virtual machine running lightweight Linux distribution, inside which was actually the Docker engine itself, so there was still the hypervisor overhead (Virtual Box was running behind). Docker Toolbox is still the only option for machines running older versions of operating systems [20].

Later, since the Docker 1.11.0 came out, it became possible to run it natively on Windows 10 or newer (Hyper-V enabled) and Mac OS X 10.10.3 or higher. But, there is still a huge drawback, even when we have native Docker engines – we still have no ability to mix and match operating systems with containers the same way we can with virtual machines. For example, we can run a Linux virtual machine on Windows using Hyper-V, or Windows virtual machine on a Linux host, but if we create a Linux container – it is not going to run on Windows, unless we set up a Linux virtual machine with container engine running in which we have to put the container [20].

And finally, the most interesting case – running the Docker containers on ARM-based devices running Linux (Raspberry PI) and moving them to other devices with other architectures. As it was previously mentioned – Docker Engine for ARM-based Linux is available[12]. So, moving Docker containers from one to another Raspberry PI is not a problem. But, what happens if we want to run container built for ARM on device based on another architecture. Of course, it is not possible to be done natively, even in the case of Linux for x86/x64.

Introducing QEMU[13] – it is possible to run an ARM container, which was built on Raspberry PI, on Windows and Mac OS X, which gives ability to emulate ARM architecture on x86/x64. And, one very important thing – we don't change the original image to get this done. As long as there is the right QEMU interpreter called „qemu-arm-static[14]" placed in the ARM container, this container can be executed, or better – emulated even on a foreign architecture.

---

[12] https://blog.hypriot.com/getting-started-with-docker-on-your-arm-device/
[13] https://en.wikipedia.org/wiki/QEMU
[14] https://github.com/sedden/docker-rpi-raspbian-qemu

It has two modes of emulation – system emulation and user emulation. In system emulation we will have that the emulated system will behave like a virtual machine with its own emulated kernel. For our container use case – it means having to spin up a VM for every container and does not sound appealing. Fortunately, user mode emulation is a much better solution. In that mode, QEMU will run the binary code of foreign architecture as a host process – and at the same time translate ant guest system calls to host system calls. Therefore, we won't need a new VM for each container. Of course, in both situation we will have large decrease in performance as QEMU introduces significant overhead [21].

Considering the opposite case - due to lack of processing power is not possible running x86/x64-built container on ARM-based device.

Finally, an overview in form of a matrix summarizing the portability (Table 3.2 below) is going to be given.

Build/Run	Windows	Mac	Linux	Linux ARM
Windows	YES	VM	VM	NO
Mac	VM	YES	VM	NO
Linux	VM	VM	YES	NO
Linux ARM	QEMU	QEMU	QEMU	YES

Table 3.2 - Docker container portability matrix

In the left column, build environment is given, while in the top row, we have the running environment. The rest of the table contains the conditions of Docker container migration:

- YES (means that it is possible without any modification to the image)
- NO (means that it is not possible)
- VM (means that it is possible running in virtual machine of the operating system which was used to build the image)
- QEMU (means that QEMU is needed in order to run)

As our goal is to perform data and computation task movement without introducing further performance decrease and delay, considering the low performance of the QEMU, we decided not to use it.

Performance-wise, in order to utilize the most of the potential offered by our devices, the strategy is to use keep different versions of images of a certain task inside the repository (Linux ARM and x86, for example), so the corresponding image can be downloaded by corresponding device with matching architecture, based on tags or image names.

Figure 3.4 illustrates the Docker image lifecycle.

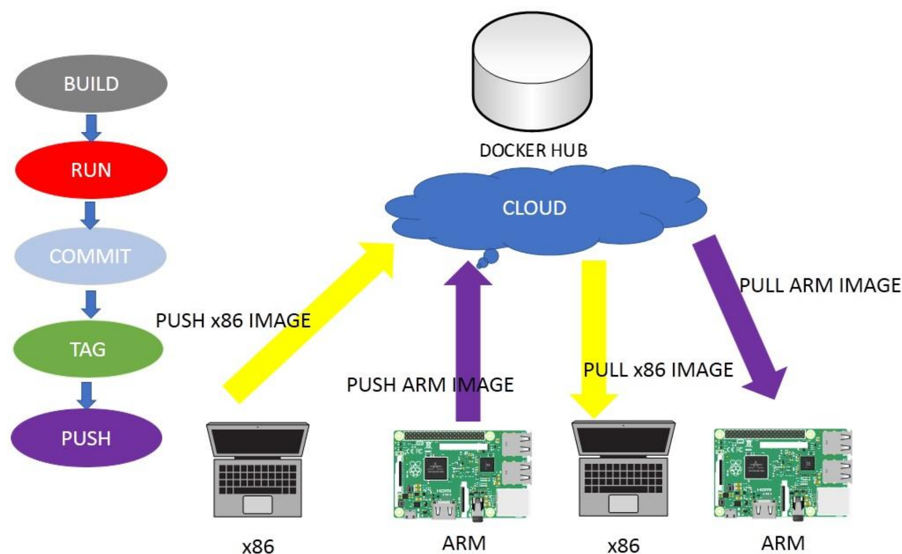

Figure 3.4 – Docker image lifecycle

First, the image is built. Then, it is run. By committing, we save the changes. Then, the image is tagged in order to be prepared for pushing to the Docker Hub repository. Once it is pushed to a public Docker Hub repository, it is available to be pulled by other machines. By using tags, we can specify which version of the image to download – ARM or x86 version.

Data in Docker containers

When it comes to containers, in general, we can make distinction between two different types of data inside them:

- *Ephemeral data:* When a container is removed, this data is lost

- *Persistent data:* When a container is removed, this data is still visible

In most of the cases, having persistent container data is not necessary (for example - container applications dealing with computation tasks) – as the final results produced are handled and stored outside the container and they only need ephemeral data for intermediate results. As containers were initially designed as a stateless concept, they only supported ephemeral data in the beginning. It practically means after container is destroyed all the created data inside is lost.

But, taking into account that more and more applications are becoming containerized, among container-based applications we often have database servers, which hold data. This data is important to keep even after the container is stopped or removed – so it belongs to persistent data category. This affected the initial concept of stateless containers, so container-oriented technologies started introducing the concept of persistent data inside containers. Considering the persistent data inside containers and its movement become highly relevant issues related to container-based applications.

There are many situations in which the organization needs to perform data movement:

- *Business change:* Acquisition and merging of the business unit

- *Workload balance:* Improve efficiency, performance and scalability of apps

- *Hardware/system upgrade:* Adapt new changes if technology, updates, practices etc.

- *Cost cutting:* Cut down the cost of data movement

- *Fault:* Network or disk failures

As we can see, all of these situations should be considered and taken very seriously, as data loss could be critical for the organization – leading to huge financial consequences. Thus, understanding the mechanisms of persistent data within containers and finding appropriate way to perform data migration is of huge importance, as the number of information system components run as containers in increasing.

When it comes to Docker containers are also two different concepts related to data[22]:

- *Union file system:* Holds ephemeral data

- *Container volume:* Holds persistent data

When a container is created, its file system can be thought of as an additional, empty layer on top of the stack: every modification to the file system is then stored in such layer – private to the container.

A data volume is a specially-designated host directory within one or more containers that bypasses the Union File System. Data volumes are native mechanism in Docker Engine providing useful features of persistent or shared data.

Important properties of Docker container data volumes are [22]:

- can be shared and reused among containers
- changes to a data volume are made directly
- changes to a data volume will not be included when we update an image
- data volumes persist even if the container itself is deleted
- data volumes are designed to persist data, independent of container's lifecycle
- Docker never automatically deletes volumes when we remove a container, nor will it "garbage collects" volumes that are no longer referenced by a container
- backup, restore and migration could be performed
- multiple containers can also share one or more data volumes

The mechanisms of sharing and moving the persistent data between containers works flawlessly in the case of containers running within a single host, while it is not possible to achieve the same in case of containers running on distinct hosts.

3.2.2 Container management systems

Container management system provide ability to schedule, manage and monitor container-based applications running across number of hosts [17][18][23].

When it comes to Docker containers, two most popular technologies which belong to this category are Docker Swarm and Kubernetes.

Kubernetes is based on Google Borg[15], and has adopted many of its ideas, while Docker Swarm is native Docker Engine solution available starting from version 1.12.0. While these two technologies can be used as alternatives for similar purposes, there are some aspects where one of them takes advantage.

In what follows, Kubernetes and Docker Swarm are going to be presented, highlighting their main concepts, ideas, features and architecture in order to be able to understand how they can be used for the framework that is going to be developed, compare them and figure out which one is more suitable for that purpose.

Docker Swarm

Docker Swarm[16] is a native Docker Engine clustering, orchestration and scheduling tool for containers. It is also considered as a container management system, as it gives ability to establish, manage and monitor a cluster of Docker Engines across a number of distributed hosts as single virtual host running Docker Engine.

Each host in a Swarm cluster runs a Swarm agent and one host runs a Swarm manager. The manager will orchestrate and schedule containers on the hosts. A discovery service will find and add new hosts to the cluster. Therefore, we have a cluster in which there is one master node and all the communications are done through this node. Through this way, there is no need to communicate to each Docker engine, thus removing the overhead of accessing multiple hosts. The client

[15] http://blog.kubernetes.io/2015/04/borg-predecessor-to-kubernetes.html
[16] https://docs.docker.com/swarm/overview/

communicates with manager to execute commands and run containers on different nodes (workers) attached to the master node.

Figure 3.5 shows the Docker Swarm architecture. The swarm is initialized on master node, which runs the Swarm manager. Then, the swarm token is generated, giving ability to any host to join the swarm if it has the token. Docker Hub is used as a discovery service when it comes to Swarm tokens (for Docker images also), so an active internet connection is required [17][18] [25].

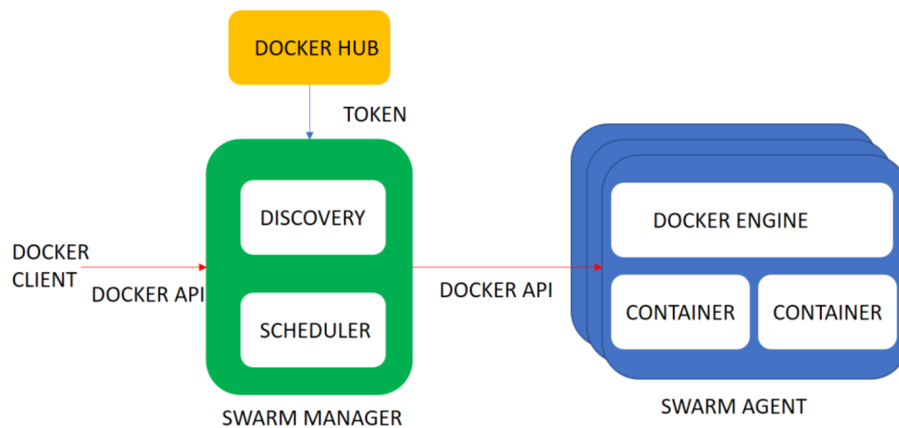

Figure 3.5 – Docker Swarm architecture

The main features of Docker Swarm include [24]:

- Launching a set of containers for a particular Docker image on the selected nodes, using filtering
- Scaling the number of containers up and down, depending on the load
- Health checks on the containers
- Performing rolling update of software across containers

Clustering is an important feature for container technology, as it creates a cooperative group of systems that can provide redundancy, enabling Docker Swarm failover if one or more nodes are experience an outage. A Docker Swarm cluster also provides administrators and developers with the ability to add or subtract iterations as computing demands change. Third party tools like Consul, Zookeeper or etcd are required to ensure high availability and failover to a secondary Swarm manager.

Swarm has five filters for scheduling containers [17]:

- *Constraint:* Also known as node labels, constraints are key/value pairs associated to particular nodes. A user can select a subset of nodes when building a container and specify one or multiple key-value pairs. These labels are very useful for our framework, as we need to perform node selection for task allocation, according to their location (Cloud/Edge) or computing architecture (ARM/x86).

- *Affinity:* To ensure that containers run on the same node, this filter tells one container to run next to another, based on an identifier, image or label.

- *Port:* With this filter, ports represent unique resource. When a container tries to run on a port that is already occupied, it will move to the next node in the cluster.

- *Dependency:* When containers depend on each other, this filter schedules them on the same node.

- *Health:* In the event that a node is not working properly, this filter will prevent scheduling containers on it.

Moreover, Docker Swarm follows the principle of "swap, plug and play". As an initial development, it serves the standard Docker API which enables pluggable back-ends which can be easily swappable with the back-end we prefer. Discovery service of master node is used to provide those back-ends. The service maintains a list of IP addresses inside the cluster which are assigned to each back-end service. This way we can allocate tasks to different nodes without having to change the configuration of our application each team we change the node executing the task. It is just enough to always send requests to the master (using correct port) and it is able to redirect requests to the right worker node.

Kubernetes

Kubernetes[17] is an open-source system for automating deployment, scaling and management of container-based applications. Kubernetes was built by Google,

[17] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

based on their experience running Linux containers in production over the last decade.

When it comes to the architecture of Kubernetes, we can make distinction between two types of components – master node and worker node components[18].

In what follows, these components are going to be explained. Figure 3.6 shows the Kubernetes architecture[19].

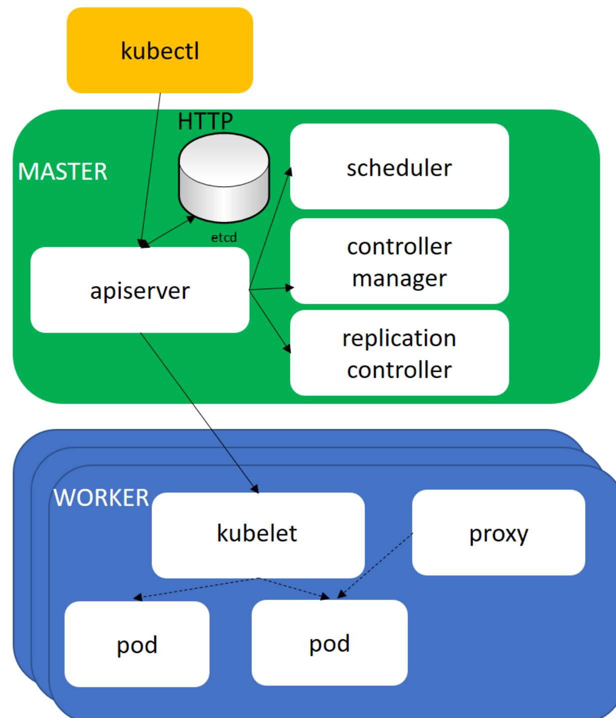

Figure 3.6 – Kubernetes architecture

Master node components

The Kubernetes master refers to a collection of processes managing the cluster state. Typically, these processes are all run on a single node in the cluster, and

[18] https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes
[19] https://github.com/kubernetes/kubernetes/blob/release-1.1/docs/design/architecture.md

this node is also referred to as the master. The master can also be replicated for redundancy and availability.

The Kubernetes master is responsible for maintaining the desired state of our cluster - the controlling unit of Kubernetes cluster, which serves as a main contact point for system administrators and users. It deploys, manages and monitors containerized applications on the worker nodes When we interact with Kubernetes, such as using the kubectl command-line interface, we are communicating with our cluster's Kubernetes master.

This is achieved with the help of few dedicated components that are described in what follows [17][18]:

- *API server:* A key component of the entire cluster. It allows users to configure organizational units and workloads for enabling communication to the other nodes in the cluster. This server also validates and configures data by implementing RESTful interface which includes API objects such as pods, replication controller and many others. AS separate client called kubectl is used along with other tools on the master side to connect users to the cluster. This gives user a control over the entire cluster.

- *kubectl[20]:* A command line interface for running commands against Kubernetes clusters.

- *Replication controller[21]:* A replication controller is a work unit for handling more complex version of a pod, which is known as a replicated pod. It is, in fact, a framework for defining pods that are mean to be horizontally scaled. A template is provided, which is basically a complete pod definition. This is wrapped together with further details about the replication work that needs to be done. The replication controller has responsibility over maintaining a desired number of copies. It ensures that a specified number of pod replicas are running which are continously monitored by Kubernetes controller. Unlike manually created pods, the replication controller pods are automatically started back if they fail, get deleted or are terminated in case of system

[20] https://kubernetes.io/docs/user-guide/kubectl-overview/
[21] https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/

malfunction. This means that if a container temporarily goes down, the replication controller might start up another container. But, if the first container goes back online, the controller will destroy one of the containers. We can consider rc pod as a process supervisor that is able to easily manage multiple pods across multiple hosts instead of just managing a single pod.

- *Controller manager :* The controller manager service is a daemon which is used to handle replication tasks defined by the API server. This service is built using a control loop whose purpose is to watch the changes in the state of the operations that are also stored in etcd side-by-side. Whenever a change occurs, the control manager reads the new information and moves the current state to the desired state. One of the controller examples is replication controller which is going to be described.

- *Scheduler :* It actually assigns workload to specific worker nodes in the cluster. This is primarily used to analyze the working environment, read the operating system environment and then place workload on acceptable nodes. It is also taking the responsibility of tracking the resource utilization on different nodes, and keep the track of the total hardware resources that are either assigned to processes or free to occupy.

- *etcd[22]:* etcd is a distributed, lightweight, reliable and consistent key-value data-store for shared configuration and service discovery across multiple nodes, and considered as one of the core components of the Kubernetes cluster. It provides a simple, secure, fast and reliable storage space for storing configuration data that can be used by the cluster in order to be used by each node of the cluster. This can be used for service discovery and represents the state of the cluster that each component can reference to, with reason to configure or re-configure itself. Etcd can be configured on a single master server, or, in production scenarios – distributed among machines. The only requirement is that it can be accessed from each machine in the Kubernetes cluster.

[22] https://coreos.com/etcd/

Worker node components

Each worker node requires a few components that are necessary to communicate with the master node and runs actual workload assigned to them. For networking, each worker node is also assigned a subnet which is used by the containers. In what follows, the components are described [17][18]:

- *Docker service:* The first requirement is to run Docker service on each worker node that is used to create containerized applications and perform dedicated work. Each unit of work is deployed as a series of pods and different IP address is assigned to each pod from the same subnet.

- *Kubelet service :* Kubelet is the central component of Kubernetes. Which is primarily used in order to manage pods and their containers on the local system. Kubelet service functions as a node agent and enables worker nodes to interact with the etcd store for configuration update, as well as to receive commands from the master node. It is also responsible for registering a node with the Kubernetes cluster and reporting resource utilization.

- *Proxy :* Proxy is running on each worker node which is used to make applications available to the external world. This service forwards requests to the correct container.

Kubernetes work units

Kubernetes deploys containers for performing user specific tasks in terms of workload. The deployment is carried in the form of work entities that are easily managed by Kubernetes. These work entities are [17][18]:

- *Pod[23]:* A pod is a basic unit of work that is created and managed by Kubernetes. It allows user to bundle closely related containers into one pod which then acts as a single application. Instead of running a single

[23] https://kubernetes.io/docs/concepts/workloads/pods/pod/

individual container, we can put together one or more containers and then deploy them as a pod. This in turn schedules all related containers into one host which shares the same environment and manages as a single unit. This means that they can also share IP space and volume directories. In other words, we can conceptualize pods as a single virtual host, including all the resources needed to carry out their application. If we take a look from the design perspective – a pod usually contains the main container (called „the front-end container“) and auxiliary containers („the back-end“), that facilitate related tasks by providing useful data. In our case, the main container would be the one holding Node-RED dataflow application, while the auxiliary containers can be used to hold database servers for the necessary data.

- *Service*[24]*:* A service is an abstraction that provides a policy to access multiple sets of pods containing distinct IP addresses. It usually uses label selector field to target specific pods and combine them in a set. While pods are ephemeral, they can easily die and they are not restored. When a pod is created, a unique IP address is automatically assigned to it, but it is not stable over time. For instance, we consider a pod which is created by the replication controller and multiple replicas are running. If, for some reason, one of replicas get terminated, then the replica controller will start another pod replica in order to balance the specified number of replicas. This causes a newly created pod to have a new IP address. This leads to a problem in which some sets of pods (front-ends) are unable to find other pods (back-ends) in the same set. Hence, Kubernetes service is created which keeps track of the pods that belong to the same set. In addition, Kubernetes also offers simple endpoint API which is used to access the service externally. The endpoints are updated whenever the set of pods in the service changes. This causes pods to provide communication between front-end and back-end pods quite easily.

- *Label:* A label is an arbitrary tag that can be placed on the above work units in order to mark them as a part of a group. These can then be selected for management purposes and action targeting. Labels are given as key-value pair. Each unit can have more than one label, but each unit can only have one entry for each key. For example, we can

[24] https://kubernetes.io/docs/concepts/services-networking/service/

stick with giving pods a „name“ key as a general-purpose identifier, or we can classify them by various other criteria, such as version, public accessibility etc. In many cases, we assign labels for fine-grained control, then we can perform selection based on a single or combined label requirements. In our situation, where we expect to have different types or architectures, we could make use of labels for marking the architecture (type=ARM or type=x86, for example) or the location tag (location=edge or location=cloud).

Kubernetes vs Docker Swarm

As it had been stated – Kubernetes and Docker Swarm are two most commonly used tools to deploy containers inside a cluster. Both are created as helper tools that can be used to manage a cluster of containers and treat all the servers as a single unit. However, they differ greatly in their approach.

Kubernetes is based on Google's experience of many years working with Linux containers. It is, in a way a replica of what Google has been doing for a long time, but adopted to Docker. In early days of Docker technology, using Kubernetes provided many additional features compared to what Docker Engine offered at that time. It could solve many of the issues Docker had – we could mount persistent volumes that would allow us to move containers without losing data, it introduced flannel to create networking between containers, it used etcd for service discovery. However, Kubernetes comes at a cost – as it uses different CLI, different API and different YAML definitions. In other words, everything needs to be done from scratch exclusively for Kubernetes. It has brought clustering to a new level, but at the expense of usability and steep learning curve.

