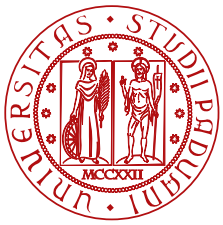


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER THESIS IN ICT FOR INTERNET AND MULTIMEDIA

Ensuring Service Level Agreement Compliance for Smart Grid Communications

Supervisor
PROF. STEFANO TOMASIN

Master Candidate
NICOLA GREGGIO

ACADEMIC YEAR: 2023-2024

DATE OF GRADUATION: 04/04/2024

Alla mia famiglia

Abstract

The rapidly evolving landscape of critical communications infrastructure is characterized by the integration of advanced technologies and the growing demand for efficient systems. Industrial Internet of Things (IoT) systems in smart grids are at the forefront of this transformation, promising improved energy distribution and management. This research focuses on ensuring Service Level Agreement (SLA) compliance of distributed software systems, which are critical to critical communication infrastructures. Research objectives include contextualization of critical communication infrastructure and industrial IoT in smart grids. It evaluates redundancy and software deployment patterns in industrial IoT software development, including continuous integration and continuous delivery (CI/CD) practices. It also explores observability data in running systems for predictive maintenance, applying deep learning and machine learning for automated fault management and autoscaling in computer clusters.

Sommario

Il panorama in rapida evoluzione delle infrastrutture critiche di comunicazione è caratterizzato dall'integrazione di tecnologie avanzate e dalla crescente domanda di sistemi efficienti.
I sistemi industriali dell'Internet of Things (IoT) nelle reti intelligenti sono in prima linea in questa trasformazione, promettendo una migliore distribuzione e gestione dell'energia. Questa ricerca si concentra sulla garanzia di conformità al Service Level Agreement (SLA) dei sistemi software distribuiti, che sono fondamentali per le infrastrutture di comunicazione critiche. Gli obiettivi della ricerca includono la contestualizzazione delle infrastrutture di comunicazione critiche e dell'IoT industriale nelle reti intelligenti. Valutare la ridondanza e i modelli di distribuzione del software nello sviluppo di software IoT industriale, comprese le pratiche di integrazione continua e consegna continua (CI/CD). Verranno esplorati inoltre metodi di osservabilità nei sistemi in funzione per la manutenzione predittiva, applicando il deep learning e il machine learning per la gestione automatizzata dei guasti e l'autoscaling nei cluster di computer.

Acronyms

IoT = Internet of Things
IIoT = Industrial Internet of Things SLA = Service Level Agreement
SaaS = Software as a Service
WSN = Wireless Sensor Networks
CI/CD = Continuous Integration/Continuous Delivery
QoS = Quality of Service
LoRaWAN = Long Range Wide Area Network
NB-IoT = Narrowband IoT
WM-Bus = Wireless M-Bus MQTT = Message Queuing Telemetry Transport
CoAP = Constrained Application Protocol
HTTP = Hypertext Transfer Protocol
AMQP = Advanced Message Queuing Protocol
IaaS = Infrastructure as a Service
VM = Virtual machine
sFTP = Secure File Transfer Protocol
SLSQP = Sequential Least Squares Programming
BFGS = Broyden-Fletcher-Goldfarb-Shanno
CG = Conjugate Gradient
OS = Operating System
VPS = Virtual private Server
TCP = Transmission Control Protocol
HTTP = Hypertext Transfer Protocol
DNS = Domain Name System
VIP = Virtual IP
VRRP = Virtual Router Redundancy Protocol
SVM = Support Vector Machine
DT = Decision Tree
RF = Random Forest

Contents

List of Figures

Chapter 1

Introduction

Today's world of critical communications infrastructures is constantly changing, driven primarily by the use of advanced technologies and the growing demand for efficient and intelligent systems. This evolution has occurred through the adoption of Industrial Internet of Things (IIoT) systems within several system including smart grids, which are needed to improve energy distribution and management.

The development of IoT has led to a digital revolution in the fields of electronics and computing, with low-cost data storage, mobile computing, artificial intelligence, Software as a Service (SaaS) and cloud computing. The number of IoT devices is currently at the peak of inflated expectations, expected to increase by 21% between 2016 and 2023 to around 18 billion. IoT devices with cellular connections are expected to reach 1.8 billion in 2023, or approximately 70% of a large-scale IoT category, [1]. Indeed, IoT is supported by an alliance of key resources, including the massive proliferation of intelligent devices, the confluence of low-cost technologies such as sensors, large amounts of data (Big Data), high-performance computing capabilities (HPC) and Wireless Sensor Networks (WSN).

This research aims to explore and contribute to the development of industrially viable methods that ensure Service Level Agreement (SLA) compliance in distributed software systems underlying critical communications infrastructures.

The importance of reliable and secure communications networks in critical infrastructure is fundamental and can't be underestimated. As societies become more interconnected, the robustness of these networks proves to be critical to the continued functioning of essential services. In this context, the energy sector plays a fundamental role, with smart grids being a revolutionary solution for efficient energy distribution. Adopting industrial IoT within these networks introduces myriad possibilities but also requires an in-depth understanding of the challenges and opportunities inherent in ensuring high availability and SLA compliance.

This research aims to achieve several interconnected objectives. Firstly, it seeks to contextualize the importance of critical communication infrastructures and the adoption of Industrial IoT systems in the specific domain of smart grids. The impact of network topology on service availability provides a starting point to understand the complexities of these systems. Secondly, the study aims to evaluate popular redundancy and software-distribution patterns within the entire chain of industrial IoT software development and deployment. This includes exploring continuous integration and continuous delivery (CI/CD) practices, as well as the deployment of software artifacts on both private and public cloud infrastructures.
Furthermore, the research delves into the gathering of observability data in live systems, considering its role in predictive maintenance of high availability (HA) software services. It also explores the application of deep learning and machine learning techniques in creating a decision support tool for automated fault management and auto-scaling in computer clusters.

The thesis is structured as follows:
In Chapter 2, the Service Level Agreement (SLA) is discussed to establish the framework for assessing the performance and reliability of the critical communication infrastructure. An analysis of critical communication infrastructure and industrial IoT in smart grids will also be highlighted, showing the current system implemented in Inkwell Data, the company where I did my Internship. Some practical problems that can be solved using the tools described in the next chapter will also be illustrated. Chapter 3 will describe some practical methods to observe the system and to ensure high service availability. In addition, a description of the software development and deployment chain in private and public cloud environments will be provided, with a specific focus on CI/CD and architectures based on virtual machines and software containers like Docker or Incus. In Chapter 4, the observability system presented earlier will be put into practice, focusing on its implementation in the local area network. It will then turn to the collection of data in production environments, emphasizing the importance of such data for predictive maintenance of high-availability services. Subsequently, some machine learning and deep learning techniques will be explored to develop a decision support tool in the context of critical communication infrastructure (Chapter 5). At last, Chapters 6 will discuss and present the results obtained in the previous Chapters.

Chapter 2

Critical Infrastructures and SLA Analysis

2.1 Service Level Agreement

Over time, there has been a significant focus on defining and targeting end-users, capturing the attention of service providers, clients, and network infrastructure providers. The past decade saw the initiation of liberalization and deregulation in the telecommunications sector. The surge in competition, driven by evolving client performance expectations, places substantial pressures on both service providers and networks. In addition, following significant cost reductions in recent years, suppliers are trying to improve their quality of service (QoS) to distinguish their products from the competition and meet customer demands. Hence, the duty of each entity that participate in service provision and their relations need to be explained. [2]
The Service Level Agreement (SLA) is a key part of business relationships in the Software as a Service (SaaS) industry. It is a contract between a service provider and its internal or external customers that documents what services the provider must provide. Let's delve deeper into the various types of SLAs and their significance. [3]
There are three main types of SLAs:

- **Customer-based:** contractual agreement between a company and a third-party service provider, which outlines expectations and guarantees regarding the provision of the service;
- **Internal:** agreement between different units or members within the same company. Defines responsibilities, performance metrics and expectations between various parties within the organization;
- **Multi-level:** single document that incorporates both general and specific SLAs. It covers a broad range of agreements, ensuring a comprehensive understanding of expectations at different levels or among different entities involved.

All SLAs are composed of two main elements: service agreements and service management. The former defines the features and functionality the customer receives from the provider, while the latter outlines how the service level is measured and maintained. Over the past few years, a great deal of research has been conducted to measure the performance of SLAs agreed between entities, focusing on the development of advanced tools and sophisticated monitoring protocols in this area. These are utilised for service level measurement such as network delay variation, latency, as well as frame and packet loss ratio. In addition, they are the key performance indicators used to ensure that the service provider meets the standards and provides the functionality guaranteed by the SLA.
For greater clarity, some metrics that SLAs can specify include:

- The availability and uptime percentage of services;
- The maximum number of concurrent users that can be served;
- Application response time;
- Usage statistics.

Based on how the network is structured, SLA contracts can be classified into two types: horizontal and vertical SLAs. A horizontal SLA involves an agreement between two service providers operating at the same architectural level, such as two Internet Protocol (IP) domains or two optical transport network domains. Conversely, a vertical SLA is a pact between two different vendors working at distinct architectural levels, like an agreement between an optical network and an MPLS core network. Figure 2.1 illustrates a network topology, showcasing potential SLA agreements across different architectural layers, providing a visual representation of the interconnected components within the system architecture. The diagram highlights the potential Service Level Agreement (SLA) agreements across different architectural layers. These agreements define the quality of service expected between various components and layers of the network infrastructure.
Let's consider a web application deployed on a cloud infrastructure consisting of multiple layers, including the presentation layer, business logic layer, and data storage layer. A vertical SLA defines the expected response time for requests originating from the presentation layer and traversing through the business logic layer to access data from the storage layer. On the other hand, let's consider now a platform with redundant data centers located in different regions to ensure high availability. A horizontal SLA governs the performance and reliability of each data center, ensuring consistent service levels across all locations.

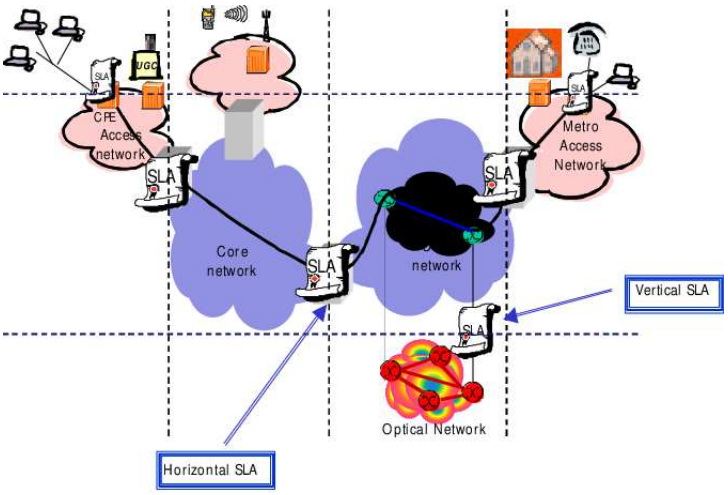


Figure 2.1: Horizontal and Vertical SLA, adapted from [4].