On the other side, we have Docker Swarm using a different approach. It is a native clustering for Docker. The best part is that it exposes standard Docker API meaning that any tool we used before to communicate with Docker (Docker CLI, Compose) can work equally well with Docker Swarm. This is an advantage and disadvantage at the same time. While the ability the use familiar tools of our own choice is great, we are bound by the limitations of Docker API. If Docker API does not support something, there is no way around Swarm API.

Comparison

In what follows, several aspects of these two technologies are going to be compared and discussed.

Setting up Docker Swarm is easy, straightforward and intuitive. All we have to do is install one of the service discovery tools and run the containers on all nodes. As the distribution itself is packed as a Docker container, it works in the same way, no matter the operating system. We run the Swarm container, expose a port and inform it about the address of the service discovery. It can be even used without any service discovery tools, but in case of more serious usage, we can add etcd, Consul or some of the supported tools.

Kubernetes setup, on the other side is quite more complicated and installation differs from OS to OS and provider to provider. Each OS or a hosting provider comes with its own set of instructions, each of them having a separate maintenance team with separate set of problems. Whatever our needs are, it is likely that online community has the solution but in most cases happens that the instructions available are valid only for older version than the most recent which we are using, as changes are frequently introduced and configuration steps are also different. For Docker, we need just few commands to get everything working.

In Docker Swarm, everything works out-of-the-box, but we still have the option to substitute one component with the other. Unlike Swarm, Kubernetes does not have all the necessary components out-of-the-box, so we need to make many choices. These decisions can introduce further problems., especially when it comes to using ARM devices, as the documentation of ARM versions is not as complete as for x86 counterpart.

Kubnernetes relies on external, third-party technologies for load balancing and networking. For example, in order to expose a service using Traefik with Kubernetes, it is necessary to provide a .yaml configuration file specifying the parametes (such as service name, port). In Docker Swarm, only one command is needed to create and expose a service (CLI offers this functionality). Thus, a number of operations is reduced (creating .yaml file and redirecting the input paramters to the shell of the master node is avoided) [24].

From the experiments with Kubernetes in mixed architecture environment, it was noticed that both networking solutions - Weave and Flannel had issues.

In Weave, a problem was related to a proxy component (also a Docker container) which was stuck in crash loop for all the nodes which have different processor architecture in comparison to the master node. The reason for it is the following – all the cluster members downloaded the same version of the container as the master does and tried to run it. As we know that container compatibilty is constrained by the arhcitecture – it means that the reason for crashing was related to the fact that these nodes (ARM worker nodes, x86 master) tried to run containers which were not compatible, beacuase they were built for different architecture. Despite the efforts and spent time trying to overcome the issue by false tagging of the image, so the nodes can can download the right image matching their real architecture (not the master architecture), the trick did not work in practice, as the component kept crashing.

When it comes to Flannel, we were not able to set it up correctly as it required separate role definition configuration files for different architectures. Despite many workarounds and suggestions found online, none of them worked in the latest version of Kubernetes (version 1.6.4), as role defintion performed by applying configuration files (.yml and .yaml) has been introduced quite recently (since the version 1.6.0) we were not able to find the appropriate configuration file allowing mixed acrhitecture [24].

After many experiments and efforts, when it comes to Kubernetes, the final result was that mixed architecture cluster was not able to run properly. But, it was possible to run a cluster using devices with same architecture.

Another thing which makes things more complicated is that we need to know things in advance, before the setup. We need to tell the addresses of all our nodes, which role of each them has, how many workers are in the cluster and so on. With Swarm, we just bring up a node and tell it to join the cluster.

The conclusion is that Docker Swarm is definitely winner here, as there was no any problem to set up a cluster of mixed architecture devices and it was done intuitively and quickly.

But, Kubernetes still does have some important advantages over Docker Swarm.

As Docker Swarm is based on Docker API and shares all the limitations with Docker. If we want to do something that is not part of Docker functionality provided by Docker API, it is just not possible.

The first major limitation was networking. Until Docker Swarm release 1.0 we could not link containers running on different servers.Actually, we still cannot link them but now we have multi-host networking to help us connect containers running on different servers. Kubernetes used Flannel to accomplish this kind of networking and now, since the Docker release 1.9, that feature is available as part of Docker Engine.

Another problem were persistent volumes. Docker introduced them in release 1.9. Until recently, if we persist a volume, that container was tied to the server and volume resides. It could not be moved around, without doing some tricks like copying volume directories from one server to another. This is itself a slow operation that ruins the goals of tools like Swarm. Besides, even if we have time to copy a volume from one to the other server, we do not know where to copy a volume from one to the other server, since clustering tools tend to treat the whole datacenter as a single entity. Since the version 1.9, Docker Swarm has become able to deal with persistent volumes. When it comes to a single host, Docker works flawlessly, but if we want to have persistent volumes among multiple hosts, it is also possible but not out-of-the-box. It requires to set up a shared file system like Ceph, GlusterFS, Network File System[25]. In clustered environment where multiple hosts participating to deliver the required service, we need to have distributed storage that is made available to all hosts and then exposed to the containers through a consistent namespace. By creating a consistent naming convention and unified namespace, all running containers will have access to the underlying durable storage backend, irrespective of the host from which they are deployed.

Both networking and persistent volumes were one of the features supported by Kubernetes for quite long time and the reason why many were choosing it over Swarm. That advantage disappeared with Docker release 1.9.

When it comes to work units – the smallest work unit in Docker Swarm is container, while in Kubernetes is pod. Pod could contain many containers which run on a single node. Kubernetes has more advanced features when it comes to

[25] https://en.wikipedia.org/wiki/Network_File_System

pod replication and auto-scaling, while Docker Swarm puts more emphasis on scheduler capabilities providing concepts as filters, node tags, affinity, strategies that assign containers to underlying nodes in oder to maximize performance and resource utilization.

Considering the other features – it can be noticed that Kubernetes is more focused on auto-scaling and pod replication customizations, while Docker Swarm has more advanced scheduling capabilities based on labels and filters. Also, it is significant that Docker Swarm offers more options for service discovery.

In what follows, Table 3.3 is given [17[18][23]. It contains parallel comparison of Kubernetes and Docker Swarm, taking into account various aspects and features.

	Kubernetes	Docker Swarm
Application definition	Applications can be deployed using a combination of pods, deployments, and services (or "microservices"). A pod is a group of co-located containers and is the atomic unit of a deployment. A deployment can have replicas across multiple nodes. A service is the "external face" of container workloads and integrates with DNS to round-robin incoming requests.	Applications can be deployed as services (or "microservices") in a Swarm cluster. Multi-container applications can specified using YAML files. Docker Compose[26] can deploy the app. Tasks (an instance of a service running on a node) can be distributed across datacenters using labels. Multiple placement preferences can be used to distribute tasks further, for example, to a rack in a datacenter.
Load balancing	Pods are exposed through a service, which can be used as a load-balancer within the cluster. Typically, an ingress is used for load balancing.	Swarm mode has a DNS component that can be used to distribute incoming requests to a service name. Services can run on ports

[26] https://docs.docker.com/compose/

		specified by the user or can be assigned automatically.
Rolling upgrades and rollback	The deployment controller supports both "rolling-update" and "recreate" strategies. Rolling updates can specify maximum number of pods unavailable or maximum number running during the process.	At rollout time, you can apply rolling updates to services. The Swarm manager lets you control the delay between service deployment to different sets of nodes, thereby updating only 1 task at a time.
Health checks	Health checks of two kinds: liveness (is app responsive) and readiness (is app responsive, but busy preparing and not yet able to serve). Out-of-the-box, Kubernetes provides a basic logging mechanism to pull aggregate logs for a set of containers that make up a pod.	Docker Swarm health checks are limited to services. If a container backing the service does not come up (running state), a new container is kicked off. Users can embed health check functionality into their Docker images using the HEALTHCHECK instruction.
Auto-scaling	Auto-scaling using a simple number-of-pods target is defined declaratively using deployments. CPU-utilization-per-pod target is available.	Not directly available. For each service, you can declare the number of tasks you want to run. When you manually scale up or down, the Swarm manager automatically adapts by adding or removing tasks.
Networking	The networking model is a flat network, enabling all pods to communicate with one another. Network policies specify how pods communicate with each other. The flat network is typically implemented as an overlay.	Node joining a Docker Swarm cluster creates an overlay network for services that span all of the hosts in the Swarm and a host only Docker bridge network for containers. By default, nodes in the

	The model requires two CIDRs: one from which pods get an IP address, the other for services.	Swarm cluster encrypt overlay control and management traffic between themselves. Users can choose to encrypt container data traffic when creating an overlay network by themselves.
Data volumes [23]	Simple shared local volumes. Docker data volumes are directories shared within one or more containers. Volumes are created separately or together with containers and can be shared between multiple containers. Data volumes also persist even when containers using them are deleted. Volumes by themselves are however only local to the node they are created on. To create global volumes, Docker engine supports volume plugins.	Volumes shared within pods. Kubernetes volumes are an abstraction to allow containers share data within the same pod. The volumes have an explicit lifetime, they are created and removed together with the pod they are enclosed in. Volumes work in basics just as any other directory, which is accessible to the containers in the same pod. Kubernetes also supports external data volume managers to transfer data between pods.
Storage	Two storage APIs: The first provides abstractions for individual storage backends (e.g. NFS, AWS EBS, ceph, Flocker). The second provides an abstraction for a storage resource request (e.g. 8 Gb),	Docker Engine and Swarm support mounting volumes into a container. Shared filesystems, including NFS, iSCSI, and fibre channel, can be configured nodes. Plugin options include Azure, Google Cloud Platform,

	which can be fulfilled with different storage backends. Modifying the storage resource used by the Docker daemon on a cluster node requires temporarily removing the node from the cluster. Kubernetes offers several types of persistent volumes with block or file support - iSCSI, NFS, Amazon Web Services, Google Cloud Platform, and Microsoft Azure. The emptyDir volume is non-persistent and can used to read and write files with a container.	NetApp, Dell EMC, and others.
Service discovery	Services can be found using environment variables or DNS. Kubelet adds a set of environment variables when a pod is run. Kubelet supports simple –SVCNAME˙SERVICE˙HOST" AND –SVCNAME˙SERVICE˙PORT" variables, as well as Docker links compatible variables. DNS Server is available as an addon. For each Kubernetes Service, the DNS Server creates a set of DNS records. If DNS is enabled in the entire cluster, pods will be able to use Service	Swarm Manager node assigns each service a unique DNS name and load balances running containers. Requests to services are load balanced to the individual containers via the DNS server embedded in the Swarm. Docker Swarm comes with multiple discovery backends: • Docker Hub as a hosted discovery service is intended to be used for dev/test. Not

	names that automatically resolve.	recommended for production. • A static file or list of nodes can be used as a discovery backend. The file must be stored on a host that is accessible from the Swarm Manager. You can also provide a node list as an option when you start Swarm.
Performance	With the release of 1.6, Kubernetes scales to 5,000-node clusters.	According to the Docker's blog post on scaling Swarm clusters, Docker Swarm has been scaled and performance tested up to 30,000 containers and 1,000 nodes with 1 Swarm manager.

Table 3.3 – Kubernetes and Docker Swarm comparison of features

Conclusion

Taking all the aspects previously discussed into account, the decision is to use Docker Swarm as the underlying technology for the framework that is going to be developed.

Kubernetes is too hard to set up, does not have out-of-the-box functionality as Docker (load balancer and networking are done using additional services and tools) especially in mixed architecture environment, Kubernetes CLI is much different and less comfortable then Docker CLI (Docker CLI/API is one of its strongest points due to its simplicity and functionality) and at the same time it does not provide any real advantages over Swarm, since the Docker release 1.9.

That does not mean that there are still no some features available in Kubernetes that are not supported by Swarm.

Kubernetes is more focused on auto-scaling and fault tolerance-related capabilities, while Swarm putting more emphasis scheduling segment and simple service discovery. For this framework, auto-scaling and fault tolerance are not taken into account, but scheduling and service discovery are very important as the framework exactly acts as a kind of scheduler, which allocates tasks to nodes. Swarm's container-level granularity is much more suitable for task allocation, as it is more convenient to consider a single task as container than a pod. Inisde a pod, we can have also a single container, but it is introducing overhead when we have many single-container pods in comparison to a single pod of the same number of containers.

There are differences in both directions, but they are not major as they were in early days of Docker Swarm, and becoming even smaller with each Docker release. Actually, for many use cases there is no gap at all, while Docker Swarm is much easier to set up, learn and use.

4 Data-Intensive Applications Development in Node-RED

Node-RED[27] is an open-source mashup tool developed by IBM, based on the server-side JavaScript framework Node.js[28]. It uses an event-driven, non-blocking I/O model suited to data-intensive, real-time applications that run across distributed devices, which makes it suitable. Also, it provides a GUI where users drag-and-drop blocks that represent components of a larger system which can be either devices, software components or web services that are about to be connected. These blocks are called nodes.

A node is a visual representation of a block of JavaScript code which is designed to carry out a specific task. Additional blocks (nodes) can be placed in between these components, to represent software functions that manipulate and transform the data during its flow. Two nodes can be wired together by connecting the output port of a node to the input port of the other node. After connecting many such nodes, the final diagram is called - a flow.

For the development of the framework, due to its characteristics (especially the flexibility and ability to extend the standard functionality), we adopt Node-RED environment.

IoT solutions often require wiring different hardware devices, APIs, online web services in interesting ways. For example, using the traditional approach, accessing the temperature data from a sensor connected to a device's serial port would require a lot of "boilerplate" code. In contrast, to use a serial port in Node-RED, all a developer has to do is to drag on a node and specify the serial port details. Hence, using Node-RED, the time and effort spent on writing "boilerplate" code is greatly reduced, so the developer could put more focus on the business logic of the application itself [2].

Node-RED flows are represented in JSON, and can be serialized in order to be imported or shared online. There is a new concept of sub-flows being introduced to Node-RED. This allows creating composite nodes, encapsulating complex logic represented by internal data flows.

[27] http://nodered.org/
[28] http://nodejs.org/

Since in Node-RED, we have blocks which are actually blocks of JavaScript code, it is possible to wrap any kind of functionality and encapsulate that as a node in the platform. So, this makes Node-RED applications easily extensible, without the need to restructure the complete application. For example, if new database type, or sensor is introduced, the user needs to write JavaScript code to deal with it and connect it to the existing dataflow, which makes it suitable for data-intensive applications. Also, we should mention that new nodes for interacting with new hardware, software and web services are constantly being added, making Node-RED a very rich and easily extensible system. Lastly, the learning curve to develop a new node for the platform is low for Node.js developers, since a node is simply an encapsulation of Node.js code.

It should be taken into account that palette of available nodes can be extended by custom nodes which can be developed or downloaded using palette manager. In what follows is only the elementary classification with the most representative examples of nodes from certain categories:

- *Input nodes:* Generate messages for downstream nodes. For example, it could be the case of receiving HTTP requests, reading serial port, reading data from textual file or triggering an inject node which is sending a message (string, number, timestamp) once or many times (periodically).

- *Output nodes:* It is used for consuming messages. For example to send data contained in msg.payload to an external service or pin on a device, and may generate response messages. It could be, for instance – sending HTTP response to the web client and displaying it, using debug node for redirecting the output to the debug console of Node-RED, sending the value via serial port or storing the obtained result into a file.

- *Function nodes:* Processing data in some way, emitting new or modified messages. Here we write our own JavaScript-alike code to satisfy our own needs for the application. As the name implies, a function node exposes a single JavaScript function. Using the function node, we can write your own JavaScript code that runs against the messages passed in and returns zero or more messages to downstream nodes for

processing. To write function nodes, we write a JavaScript function using the built-in code editor.

In addition to these core types, there are some more specific node categories, used for special purposes:

- *Data storage nodes:* Used for database opeartions. They are able to execute queries over different types of databases (MongoDB, MySQL). For each type of database, a specific node is downloaded and added to the palette.

- *Credentials nodes:* These are nodes that hold the credentials used by one or more nodes to connect to an outside system or service., such as database or ssh.

- *HTTP request nodes:* This category of nodes provides ability to send HTTP requests to the URL, specified by msg.url property of the node.

- *Social networking nodes :* performing operations related to socail networking and communication as: receiving/sending e-mail, reading/posting tweets.

- *Execution nodes:* Among the available pre-installed nodes, there is so-called „exec" node, whose function is to redirect the msg.payload property of the message as a CLI command to the host operating system and execute it. Also, a subtype of execution nodes is downloadable ssh node, which is providing full ssh functionality.

- *User-created nodes:* A concept of subflows is providing ability to the developer to create own nodes, by using the existing nodes and connecting them.

Using the previously mentioned node types – the flows are created. Nodes are the primary building blocks of Node-RED. Nodes consist of code that runs in the Node-RED service (JavaScript .js file), and an HTML file consisting of description of the node, so that it appears in the node pane with a category, color, name and icon, code to configure the node and help text. Inside the flow, nodes exchange messages along pre-defined connections or wires. Under the hood,

a flow consists of a list of JavaScript objects that describe the nodes and their configurations, as well as the list of downstream nodes they are connected to, the wires.

Messages are the primary data structure used in Node-RED, and are, in most cases the only data that a node has to work it, when it is activated. These messages often contain a msg.payload property which actually represents the useful content of the message. Nodes may, additionally, attach other properties to a message, which can be used to carry other information onto the next node in the flow. When this happens, these extra properties will be documented in the node documentation that appears in the node info pane when you select a node in the Node-RED workspace.

This ensures that the flow is conceptually stateless – each node is self-contained, working with input messages and creating output messages. Apart from the context data, this means that the effect of a node's processing is either contained in its output messages, or caused by internal node logic that, for example, changes files or I/O pins on Raspberry Pi.

Wires define the connections between node input and output endpoints in a flow. They are typically used to connect the output endpoints of nodes to inputs of downstream nodes, indicating that messages generated by one node should be processed by the next connected node. It is also possible to connect more than one node to an endpoint using wires. When multiple nodes are connected to an output endpoint, messages are sent to each connected node in turn in the order they were wired to the output. When more than one node output is connected to an input endpoint, messages from any of those nodes will be processed by the connected node when they arrive. It is also possible to connect downstream nodes to upstream nodes to form loops.

While it is generally true that messages are the only way of communication between nodes, there is still one exception to this rule which is available to function nodes.

Function nodes have access to a special object called context that is used to hold data. This is important for nodes that need to maintain an index or count or sum data in messages. Two types of context variables exist: flow-level and global-level context variables. The flow-level context is shared by all nodes on a given

tab. The global-level context is shared by, and accessible to all nodes of the application.

Node-RED development environment is available in form of a native Windows, Mac OS and Linux application. Also, it is available for Raspberry Pi (Linux ARM port) and as a Docker container, compatible with both x86 and ARM Linux platforms.

Node-RED comes with a huge set of pre-installed nodes which provide rich functionality. Additional nodes can be installed and imported also. The decision of using Node-RED as an environment for the framework development is brought due to following reasons:

- Node.js server-side functionality embedded, which means that developed functionality (node flows) can be exposed as a service on the server, and executed by sending HTTP requests to the server executing the application (if HTTP nodes are used as an input)

- Execution nodes provide ability to execute any external application by using its command line interface and receive the console output as a msg.payload, which gives endless capabilities. In the case of our framework this was crucial, as execution nodes are used to send commands to Docker client via command line terminal, which further redirects them to Docker daemon. This way, by executing Node-RED application on Swarm master, we can manage the container-based application cluster. Concretely, this is used when we want to allocate a task (container) to a worker node in Swarm.

- HTTP request nodes are extremely useful feature, as we can use them to send HTTP requests to any resource specified by its URL. This provides an ability to communicate with any service within Edge or in Cloud, which is crucial for our framework. Also, by processing msg.url parameter, we can set the parameters during the execution.

- Inject nodes, which are set to trigger the flow immediately after the application deployment in order to perform task allocation (late binding).

- Nodes for file input and output, providing means for file manipulation.

- JSON representation and exporting of Node-RED flows provides an easy way to store, share and process them. This way, we have ability to generate new flows using templates and changing only the necessary parameters, while the rest of the flow remains untouched, even inside the Node-RED environment itself, as it has all the necessary mechanisms (converting JSON files to JavaScript objects and parsing). Each node represents a single JSON object, while the flow is an array of objects representing nodes. Together with file manipulation-related capabilities, we get a solution for generating new flows from template files.

- Using a concept of subflows, we can connect many existing nodes into one. This provides the ability to develop specific nodes which actually correspond to the framework capabilities and gives more value to the overall work – making it to really be considered as a framework as it is possible to easily reuse these nodes.

5 Framework for Computation Task Movement in Fog Environment

As a result of the research, a framework for computation task movement has been developed providing ability to move the tasks between the available nodes in both Cloud and Edge, taking into account the device heterogeneity.

In this chapter, the framework is going to be described in details, taking into account various aspects - from framework overview, requirements, use cases, conceptual description including the data model, selection criteria and algorithm mechanisms, via underlying infrastructure and architecture, to the actual implementation as a Node-RED framework and its limitations and constraints.

5.1 Framework overview

This section provides a high-level overview of the developed framework from the perspective of its users, their roles, responsibilities and lifecycle phases of the application developed using our framework from single task development to service delivery in order to introduce the idea of the provided solution.

In what follows, users and application lifecycle phases relevant to the scope of this framework are presented.

We can identify three types of users involved into framework lifecycle:

- *Task developers:* Their responsibility is to create containers corresponding to tasks and generate Node-RED abstractions – nodes using the task development environment, which they publish and make them available to application developers. Also, they need to provide the necessary framework data.

- *Application developers:* Import tasks created by task developers into their Node-RED application development environment and use them to develop data-intensive applications.

- *End-users:* Use the services created as a result of application developers' work.

There are six phases in the framework lifecycle:

1. *Task creation*: In this phase, task developers create containers corresponding to tasks and generate corresponding nodes which will be imported in application developer's Node-RED environment. They have have to specify the cirteria for selecting the device which will execute a given task according to defined policy. Also, they need to provide certain data which is necessary to utilize the framework capabilities, including information about tasks, devices and mapping between tasks and devices (which device can execute which tasks).

2. *Task publishing*: Once the necessary steps are performed, task developers export and publish Node-RED components (nodes) necessary to execute the tasks and make them available to application developers. They are also generated inside Node-RED environment, and have two components – allocator and execution proxy. They are exported in JSON format. The allocator part is used for allocating a given task to device according to criteria specified by task developer, while execution proxy is the component which is connected to the flow of the developed Node-RED data-intensive application. They are stored inside the framework database.

3. *Task retrieval*: Application developers retrieve and import tasks in JSON format previously published by task developers into their Node-RED environment from the framework database.

4. *Application development*: Application developers use imported nodes in their Node-RED application development environment as black boxes to create their own data-intensive applications, without taking care where is the task actually executed.

5. *Deployment*: The services offered by data-intensive application are deployed by using Node-RED environment, more precisely the NodeJS back-end components embedded in Node-RED. Immediately after the deployment, each task is dynamically allocated to the node selected by criteria previously chosen by task developers. In this case, Node-RED

interacts with Docker Swarm in order to create the services corresponding to tasks and allocate them to selected nodes.

6. *Execution*: Once all the tasks are allocated to selected nodes, the application is ready to be executed and its service delivered in order to satisfy the needs of end-users. In this situation, we also have the interaction of Node-RED and Docker Swarm, as containers running as Docker Swarm services are actually performing the computation itself and providing the necessary results. Therefore, the actual requests are forwarded from Node-RED comonents (nodes) to containers running within Docker Swarm.

In what follows, Figure 5.1 is given, presenting the previously described lifecycle of the application developed using our framework and involved users.

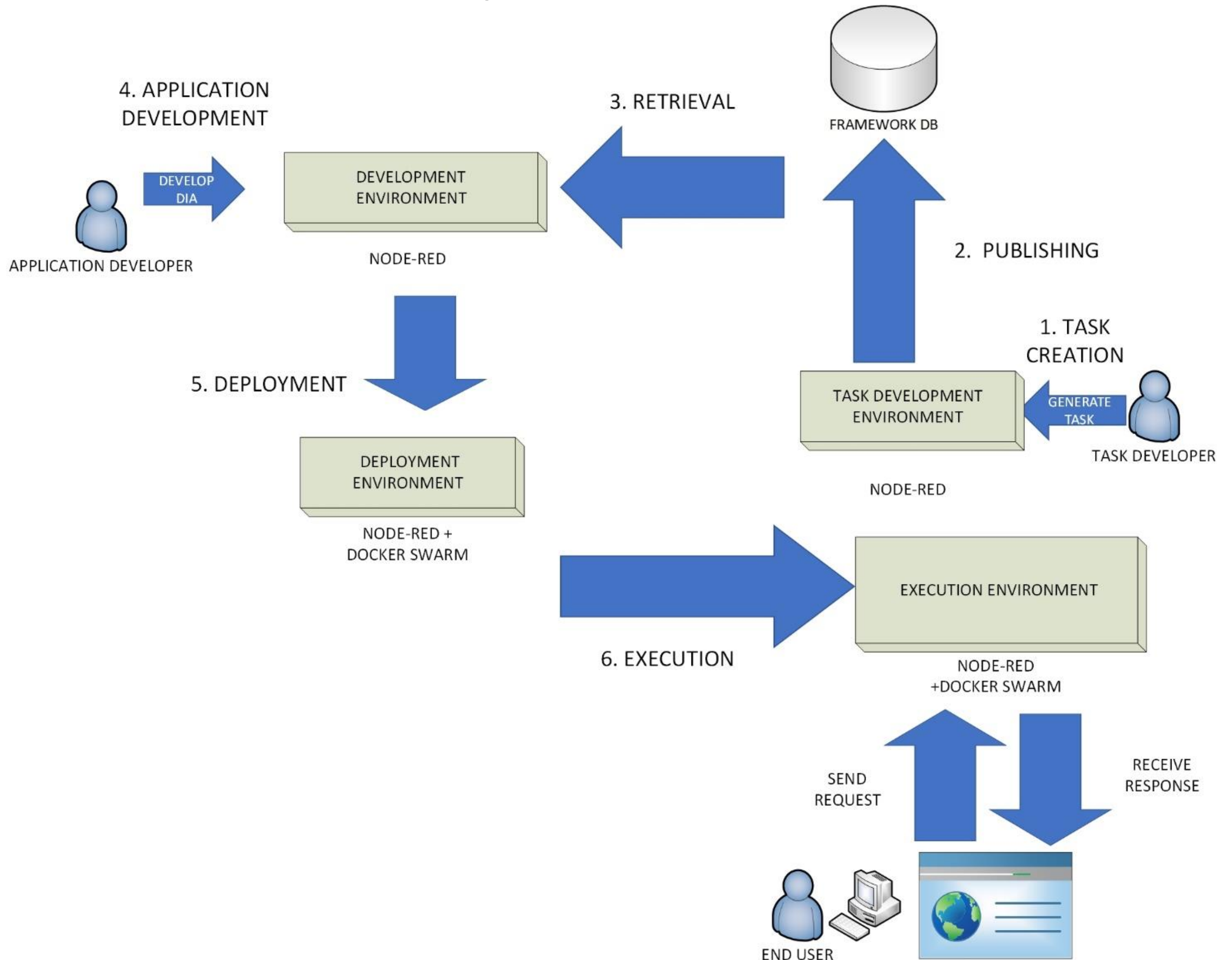

Figure 5.1 – Framework overview

The source code and other assets related to the framework implementation are available on GitHub, inside the repository that can be accessed through the following link:

https://github.com/nenad-petrovic/thesis

5.2 Framework database design

Task developers should fill the framework database with the necessary data, which gives ability to fulfill the needs of maintaining the information about tasks and nodes necessary for the framework mechanisms.

Figure 5.2 displays the UML class diagram of the concepts which are stored in the framework database keeping the necessary information.

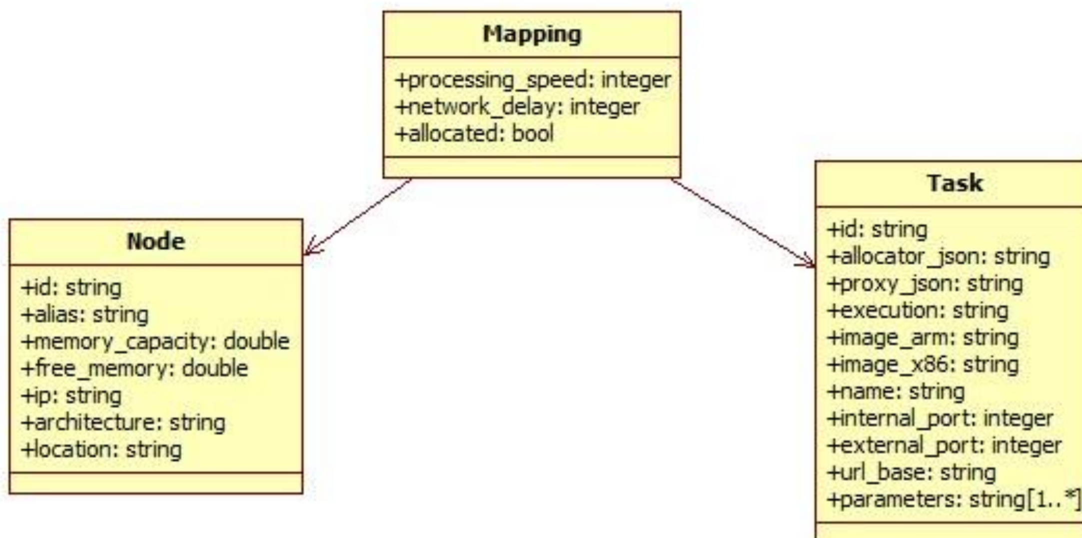

Figure 5.2 – UML model of the concepts stored in framework database

As it can bee seen, there are two main concepts: node and task. They are connected by mapping – which means that the node has ability to execute the task it is connected to. Each node can be mapped to many tasks, and vice-versa. The mapping itself does not mean immediately that the node is executing the task – it just means that the node is a candidate for the execution.

5.2.1 Nodes

Node is a concept of the machine able to execute a task. It could be either conventional personal computer, high-performance server or even a single-board computer, such as Raspberry Pi. The main information stored about nodes includes: architecture, ip address, alias, total memory capacity, free memory, location.

Architecture refers to the processing unit architecture which could be x86 (as in case of conventional personal computers and most modern high-performance servers) or ARM (in case of most single-board devices) – so the distinction could be made between these two, which is of utmost importance when it comes to task allocation – as different Docker images are used for different architectures, as it was previously explained.

IP address is used in order to be able to reference to the nodes in network. In this framework, it is only used for external services, while for all the tasks executed by both Cloud and Edge nodes, the standard localhost address is used (127.0.0.1) for the reference, as it is made possible by the mecahnisms of Docker Swarm architecture, which are going to be explained later.

Alias is user-given name to the node (such as "pc1","rpi1"), which is used by Docker Swarm in order to constraint the task allocation to a single node.

Memory capacity and free memory are important in order to prevent overloading the node by allocation of too many tasks which could lead to the allocation failure. This information is used during allocation in conjunction with the task attribute identifying the memory needed for a certain task, in the following way:

```
node.free_memory=node.free_memory – task.memory
```

Location could have the following value: "edge", "cloud" or "external". "Edge" means that node is physically located inside the organization boundaries and is under its total jurisdiction. "Cloud" means that node is provided by external provider (IaaS), and, is, in most cases a virtual machine. So, this node is not within the physical boundaries of th organization. The distinction between these two is crucial when it comes to data privacy issues and is taken into acocunt during the node selection. But, for these two it is common that they are both

fully accessible from the perspective of the organization. It means they can be managed in terms of task execution – we can make them execute a desired task by our choice. On the other side, "external" means that the node is not under organization management, and is used for special purpose as external service (SaaS, such as database application or image processing service or PaaS). These nodes are, in fact, also Cloud nodes, but distinction should be made, because we are not able to fully access these devices and make them execute any other task, but just execute the specific, pre-defined tasks, offered by their providers. From the perspective of the underlying infrastructure, they are not part of Docker Swarm cluster, while the previously mentioned IaaS-based Cloud nodes are able to join the cluster as any other node.

5.2.2 Tasks

Another crucial concept of the framework is the task. Information about tasks which is necessary for the framework, in order to achieve the desired functionality, consists of: task name (id), JSON strings of the Node-RED components, execution environment, memory needed, ARM and x86 image locations (available on Docker Hub), task URL, internal and external port.

Task name (represented as id) is used in order to distinguish between different tasks inside data-intensive application, and is also the Docker Swarm service name at the same time.

JSON strings are the task representation that is used inside Node-RED development environment. They are constructed and published by task developers, while it is imported and used by application developers. Two of them are stored – `allocator_json`, which corresponds to the task allocator node and `proxy_json` which corresponds to task execution proxy.

Execution environment is specifying where the task is performed. It can have values: "cloud" or "edge". So, when the node selection is performed, nodes are filtered according to the execution environment. It is used in conjunction with node attribute named "location". "Cloud" value of task attribute is compatible with both "cloud" and "external" node attributes.

Memory needed is information used to determine whether the node is able to accept the task or no, depending on the free memory available.

ARM and x86 image locations are necessary in order to provide ability to identify images during the service creation. If it happens that node does not have the image locally, it needs to download it from the Docker Hub. It is crucial to distinguish between ARM and x86 images, as image compatibility constraints are imposed by the architecture – the image built for x86 cannot be executed on ARM devices, and vice-versa (although, the later is not always true, as QEMU provides ability to execute ARM images on x86 architecture, but is not considered due to huge performance impact). This attribute is a string, with following structure: `repository_name/image_name.`

Task base url is useful in cases when we have to send HTTP requests and is used in order to form the whole URL request string, together with external port and parameters (stored as multivalued attribute), which has the following form:

```
http://host_ip:external_port/url_base?parameter1=value1&paramter2=value2
```

External port has also role in Docker Swarm service creation, together with internal port. External port, as it was explained, is used to access service externally, from other hosts, while the internal port is used locally by Docker engine. Both of them are equally important, when it comes to service creation, as the Docker Swarm service creation command has the following structure:

docker service create -p ***external_port:internal_port*** --replicas **number_of_replicas** --network **overlay_network_name** --name ***task_name*** --constraint 'node.label.alias==***node_alias'*** ***image_name***

Words given bold are user-defined values of the parameters, while the words which are both bold and italic are the parameters obtained from the database which maintains information about tasks and nodes.

5.2.3 Mappings

And, the third concept maintained inside the framework database is related to node-task mapping. It is capturing the information which node is able to execute which task. As it was previously said, it does not mean that the task is allocated to the node. It actually means that the node is able to execute the task, and is a

candidate for the execution, while the selection algorithm is used to further decide whether the node is the most suitable one according to the given criteria or not.

There are three attributes encapsulated into mapping – network delay, processing speed and allocated. The first two parameters are used for illustrative purposes only providing the information about the performance of the task when executed by the mapped node. They represent integer numbers, ranging from 1 to 5, which is actually a degree of the given attribute. For example, when it comes to network delay – higher value means longer delay (which means worse performance), while for processing speed is the opposite. Having higher processing speed means better performance. The idea is to perform benchmarks and maintain the results obtain, scale them on a relative scale, which would give relative values from 1 to 5. Network delay could be a relative comparison of network pings between available nodes. These values could be used to make more refined selection of the node for the task allocation. Allocated attribute provides the information if the given mapping is actually done. In the case of value 1, it means that the task is allocated to the given node and that node is actually executing it. Otherwise, if value is 0, it means that the allocation is not done yet. This information is set in late binding phase of the deploymeny and used in the execution phase to detect where is the task allocated.

5.3 Task development environment

The goal of this module is to support the task developers create the corresponding Docker containers executing these tasks, to provide Node-RED components which represent these tasks and can be imported by application developers inside their Node-RED environment. For each task they generate two Node-RED components - execution proxy nodes and late binding task allocation node.

Figure 5.3 displays the use case diagram for task developers.

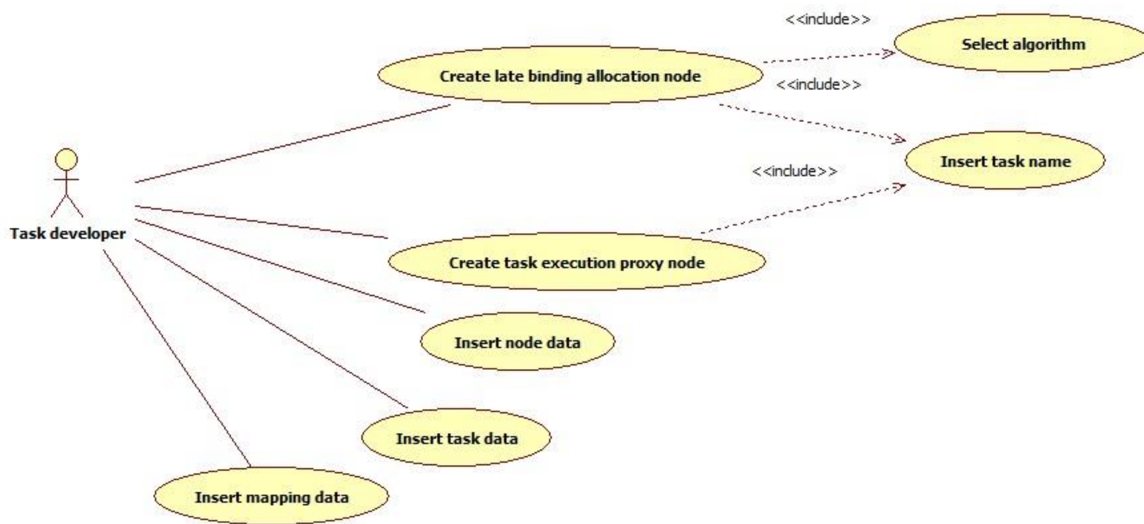

Figure 5.3 – Framework use case diagram from the perspective of task developer

As it can be seen in use case diagram, task developers need to specify the task name and algorithm (from the list of available algorithms) in order to create allocation nodes. For task execution proxy nodes it is enough to provide the task name .

They also need to provide the necessary data related to devices (nodes) that are considered as candidates for execution, tasks and mapping between nodes and tasks to fill the framework database, that holds all the relevant data enabling the functionality.

5.3.1 Template-based task creation mechanism

Before the execution of the task we have to select the most suitable node, and then allocate the task to the node, by creating a corresponding Docker Swarm service to the desired task, which is located on the selected node. The components having these responsibilities in the framework are node selector and service creator nodes, and they together make a previously mentioned task allocator. These operations, which are actually done in late binding phase. It means that immadiately after the deployment of the application, the node selection for each of the tasks is done and the corresponding Docker Swarm service created, which

is going to be executed on the selected node. Also, a task-node allocation record is stored into database which keeps the allocation records, as it is necessary to use this information later, in execution phase to know where is the task located.

The next is phase is the task execution. For this part, the execution proxy node takes its turn. Its role is to send HTTP requests to the right node, using the right port and URL parameters. The execution proxy component takes its responsibility in the execution phase. In order to know where is the task located it needs to look up the allocation database.

Both components – task allocator and task execution proxy are generated by task developers in task creation phase. Templates for these nodes are taken and processed in order to create custom nodes by changing the necessary parameters.

Figure 5.4 is displayed in order to illustrate the idea of previously described concepts.

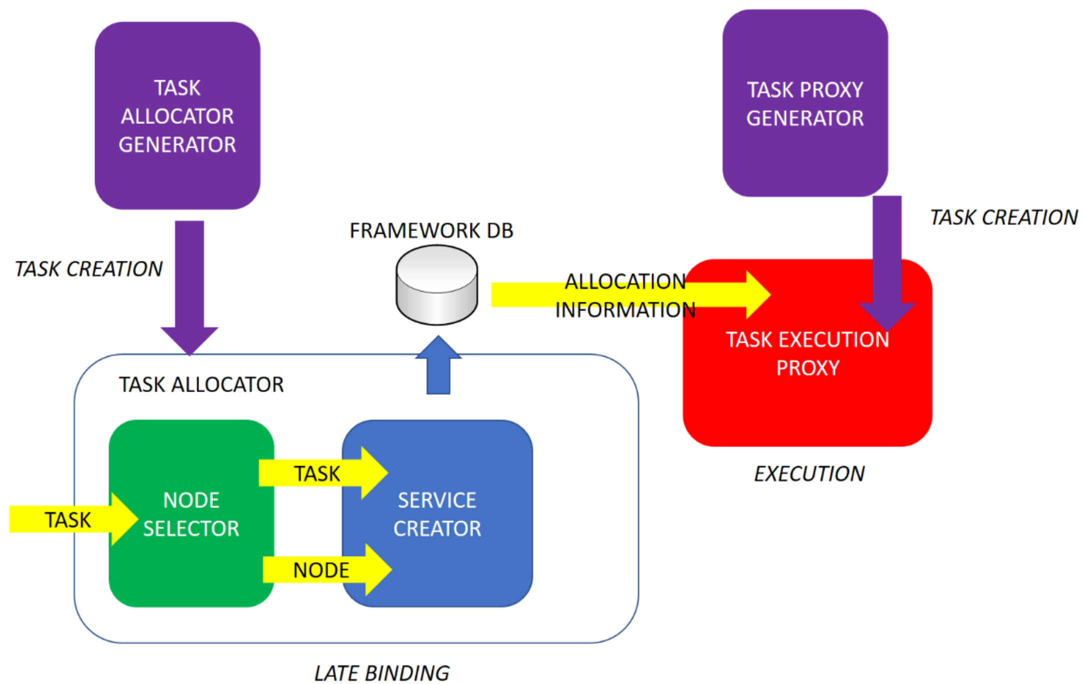

Figure 5.4 – Implementation mechanism overview

When it comes to the implementation of the previously described mechanism, node templates are used in order to generate custom nodes, exploiting the fact that Node-RED exports and imports flows and subflows in JSON format natively, which is textual, readable by human and easy to manipulate. Every node of the flow is represented as a JSON object, and all its properties are presented as JSON attrbiutes and values. Among these attrributes, we have the node type, name, x and y coordinates, and many others which could be specific for a certain type of nodes. A Node-RED flow is represented as an array of JSON objects, where each JSON object corresponds to one of its nodes.

Figure 5.5 illustrates the concept.

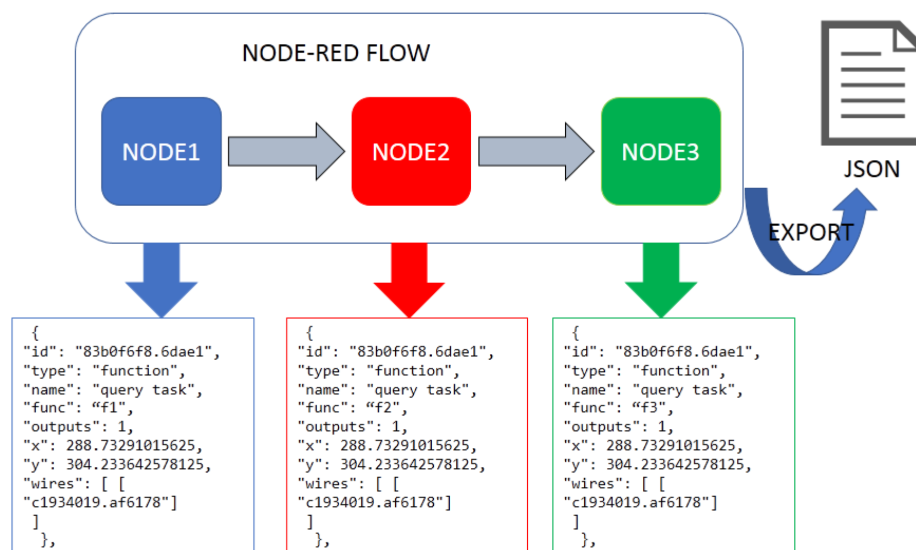

Figure 5.5 – Node-RED flow as array of JSON objects

The idea is to understand the generic structure of two types of nodes (which are actually subflows) – late binding task allocation and task execution proxy, so we can keep the templates stored and customize them everytime we need a specific node, by parsing them and changing the specific parameters.