2.2 Critical Communication Infrastructures for smart IoT

As Industrial Internet of Things (IIoT) technologies grow and advance, the integration of smart devices and real-time applications within industrial environments becomes increasingly popular. However, the successful implementation of IIoT for real-time applications faces notable challenges, particularly in ensuring high availability and reliability. This section explores the critical communication infrastructures essential for supporting Smart IoT in industrial environments.

Industrial real-time systems impose stringent availability requirements, typically ranging from 99.9% to 99.999%. These demands are considerably higher compared to architectures based on local cloud services, which often do not offer comparable guarantees or service level agreements. The stark differences in Quality of Service and SLA availability between operational technologies and computing technologies present a significant obstacle in the widespread adoption of IIoT for real-time applications.

The adoption of IIoT in real-time applications is hindered by the inherent challenges posed by the differences in QoS and SLA availability. The reliance on architectures not specifically designed for high availability in industrial environments raises concerns about the stability and performance of critical systems. Ensuring that the communication infrastructure can meet the stringent demands of real-time industrial processes becomes a pivotal aspect of successful IIoT integration.

Critical communication infrastructures play a central role in addressing the challenges associated with IIoT adoption. These infrastructures are tailored to meet the specific requirements of industrial applications, providing robust, reliable, and low-latency communication pathways. Such dedicated communication networks become indispensable for ensuring that real-time data

transmission, control signals, and mission-critical information flow seamlessly within the industrial ecosystem.
Key characteristics of critical communication infrastructures include:

- **Reliability:** These infrastructures prioritize reliability to ensure consistent and uninterrupted communication.
- **Low Latency:** Minimal communication delays are crucial for real-time applications, making low-latency communication a fundamental attribute.
- **Security:** Industrial environments demand secure communication channels to safeguard sensitive data and prevent unauthorized access.
- **Scalability:** The ability to scale the infrastructure to accommodate the growing number of connected devices and evolving industrial needs.

In the next chapters it will be illustrated the system implemented at Inkwell Data, the company where I interned, highlighting a real Critical Communication for IIoT.

2.3 IoT General Architecture

Before analysing in detail the infrastructure used in the company, it is important to highlight some key concepts of the IoT world. These include the notions of digital twin, gateway and smart meters. Knowledge of these concepts is essential for understanding the operation of more sophisticated concepts that will be explained in the following chapters.
The digital twin is a structure that represents physical objects in the cloud, e.g. sensors or smart meters. Real devices send data to their counterpart in the cloud and at any time update their state to be as close to real time as possible. Cloud-based applications and services normally use the digital twin to access device data, instead of accessing the endpoints directly. The most important advantage of using digital twins is certainly that they are always available, even when the physical object is switched off or unavailable. In this way, delay-tolerant applications can proceed without having to waiting for temporarily inaccessible devices.
In other words the digital twin is a mould that describes the attributes a real object instance must have. Attributes can be divided into two types:

- **General attributes:** the general attributes are shared across all instances created with this digital twin and remain consistent across all instances. One example of this attribute can be the ”manufacturer”.

- **Instance attributes:** this kind of attributes are specific for each instance and they capture a unique characteristic or property. An example could be the ”Temperature Setpoint”, which uniquely reflects the desired temperature set by the user for that specific device.

An IoT gateway is a hardware or software device that connects physical devices, such as sensors or smart meters, and cloud platforms or management systems. This component is responsible for collecting and sending data generated by the devices to the cloud, as well as sending commands or instructions to the devices themselves. Gateways can be implemented on dedicated devices or integrated directly into the devices themselves. The digital twins of physical devices typically reside in cloud platforms or centralised management systems. When peripheral devices generate data, it is sent to the nearest IoT gateway. The most commonly used protocols are LoraWan, NB-IoT and WM-Bus. The gateway, in turn, communicates with its corresponding digital twin in the cloud platform and transmits the collected data. This communication can take place via standard communication protocols such as MQTT, CoAP or HTTP. A general representation of the IoT world is shown in Figure 2.2.

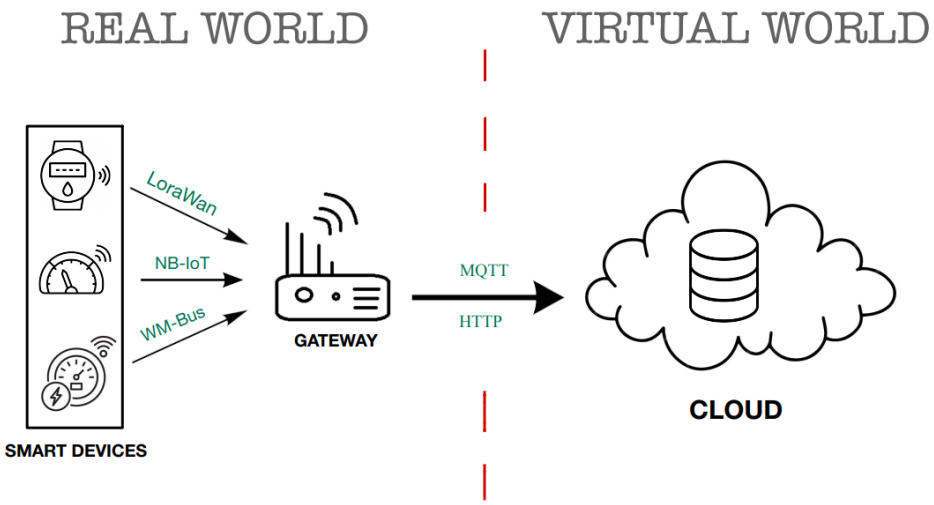


Figure 2.2: General representation of IoT

2.4 Real-world case study

It will be now described the system currently implemented in Inkwell Data.
The company is involved in the development and management of IoT systems. The main infrastructure behind the company is Altior, designed to help the user deploy industrial IoT solutions without the technology getting in the way. More specifically it is a network technology-agnostic service design and delivery platform for industrial IoT network operators, a data-oriented approach to the development of industrial IoT, supporting LPWAN network technologies for operations on license-free spectrum. It is also designed for carrier grade operations and five nines reliability with a distributed approach. Altior can be rolled out on public or private clouds or on local premises. [5]
The system is described in Figure 2.3.
We can consider the system divided into two major areas:

- **Altior**: represents the main core of the IoT infrastructure, where all messages from the meters and gateways converge. Here, raw data is received, processed, decoded and decrypted, while the status of the digital twins, which reside right inside Altior, is constantly updated.

- **Swarm**: it represents the application part of the IoT infrastructure. Swarm is responsible for managing the back-end and front-end of IoT applications. In addition, communication with the database takes place here, which plays a key role as a central repository for all meter data and readings from the IoT infrastructure.

Communication between Altior and Swarm is established by **RabbitMQ**, an open source messaging system based on Advanced Message Queuing Protocol (AMQP). RabbitMQ functions as a reliable intermediary for the exchange of messages. [6]
The Altior system is itself composed of five main applications:

1. **Network Adapter Manager**
   The aim of this component is to collect information about the running network adapters. A network adapter is a component that enables IoT devices to communicate over a specific network, serving as an interface between the specific device and the network it operates on. It has a registry where the digital twin instance can register to. That registry is necessary for routing the incoming messages to the right digital twin instance.

2. **Digital Twin Runtime**
   The digital twin runtime manages the digital twin instances and it is the process that interacts with external processes (e.g., the network adapters) and forwards the incoming

messages. Each digital twin instance is built by populating the attributes described in Chapter 2.3 into a specific digital twin.

3. **Codec Manager**
   This component manages the codecs. A codec is a process that receives a Digital Twin Instance state and a binary message, takes the binary message and update/upgrade the Digital Twin Instance state. The component give the following functions:

   - Uploading a new codec: the function receives the beam files and the information to add to the codec.
   - Refreshing the codecs folder.
   - Stopping a codec.
   - Running a function of specific codec: this can be done sending a message to a specific process.

4. **Aegis**
   This component contains all security information and it interacts with the external process and forwards the requests to the correct sub-component. It manages the Access Control and the related resources access, the users, groups of users and user themselves.
   It is also responsible for the encryption and decryption of messages from the meters. The codec manager relies on this application to decrypt the message. Once this step is successful, the codec is allowed to proceed with decryption.

5. **Altior Application Manager**
   This component manage the Altior applications. The Altior application is an entity with the following fields:

   - Name
   - Code
   - Version
   - Creator ID
   - Security

   They all take as input the current state of the digital twin instance with some additional parameters and return the updated state of the instance. It is responsible to send the messages to RabbitMQ.

The general overview of the infrastructure is shown in Figure 2.3:

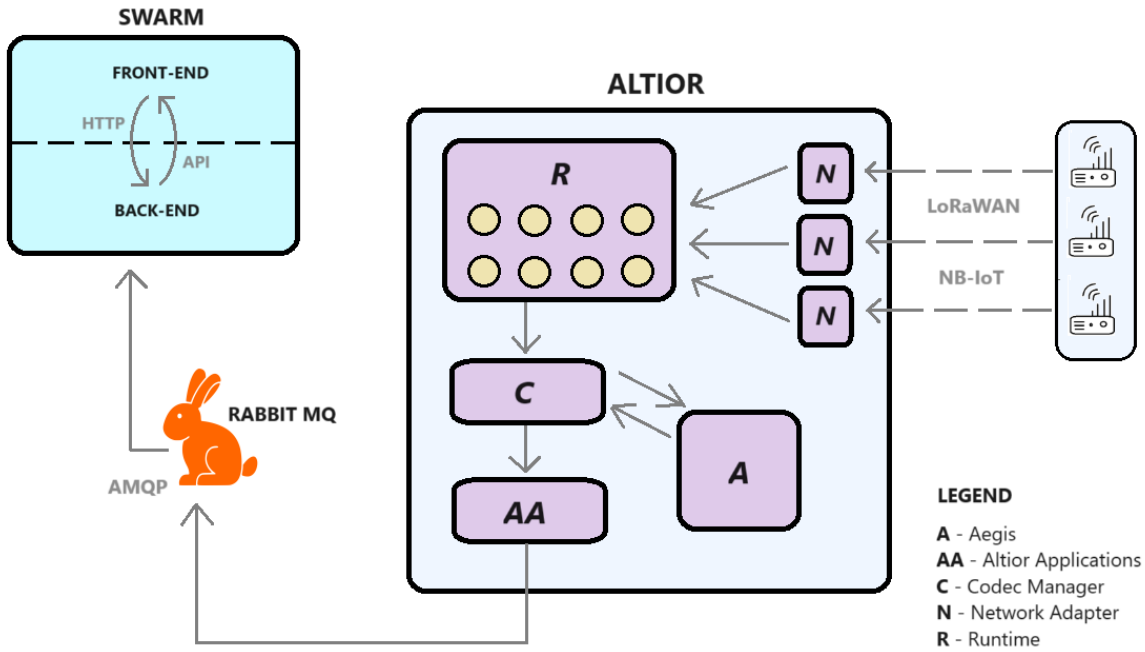


Figure 2.3: Infrastructure scheme

The physical architecture can change during the Altior system's working life.
In order to remain compliant with the programming language used by the company for application development, all code will be written in Erlang/Elixir and Python. This is not a limiting factor for the general concepts that will be discussed and analysed in this thesis. Each application that runs in the system is a node and the number of nodes can increase if the system load increase. All of these nodes may be on the same host or on different hosts. The key aspect is that they maintain shared parameters to ensure system consistency. In Elixir (and the Erlang/OTP environment), the basic configuration files used are:

- sys.config: where there are application-specific configurations, such as database connection parameters, authentication settings and network configurations
- vm.args: used to configure the behaviour of the Erlang VM itself, such as the maximum memory size or log file paths. In addition, the node identifier and the Erlang distribution cookie are configured here:

```
-name node_name@server_hostname     # Node identifier
-setcookie cookie_name      # Erlang distribution cookie
```

  The node identifier is unique to each, while the cookie is shared and the same with all the other nodes that make up the system's infrastructure.

Keeping the same cookie, many replicas of the same node can be hosted in different servers. This will be discussed in Chapter 3 and it is the first step to increase the reliability of the sys-

tem. In fact, in the event that a server is unavailable, and thus all nodes associated with it are unreachable, the service will be interrupted. If, on the other hand, the technique of replicating nodes on several different hosts is used, the service will remain active, thus guaranteeing greater reliability of the system.
In Chapter 3 a theoretical analysis will be made to guarantee 99.999% availability of the service, i.e. with a maximum of approximately 5 minutes and 15 seconds of interruption per year. However, it's essential to note that achieving this level of availability also requires considering real-world factors such as network latency between servers. Latency between servers can impact the overall performance and availability of the system, and it's crucial to account for it in our analysis and implementation strategies.

Chapter 3

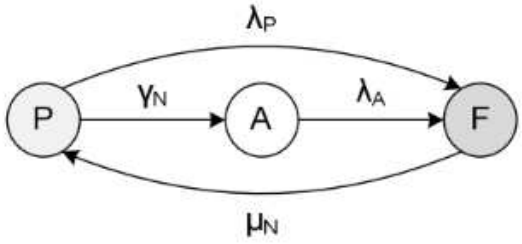
Methods and Tools

3.1 Reliability theoretical model of a computing node

In this Chapter, a mathematical model is highlighted to determine the optimal number of replicas required to achieve a specific level of reliability. The model is based on Markov chain with the next state-space and conditions of transitions between states:

- State P (passive): the node is up, but passive, and does not provide computing operations because of software initialization. From this state, the node can either pass to state A with rate γ_N (activation rate), or pass to state F with rate λ_P (failure rate in the passive state).
- State A (active): the node is up and active, and provides the computing operations. From this state, the node can pass to state F with rate λ_A (failure rate in the active state).
- State F (fault): the node is down. From this state, the node can pass to state P with rate μ_N (rate of the repair operations).

The state-space and transition conditions, according to the discussed reliability model, is shown in Figure 3.1.
According to [7], the differential equations of Kolmogorov-Chapman for this Markov chain are



h

Figure 3.1: The base three-state reliability model of the computing node

described as follows:

$$\begin{cases} P_P(0) = 1; \quad P_A(0) = 0; \quad P_F(0) = 0; \\ P_0(t) + P_1(t) + P_2(t) = 1 \\ \dfrac{dP_P(t)}{dt} = -(\lambda_P + \gamma_N)P_P(t) + \mu_N P_F(t) \\ \dfrac{dP_A(t)}{dt} = \gamma_N P_P(t) - \lambda_A P_A(t) \\ \dfrac{dP_F(t)}{dt} = \lambda_P P_P(t) + \lambda_A P_A(t) - \mu_N P_F(t) \end{cases} \tag{3.1}$$

Where λ_A is the failure rate of the computing node in the active state, λ_P is the failure rate of the computing node in the passive state, μ_N is the repair rate of the computing node and γ_N is the activation rate of the computing node (the rate of passing from the passive state to the activate state).

Considering $t \to \infty$, when Markov process reaches the steady state and derivatives of probability functions tending to zero, we obtain the following formulas for stationary probabilities of the states:

$$\begin{cases} P_P(\infty) = \dfrac{\mu_N \lambda_A}{\mu_N \gamma_N + \lambda_A(\mu_N + \gamma_N + \lambda_P)} \\ P_A(\infty) = \dfrac{\mu_N \gamma_N}{\mu_N \gamma_N + \lambda_A(\mu_N + \gamma_N + \lambda_P)} \\ P_F(\infty) = \dfrac{\lambda_A(\gamma_N + \lambda_P)}{\mu_N \gamma_N + \lambda_A(\mu_N + \gamma_N + \lambda_P)} \end{cases} \tag{3.2}$$

The state A is the only operable state for the computing node, so we can also obtain the formula for the stationary availability factor K of the computing node:

$$K = P_A(\infty) = \frac{\mu_N \gamma_N}{\mu_N \gamma_N + \lambda_A(\mu_N + \gamma_N + \lambda_P)} \tag{3.3}$$

Moreover, the formula for the mean time to failure of the computing node can be easily obtained by using the topological method for the reliability models based on the Markov chains. It can be calculated as a ratio of the availability factor to the weighted sum of probabilities of all operable states, each of which is multiplied by the sum of all rates of transitions to all inoperable states. In our case, we have only one operable state, A, with only one transition to state F with rate λ_A, thus we obtain the following simple formula for the mean time to failure:

$$T_F = \frac{P_A(\infty)}{\lambda_A P_A(\infty)} = \frac{1}{\lambda_A} \tag{3.4}$$

At last, we can easily obtain the formula for the mean time to the recovery of the computing node, which takes into consideration both repair and activation rates, by using the fundamental

relation between the availability factor, mean time to failure and mean time to recovery:

$$T_R = \frac{1-K}{K} T_F = \frac{\mu_N + \gamma_N + \lambda_P}{\mu_N \gamma_N} \tag{3.5}$$

This model provides a sound theoretical basis for designing reliable systems, taking into account the reliability of a single node.
Thanks to this, we can now apply these concepts to an interconnected system of interacting nodes, e.g. all the applications described in Chapter 2.4.

3.2 Interconnection of nodes

Having considered the availability of each individual node, we must now consider the interconnection of all these nodes from a broader point of view.
Before analysing the availability of the system as a whole, it is essential to understand the true meaning of availability and the difference between the various levels of it. Availability indicates the percentage of time the infrastructure, system, or solution remains operational under normal circumstances to fulfil its intended purpose. Despite a seemingly negligible change in availability, such as going from 99% to 99.999%, there is a significant change in service downtime, as highlighted in the Table 3.1, where downtime is highlighted depending on system availability.

	Downtime		
Availability	**Per year**	**Per month**	**Per week**
90%	36.5 days	72 hours	16.8 hours
99%	3.65 days	7.2 hours	1.68 hours
99.9%	8.76 hours	43.8 minutes	10.01 minutes
99.99%	52.56 minutes	4.32 minutes	1.01 minutes
99.999%	5.26 minutes	25.9 seconds	6.05 seconds

Table 3.1: Downtime for different levels of availability

We can see that increasing the reliability of the system from 99% to 99.999% allows us to significantly decrease the downtime, from 3.65 days to 5.26 minutes per year.
Let us now analyze a series of nodes that interact with each other. We can calculate their availability as shown in Chapter 3.1, but we must understand how to calculate the availability of the whole system. We can consider the whole system as a series of interacting applications, each of which has a specific availability. If even one node in the system is inactive, the entire system suffers from the unavailability and does not function properly. For example, in our case study, if the codec manager application is inactive, messages from the network adapter - which in turn has received messages from the meters or gateways - are not decoded and transmitted to the

runtime for processing. As a result, as long as the codec manager is not operational, the entire system experiences a drop in availability. Therefore, it is critical to ensure that every node in the system is up and running at all times to maintain high overall system availability.
Let us consider the individual availability of each node K_1, K_2, ... , K_n of a system with *n* nodes. The relationship linking overall system availability K_T to individual node availability is described as:

$$K_T = \prod_{i=1}^{n} K_i \qquad \forall i \in \mathbb{N}^+ \tag{3.6}$$

However, this is not an optimal solution because upon the unavailability of a single node, the whole system encounter the problem. In an attempt to enhance overall system reliability, we could have introduced a concept labelled as replication whereby we create multiple systems that are copies of the original and operate in parallel. This technique would ensure that the system would continue operating even with the failure of some or other single nodes such that the service would remain available. The approach would entail making multiple copies of the system in two regions, which could be servers, clusters, or virtual machines, two copies of the similar system, respectively. Considering that total availability is equal to the complementary availability, it implies that there is a higher probability that at least one of the replicas is operational. Considering K_1, K_2, ... , K_n the availability of individual nodes in each region, the total availability of the parallel system will be:

$$K_T = 1 - ((1 - \text{Region A availability}) \cdot (1 - \text{Region B availability})) \tag{3.7}$$

The availability of each region is given by the availability of the set of nodes in that region. Therefore we can rewrite (3.7) as follow:

$$K_T = 1 - ((1 - K_{T1}) \cdot (1 - K_{T2})) \tag{3.8}$$

Where K_{T1} is the total availability of the first region and K_{T2} is the total availability of the second region.
We can now generalize this concept by considering *n* replicas of the same system:

$$K_T = 1 - \prod_{i=1}^{n} (1 - K_{Ti}) \tag{3.9}$$

In Chapter 4.2, a practical analysis will be conducted to explore these concepts, focusing on the number of nodes and the number of replicas in the system. This approach will allow us to empirically evaluate the impact of different scenarios on the overall system availability.