In what follows, the implementation of node generation mechanism by using JSON node templates is going to be explained. The mechanism for generating new nodes is also implemented using Node-RED.

5.3.2 Task allocation (binding) node generation

For generating the task allocation late binding node from the tempate, it is necessary to change the parameters which hold the task name and algorithm name, in the step of task and algorithm specification in task allocation late binding node.

The first necessary step is to load a generic template JSON file located on a disk into a Node-RED flow. This is done by using file input NodeRED component, which comes pre-installed.

Then, the next step is modify the template node. Before that, the JSON file is read and converted to JavaScript array of objects (also by using Node-RED pre-installed component called "JSON", so it can be easily parsed and modified. In that form, a node in the flow which contains these parameters is located by its name (given as JSON attrbiute) and the string holding the code changed. The node inside the task allocation flow which is modified belongs to function node category and the code of the actual function code, containing the parameter specification is placed in JSON attribute of the node called 'func'. Therefore, we can use this fact and change these values with user-specified values for task and algoirthm. Once we change all the parameters, we can export a new JSON file.

In this case, a kind of „code-as-data" approach was used, as the code of the function which is going to be executed in the node which is going to be produced is actually located in a JSON attribute which is set by node generator.

Figure 5.6 displays the implementation as of the node generator as a dataflow, utilizing the previously described concepts, while the code in Listing 5.1 shows how the processing of JSON file is done. In fact, this code corresponds to the node called „GENERATE BINDING FROM TEMPLATE“ from the upper dataflow of the Figure 5.6.

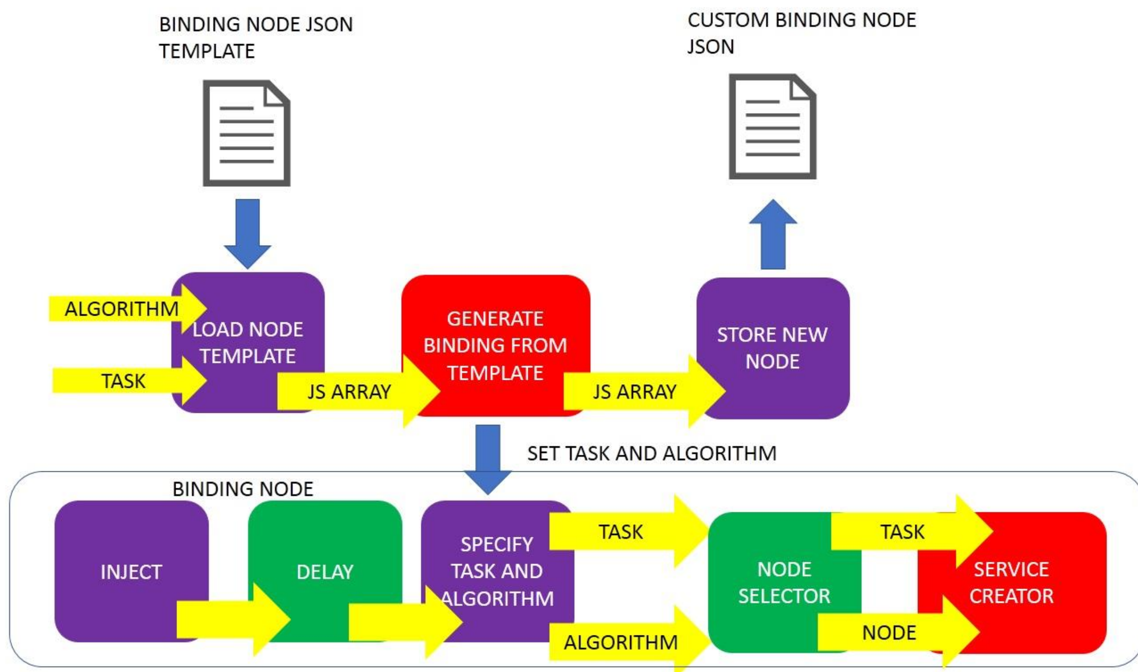

Figure 5.6 – Generating task allocation node from template

```
    var result={};
    var algorithm=msg.req.query.algorithm;
    var task=msg.req.query.task;

    for(var i =0; i<msg.payload.length;i++)
    {
        if(msg.payload[i].name=="specify_task_and_algorithm")
        {
        msg.payload[i].func='var          task=\"'+task+'\";\nvar
node=\"'+node+'\";\nmsg.payload=\"\";\nmsg.algorithmname=algori
thm;\nmsg.taskname=task;\n\nreturn msg;';
        result=msg.payload[i];
        }
    }
    return msg;
```

Listing 5.1 – Code of the late binding task allocation node generator

5.3.3 Task execution proxy generation

The next step is to do the same for a task execution proxy node. Here, more problems occur – as these nodes which need to be generated are going to be used in execution phase, after the allocation is done. It means that all the data related to the allocation of tasks is known in this step inside the execution task proxy, but completely unknown to the node generator – so there is a gap between them in terms of the information available. It means that the code which needs to be used in execution phase should be completely parametrized and require, as node generator does not know anything about the allocation decisions as this information is unavailable at the node generation step.

In this case, we need to change the code of two nodes. The flow of the operations to be done in order to generate the desired node is presented in the upper of Figure 5.7, while the flow below represents the template flow which is used to generate the task execution proxy. The arrows between these two going from the

first to second represent the changes and modifications which need to be done in order to customize the template.

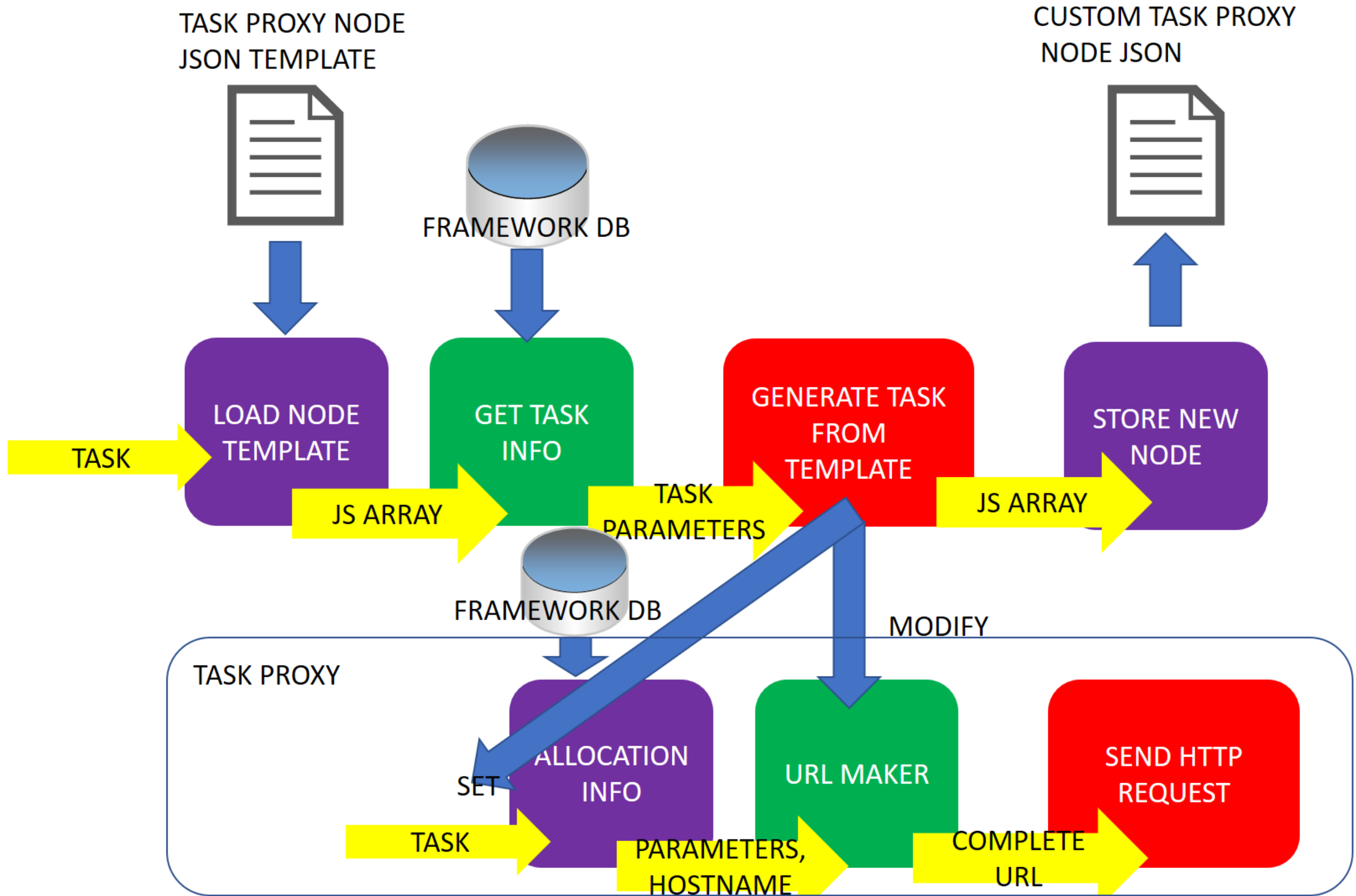

Figure 5.7 – Generating task execution proxy node from template

The first one is to change the task name, which is done in a similar way as explained and displayed in Listing 5.2, and consists only in providing the right task name.

However, the critical part of the task proxy generator code is related to URL maker, as URL needs to use information which becomes available after the allocation is performed and is not available at the moment when the task

execution proxy node is generated by task developers. Therefore, the code needs to be adapted in order to utilize „code-as-data“ approach.

In order to find the best form of the code for this purpose, it is necessary to first take a look at the code which needs to be generated (Listing 5.3).

The example given in Listing 5.2 considers the situation when we generate URL which has two parameters. The code given by black font is generic and part of a template, while the code in blue and red is generated, depending on the number of parameters of the task. The code in blue represents operations related to variables and value assignment, while the code in red is the actual URL cosntruction.

```
msg.limit = 5;
msg.skip = 0;
var result=msg.payload.url;
var url_base=result;
var final_url=url_base+"?";

var parameter1=msg.parameter1;
var parameter2=msg.parameter2;

final_url=final_url+"parameter1="+parameter1;
final_url=final_url+"parameter2="+parameter2;


msg.url=final_url;
return msg;
```

Listing 5.2 – Code that is generated by task execution proxy node generator

The colored code is the part that we should generate, so it should be as parametrized as possible, in order to generate it using only the inforamtion available before the task allocation, while the code which is generated is refering to the execution phase, which is after the task allocation.

As it can be concluded, the data necessary to generate this code include the task parameters, which are available in TASK database. This is the reason why the task information is read before generating the code, as it is displayed in Figure 5.7.

In what follows the code for generating the URL maker inside task execution proxy is going to be explained (given as Listing 5.3).
The variable `url_maker_code` is the one which holds the code that needs to be inserted into `url_maker` node which is a part of the task execution proxy template node flow, and the goal is to fill it with the content as given in Listing 5.2.

Step by step, the parts of the code are going to be concatenated.

The first step is to assign to it the beginning part of code which is generic and always the same (Listing 5.2, lines 1-5).

The second step is to construct the code given in blue and concatenate it to the result (`url_maker_code`). The for loop which generates the code this part of the Listing 5.2 code is given in Listing 5.3, also using the same color (blue). Here, we generate the code for variable assignement.

The third step is to generate the code previously given in red (Listing 5.2). The corresponding code for generating it is given as for loop in Listing 5.3 (also red fonts). A variable `url_code` is used in order to store that part of the code which is generated by the for loop.

The fourth step is the code which is also generic and just concatenated to the result. In this part the the previously obtained strings in for loops are concatenated and generic part added to them.

Now, a for loop iterates through all the elements of the flow in order to find the nodes where these modification need to be applied.

The fifth step, given in green, corresponds to the first blue arrow in Figure 5.7, where we introduce the right task name into the template.

And, the final step, given in violet fonts is the moment when the node titled „`url_maker`“ is found inisde the JavaScript array obtained from the JSON template flow. This node belongs to function type. Therefore, its attribute called func actually holdds the fucntion code which is going to be executed. This is the right place to insert the previously obtained result stored in `url_maker_code`.

Finally, after performing all the mentioned steps, the generated task proxy can be stored as JSON file which can be later imported by application developers.

Once application import the JSON file into their Node-RED environemnt, they get a subflow which is represented as a single node with inputs corresponding to URL parameters as msg properties and a single output, which is the result of the operation.

Figure 5.8 displays the task execution proxy node which is generated – from the perspective of service application developers.

```
   var taskinfo=msg.payload;
   var task=msg.req.query.task;
   var url_maker_code="msg.limit = 5;\nmsg.skip = 0;\nvar result=msg.payload.url;\nvar
url_base=result;\nvar final_url=url_base+\"?\";\n";

   for(var k=0;k<taskinfo.parameters.length;k++)
   {
       url_maker_code=url_maker_code+"\nvar                                                     "+
taskinfo.parameters[k]+"=msg."+taskinfo.parameters[k]+";\n";
   }
   var url_code="";
   for(var l=0;l<taskinfo.parameters.length;l++)
   {
       url_code=url_code+'\nfinal_url=final_url+'+'"'+
taskinfo.parameters[l]+'="+'+taskinfo.parameters[l]+';\n';
       if(l!=taskinfo.parameters.length-1)
       {
           url_code=url_code+'\nfinal_url=final_url+"&";\n';
       }
   }
   url_maker_code=url_maker_code+"\n"+url_code;
   url_maker_code=url_maker_code+"\nmsg.payload=final_url;\n";
   url_maker_code=url_maker_code+"\nmsg.url=final_url;\nreturn msg;\n";
   for(var i =0; i<msg.flowtemp.length;i++)
   {
        if(msg.flowtemp[i].name=="query_allocation")
        {
        msg.flowtemp[i].func='msg.limit = 5;\nmsg.skip = 0;\nvar task=\"'+task+'\";
\nmsg.payload={ task: task};\nmsg.projection = {\"_id\":0, \"url\":1,\"parameters\":1};
\nreturn msg;\n';
        }

       if(msg.flowtemp[i].name=="url_maker")
       {
        msg.flowtemp[i].func=url_maker_code;
       }

   }
   msg.payload=msg.flowtemp;
   return msg;
```

Listing 5.3 – Code for generating URL maker inside task execution proxy

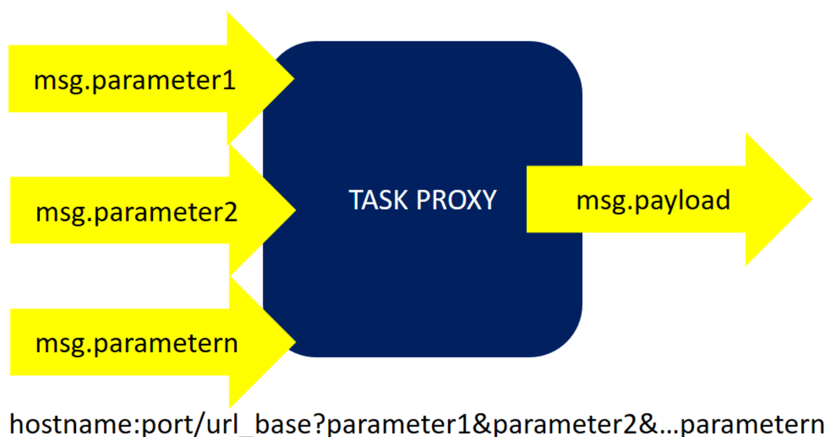

Figure 5.8 – Task execution proxy obtained as a result of node generator

5.3.4 Task creation user interface

To perform all the necessary operations, HTML-based GUI is provided (Figure 5.9).

As we can see on the GUI contains two parts. The first one deals with data management, while the second is used for template-based task node generation.

Furthermore, the data management part has three separated sections – for nodes, tasks and mapping.

Task generator is used for generating the necessary nodes in JSON format – task allocator and execution proxy node.

For allocator node, task developer has to specify both the task name and select an algorithm from the list, while for execution node she has only to specify the task name.

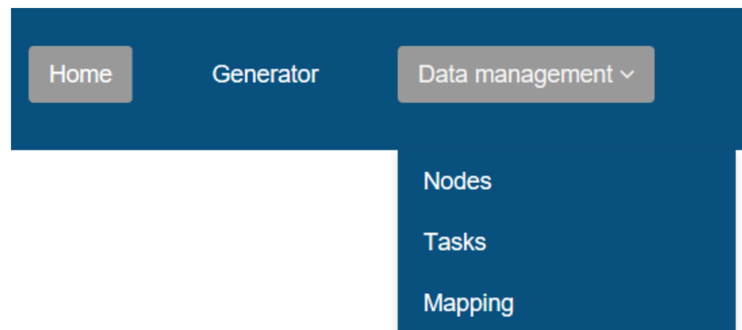

Figure 5.9 – Web application GUI for service task developers

5.4 Application development environment

Application developers create their data-intensive applications inside Node-RED environment using provided blocks of functionality – called nodes in Node-RED terminology. Each of them represents a task in terminology of data-intensive application.

Application developers import nodes previously created by the task developers inside the Node-RED development environment in order to use them and develop their own applications by combining them with other available nodes.

From task developers, they receive two types of nodes for each task – task allocation and execution proxy. They are both used as black boxes in the application developer's Node-RED environment.

Figure 5.10 displays the use case diagram of the framework from the perspective of application developer.

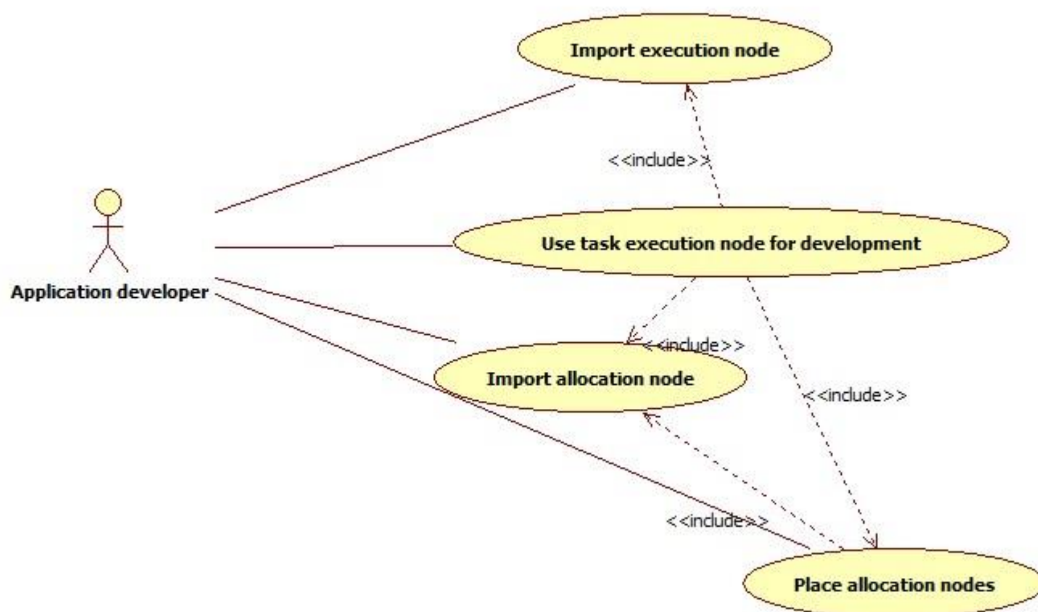

Figure 5.10 – Framework use case diagram from the perspective of application developer

As it is presented in Figure 5.10, application developer can import execution node, import allocation node, place allocation nodes and use task execution nodes in their Node-RED enviroment while developing their own applications. In order to use task execution nodes, they must import both allocation and execution nodes corresponding to the task, and just place the allocation node in their current flow as allocation is done automatically immediately after the deployment and is condition for the actual task execution, while the task execution node needs to be connected to the rest of the flow, taking care of input and output parameters.

Figure 5.11 presents the structure of the nodes imported in application development environment.