3.3 Optimization of node replicas

Deploying numerous copies of each node on different servers can result in significant cost increases, underscoring several financial considerations. First, the hardware investment is substantial and involves costs in terms of CPU, memory, and storage space for each additional server. This amount can become substantial when multiplied by the number of replicas required to ensure satisfactory system reliability. In addition, software licensing expenses must be considered, as some solutions require a license for each active instance. In addition to initial costs, operational expenses also emerge: periodic maintenance, software upgrades, ongoing monitoring, and life-cycle management, which require human resources and contribute to overall operational costs. Therefore, while recognizing that increased replication can potentially increase system reliability, it is imperative to carefully evaluate the costs associated with this strategy and balance them carefully against the expected benefits. Understanding whether using multiple replicas distributed over different regions is the optimal choice is not always obvious. The trade-off between the number of replicas and the overall reliability of the system needs to be carefully evaluated. In practice, this involves identifying the minimum number of replicas needed to achieve the desired reliability. This assessment may involve analysis of data on the reliability of individual nodes and the system as a whole, as described in Chapters 3.1 and 3.2. The results of these evaluations can help determine the appropriate balance between the number of replicas and the availability of the system, while also considering the costs associated with implementing and managing the additional replicas. It is critical to find an optimal point that ensures a robust and sustainable infrastructure over time.

We can formulate the problem as one of cost optimization, where the goal is to minimize the number of replicas needed subject to the constraint of achieving a specific availability. Suppose that the availability of each node is represented by *K*, we need to determine the optimal number of replicas r^* so that the total system availability K_T is greater or equal to a desired threshold *T*.
So, formally, we look for:

$$r^* = \min\{r \in \mathbb{N}^+ \mid K_T(r) \geq T\} \tag{3.10}$$

The problem formulation can be expressed as an optimization problem:

$$\begin{cases} min \quad r \\ 1 - ((1-K)^r) \geq T \\ T \in [0,1] \\ r \in \mathbb{N}^+ \end{cases} \tag{3.11}$$

To solve this problem, a Python script has been implemented that can determine the optimal number of replicas required to achieve a specific system availability. This script considers the reliability of individual nodes and finds the minimum number of replicas required to meet the desired overall availability requirements.

```
import time
from scipy.optimize import minimize

def objective_function(r, K, T):
    return abs(1 - (1 - K) ** r - T)

K = 0.9
T = 0.99999
initial_guess = 1
start_time = time.time()
result = minimize(objective_function, initial_guess, args=(K, T), bounds
    =[(0, None)], method='SLSQP')
end_time = time.time()
execution_time = (end_time - start_time)*1000
optimal_r = round(result.x[0])

print("The optimal value of r is:", optimal_r)
print("Execution time:", execution_time, "ms")
```

The Python code uses the `minimize` function of the `SciPy` library to optimise the number of replicas required to achieve a given system availability. In particular, the method of sequential least squares programming (SLSQP) is used to minimize the objective function. SLSQP is a non-linear optimisation algorithm that can handle equality and inequality constraints.
The choice of using SLSQP depends on the nature of the problem and the constraints imposed. In this case, SLSQP is suitable because it can handle both variable domain constraints (e.g., $r \geq 0$) and inequality constraints. In addition, it is effective for nonlinear optimization problems such as the one presented, in which the objective function is not necessarily convex. In addition, it has been shown that the computational time using this algorithm is lower than other methods available in the *SciPy* library.

In the conducted analysis, we considered a configuration in which all nodes have an availability of 0.9, and the desired availability for the system was set at 0.99999. It is important to note that, in practice, each node could have a specific level of reliability, different from the others. However, this doesn't alter the concepts highlighted in this analysis.

Results are shown below:

```
The optimal value of r is: 5
Execution time: 15.432 ms
```

To evaluate the effectiveness of the minimization method used in our study, we compared the results obtained with different approaches offered by the *SciPy* library. In particular, we examined Powell, Broyden-Fletcher-Goldfarb-Shanno (BFGS) and Conjugate Gradient (CG) minimization methods.
The results obtained are as follows:

Powell Algorithm:

```
The optimal value of r is: 5
Execution time: 45.843 ms
```

BFGS Algorithm:

```
The optimal value of r is: 5
Execution time: 34.279 ms
```

CG Algorithm:

```
The optimal value of r is: 5
Execution time: 28.528 ms
```

We can see that the SLSQP algorithm emerges as the fastest among the considered options, leading to its recognition as the most efficient choice based on execution time.

3.4 Virtualization and Containerization

Another key solutions to ensure an even higher level of reliability and availability of services are the concepts of virtualization and the usage of containers. Virtualization is the main technology powering cloud computing and it allows multiple operating systems and applications to run simultaneously on the same server. In cloud computing services, IaaS providers provide customers with a set of virtual machines, each with its own operating system instance, where customers can install their own applications and dependencies. However, this approach is onerous in terms of complexity and manual configuration of the virtual machines, as well as significant overhead due to the presence of an entire operating system on each individual virtual machine. A newer and more efficient alternative is to use containers to deploy applications and their dependencies, thus solving the main problems of virtual machines. [8]

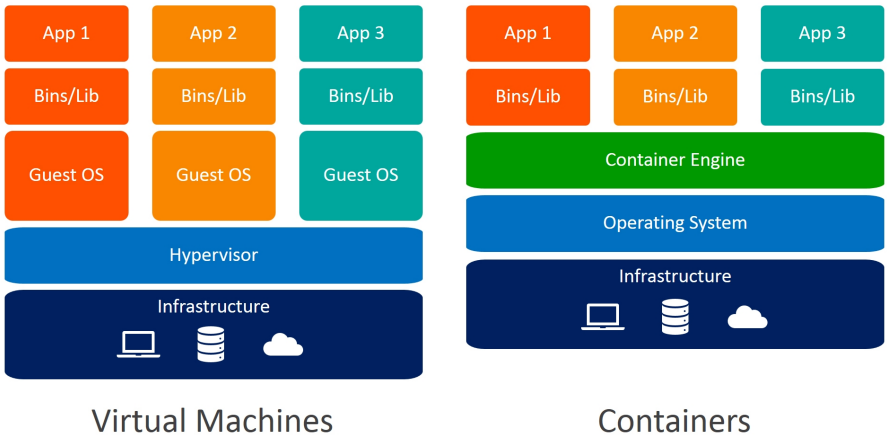


Figure 3.2: Comparison between virtual machines and containers, adapted from [9]

As shown in Figure 3.2, the main difference from traditional virtualization techniques is that there is no need for a hypervisor layer needed to provide the virtualization functionality. Instead, the kernel and the OS provide the virtualization features. Containers in the context of a services architecture are useful to package, deploy and run each single service. A container is easily mapped to a service. Traditional virtualization allows a processing node to run multiple isolated operating system images simultaneously. For reliability reasons, it is preferable to install each service on a different machine. By placing each service on a different machine, if a service crashes, at least the other ones are not affected. This aspect is good also for what concerns security, if a service has security flaws, these flaws have no effect on the other services. Instead of installing each service into a different computing node, virtualization can be leveraged so that each service is installed inside a different VM. Virtualization helps substantially by reducing the amount of computing machines needed and consequently it lowers the costs of supporting a complex computing environment.

	Virtual Machines	Containers
Resources used	Heavyweight	Lightweight
Kernel	Emulated Kernel	Native Kernel
Operating system	Dedicated (each VM has its own)	Shared with other containers
Virtualization	Hardware	Software
Start-up time	Minutes	Seconds
Memory management	Pre-allocated (more RAM needed)	Dynamic (less RAM needed)
Isolation	Full	At the thread/process level

Table 3.2: Main differences between VMs and Containers

As highlighted, containers offer superior performance and reduced cost, making them ideal for many applications. However, virtual machines offer a higher level of isolation, making them more suitable in scenarios that require total separation of environments. Therefore, at present, the optimal solution is a setup that integrates both technologies, leveraging the flexibility and security of virtual machines along with the economic and performance advantages of containers.

One possible approach might be to organize the infrastructure in the following way:

- **Private Cloud Service**, appropriately replicated/redundant, providing one or more hosts managed by a hypervisor with pre-allocated hardware resources. On these hosts it would be possible to create a series of virtual machines based on the number and size of services required by various clients, considering factors such as data volume and computing power required.
- **One or more VMs** for each customer, thus ensuring an adequate level of isolation for processed and stored data.
- **A containerization system** to be deployed on virtual machines dedicated to deployment, thus enabling the efficient delivery of services and applications required by each customer.

3.5 Container orchestration

The primary benefit of this approach to containers is that it makes a substantial contribution to efficiency and convenience in the software development cycle. It is even more important for teams that use the above-mentioned agile methodologies and Continuous Integration and Continuous Delivery (CI/CD) . Thus, in the context of containers, efficiency and consistency of development and release processes cannot be provided without these practices. There are two options for introducing them:

Continuous Integration (CI):

- Test automation: containers allow developers to build isolated and predictable test environments. CI systems execute an automated build and test process for every commit in the code repository. The test environment is identical and replicable with the use of containers.
- Automated Builds: containers have all prerequisite software necessary to build and test an application. It makes the setup for the CI builders easier and eliminates the possibility to generate different build results from individual team members due to environmental differences.

Continuous Delivery (CD):

- Automated releases: automate the release pipeline as much as possible, which allows you to automate the deployment of the application's test, staging, and production versions. Containers allow each application and its dependencies to be packed into a separate portable unit, making it easier to deploy the application to different environments.
- Simplified rollback: containers allow you to store several versions of an application at once and switch between them in a few simple steps, which allows, in case of problems with the release, to return immediately to the working state of the previous version of the application.

Figure 3.3 demonstrates the process of developing and implementing an application in the Altior system from Chapter 2.4.