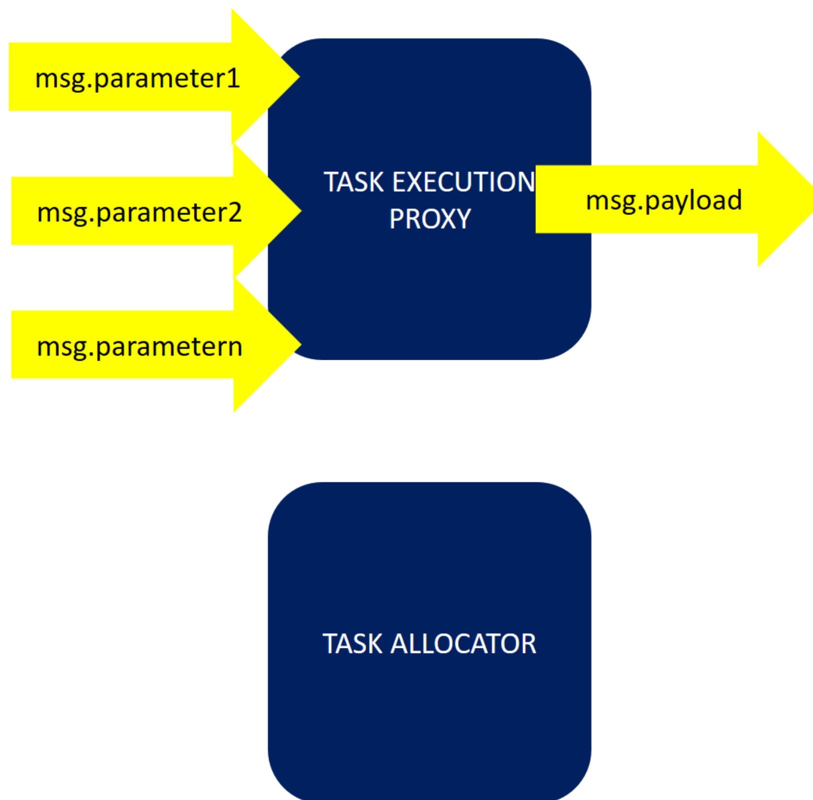

Figure 5.11 – Framework-generated components used by application developers

As we can see, task execution proxy can have many input parameters provided as distinct message attributes, while it has a single output placed inside the

msg.payload atttribute. These nodes are connected to the rest of the flow, while it is up to application developer to correctly provide the input parameters – either from GUI or previous nodes in the flow.

Task allocator does not have any input or output parameters. It just needs to be placed inside the application development environment – it does the rest of the job automatically.

Figure 5.12 illustrates the examples how the Node-RED environment of the application developer could look when they use the imported components.

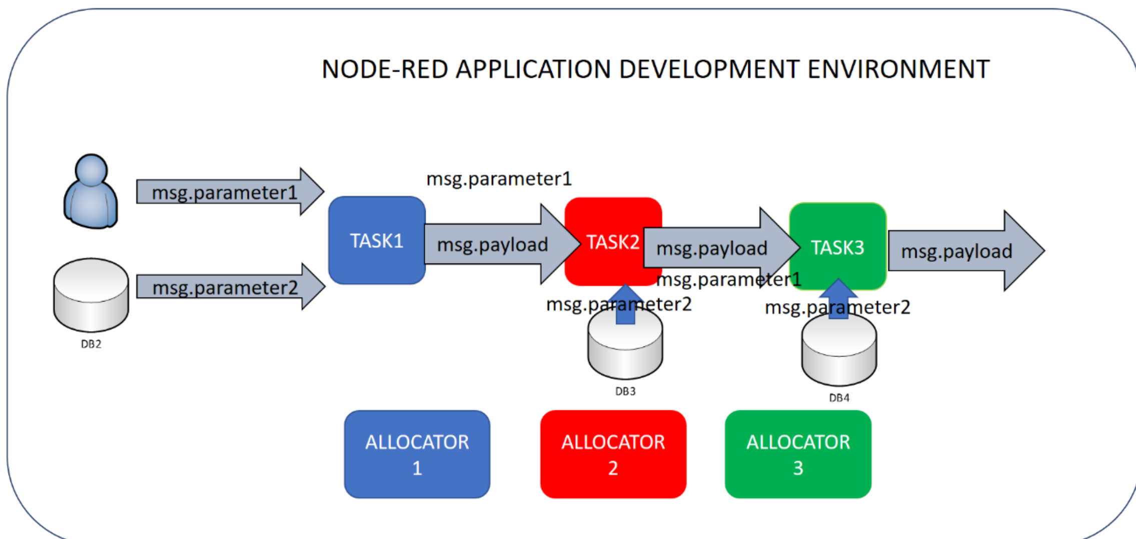

Figure 5.12 – Example of Node-RED application developed using the components generated by the framework

As we can see in Figure 5.12, there are three different tasks in the example application. Each of them has two input parameter and produces one output. The first task reads one input from the user input and another one from the database databases, while the second and the third task accept one input parameter from the output of the previous task and another from the database.

In order to use these task nodes, application developers need to also place corresponding allocation nodes (ALLOCATOR 1..3) as they are responsible for task allocation to right devices. Allocator nodes are not connected to the rest of the flow, but they have to be placed in the workspace in order to provide their functionality, as they are triggered immediately after the application deployment.

5.5 Deployment environment

Once the application development is done, the application has to be deployed in order to become available to the end-users. In deployment phase, we can distinguish between two steps:

-Node-RED application deployment
-Late binding task allocation achieved by interaction with Docker Swarm

While the first step is the same as for any other Node-RED application, the second one is the situation where the framework components related to task allocation play their roles.

Figure 5.13 provides an overview of deployment environment.

In this step, task allocator component takes its turn. As it was said, the application developer just needs to put it inside the Node-RED environment in order to make it work. It is not connected to the rest of the flow. When application developers deploys the Node-RED application, immediately the task allocator component is triggered.

First, it selects the device where the task is going to be excuted. After that, the container-based Docker Swarm service corresponding to this task is deployed on this device, by constructing and executing the Docker Swarm service creation command from the Node-RED application. Figure 5.13 illustrates the previously described idea.

After that, the selected worker node, given by its alias looks up its own Docker image directory to check if it already has the image. If it is present, the container is created from the image and it runs. Otherwise, the node communicates the Docker Hub repository, given by image name (x86 or ARM, depends on device

architecture). If the image is present – it starts downloading it. Once the downloading is finished, the container is created and service becomes available

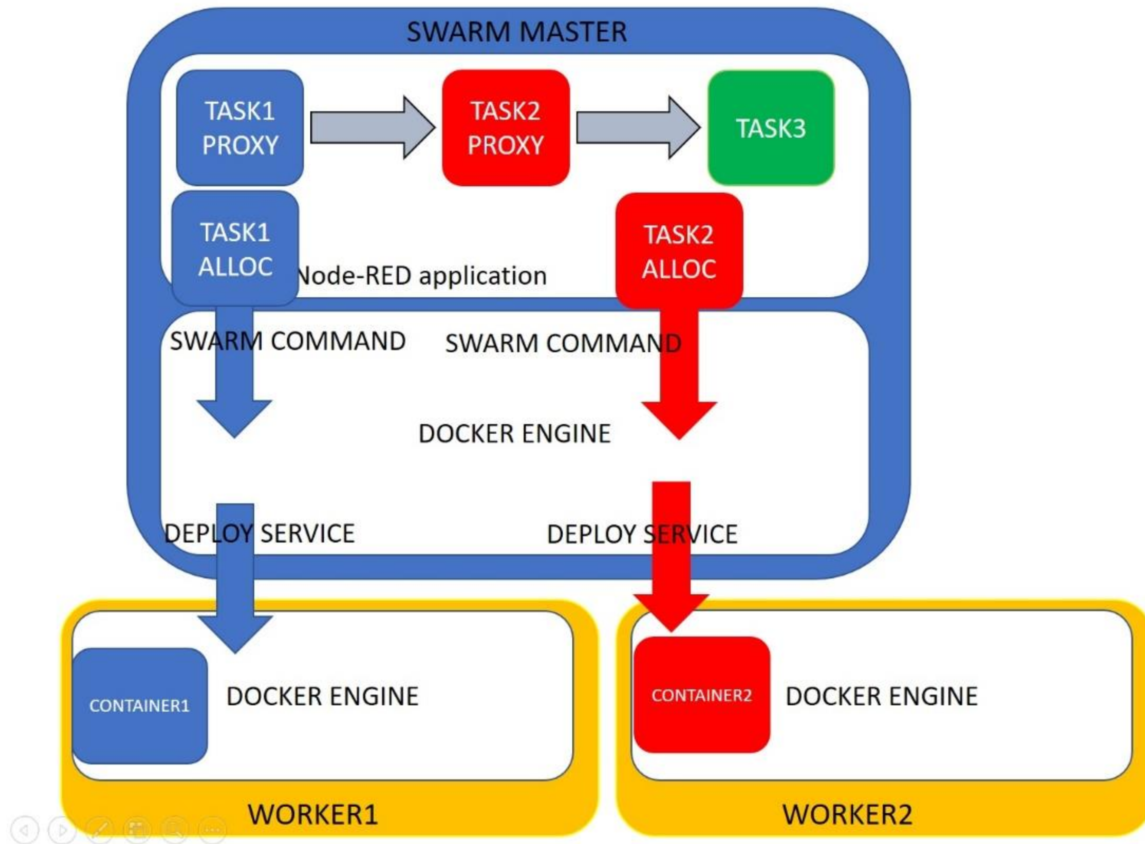

Figure 5.13 –Deployment environment overview

Figure 5.14 provides the flow of the operations when a node tries to run the image by executing the following command automatically:

```
docker run -d --name new_name run -p external_port:internal_port
repository/image_name
```

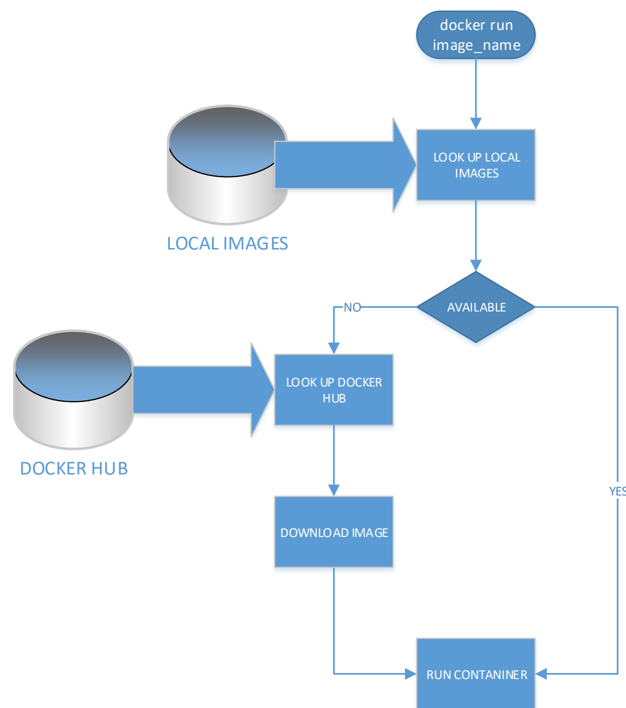

Figure 5.14 – Operations performed after docker run command

Figure 5.15 displays the complete overview of the previously described situation from the perspective of infrastructure and device communication.

One of the main advantages using Docker Swarm is a simple service access. We can access the service by hitting any of the master or worker nodes. It does not matter if the particular node does not have a container scheduled on it. That is the whole idea of the swarm – we can send a request to any node of the cluster and get the expected response.

In our case, it practically means that Node-RED application does not need to be changed after the task allocation, which makes the framework much simpler. That means that we do not need to know where is each task executed (on which

node exactly). It is enough just to use the localhost address (127.0.0.1) and correct port for each of the tasks, and it will behave in an expected manner.

In what follows, the Node-RED components taking part in this phase are going to be presented and their implmentation.

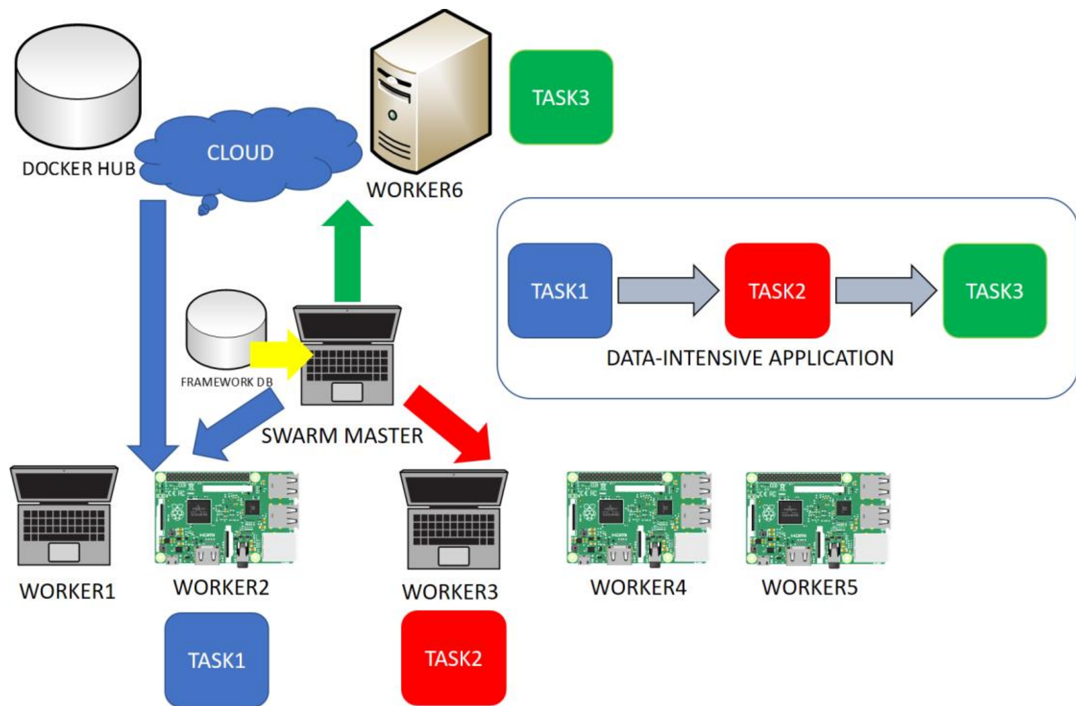

Figure 5.15 – Task allocation from the infrastructure perspective

Table 5.1 provides a complete overview of implemented components relevant to application deployment. Column named "Component" provides the name of the considered component, "Phase" denotes the lifecycle phase from *Framework overview* when the component takes its turn, "Part of" denotes a higher level component where it is contained, while "Role" explains what is the responsibility of the component inside the framework and what it is used for.

Component	Phase	Part of	Role
Node selector	Late binding task allocation	Task allocator node	It accepts a task name as an input and gives a node name as an output. Its role is to select an appropriate node to execute the given task according to a specified criteria by task developer.
Service creator	Late binding task allocation	Task allocator node	This component is responsible for the allocation of the task to the node. It creates a Docker Swarm service corresponding to the task.
Task allocator	Application develpoment Late binding task allocation	Stand-alone	This component is also imported into application developer's enviroment and is based on previously described Node selector and Service creator. It selects and allocates the task to the selected node.

Table 5.1 – Overview of implemented components involved in deployment phase

Task allocator component is created by task developers in task development phase and used by application developers in application development. It is triggered in late binding task allocation phase in order to allocate select and allocate the task to the right node. It consists of two sub-components: node selector and service creator.

In what follows, the implementation of this components is provided.

5.5.1 Node Selector

The first part of the task allocator from Figure 5.16 is implemented as a sub-component named "node selector". The goal of this node is to, depending on the given task name, determine the best node to execute the task according to the given criteria.

First, the high-level concpet of the selection algorithm is going to be presented. After that, the idea is going to be explained from the implementation perspective in terms of Node-RED primitives.

It is important to state that selection algorithms are not focus of this thesis and they are presented for illustrative purposes only. It is quite simple and does not perform any real performance metrics and cost analysis, as it is considered as a future work.

On the next page, the Figure 5.16 displays the main algorithm idea. In what follows, it is going to be described.

As an input, the algorithm accepts a task name. The first step is to obtain all the nodes which can execute the task, by using the master database.

After that, the remaining node memory and task memory occupation size are checked in order to eliminate nodes which are not able to handle more tasks due to overload.

Furthermore, the nodes are filtered according to their location. Node location must belong to the set of available task execution enviroments.

Previously mentioned node filtering criteria is considered as mandatory mechanism, while further filtering is left to the application developer. So, further

possible criteria include node filtering according to the parameters such as processing speed and network delay, memory available – the least loaded node etc. These blocks can be used in cascade for filtering, giving freedom to define specific constraints.

Finally, after all the filtering performed over the set of nodes mapped to the task, one of the nodes is selected, and the task is allocated to the node (actually the Docker Swarm service is created).

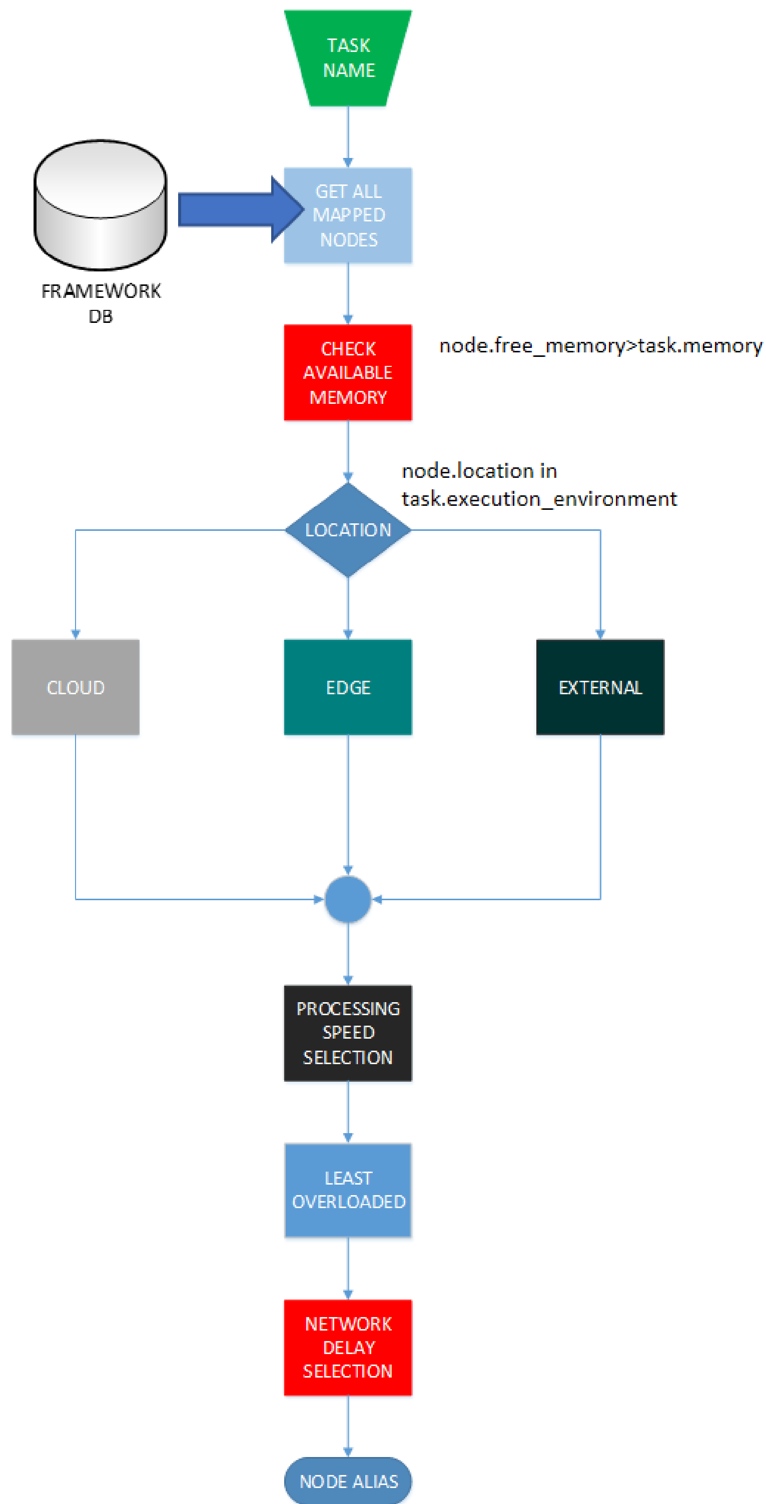

Figure 5.16 – Selection algorithm and criteria

In the implementation, there are three types of nodes participating in node selection. They are used in order to make a custom node selection strategy.

The Figure 5.17. presents the structure of the selector node.

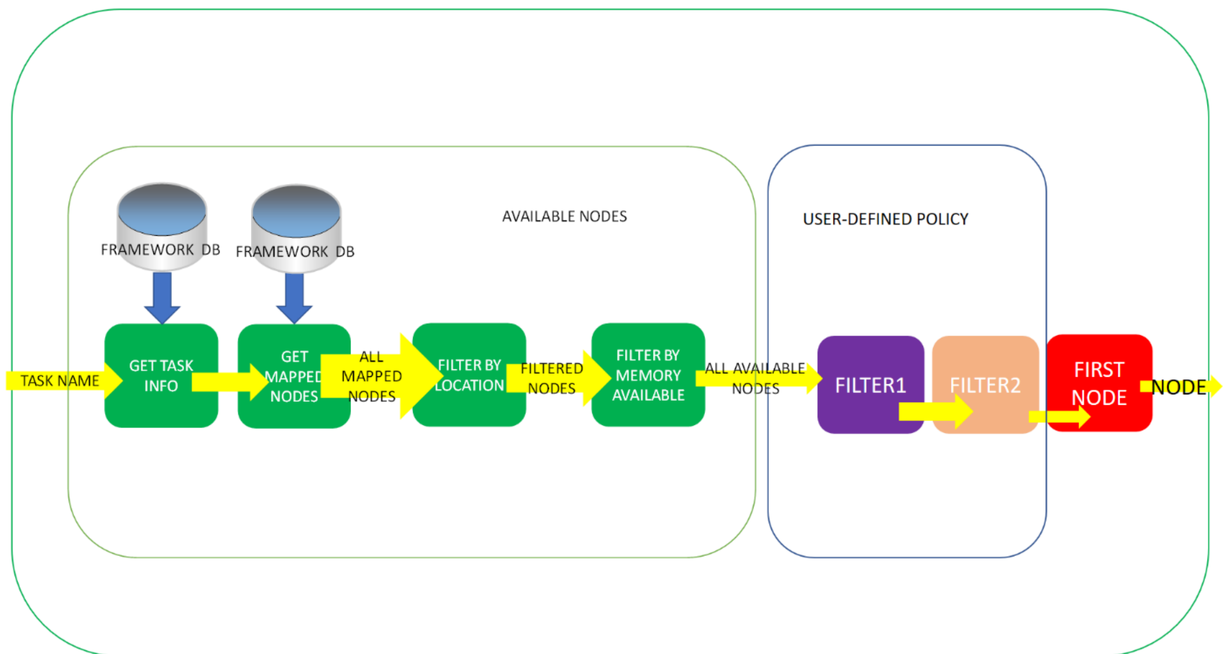

Figure 5.17 – Structure of the selector node

The first part of this sub-component is called “available nodes”. Its role is to return all the available nodes and it is a mandatory part in defining the policy and not meant to be customizable. It is applying the fundamental constriants related to task allocation, as previously told – for a given task, it checks all the mapped nodes in terms of location and available free memory. After that, all the nodes satisfying the criteria that they are performed in one of the allowed task locations and have enough available memory at the same time, are returned as an output of this node. All the node and task inforamtion used in checking these constraints is taken from the framework database. In implementation, this is a MongoDB database hosted by Mlab Cloud database provider.

The second part consists of many filter nodes nodes in cascade, defined by user in order to performed the desired policy. Among the filter nodes, there are many policies:

- *Fastest filter:* Returns the nodes which have the fastest processing speed for a fiven task.

- *Delay filter:* Returns the nodes with lowest network delay for a certain task.

- *Least loaded filter:* Returns the nodes with least ram memory occupied.

The third part is called "first node", and presents also the mandatory part of the policy defiition. The role of this node is to take the first node from the available nodes, as there can be many nodes satisfying all the given constraints. Furthermore, this node can be customized in order to return the one with the least memory occupied for example, ensuring this way that the selected node is the one which is not loaded with too many tasks.

5.5.2 Service creator

After selection of the node suitable for the task allocation, the next step is to perform the task allocation, by creating the corresponding Docker Swarm service. The Node-RED node taking this responsibility inside the framework is called "service creator".

At this point, we already know which node exactly is going to execute the task. Thus, the main function of this component is to construct the Docker Swarm command for service creation that will allocate the service to the desired node, given the task and node names.

Figure 5.18. displays the structure of the service creator node. As it can bee seen, it conists of six parts and it takes the task and node names as input and stores tha allocation information containing which node is executing the task, together with more necessary details which are going to be used in execution phase. Also, as the part of the base URL (without parameter values) which is obtained, the hostname is resolved. It can be a local host (for local devices and IaaS nodes which are part of the Docker Swarm cluster) or external IP address (in case of SaaS/PaaS).

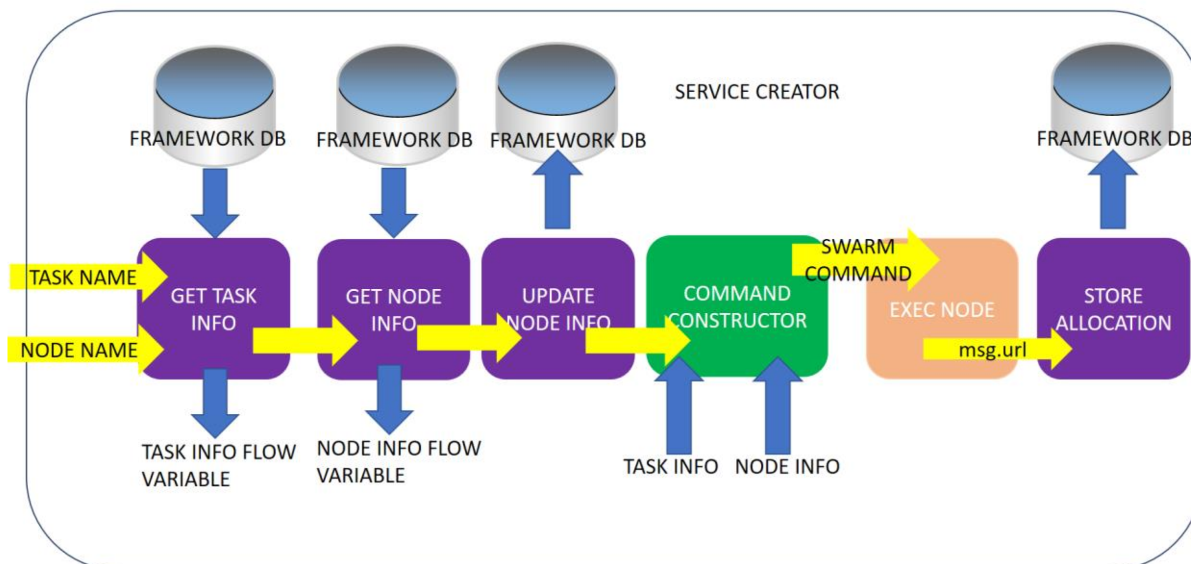

Figure 5.18 – Structure of the service creator node

The first two components find the complete information about the task and node from the framework database and store them as flow variables, so they can be used later in order to construct the Docker Swarm command.

The third part updates the remaining node free memory after the task allocation:

```
node.free_memory=node.free_memory - task.memory
```

After that, the most important component takes its role – the command constructor. It takes the node info and task info as input (previously stored as flow variables) and constructs the command that is going to be executed by the Swarm master, in order to create the service, which is actually the task allocation itself. The code inside the command constructor is presented in Listing 5.4. In what follows, this code is going to be explained.

First, the flow variables capturing the task and node info are obtained. Furthermore, the node location is checked. We can distinguish particularly between two major cases: the first one covering the usage of external nodes (SaaS/PaaS), and the second - using Cloud (IaaS) or Edge nodes. The major reason making this distinction is related to the fact that, in the second case the proper Docker Swarm command has to be constructed and executed, while in the

first case we are using external service and we do not have to create Docker Swarm service.

As the next node is in the flow of the allocator component is Node-RED pre-installed „exec" node. This node gives the ability to execute commands using the default command line interface on the machine (operating system dependent) executing the Node-RED application. In our case, this is the Swarm master itself (constraint is to use Linux, as it is going to be explained later, in framework *Requirements and constraints* section), so we need to construct the command which can be accepted by its command line interface. The exec node accepts the command to be executed as the msg.payoload parameter sent by the previous node. Thus, in the case of the external service, we just need to assign the msg.payload an empty string in order to avoid the service creation.
Otherwise, in the case of Cloud or Edge location parameter values, a command is constructed using the node and task info, in the following way:

```
docker service create -p task.port_external:task.port_internal –
-replicas 1 --network overlay_network --name task.name --
constraint 'node.label.alias==node.alias' image_name
```

Remembering the story of Docker image compatibility constraints when it comes to execution in different processor architecture environments, we will have to also distinguish between the case of x86 and ARM images. So, if the node has ARM architecture we select for image name the `task.image_arm` value from task info, while, on the other side, for x86 nodes we take the value from `task.image_x86.` Once the service create command is constructed, we assign it to msg.payload as exec node accepts it for execution.

One more important thing left to explained is how to form a msg.url, for the tasks which utilize the HTTP requests. In this case we also make distinction between external services and others.

For the case of external service, it the msg.url is formed in the following way:

```
msg.url=node.ip+":"+task.port_external+   "/"+task.url;
```

In all the other cases:

```
msg.url="127.0.0.1"+":"+task.port_external+ "/"+task.url;
```

The reason for using 127.0.0.1 as ip address has been already explained, but once again – if the service is allocated to a machine which is a part of a Docker Swarm, we can use the standard localhost address. Of course, in both cases we are using external port, as this port is exposed for the service access.

The previously URL is stored in database as a part of allocation record, which holds the information of which task is allocated to which node. It also holds the information about the paramters, host address and URL base, so it can be used by the execution proxy later, in the execution phase. Actually, the URL obtained at the late binding phase is not including the paramter values, as the values are going to be specified in the execution phase.

```
   var node=flow.get('nodeinfo');
   var task=flow.get('taskinfo');

   if(node.location=="external")
   {
       msg.payload=” ”;
       msg.url=node.ip+":"+task.port_external+   "/"+task.url;
   }
   else
       {
           if (node.arch=="x86")
           {
               msg.payload="docker      service      create      -p
"+task.port_external+":"+task.port_internal+
                   "  --replicas  1  --network  my-net  --name  "+
task.id+
                   "  --constraint  'node.labels.alias=="+  node.id+
"' "+task.image_x86;
           }
           else
           {
           msg.payload="docker       service       create       -p
"+task.port_external+":"+taski.port_internal+
             " --replicas 1 --network my-net --name "+ task.id+
             "   --constraint   'node.labels.alias=="+   node.id+   "'
"+task.image_arm;
           }

           msg.url="127.0.0.1"+":"+task.port_external+
"/"+task.url;
       }
   return msg;
```

Listing 5.4 – Code of the command constructor

5.5.3 Task allocator

As it was said prevoiously, the task allocation steps (node selection, service creation) are done in late binding phase, immadiately after deployment. In what follows, the implementation of this mechanism in the framework combining the node selector and service creator is going to be explained (Figure 5.19).

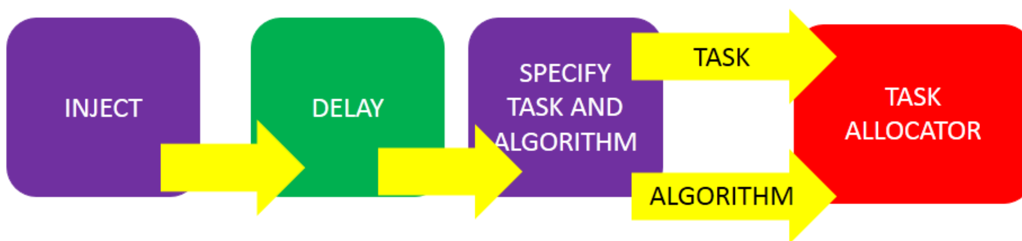

Figure 5.19 – Late binding task allocation implementation

The first node in the flow is the "inject" node. It belongs to set of standard pre-installed Node-RED input nodes and is used to inject a message payload, based on a certain time template. The message content could be a string, number, flow or timestamp. The time template can be – interval (repeating injection after a given period of time), certian moment (select the hour and minute of the day when to trigger the inject node) or just once at the start (after deployment).In our case, as we wanted to achieve an effect of late binding, we selected „once at the start" , as we want to trigger the task allocation flow only once, immediately after the deployment of the applicaiton.

The second node is „delay" node. It also belongs to pre-installed set of nodes and is used to hold the flow for a given time. In this case, it is necessary for the implementation, in order to avoid the loss of the inject trigger. When we deploy the applicaiton, it might take a while to initialize all the flows, especially if the application contains a lot of nodes. If the injection trigger is done during the initialization, it is not going to be caught and is lost. So, a delay of few seconds is introduced in order to avoid the injection trigger loss.

The next node specifies which task is going to be allocated, and by using which algorithm from the available pre-defined algorithms. It explicitly sets the value of msg.task and msg.algorithm properties. This is done explicitly this way due to idea of generating nodes from templates, which is going to be explained later.

And, finally the task allocator takes the task and algorithm names in order to allocate a given task according to the given criteria to the right node. Figure 5.20 displays the task allocator structure. As it can be seen, it is different than one presented as a part of Figure 5.4. This is due to introduction of algorithm selection switch, which gives capability to dynamically select the algorithm according to the given name of the algorithm.

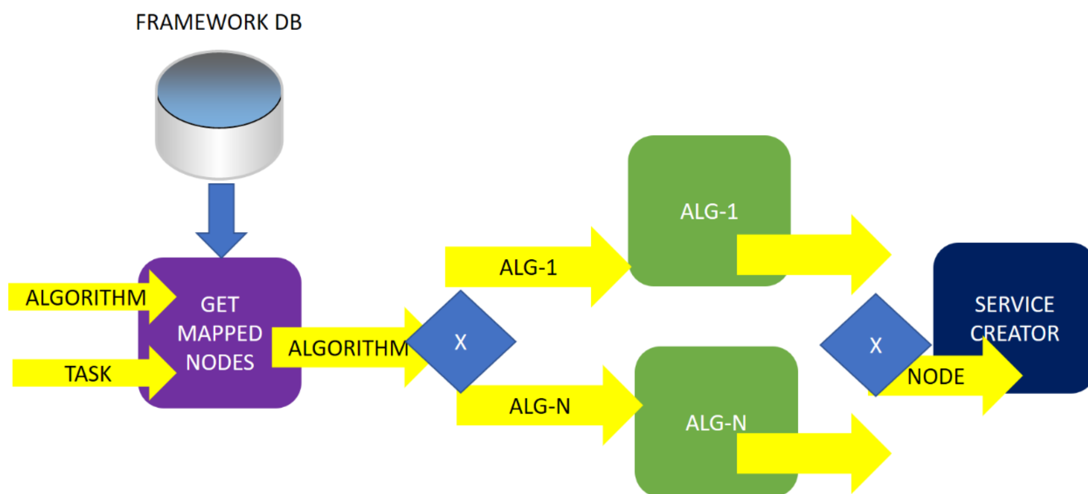

Figure 5.20 – Modified task allocator for late binding

First, all the mapped nodes are obtained. Then, according to the algorithm input parameter, the corresponding algorithm is selected to find an appropriate node. Once the node is selected according to the specified algorithm, the Docker Swarm service can be crearted. After that, the service is ready to be executed and late binding task allocation phase is completed.

5.6 Execution environment

Once the tasks, are allocated to right nodes, the execution can be performed. The execution is actually sending the HTTP request submitted by end-users to the right device, which was selected in late-binding phase of the deployment.

If the selected device belongs to Docker Swarm (either within Edge or IaaS), the HTTP request is sent to the local host, which has a role of the Dokcer Swarm master at the same time. Internal host resolution mechanism of Dokcer Swarm master locates the device responsible for task execution and redirects it to the correct device, based on service ports. In this case, we have interaction of Node-RED application and Docker Swarm.

In other case, if the device belongs to SaaS/PaaS category, the HTTP request is sent directly to the device, using its public IP address. This time, we do not have any Interaction with Docker Swarm.

In execution phase, the main role in our framework belongs to the component called task execution proxy. It sends HTTP request and outputs the HTTP response.

Figure 5.21 presents provides an overview of the executing environment by using an example of the data-intensive application containing three tasks. For first two tasks, HTTP requests are sent to the local host. Then, the Docker Swarm resolves the host where is the task allocated and HTTP request sent to this address taking into account the values of the parameters defined by the end-users. For the third task, the HTTP request is just sent to the host address without Docker Swarm interaction.

In what follows, the task execution proxy component and its URL making mechanism are going to be described.

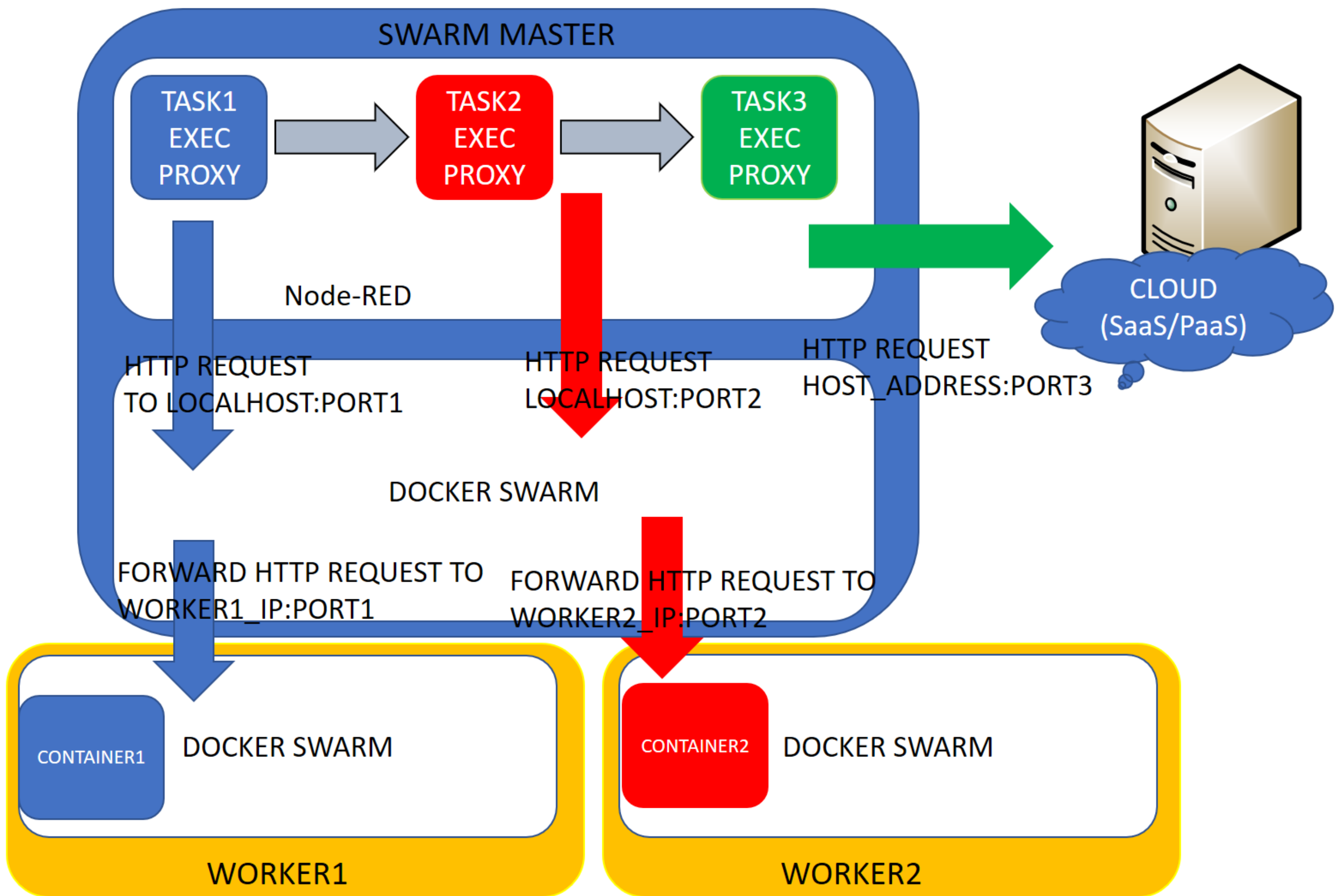

Figure 5.21 – Execution environment overview

5.6.1 Task execution proxy

It is created and exported by task developers using the task development environment based on Node-RED, imported and used by application developers inside their Node-RED application development environment in order to develop their own data-intensive applications. The role of this component is to provide the service defined by the task developers to the application developer regardless of the location of the device executing the actual task.

Figure 5.22 displays its structure.

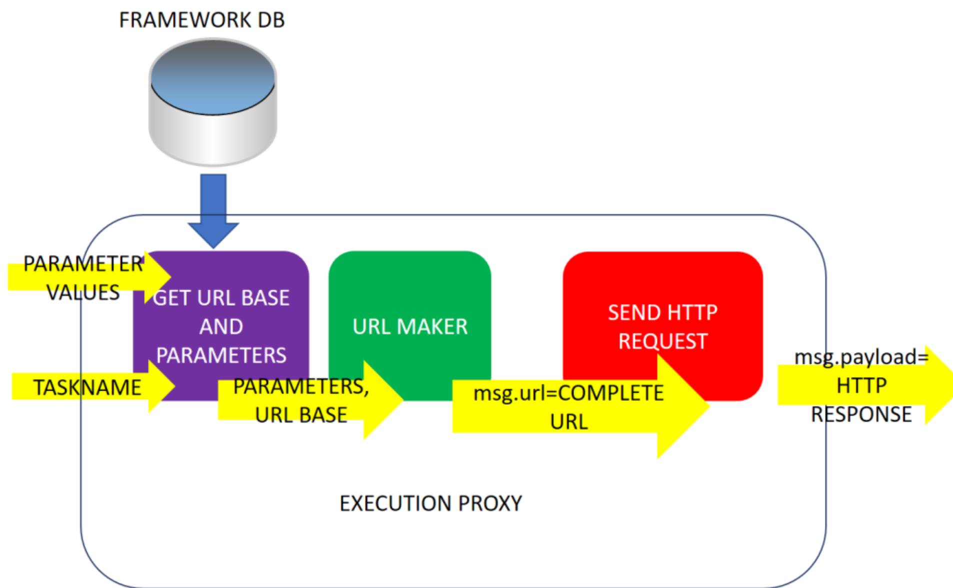

Figure 5.22 – Structure of the execution proxy

As an input, the execution proxy takes the task name and the parameter values. Parameter values are set by the end-user by using GUI, while the task name is hard-coded for a specific task during the task development phase.

Then, the next step is to read the allocation information from the database (URL base containing the host address and parameters), which was inserted in late binidng phase immediately after the deployment.

Furthermore, based on this inforamtion, a proper URL can be constructed (URL maker taking this role) using the URL base (in form *`host_address:port`*) concrete parameter values, specified by the user.

Once the URL is completed, it can be used forwarded to the HTTP request node, as msg.url parameter. HTTP request node is a part of the standard pre-installed nodes of the Node-RED develpoment enviroment. It takes the URL as a msg.url property of the message and sends an HTTP request, so the URL MAKER has to assign this result to the msg.url property in order to forward it correctly to the HTTP request node.

In what follows, the URL maker code is provided (Listing 5.5):

```
msg.limit = 5;
msg.skip = 0;
var result=msg.payload.url;
var url_base=result.url_base;
var final_url=url_base+"?";

var parameter1=msg.parameter1;
var parameter2=msg.parameter2;

final_url=final_url+"parameter1="+parameter1;
final_url=final_url+"parameter2="+parameter2;



msg.url=final_url;
return msg;
```

Listing 5.5 – URL MAKER code

As we can see, the URL base is taken from the result and parameter values concatenated in order to construct the valid URL.