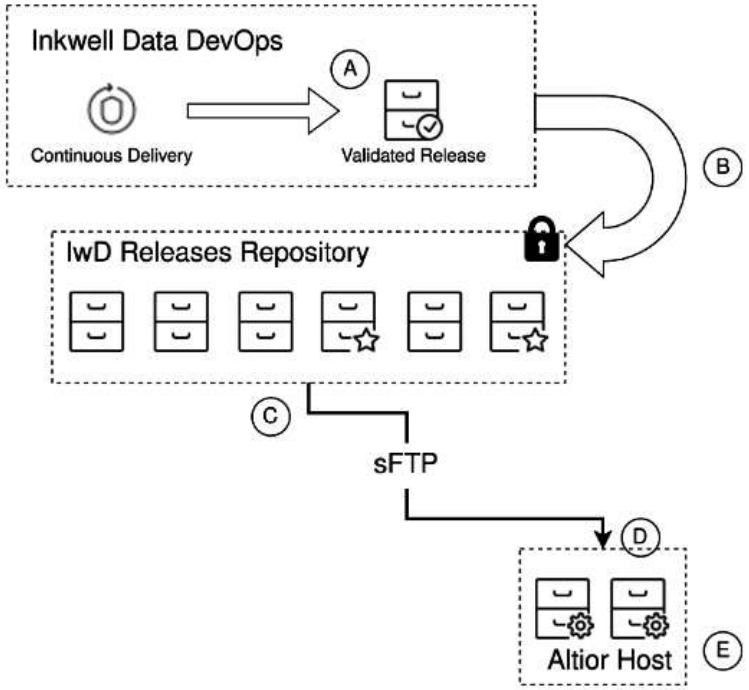


Figure 3.3: Deployment Process

This process goes through several stages, each playing an important role in ensuring that a new feature or a fix for a bug is reliably and safely delivered to the customers, and end-users. The process starts at the developing stage, where the new code is written and tested. Afterward, the code implemented into the embedded is validated. Releases that pass the validation are stored in a separate repository, where they would be easy to access and track. Finally, the validated releases are transmitted to the Altior hosts via Secure File Transfer Protocol (sFTP). Altior hosts are the ones representing the production servers, where the releases are available to the end-users.

Automatic deployment of new application versions to a cluster of hosts is managed through container orchestration. The orchestrator manages application scalability, starting new containers or terminating existing ones based on workload and resource needs. In addition, in case of problems during deployment, the orchestrator facilitates rollback to the previous version of the application, ensuring safe and simplified release management. This facilitates the virtualization of hardware resources, enabling the installation of distinct applications that remain isolated from each other. Consequently, customers are furnished with a network-connected array of machines, each capable of running one or more servers or applications.
In the solution we implemented, a server cluster is logically organized into three tiers, as shown in Figure 3.4:

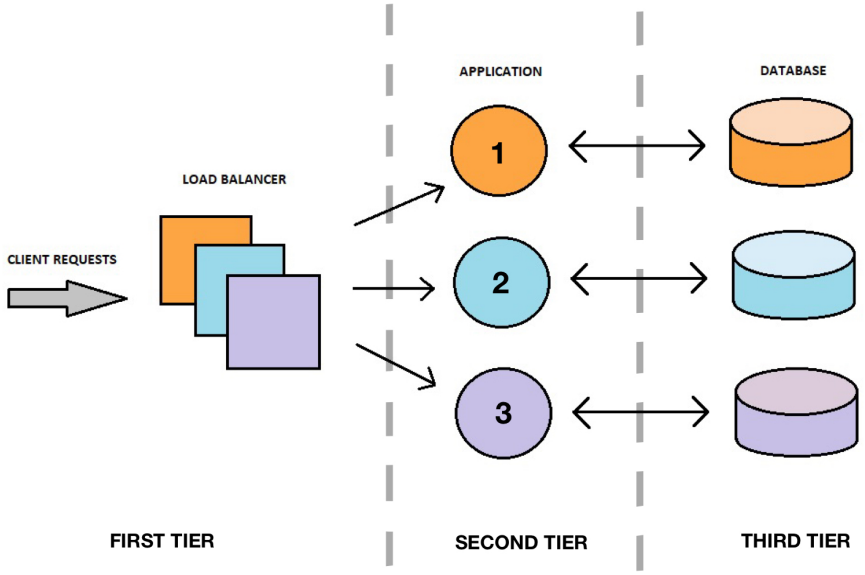


Figure 3.4: The general organization of a three-tiered server cluster

The **first tier** is the first stage in the process. As the system entry point, it mainly involves various customer requests. More specifically, when the client requests some data or certain services he or she is referred to the system. These requests are mainly to the application or compute servers which are reached through the load balancer. The load balancer generally helps in the effective functionality of the system. It mostly serves as a focal point for routing the client's requests concerning the client's location. The load balancer uses efficient algorithms to apportion the workload among client's specified and available application or computing servers which can be found in the next tier. These external servers help ensure that no individual server is overburden thus maximizing on the system's output which helps in a quick and efficient response to clients.

In the **second tier** of the architecture, requests are further processed by application or compute servers. The data processing occurs, and the model presented in Chapter 3.3 determines the optimal number of replications.

In the **third tier** of the architecture of the architecture, the distributed File/Database system plays a crucial role in coordinating and managing the data throughout the system. In this environment, the data is stored and retrieved, ensuring persistence and availability. More specifically, the system manages data storage in a distributed nature, which guarantees quick and efficient access to the stored information. Additionally, the system ensures the data replication and distribution in a way that guarantees the system resilience and business continuity even in individual hardware failures or software errors.

3.6 Load balancing

Load balancing is a process that involves the redistribution of workload among nodes in a system to optimize resource utilization and improve job response times. The purpose is to alleviate situations where some nodes are overloaded while others remain underutilized. Dynamic load balancing algorithms operate based on the current state of the system, without considering its previous behavior. When developing such algorithms, several factors must be considered, including load estimation, load comparison, system stability, performance, node interaction, nature of the work to be transferred, node selection, and others. The load to be considered may include CPU load, memory utilization, network delay, or overall network load. [10]
The main goal of load balancing is to redistribute workloads across computing resources to significantly improve performance. The currently most widely used technology for sharing or load balancing is HA Proxy, which is an open source software licensed under GPLv2. HA Proxy is used to share loads of TCP / HTTP requests or load balancers and proxy solutions that can run on Linux, Solaris and FreeBSD operating systems. [11]
Figure 3.5 illustrates the architecture adopted in the company to implement an HTTP request management solution aimed at directing traffic to the most available replication at the time.

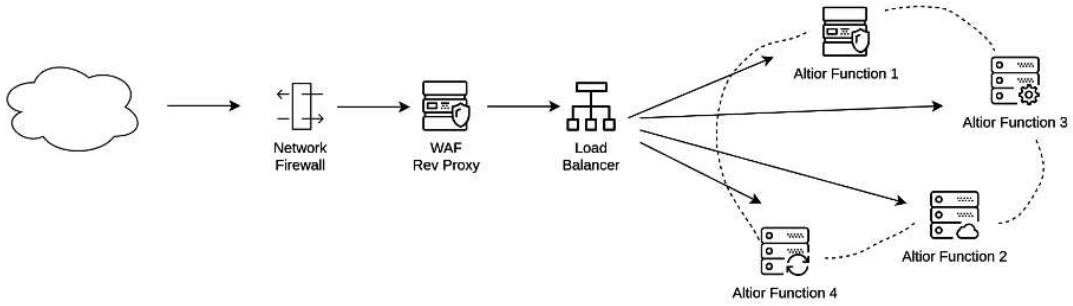


Figure 3.5: Architecture for HTTP request management

The reverse proxy is the intermediary between the clients making requests and the back-end servers providing the content. Unlike a traditional proxy server, which forwards client requests to the target servers, a reverse proxy forwards client requests to the appropriate servers and returns responses to the clients. More specifically, in Chapter 4.3 we will implement HA Proxy both as a reverse proxy and as a load balancer.

Chapter 4

Observability Data Collection and Analysis

4.1 Implementation of the base three-state reliability model

Before analyzing the whole system, this Chapter will show how to apply in a practical way the theoretical concepts discussed in Section 3.1. The only parameters needed to be derived are the rates of three-state reliability model λ_A, λ_P, γ_N, μ_N.
This study aims to develop a method for calculating the above parameters based on a server with Linux architecture, but the idea and general approach proposed to derive these parameters can also be applied to all other operating systems.
The first parameter to consider is λ_A, the failure rate in the active state. To obtain this empirical information, we can use the event log (e.g., the registry on Linux) to monitor all the times the service related to a node is stopped, either voluntarily or due to a failure. We can use the registry on Linux with `journalctl`, used to print the log entries stored in the journal and where all information about events occurring in the system is saved. In this way we can filter out and count all the times that the application we want to analyze is voluntarily stopped due to an update of the application, or due to a sudden stop:

```
start="start_date"
end="end_date"
sudo journalctl --since "$start" --until "$end" | grep "Stopped
    name_application | wc -l"
```

This command shows how many times the string "`Stopped name_application`" is present in the system registry in the last month. To increase the accuracy of the data, this method can be applied over several months within the year. By periodically recording and analyzing the output for each month, a set of values representative of service activity over the year can be obtained.

Subsequently, these values can be averaged to obtain an annual average figure. This approach makes it possible to mitigate any seasonal variations in service activity and to obtain a more accurate estimate over the entire period.
Based on the total number of failures and the time period during which the observation was conducted, we can calculate λ_A using Equation 3.6:

$$\lambda_A = \frac{\text{Total number of failures}}{\text{Total observation time}} \tag{4.1}$$

The repair rate of a system μ_N can be considered as the number of times the system is able to independently resolve a problem and restore proper operation after an outage. This concept is related to the system's transition from failure to passive status, and then to active. For example, when a system detects an error and initiates automatic recovery procedures that lead to the resolution of the problem and return of the system to operation, this process can be considered as repair. We can find the number of times the system automatically reboots as follows:

```
start="start_date"
end="end_date"
sudo journalctl --since "$start" --until "$end" | grep "Restarting
    name_application | wc -l"
```

In practice, repair rate can be calculated by counting the number of times the system goes from the failure state to the passive state and then to the operational state during a specified period of time as shown in Equation 3.7. This can include situations in which the system independently detects and resolves a problem without requiring human intervention.

$$\mu_N = \frac{\text{Total number of restarts}}{\text{Total observation time}} \tag{4.2}$$

When a node is in the passive state, two distinct scenarios can occur: the first is that the node becomes active with the rate γ_N (activation rate), while the second is that it goes directly to the failure state with rate λ_P due to a configuration problem, as a result of maintenance of the node itself. To analyze the first scenario, we can use an approach similar to that used to calculate the failure rate. In this case, we need to consider how many times the application under consideration was successfully started. Basically, we can monitor the system log to identify application startup events and record how many times the startup occurred correctly:

```
start="start_date"
end="end_date"
sudo journalctl --since "$start" --until "$end" | grep "Started
    name_application | wc -l"
```

We can then calculate the activation rate as shown in Equation 3.7:

$$\gamma_N = \frac{\text{Total number of successful starts}}{\text{Total observation time}} \tag{4.3}$$

Lastly, but not least, we can calculate the failure rate λ_P in a similar manner, by determining the number of times the startup of an application has failed:

```
start="start_date"
end="end_date"
sudo journalctl --since "$start" --until "$end" | grep "Failed to start
   name_application | wc -l"
```

We can finally calculate the failure rate as described in Equation 3.9:

$$\lambda_P = \frac{\text{Total number of failed starts}}{\text{Total observation time}} \tag{4.4}$$

In an active production environment with many applications generating a large volume of messages in a short period of time, a more efficient approach might be to extract the information useful to us from the `journalctl` system log and store it in a separate log file. This log file turns out to be more accessible and manageable for data analysis, allowing easier and more efficient processing.