After the URL construction, HTTP request node sends an HTTP request to the device offering the service. HTTP response is taken and placed into msg.payload parameter, so it can be displayed or used an input within a node chain inside the flow.

5.7 Prerequisites and limitations

While the framework is providing the fucntionality in the environment composed of machines with different architectures, and different locations, there are still certain constraints and limitaions which restrict the freedom of hardware and operating system chioce for some parts of the framework. Most of these constraints are heavily influenced by the technology behind it. In this section, conditions in which the framework is functional are going to be presented, together with the most important limitations and constraints. Also, the framework needs a certain set up and configuration before it is ready to be used.

5.7.1 Prerequisites

The first prerequisite is that all the machines which are about to join the Swarm cluster must have the Docker Engine installed. When it comes to the master node, it also must be have Node-RED installed and have access to a database which holds the necessary information about nodes, tasks and mapping described in *Framework database design*. In this case, a MongoDB SaaS database in Cloud by Mlab was used for that purpose.

Furthermore, the all the nodes under application developer's control (both in Cloud (IaaS) and Edge) need to be part of a Docker Swarm, which is properly configured. Thus, before using the framework we must generate a Swarm token by master, distribute it among the nodes to join them to the cluster. [25]

Docker Swarm configuration

The main machine of the cluster based on Docker Swarm is called master and is the one responsible for cluster management (it can be replicated, so there could be more than one master, in fact). In Docker Swarm terminology, tasks are called services. Main cluster management operations include creating and removing services, scaling them up and down (as each service can be replicated and executed on multiple machines). Master can also execute tasks itself.

Other machines belonging to the cluster are called workers. Their main role is task execution. There are no constraints when it comes to processor architecture – the only requirement is to run the Docker engine and the machine could be

within Edge or in Cloud. Masters and workers could be either ARM or x86 devices, but it is recommended to use high performance servers for master role, as the master itself is a single point of failure.

The first step, before using the framework is to create a swarm on the master. So, in order to create the swarm the following command is performed on the master node:

```
docker swarm init
```

After that, the swarm token is generated. This token is used by worker nodes in order to join the swarm.

So, the next step is to distribute the token to future worker nodes, so they can join the swarm. The command executed on the worker nodes has the following structure:

```
docker swarm join –-token token_value master_ip:port
```

Token value is a string which is used to for access, while master node ip address and port for swarm are used also for the join command.

The final step is to create an overlay network, so we can provide ability for multihost container networking. The swarm makes the overlay network available only to nodes in the swarm that require it for a service. When we create a service that uses an overlay network, the manager node automatically extends the overlay network to nodes that run service tasks. The command which creates an overlay network is the following:

```
docker network create --driver overlay --subnet=10.0.9.0/24 network_name
```

Figure 5.23 summarizes the steps to be performed in order to create a cluster based on Docker Swarm technology [25] (on the right side of the figure) and illustrates the underlying infrastructure (on the left).

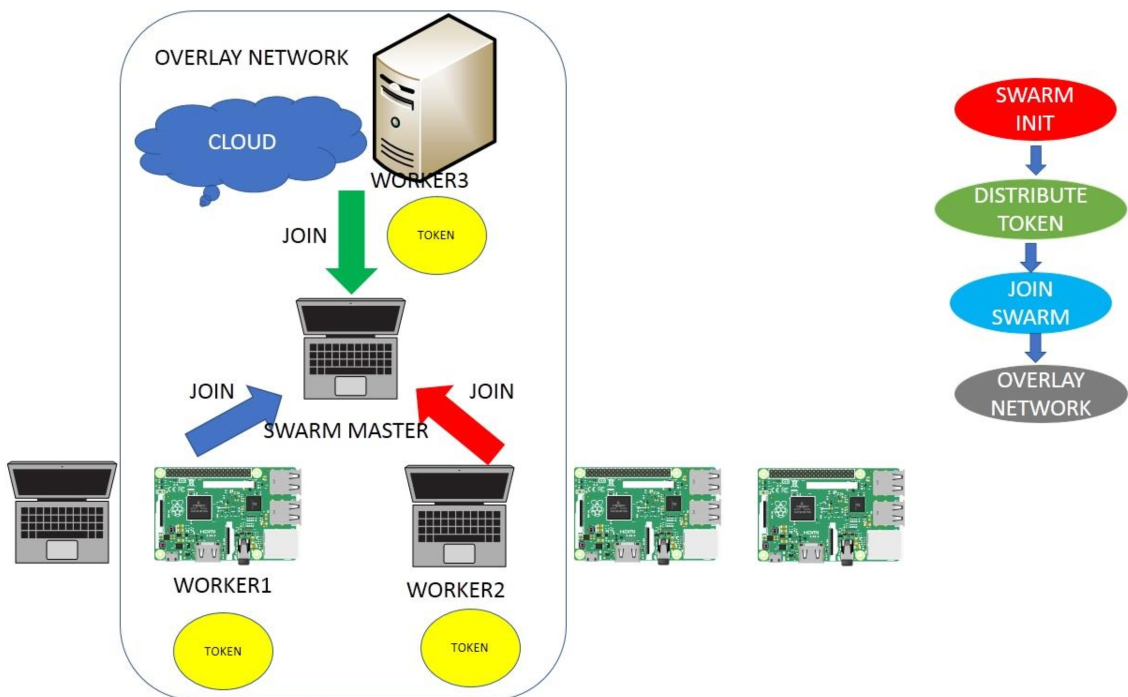

Figure 5.23 – Docker swarm initialization

Also, as attribute named *alias* is used for node selection during the phase of Swarm service creation (discussed in *Conceptual overview and data model*), all the nodes must have labels which are assigned by the master node, by execution of the following command:

```
docker node update  --label-add alias=node.alias node_id
```

Once the swarm is created, the framework is ready to be used for task allocation.

5.7.2 Constraints

Considering the Docker Swarm, there is an important constraint coming from its architecture. Despite the advantage of using localhost address for all URLs, there is a limitation related to the ports used as a drawback. This is related to external ports which are used for external service access – we cannot use the same external port for more than one service, if the master has exposed that port previously. But, when it comes to internal ports – there are no such constraints, as they are used internally by the devices where the service is actually created.

When it comes to the Node-RED framework, there are two limitations of the execution environment. Both of them are related to operating system constraints.

As it was explained in *Task allocator* sub-section, the framework component is using an exec node to redirecting the value of `msg.payload` to the command line interface of the operating system. As different operating systems have different CLI, these commands are going to work only on Linux. So, the constraint is that the master node must run some of the Linux operating system distributions (either x86 or ARM).

Another constraint related to execution enviroment is that Node-RED framework for task allocation should not be run as a container, as process isolation mechanisms of containers prevent the direct interaction with the command line interface of the real machine.

It is also important to state that all the containers used for task considered are Linux containers (both x86 and ARM). Windows and Macintosh native container technologies do exist for newer versions of these operating systems (called Docker for Windows and Macintosh), but they are quite new concept and would make the situation more complicated, when it comes to support and base image availability, managing additional images for each task - so they were avoided for this thesis. The original container concpet orignates from Linux and is considered as something already mature enough, with large community online contributing to both x86 and ARM architecture Docker Hub base image availability. One reason which makes Linux containers as universal solution is the fact that the original Docker Engine for Windows and Macintosh (Docker Toolbox) was based on Linux virtual machine running behind. Thus, these containers run with no problem on Windows and Macintosh machines using Docker Toolbox. For older Windows and Macintosh versions, this is still the only way to run contaniers.

But, in general, nothing prevents worker nodes from running any other operating system which has support for Docker Engine and the framework is still functional in this case, given that the corresponding operating system-specific images are provided.

5.7.3 Hardware and software specification

In previous two sub-sextions, prerequisites and constraints were discussed. Taking these facts into account, hardware and software specification is derived.

In what follows, a table presenting the summary of necessary hardware and software specification to utilize the framework and its constraints is provided (Table 5.3):

	Master	Worker
Operating system	Linux distribution	Any operating system supporting Docker Engine
CPU architecture	X86 (recommended) ARM	X86 ARM
Docker Engine	Installed	Installed
Node-RED	Installed	-
Internet connection	Necessary	Necessary

Table 5.2 – Hardware and software specification summary

It is necessary that all the nodes have an active internet connection, as Docker Hub registry is used as container image and Swarm token discovery service. The alternative is to have a local private registry, accessible only by the organization.

When it comes to the choice of Swarm master machine, it is recommended to avoid using ARM-based devices, as master has more responsibility than worker nodes and it would have a huge impact on whole cluster performance. Also, as the master itself is a point of failure, it should be taken into account during the machine selection – so more realiable machine is recommended for the master role.

6 Case study

In this chapter, an example from healthcare domain is going to be considered in order to show how the framework can be used in real situations. This domain is selected as constraints related to data privacy are here common, and it is designed to demonstrate the capabilities of task movement. The application has been implemented and serves as a proof-of-concept of the previously built framework. In what follows the scenario of the application is going to be presented, overview and architecture of the developed application, implementation and experiments performed in order to test the capabilities of the developed framework.

6.1 Healthcare scenario example

In what follows, the scenario of application usage is going to be presented.

Let us assume that patient comes with symptoms such as chest pain, which make the doctor suspect that the cause of them is actually an infraction. In order to make sure if she is right, the doctor needs to perform two examinations: troponin test and ECG. If both of them show state different than normal, it means that they were right that patient had an infraction.

Troponin test is used to determine whether the amount of present troponin in blood is higher than normal. During the infraction and some hours after, its value remains relatively high compared – more than 0.1 ng/ml in case of troponin T. We assume that there is a device in hospital performing this measure and it is controlled by single-board computer, such as Raspberry Pi. The value obtained is stored into MongoDB database inside the Raspberry Pi, together with the corresponding patient identification. It is important that this information remains within the boundaries of the organization (hospital) in raw form (non-processed), because of the legal constraints.

So, the information stored after the measurement in this case is:

TROPONIN-TEST (patient, value)
patient -a string used to identify patients
value – a float value representing the amount of troponin in blood

After that, the value recorded needs to be checked whether it is greater than the given threshold or not. For example, let us assume that for the troponin test result determination we use a php script as given by code in Listing 6.1. This php is executed also on a device inside the organization, as we still need to use raw patient data.

```
<?php
    if (isset($_GET['troponin']))
    {
        if(floatval($_GET['troponin'])>0.01)
        {
            echo "1";
        }
        else
        {
            echo "0";
        }
    }
    else
    {
    }
?>
```

Listing 6.1 – Troponin test PHP script

Once the processing is performed, we can store the test results into a database. In this case we do not have raw data with restricted movement due to legal issues, so we are also free to use an online database Cloud provider. The information of the troponin test result obtained consists of following:

TROPONIN-RESULT (patient, result)
patient -a string used to identify patients
result – an integer number providing us information whether the troponin value is increased (1) or not (0).

ECG is captured by a medical device connected to a Raspberry Pi, and the result obtained is stored inside the Raspberry Pi (an image). The Raspberry Pi has its own database which keeps evidence of the ECGs performed for different patients. This image movement is legally constrained due to law regulations related to

privacy and ownership, so it should remain within boundaries of the healthcare organization which performed the test, in its raw form. The information obtained is:

ECG-TEST (patient, value)

patient -a string used to identify patients

image – a string representing the path of the image obtained during the test

In order to determine whether the ECG is normal or shows anomalies, it is needed to analyze the image obtained. As the test itself is using an image whose movement is restricted outside the healthcare organization, it means that it is also going to be performed within the boundaries of the organization by the machine which belongs to the organization. The code performing the test is shown in Listing 6.2.

```
<?php
    if (isset($_GET['ecg'])) {
      $ecg_value=rand(1,100);
      if($ecg_value>50)
      {
         echo "1";
      }

      else
      {
            echo "0";
      }
    }else{
    }
?>
```

Listing 6.2 – ECG test PHP script

As it can be seen, this is not a real ECG image analysis code, but only a script for illustrative purposes, as the goal is not to have a real ECG analysis, but just an example of a code which is executed within boundaries of the healthcare organization. What this code does actually is checking whether the image path is provided and then, a random number between 1 and 100 is generated. If the number is greater than 50, which would have a meaning of presence of anomalies in ECG image - than the output is 1; while it is 0 otherwise.

After the result obtained, it is stored in database. The database could also be provided by online provider, as the data is considered in processed form (not raw) so we have a freedom of movement.The information stored in this case is:

ECG-RESULT (patient, result)
patient -a string used to identify patients
result – an integer number providing us information whether the ECG contains anomalies (1) or not (0).

And, finally the infraction test is performed by using the both values of ECG and troponin test. It works this way – if both ECG shows anomalies (value 1) and troponin has increased value (value 1), it is considered as an infraction state. The code performing the test is provided as Listing 6.3.

As this script uses the processed patient data, it is not mandatory to constraint the execution only to the boundaries of the organization.

But, as the result is sensitive to legal issues, it has to be stored within the boundaries of the healthcare organization. The information created as a result of of an infraction test:

INFRACTION-RESULT (patient, result)
patient -a string used to identify patients
result – an integer number providing us information whether the patient has infraction (1) or not (0).

```
<?php
    if (isset($_GET['ecg'])  && isset($_GET['troponin'])   ) {
      $ecg_value=intval($_GET['ecg']);
      $troponin_value=intval($_GET['troponin']);
      $outcome=$ecg_value*$troponin_value;
      echo $outcome;
    }else{

    }
?>
```

Listing 6.3. – Infraction test PHP script

6.2 Application overview and architecture

In this section, the previously described scenario is going to be presented from the perspective of data-intensive application (which was implemented and used for task allocation experiments), broken down into single tasks allocated to nodes. In the later part of this section, the architecture of the implemented application is going to be presented.

The tasks can be divided, in case of this application in two categories: database-oriented, on one side and processing tasks, on the other side.

All the tasks related to troponin, ECG and infraction test result data belong into this category. Among them, we can make further distinction between data reading and data storage tasks.

On the other side, we have data processing tasks, which are actually executed as the PHP scripts presented in code Listing 6.1-6.3.

The illustration of the case study task flow and infrastructure used, is presented in Figure 6.1.

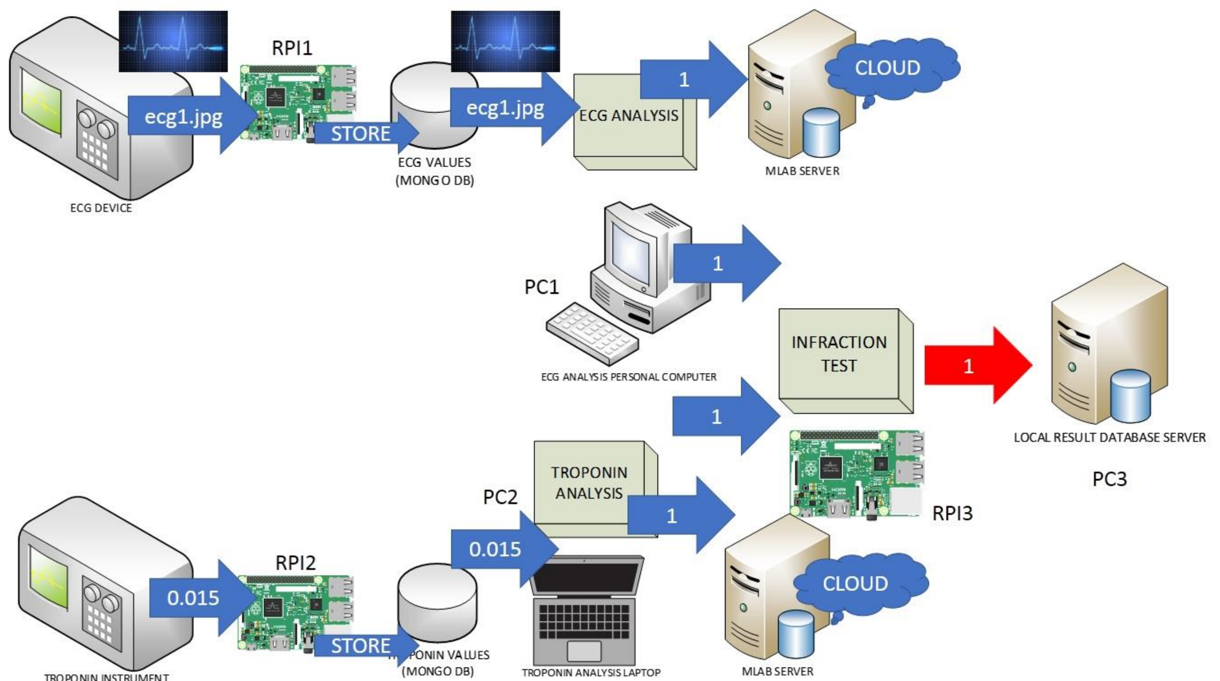

Figure 6.1 – Case study illustration

The situation displayed is the case when the patient infraction test result is 1, because troponin is larger than the threshold and ECG image contained anomalies.

As we can see, there are three types of processing tasks, displayed as gray boxes: troponin analysis, ECG analysis and infraction test. All of them are encapsulated inside Apache PHP server containers with both ARM and x86 versions.

When it comes to data-oriented tasks, they are either packed within database servers (troponin data, ECG data and infraction results or external services are used (troponin and ECG results). All of the databases used are based on MongoDB technology except the results databases which are using MySQL technology.

In what follows, Table 6.1 is presented summarizing, the information about tasks – including the container images, ports, execution location, URL and execution node.

As it can be seen, the image information about the troponin and ECG results is omitted as external services are used and service is not created in these situations.

For external ports, for the tasks based on same technologies, consecutive values are used due to master node external port limitation explained in the last part of *Framework for Computation Task Movement in Fog Environment* chapter, titled *Constraints*.

Also, URLs are available only for data processing tasks as they are used for making HTTP requests.

Execution node is corresponding node in Node-RED used for the execution. When it comes to PHP-based Apache server tasks, HTTP request nodes are used in order to send HTTP requests formed by using URL, port and node IP information (if necessary). These nodes come pre-installed with Node-RED. For data-oriented tasks, specific nodes are used for MySQL and MongoDB. However, these nodes are not pre-installed, so they have to be manually added to the node palette using Node-RED palette manager, downloaded and installed before use.

Task name	Image x86	Image ARM	External Port	Internal Port	Execution	URL	Execution node
Troponin analysis	vicviper/troponin	vicviper/troponin-arm	82	80	Edge	troponin.php	HTTP request
ECG analysis	vicviper/ecg	vicviper/ecg-arm	81	80	Edge	ecg.php	HTTP request
Infraction test	vicviper/infraction	vicviper/infraction-arm	83	80	Edge	test.php	HTTP request
Troponin data	vicviper/mongo-troponin	vicviper/mongo-troponin-arm	27017	27017	Edge	-	MongoDB
ECG data	vicviper/mongo-ecg	vicviper/mongo-ecg-arm	27018	27017	Edge	-	MongoDB
Troponin results	-	-	27017	-	External	-	MongoDB
ECG results	-	-	27017	-	External	-	MongoDB
Infraction results	vicviper/mysql-infraction	-	3306	3306	Edge	-	MySQL

Table 6.1 – Tasks and information about ports used and images

When it comes to machines executing these tasks, the application is executed in mixed-architecture environment, with nodes located in both Cloud and Edge.

Among the available devices, there are three Raspberry Pi model 3 single-board computers, three laptops and two Cloud hosting database proveder services – one for MySQL (MySQLFree Hosting) and one for MongoDB databases (Mlab). The real Cloud virtual machines (Amazon AWS, Google Compute Engine) were not used due to high pricing plans not suitable for this purpose. POLIMICLOUD virtual machines were considered, but, in the end, they were not used due to number of network access restrictions and port usage limitations.

In what follows, a table (Table 6.2) of devices used as nodes executing the allocated tasks and its main characteristics is presented.

It can be seen that for external services some attributes are left out. The reason for this is because they are unknown and irrelevant for us, as they are used as a black box from the perspective of our framework. As they are not part of our

Docker Swarm, we are not able to execute arbitrary tasks, but just use them for a single purpose.

Node name	IP	Architecture	Model	Provider	Location
rpi1	10.79.4.153	ARM	Raspberry Pi 3	Local	Edge
rpi2	10.79.4.159	ARM	Raspberry Pi 3	Local	Edge
rpi3	10.79.4.246	ARM	Raspberry Pi 3	Local	Edge
pc1	10.79.5.169	x86	Toshiba c55	Local	Edge
pc2	10.79.4.225	x86	Acer 5553g	Local	Edge
pc3	10.79.5.179	X86	HP 15bc201nl	Local	Edge
external1	sql11.freemysqlhosting.net	-	-	FreeMySQL Hosting	External
external2	ds113841.mlab.com	-	-	Mlab	External

Table 6.2 – Devices participating as nodes in healthcare application case study

In what follows, the architecture of the application is going to be presented.

From the end-user perspective, the application has client-server architecture. On the client side, we have HTML pages run inside web browser, while three is a Node-RED application running on the server side, supported by a cluster (Docker Swarm) of machines executing the corresponding tasks. The Node-RED application server is actually a Docker Swarm master. It is allocating the application tasks to worker nodes and providing services for the client. The user sends HTTP requests using the web page which are then executed in Node-RED. During the execution, various tasks occur, so the master is redirecting the execution of these tasks to the corresponding worker machine, according to the port of the service which is used.

Figure 6.2 presents the architecture of the implemented application.

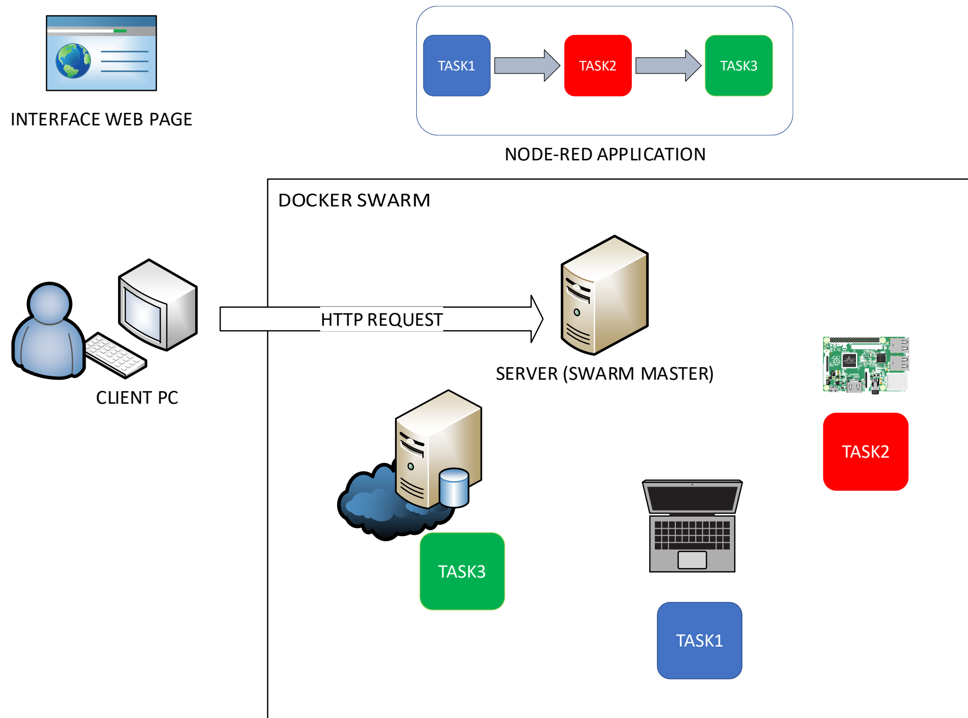

Figure 6.2 – Case study application architecture

6.3 Application interface

The application consists of several services, which are called by HTTP requests sent to the Swarm master via port 1880 (which is publicly exposed for Node-RED application deployments).

HTTP requests which are generated, have the following form:

http://***master_ip***:1880/***service_name***?parameter1=value1¶meter2=value2

In what follows, these services are presented in Table 6.3, together with their names, necessary input parameters and effects they cause. The first six of them are end-user services, supporting the operations related to the discussed healthcare scenario itself, while the last three are administrator-oriented and are used for task allocation and deallocation. Actually, they are based on the previously defined framework and demonstrating its capabilities.
The decision to separate task allocation and the execution itself is made due to the following reason - the download speed of the images corresponding to the allocated task, if image is large (hundreds of megabytes), is not fast enough to perform it immediately, so it could take more time to download the image and run the container. Thus, having a task proxy which is performing both the allocation of the task and its execution in sequence, would lead to errors and delays, as it is possible that larger images could take more to download and run and might happen that some of containers in not running yet. It is also possible to perform allocation of tasks in "late-binding" style, but could lead to undesired slowdowns and errors if image is not already downloaded to device which is going to perform the execution of the task. It means that task allocation needs to be done explicitly before the task execution.

Service name	Input parameter	Effect
`ecg_test`	patient – string which represents an id of the patient	Performs a test based on ECG image of the patient given by her id and stores the outcome in g result database.
`troponin_test`	patient – string which represents an id of the patient	Performs a test based on troponin value of the patient given by her id and stores the outcome in troponin result database.
`final_test`	patient – string which represents an id of the patient	Performs a test based on troponin and ECG test results of the patient given

		by her id and stores the outcome in infraction result database.
ecg_result	patient – string which represents an id of the patient	Returns an outcome for the ecg test stored in ecg result database for a given patient.
troponin_result	patient – string which represents an id of the patient	Returns an outcome for the troponin test stored in ecg troponin database for a given patient.
final_result	patient – string which represents an id of the patient	Returns an outcome for the infraction test stored infraction result database for a given patient.
mapper	task – name of the task we want to allocate to a node node -name of the node where we want to allocate a given task	Task given by its name is allocated to the given node.
destroyer	task – name of the task we want to remove	Task given by its name is destroyed.
algorithm_selector	task – name of the task we want to allocate to a node algorithm -name of the algorithm which we want to use for allocation of the specified task.	Task given by its name is allocated to the given node.

Table 6.3 – Case study example application service parameters

For easier demonstration and presentation of the application, HTML-based web interface has been created which is providing graphical user interface where previously mentioned parameters of the functions can be specified. HTML web

pages are actually providing a graphical interface to specify parameters, and everything is translated into HTTP requests.

In the interface, there are three major parts with separate web pages: tests, results and dashboard.

The first part of the GUI is providing interface for first three functions from the Table 6.3. The user specifies the patient id in order to perform the tests for the desired patient – troponin test, ECG test and infraction test. In order to get the result for infraction test make sure that both troponin and ECG tests have been performed already. Figure 6.3 displays the corresponding web page.

The second part is providing interface for next three functions from the Table 6.3. The user specifies the desired patient id, and gets the result of the previously described tests. Figure 6.4 displays the corresponding web page.

INFRACTION TESTS
patient:
Troponin test
patient:
ECG test
patient:
Infraction test

Figure 6.3 – Infraction tests HTML web page

INFRACTION RESULTS
patient:
Troponin result
patient:
ECG result
patient:
Infraction result

Figure 6.4 – Infraction tests HTML web page

The third part of the web-based interface is a dashboard, which is a system administrator tool, giving ability to allocate/deallocate tasks to nodes. Allocation of tasks could be done in two ways: specifying both the task and node names in order to manually allocate the task to the desired node or by specifying the desired algorithm which is going to be used for the allocation, from the list of available algorithms. These algorithms give ability to select a node to execute the given task, following some policy – first available node which can execute the task, first node with ARM architecture, node which is the fastest for certain task, the one with smallest network delay or largest size of free RAM memory. The GUI offers a selection of the algorithm as a drop-down list of available algorithms. Also, the functionality to destroy a task is given. It is present for test purposes in order to demonstrate capabilities of task allocation to different nodes and using various algorithms. Figure 6.5 shows the interface of the system administrator dashboard.

As Figure 6.5 shows, there is a drop-down list offering a set of prepared algorithms. By pressing the "ALLOCATE", the allocation is performed. As the emphasis is not on selection criteria and cost estimation algorithms, the offered algorithms are quite simple and used for experiments and proof of concept:

- *first:* Selects the first available mapped node[29] able to execute the task

- *arm:* selects the first node from available mapped nodes which has ARM architecture

- *X86:* selects the first node from available mapped nodes with x86 architecture

- *fastest:* selects the node with the highest processing speed for the given task from available mapped nodes

- *ram:* Selects the node with the highest amount of free RAM memory available present among available mapped nodes

[29] Available mapped nodes represent a set of nodes matching the execution location and available memory criteria with respect to a given task

- *delay:* Selects the node with the lowest network delay (based on ping) for the given task rom available mapped node

TASK MAPPER

task: ecg_processing node: pc1 ALLOCATE

ecg_processing DELETE

task: ecg_processing node: pc1 ALLOCATE

Figure 6.5 – System administrator dashboard

6.4 Node-RED application implementation

In what follows, Node-RED flows implementing the most important aspects of the application are going to be presented.

Figure 6.6 presents the flow implementing the service named `ecg_test`. ECG image is taken from the ECG image database, processed to check if it contains anomalies, and in the end, the result is stored into ECG result database.

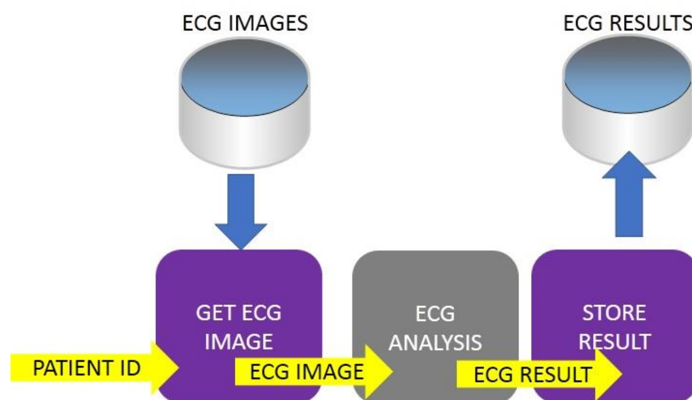

Figure 6.6 – Node-RED flow implementing `ecg_test`

Figure 6.7 presents the flow implementing the service named `troponin_test`. Troponin value is taken from the troponin value database, processed to check if it has value greater than the threshold and the result is stored into troponin result database.

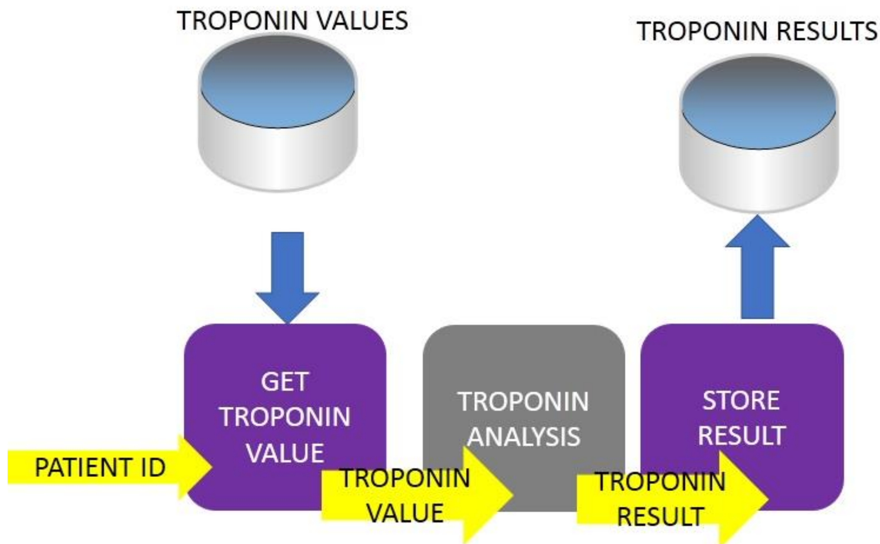

Figure 6.7 – Node-RED flow implementing `troponin_test`

Figure 6.8 presents the flow implementing the service named `final_test`. Troponin and ECG results are taken from the corresponding databases, checked if both of the values are 1, and the result is stored into infraction result database.

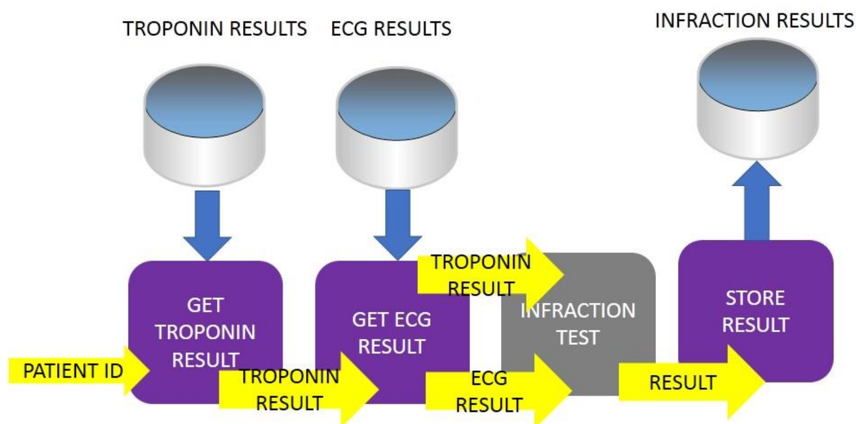

Figure 6.8 – Node-RED flow implementing `final_test`

One more aspect of the implemented Node-RED application is task mapper. It is available in two variants: the first one is allocating task to a given node (`mapper`), while the second taking task name and selection algorithm as input variables, and performing task allocation to the node selected by the algorithm (`algorithm_selector`). Figure 6.9 presents the implementation of the later variant. Algorithm selection is performed, so one of the paths corresponding to the steps necessary by the selected algorithm is executed (ALG-1 or ALG-N) in XOR manner (in terms of BPMN). After that, the name of the selected node is taken and forwarded to task allocator which is responsible for the allocation of the task to the chosen node.

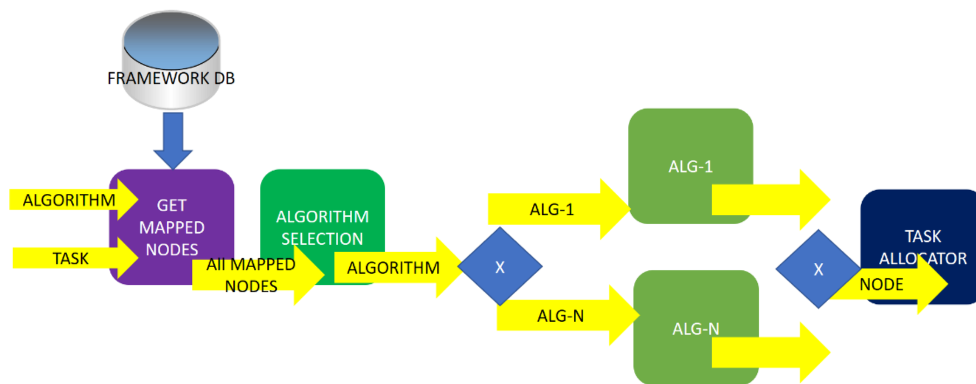

Figure 6.9 – Node-RED flow implementing `algorithm_selector`

The source code of the application, together with necessary data and GUI is available on GitHub, inside the repository that can be accessed through the following link:

https://github.com/nenad-petrovic/thesis

Docker images used for the application implementation are available inside Docker Hub repository that can be accessed through the following link:

https://hub.docker.com/u/vicviper/

6.5 Experiments

As the goal of the developed framework is to provide ability to perform task allocation automatically based on different policies (data privacy) and environment conditions (node resource availability, network delay) without affecting the structure of application developed by application developers. In practice, it means that in case of policy change task developers just need to deliver new allocation nodes (which are just put in the flow and not connected to the rest of the nodes in the Node-RED development environment) to application developers. Application developers do not have to change their flows each time the task developers change the policy, but just import the new allocation nodes and put them instead these which had been previously used.

In this case, we decided to experiment with different allocations of the computation tasks: ECG processing, troponin test and infraction test. The reason that we did not involve the movement of database server containers in the experiments is based on the fact that the framework does not provide ability to automatically move container data and features related to this issue are considered as a future work.

Therefore, we had a mixed architecture Docker Swarm cluster of three Raspberry Pi devices and three traditional x86/x64 laptops (already presented in Table 6.2). One of the laptops (Toshiba C55), running Ubuntu acted as a Docker Swarm master. The rest of the devices (including the master) were chosen to perform one the previously mentioned tasks that we decided to use for movement. The movement was done from the dashboard presented in Figure 6.5 – either by manually allocating tasks to desired nodes or using algorithms. All Node-RED components were located and run by master node – including the task execution proxies and allocation nodes.

The data-related tasks (database servers) were static during the experiments and always executed by same nodes (one MongoDB was located on laptop, while another one on Raspberry Pi). For result database Software-as-a-Service database provided by MySQL FreeHosting was used.

The result is that applications (ECG test, troponin test and infraction test) were giving the same results regardless of the node which is executing the related tasks. Of course, the condition is to provide new allocation nodes each time the new selection policy is defined.

7 Conclusion and Future Work

As a result of the research carried out during the thesis, a framework for task movement in mixed architecture fog environment has been developed, using Node-RED development environment, Docker containers as task abstraction and exploiting the capabilities of Docker Swarm container management system to allocate tasks.

Despite the fact that the framework provides capabilities to achieve its primary goal, there are still many aspects of which could be topic of further consideration for improvement and extension,

In the first place, aspects related to persistent data movement between containers would be further considered in order to find an appropriate and convenient solution for this issue. Despite their true stateless nature, containers in some cases produce data which is persistent and later needs to be moved to another container or location, as in case of database servers.

Docker Engine, starting from version 1.9 has introduced support for persistent data volumes, giving this way capability to move and share data between containers. When it comes to sharing and moving data among the containers located on the same host, Docker Engine provides quite sophisticated capabilities using the concept of data volumes, but in the case of persistent data which needs to be shared and moved among containers run on separate hosts – there are no automatic mechanisms to achieve this goal. It would be necessary to manually implement the operations based on offered primitives related to data volume backup and sending (from the source host perspective); receiving and mounting (from the perspective of the destination host) - which results in large overhead and network traffic, as data volumes could be very large.

For purpose of container data volume management inside clusters data volume, container data volume orchestrators are used. One of these technologies is called Flocker and is compatible with Docker Swarm and Kubernetes. First, it started as a plugin for Docker, but later it evolved into a standalone product. Flocker offers a convenient way for multi-host persistent data with mechanisms of data volume management similar to Kubernetes container management. During the research, it was considered to be used as a mean to achieve data volume movement inisde the framework, but many difficulties appeared. As the goal is to have mixed architecture execution enviroment, we were interested aslo in

ARM architecture version of Flocker, but it was not possible to find such product publicly available. Further – the developer team behind the Flocker started to continously drop the support for this product, which resulted in missing documentation web pages, download links and other online resources crucial for installation, configuration and understanding of this technology. So, when it comes to utilization of data volume management inside the framework – the solution would be to still consider Flocker based on available online resources left behind, or find a suitable alternative. Docker Swarm has recently introduced support for distributed file system technologies such as NFS and iSCSI. Taking this fact into account, it could be an interesting consideration in development of data movement capabilities.

Another aspect considered for extension is utilization of replication and scaling capabilities provided by Docker Swarm in order to achieve advanced scalability and fault tolerance.

When it comes to extension of already existing capabilities – selection criteria and algorithm construction mechanisms could be extended in order to provide more freedom and generality, by taking various node and task features in order to utilize more complex cost and performance analysis.

References

[1] Martin Kleppmann, "Designing Data-Intensive Applications", O'Reilly Media, ISBN: 9781491903063, March 2017, pp. 1-2.

[2] Michael Sheng, Yongrui Qin, Lina Yao, Boualem Benatallah, "Managing the Web of Things: Linking the Real World to the Web", Elsevier, ISBN: 9780128097656, February 2017, pp. 73-78.

[3] Pierluigi Plebani, David Garcia-Perez, David Bermbach, Frank Pallas, Stefan Tai, "Moving Data in the Fog: Information Logistics with the DITAS Cloud Platform", 2017, pp. 1-7.

[4] Pierluigi Plebani, David Garcia-Perez, Maya Anderson, David Bermbach, Cinzia Cappiello, Ronen I. Kat, Frank Pallas, Barbara Pernici, Stefan Tai, Monica Vitali, "Information Logistics and Fog Computing: The DITAS Approach", 2017, pp 130-136.

[5] Luigi Atzori, Antonio Iera, Giacomo Morabito, "The Internet of Things: A survey", Computer Networks, 54(15), 2010, pp. 2787–2805.

[6] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic, "Cloud Computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility", Future Generation Comp. Syst., 25(6), 2009, pp. 599–616.

[7] Matthew N.O. Sadiku, Sarhan M. Musa, Omonowo D. Momoh, "Cloud Computing: Opportunities and Challenges. Potentials", IEEE, 33(1), January 2014, pp. 34–36.

[8] Weisong Shi, Schahram Dustdar, "The Promise of Edge Computing", May 2016, pp. 78-81.

[9] Nitinder Mohan, Jussi Kangasharju, "Edge-Fog Cloud: A Distributed Cloud for Internet of Things Computations" in IEEE 2nd Conference on Cloudification of Internet of Things (CIoT), 2016, Paris, France, pp. 1-2.

[10] Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are [Online]. Available on:
http://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf

[11] What's the Difference Between Containers and Virtual Machines [Online]. Available on:
http://www.electronicdesign.com/dev-tools/what-s-difference-between-containers-and-virtual-machines

[12] Containers vs. Virtual Machines [Online]. Available on: https://docs.docker.com/get-started/#containers-vs-virtual-machines

[13] Ann Mary Joy, "Performance Comparison Between Linux Containers and Virtual Machines", International Conference on Advances in Computer Engineering and Applications (ICACEA), March 2015, pp. 342–346.

[14] David Bernstein. Containers and Cloud, "From LXC to Docker to Kubernetes", IEEE Cloud Computing, 1(3), 2014, pp. 81–84.

[15] Introduction to Control Groups (Cgroups) [Online]. Available on:
https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html

[16] Namespaces in Operation, Part 1: Namespaces Overview [Online]. Available on: https://lwn.net/Articles/531114/

[17] Container Orchestration Tools: Compare Kubernetes vs Docker Swarm [Online]. Available on:
https://platform9.com/blog/compare-kubernetes-vs-docker-swarm/

[18] Kubernetes and Docker Swarm Compared [Online]. Available on: https://platform9.com/blog/kubernetes-docker-swarm-compared/?utm_code=k8s_ds_redir

[19] Dockerfile Reference [Online[. Available on: https://docs.docker.com/engine/reference/builder/

[20] Docker Toolbox, Docker Machine, Docker Compose, Docker WHAT!? [Online]. Available on: https://nickjanetakis.com/blog/docker-toolbox-docker-machine-docker-compose-docker-wtf

[21] How to Build and Run ARM Docker Images on x86 Hosts [Online]. Available on:
https://matchboxdorry.gitbooks.io/matchboxblog/content/blogs/build_and_run_arm_images.html

[22] Manage Data in Containers [Online]. Available on:
https://docs.docker.com/engine/tutorials/dockervolumes/

[23] Docker Swarm vs Kubernetes: Comparison of the Two Giants in Container Orchestration [Online]. Available on:
https://www.upcloud.com/blog/docker-swarm-vs-kubernetes/

[24] Setup Kubernetes on a Raspberry Pi Cluster easily the official way! [Online]. Available on: https://blog.hypriot.com/post/setup-kubernetes-raspberry-pi-cluster/

[25] Docker Swarm Tutorial [Online]. Available on: https://rominirani.com/docker-swarm-tutorial-b67470cf8872