4.2 Analysis of System Reliability and Availability in a Multi-Node Architecture

Whether the system can be applied effectively depends directly on the number of nodes required in its architecture. Using more and more nodes results in a less efficient system. This can be clearly observed from the trend in total availability obtained by analysing the Figure 4.1. In fact, it can be seen that the overall effectiveness of the system decreases as the number of applications increases, as evidenced by the decrease in total availability. This phenomenon can be attributed to the fact that the failure of each of the connected nodes can compromise the functioning of the entire system, thus reducing its ability to provide services consistently and reliably.

Let us now examine a system composed of n interconnected nodes. Although each node has its own availability, for simplicity in our calculations we will assume that all nodes have an availability of 0.9. Figure 4.1 shows the model described in Equation 3.6, which shows the availability of the system as the number of connected nodes increases.

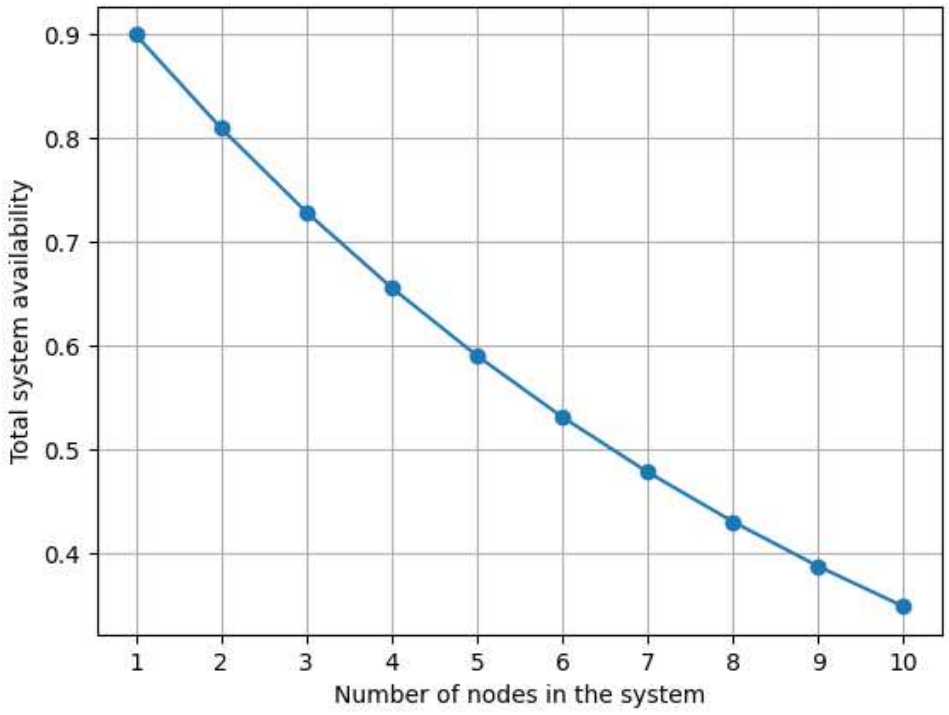


Figure 4.1: Variation of total availability with the number of nodes

We start by considering a system with a single node. Here, the entire system will be available only if that single node is functioning. Therefore, at first, with a single node, the availability will be at its maximum, assuming a value of 90%. However, as the number of nodes in the system increases, the overall availability will decrease. This is because with more nodes in the system, it becomes increasingly difficult for all of them to be functioning at the same time. For example, with two nodes, both must be functioning to ensure that the system is up and running, which means that the probability of this happening is the product of the availability probabilities of each node. This trend continues as the number of nodes in the system increases. With three, four, or more nodes, the probability of all of them being functioning at the same time decreases further, leading to a reduction in overall system availability.

We will now focus on analyzing how the introduction of two replicas in parallel can affect the overall reliability of the system. We will examine how system reliability varies with the number of nodes while holding the number of replicas constant. This will allow us to assess the impact such a configuration has on the reliability and availability of the services offered. Results are highlighted in Figure 4.2.

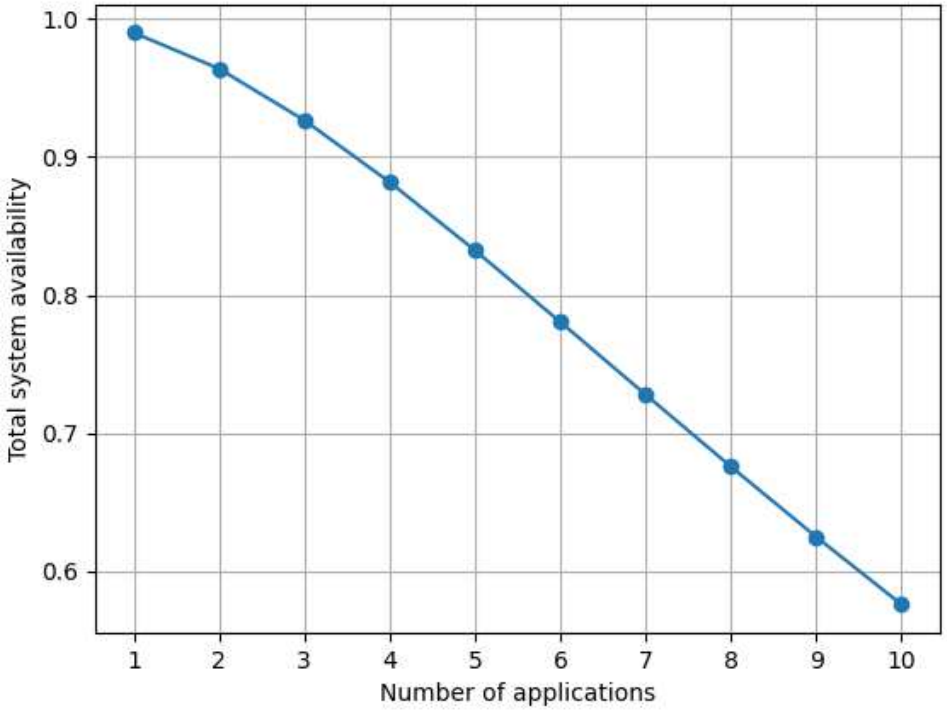


Figure 4.2: Variation of total availability with two replicas

We note that with a small number of nodes, the reliability of the system remains relatively

high, hovering around 90%. However, as the number of nodes increases, we observe a gradual decline in overall reliability. This decline is due to the increased complexity of the system and the increasing probability of failures distributed over multiple nodes.

Let us now consider *n* replicas. In this way, we can get a more complete view of the system as a whole and assess the best trade-off between resource cost and availability. As the number of replicas increases, the overall availability of the system will be higher. However, the resource costs associated with maintaining a large number of replicas must also be considered. The aim is therefore to find an optimal balance between increasing availability and the additional costs we have to bear as the number of replicas increases. Let us now look at the heat map of system availability as the number of nodes and the number of replicas change. Figure 4.3 shows the trend of overall system availability with different configurations of nodes and replicas.

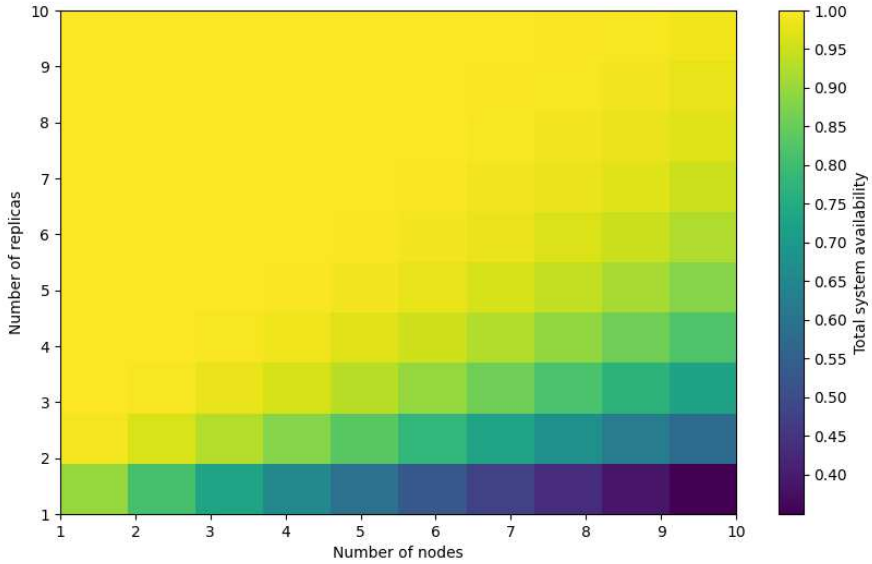


Figure 4.3: Total system availability with varying number of nodes and replicas

Each cell in the heatmap represents the reliability of the system with a certain number of nodes and replicas. Lighter colors indicate higher availabilities, while darker colors indicate lower availabilities.

4.3 HA Proxy load balancing

In this chapter, we will explore the practical implementation of HA Proxy as a solution for load balancing in our distributed system. We will begin by discussing the steps for the installation and configuration of HA Proxy and then present the results obtained through practical testing and evaluation. [12]

The configuration was implemented on a Linux server running Ubuntu 22.02.

After installing the HA Proxy service in the system, it is necessary to edit the HA Proxy configuration file:

```
sudo vim /etc/haproxy/haproxy.cfg
```

To set our HTTP load balancing, the configuration is provided below:

```
global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/haproxy
        stats socket /run/haproxy/admin.sock mode 660 level admin
        stats timeout 30s
        user haproxy
        group haproxy
        daemon
        # Default SSL material locations
        ca-base /etc/ssl/certs
        crt-base /etc/ssl/private
defaults
        log global
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
  stats enable
  stats hide-version
  stats realm Haproxy Statistics
  stats uri /haproxy_stats
  stats auth admin:admin

frontend http_frontend
   bind *:80
   default_backend http_back

backend http_back
   mode http
   option httplog
```

```
    balance roundrobin
    server server1 127.0.0.1:4002 check
    server server2 127.0.0.1:4003 check
```

Here we define a front-end called `http_front` that listens on the default HTTP port 80 and a back-end called `http_back` with two servers hosted in the same machine on ports 4002 and 4003. The statistics uri provides a simple web interface to monitor the status of HA Proxy.
At this point we can start the service and enable it so that it runs at system startup.

```
sudo systemctl start haproxy
sudo systemctl enable haproxy
```

This means that the service will be started whenever the operating system boots without the need for manual intervention, ensuring that HA Proxy is always available to handle load balancing. This helps ensure higher service availability and a lower likelihood of interruptions due to human error or configuration problems. In Figure 4.4 we can see the status of the service after configuring it.

```
● haproxy.service - HAProxy Load Balancer
     Loaded: loaded (/lib/systemd/system/haproxy.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2024-03-09 11:22:46 CET; 2h 20min ago
       Docs: man:haproxy(1)
             file:/usr/share/doc/haproxy/configuration.txt.gz
   Main PID: 149951 (haproxy)
     Status: "Ready."
      Tasks: 9 (limit: 18734)
     Memory: 43.2M
        CPU: 1.485s
     CGroup: /system.slice/haproxy.service
             ├─149951 /usr/sbin/haproxy -Ws -f /etc/haproxy/haproxy.cfg -p /run/haproxy.pid -S /run/haproxy-master.sock
             └─149953 /usr/sbin/haproxy -Ws -f /etc/haproxy/haproxy.cfg -p /run/haproxy.pid -S /run/haproxy-master.sock
```

Figure 4.4: HA Proxy service

When an HTTP request is made from the front-end, the load balancer receives the request and routes it to one of the two available back-end servers. This process occurs according to the configured load balancing algorithm. In this case we used the Round-Robin algorithm, which is a load-balancing method that distributes requests equally among a group of servers using the DNS. Each time a client makes a DNS request for a given domain, the authoritative DNS server returns a different A record with each subsequent request, ensuring that the load is distributed evenly among the available servers.
The experiment is conducted in our local network and the front-end interface is configured to listen on the standard HTTP port 80 at the private IP address 192.168.1.68. As for the back-ends, we created two virtual nodes on the same machine thanks to Docker. They were created through Pyhton and listen on ports 4002 and 4003 respectively staying running while waiting for requests. The code is provided below:

```python
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

class Server1(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

def run(server_class=HTTPServer, handler_class=Server1, address='
    192.168.1.68', port=4003):
    # Define the server's address and port
    server_address = (address, port)
    # Create an instance with the specified address and port
    httpd = server_class(server_address, handler_class)
    print(f'Server started on {address}:{port}...')
    # Start the server and keep listening for requests
    httpd.serve_forever()
```

At this point, we have created a Docker image to containerize our server application. We begin by writing a Dockerfile, which is a text file that contains instructions for building a Docker image. [13] In this file, we select a base image that provides the basic environment for our application. We copy the application code into the Docker image to make sure it contains all the necessary files. If there are dependencies, such as Python libraries, we install them inside the image. Finally, we define the configuration settings needed to run our application as a Docker container, such as environment variables, exposed ports, or startup commands.

```
# Use a base image, such as Ubuntu
FROM ubuntu:latest

# Update the system and install the backend server, for example Python
RUN apt-get update && apt-get install -y python3

# Copy the backend server files into the container directory
COPY server.py /app/server.py

# Expose port 4003 to allow access to the server
EXPOSE 4003

# Mount a volume to persist data
VOLUME /app/data

# Start the backend server when the container starts
CMD ["python3", "/app/server.py"]
```

Once the Dockerfile is ready, we use the '`docker build`' command to build the Docker image. This command reads the Dockerfile and executes each instruction, creating a new Docker image based on the specified configuration. Finally, we use the '`docker run`' command to run a Docker container based on the image created. This command starts a new instance of the container using the specified image, making our server application accessible within the containerized environment.
Figure 4.5 shows the logs of the two virtual backe-end nodes within the Docker containers. We can see how HTTP requests are handled equally between the two available backend servers thanks to the Round-Robin algorithm. Each request is identified by the backend to which it was sent, allowing the workload distribution to be monitored. This approach ensures an even distribution of requests among the backend servers, helping to optimize resources and improve overall system efficiency.

```
2024-03-09 17:07:17,444 - INFO - Server started on port 4002 with backend id Backend 1...
2024-03-09 17:07:17,444 - INFO - Server started on port 4003 with backend id Backend 2...

2024-03-09 17:07:21,058 - INFO - 127.0.0.1 - "GET / HTTP/1.1" 200 - Backend 2
127.0.0.1 - - [09/Mar/2024 17:07:21] "GET / HTTP/1.1" 200 -
2024-03-09 17:07:21,855 - INFO - 127.0.0.1 - "GET / HTTP/1.1" 200 - Backend 1
127.0.0.1 - - [09/Mar/2024 17:07:21] "GET / HTTP/1.1" 200 -
2024-03-09 17:07:22,309 - INFO - 127.0.0.1 - "GET / HTTP/1.1" 200 - Backend 2
127.0.0.1 - - [09/Mar/2024 17:07:22] "GET / HTTP/1.1" 200 -
2024-03-09 17:07:22,720 - INFO - 127.0.0.1 - "GET / HTTP/1.1" 200 - Backend 1
127.0.0.1 - - [09/Mar/2024 17:07:22] "GET / HTTP/1.1" 200 -
```

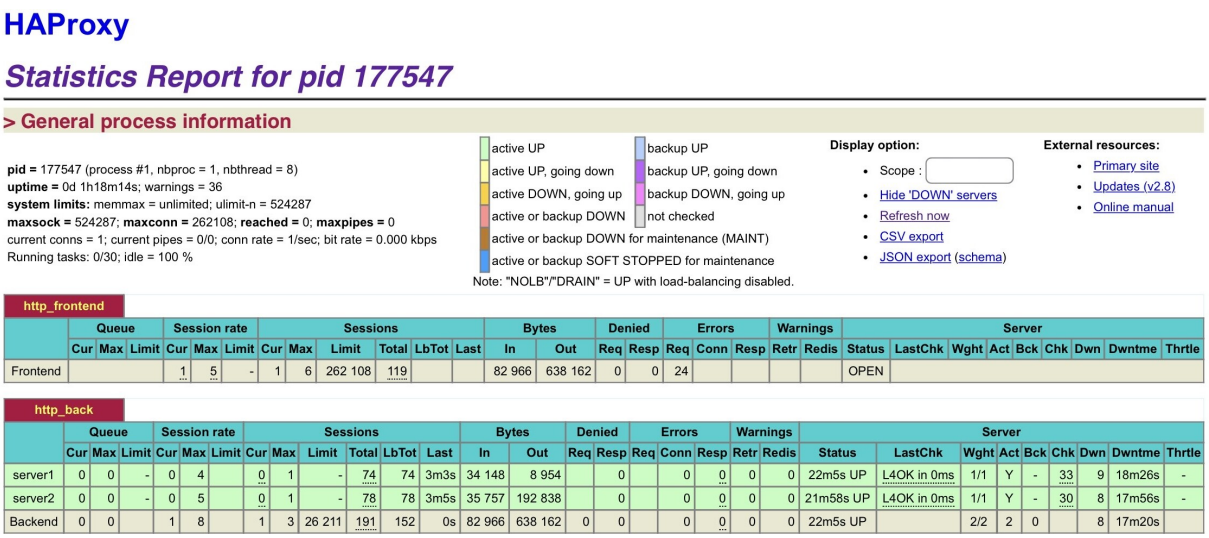
Figure 4.5: Server logs

To monitor the balancing system even more efficiently, HA Proxy provides an interface for monitoring the performance and status of the load balancer server. This web interface provides a detailed overview of server utilization statistics, including data such as the amount of inbound and outbound traffic, number of active connections, and load distribution among back-end servers. One can then monitor the performance of the load balancer server in real time and identify any issues or congestion in workload distribution. The interface is available at the following URL `http://server_ip/haproxy_stats` where `server_ip` represents the IP address of the server on which HA Proxy is installed, as shown in Figure 4.6.

HA Proxy Stats provides a detailed overview of the performance of the server load balancer and associated back-end servers. The following are some of the key insights that can be obtained through HA Proxy Stats:

- **Sessions and Connections**: HA Proxy Stats provides information on the number of active sessions and incoming and outgoing connections. This helps monitor server activity and assess current workload.

- **Request Statistics**: Users can view the total number of HTTP requests handled by the server's load balancer, along with pending requests and average response time. This allows users to monitor the efficiency of request handling.
- **Load Distribution**: HA Proxy Stats displays the load distribution among the back-end servers, including details on requests sent to each server and responses received. This allows the workload balance to be assessed.
- **Back-end Server Details**: Detailed information about each back-end server is available, such as operational status, latency, availability, and active connections. This makes it easier to identify any problems or congestion on individual servers.



HAProxy

Statistics Report for pid 177547

> General process information

pid = 177547 (process #1, nbproc = 1, nbthread = 8)
uptime = 0d 1h18m14s; warnings = 36
system limits: memmax = unlimited; ulimit-n = 524287
maxsock = 524287; **maxconn =** 262108; **reached =** 0; **maxpipes =** 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.000 kbps
Running tasks: 0/30; idle = 100 %

active UP | backup UP
active UP, going down | backup UP, going down
active DOWN, going up | backup DOWN, going up
active or backup DOWN | not checked
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

External resources:
- Primary site
- Updates (v2.8)
- Online manual

http_frontend

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	5	-	1	6	262 108	119			82 966	638 162	0	0	24					OPEN								

http_back

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
server1	0	0	-	0	4		0	1	-	74	74	3m3s	34 148	8 954		0		0	0	0	0	22m5s UP	L4OK in 0ms	1/1	Y	-	33	9	18m26s	-
server2	0	0	-	0	5		0	1	-	78	78	3m5s	35 757	192 838		0		0	0	0	0	21m58s UP	L4OK in 0ms	1/1	Y	-	30	8	17m56s	-
Backend	0	0		1	8		1	3	26 211	191	152	0s	82 966	638 162	0	0		0	0	0	0	22m5s UP		2/2	2	0		8	17m20s	

Figure 4.6: HA Proxy statistics

4.4 Implementation in a real scenario

This test was performed using three preconfigured virtual web servers, with the load distributed equally across each server using HA Proxy. The following results represent the average of ten data observations collected using Round-Robin algorithm.
In order to assess the server's ability to handle the workload, the first test conducted concerns the number of incoming HTTP requests and observing how the system responds increasing the number of back-end replicas. By measuring the server's ability to process a large amount of requests, we can gain crucial information about its capabilities and areas where resources may need to be optimized.
The experiment involves simulating a workload in a production environment. System performance, such as response time and server stability, was measured as the volume of requests increased. To assess the impact of system scalability on performance, tests were performed by increasing the number of replicas from one to three back-end servers. This allowed us to examine how increasing the number of servers affects the overall performance of the system. From Figure 4.7, it is clear that as the number of replicas of the back-end servers is increased, the system can adequately handle the requests per second. This observation indicates that, by distributing requests over independent processing back-end servers, the system can effectively handle heavier workloads.

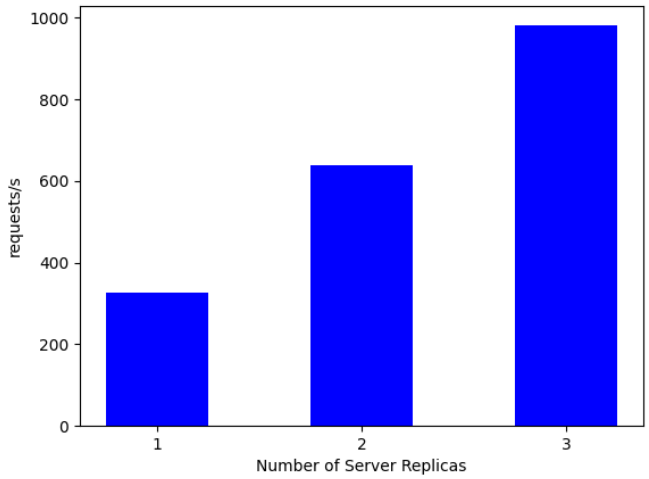


Figure 4.7: Number of Requests/s based on number of server replicas

To get a more precise metric, we can now calculate the system's throughput, which is the amount of data sent from the client to the back-end server via HA Proxy in a given time interval, as described in (4.5).

$$\text{Throughput (KB/s)} = \frac{R \cdot P_{\text{size}}}{T} \tag{4.5}$$

Where:

- R is the number of HTTP requests
- P_{size} is the payload size of the HTTP requests in KB
- T is the time interval taken into account in seconds

When looking at HTTP requests for a service in the IoT wolrd, the size of the payload can vary based on the type of request being received. For instance in a GET request the payload is typically small or empty since it involves the client just requesting data from the server. In the other hand, in a POST request the payload might include information sent from the client to the server like when creating gateways meters or digital twins.
To understand of how large these payloads are, we observed and recorded a large number of requests over time, using HA Proxy logs. By having an amount of requests and assuming they are independent and identically distributed (i.i.d) we can make use of the Central Limit Theorem. This theorem states that the distribution of the sample mean of a large number of independent and identically distributed random variables will be approximately normally distributed, regardless of the original distribution of the variables themselves. [14] In practice, this means that we can approximate the payload size distribution to a Gaussian distribution. This approximation allows us to calculate the mean value of the average payload size, which can be used to analyze and optimize the performance of the system.

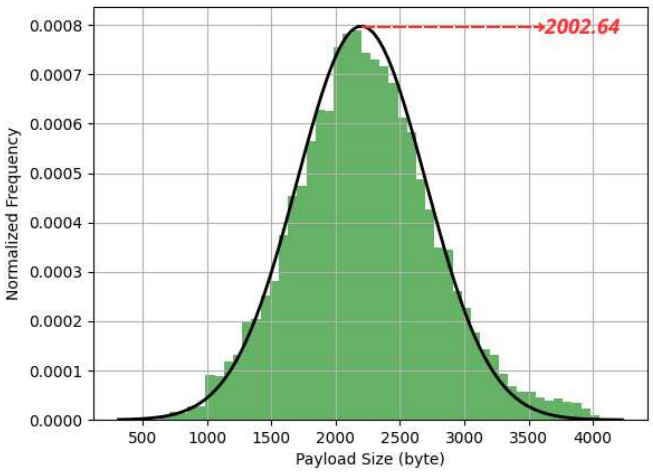


Figure 4.8: Distribution of payload size of HTTP requests

Figure 4.8 presents the distribution of HTTP requests, which, with a large number of requests, shows a tendency to resemble a Gaussian distribution. This observation allows us to calculate the expected value, identifying the point at which the curve reaches a maximum. In addition, the presence of POST-type HTTP requests is evident for higher payload size values, confirming the hypotheses made earlier.

In addition, CPU utilization can be monitored in relation to the number of replicas. By using only one replica of the back-end server, the entire load is handled by a single node. Therefore, the CPU utilization of this server is very high causing congestion or delays, especially at peak transmission times. By increasing the number of replicas of the back-end servers, the workload is divided equally between the nodes. The round-robin algorithm makes sure that requests are sent cyclically to the different servers, allowing them to share the load equally. As a result, CPU utilization on each replica will be lower than in the case of a single replica. Although the total number of requests may increase with multiple replicas, the effect of load distribution will result in more efficient utilization of CPU resources and better ability to handle traffic during load peaks.
The overall results of the analysis are presented in Table 3. The data are illustrated in a high load context, in which the HA Proxy server is subjected to more requests than its maximum processing capacity per second. This scenario allows us to assess the maximum capacity of the system and the throughput in the face of extreme workload.

N° of replicas	**Requests/s**	**Throughput (MB/s)**	**CPU Utilization (%)**		
			Node 1	**Node 2**	**Node 3**
1	7.43	14.88	35.47		
2	13.85	27.74	21.09	22.09	
3	20.22	40.49	8.72	7.97	8.60

Table 4.1: Downtime for different levels of availability

4.5 Keepalived implementation

In the analysis conducted so far, we have implemented HA Proxy as a load balancer and web server to distribute traffic across multiple back-end servers. This approach allows more requests per second to be handled more efficiently. However, if the server on which HA Proxy is installed experiences problems, the service may be interrupted. To mitigate this risk, a backup server can be used that, in the event of failure of the ”master” server, becomes the new master while keeping the connection active. This transition should occur automatically the moment the master server goes offline.

One solution to ensure this high availability is to use Keepalived, an open-source software designed to keep critical services up and running even in the event of hardware failure or system malfunction. Keepalived implements the Virtual Router Redundancy Protocol (VRRP), allowing multiple servers to share a single virtual IP address. [15]

We designed a high-availability load balancing system using HA Proxy and Keepalived within Docker containers in the same network. This approach provides service availability and redundancy in case one of the nodes fails. Now, to better understand how this system works, Figure 4.9 shows the general topology.

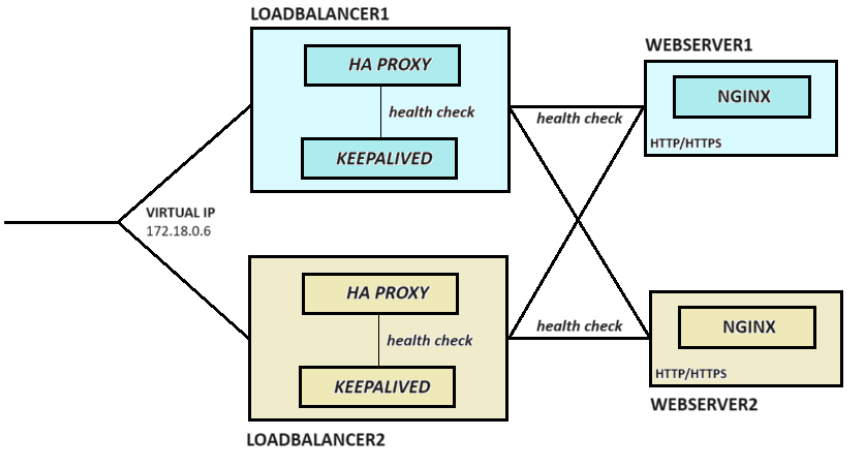


Figure 4.9: Topology of the system

There are two web server instances, named WebServer1 and WebServer2, based on Nginx web server. At the entrance of the web servers, we find two instances of load balancers: the master node LoadBalancer1 and the backup node LoadBalancer2. Both nodes use HA Proxy and Keepalived and share the same virtual IP address, which is accessible to clients. In addition, health checks are implemented both on the Keepalived side, to monitor the health of the load balancers, and on the HA Proxy side, to monitor the health of the web servers. This configura-

tion ensures business continuity and redundancy of the system, improving its overall reliability. To begin, we need to create the Docker containers to host the web servers and load balancers. To enable communication between the containers, we will create a Docker network so that the containers are isolated and can communicate securely. Once the containers are created, we will configure HA Proxy as described in the previous Chapter, setting the two web servers as back-end servers, and then start the service. Now the web servers will be accessible via the load balancer and will be able to handle incoming requests in a distributed and secure manner.
Now we need to configure Keepalived so that the LoadBalancer1 node becomes the master, while LoadBalancer2 becomes the backup node, used in case the master fails. To ensure this, some changes must be made to the service configuration file on both nodes:

```
vim /etc/keepalived/keepalived-master.conf      # In the master node
vim /etc/keepalived/keepalived-backup.conf      # In the backup node
```

This file contains crucial details related to VRRP configuration, such as node status (master or backup), virtual router identifier (VRID), priority, and shared virtual IP (VIP) address. The use of a single VIP is critical to ensure continuity of service in the event of failure or maintenance on one of the nodes. The Virtual IP is dynamically assigned to the active node, known as the master, within a VRRP group. Although there are multiple nodes in the group, only the master actually holds and uses the VIP to route incoming network traffic. In case the master node fails, Keepalived automatically steps in to transfer the Virtual IP to the backup node. This transition occurs without interruption to end users, ensuring continuity of services even if one of the nodes fails. The the configuration file for the master node is provided below:

```
vrrp_script chk_haproxy {
  script "pgrep haproxy"
  interval 2
  weight 2
}
vrrp_instance VI_1 {
  state MASTER
  interface eth0
  virtual_router_id 51
  priority 101
  advert_int 1
  virtual_ipaddress {
    172.18.0.6
  }
  track_script {
    chk_haproxy
  }
}
```

It is divided into two main sections:

- `vrrp_script chk_haproxy` allows to check the status of the HA Proxy process. It means that the VRRP instance monitors the state of the HA Proxy process using the defined script, which is executed every 2 seconds to check whether the HA Proxy process is running. If the process is active, the script is considered successful and is assigned a weight of 2. The weight is used to determine the relative importance of the script within the overall monitoring of the VRRP instance. A higher weight indicates greater importance in the overall health of the VRRP instance.

- `vrrp_instance VI_1` defines the details of the VRRP instance. The current node is configured as MASTER (state MASTER) and uses the eth0 interface for VRRP communication. The virtual router identifier must be unique for each instance (`virtual_router_id` 51). Node priority (priority 101) indicates that this node has the highest priority and should be selected as master. Backup nodes should have a lower priority. The VIP address assigned to this node is 172.18.0.6 (`virtual_ipaddress` 172.18.0.6) and should also be the same for the backup nodes to ensure continuity of service during failover. Finally, the script check `chk_haproxy` is tracked (`track_script chk_haproxy`), which means that if the script fails, the node's priority will decrease, allowing the higher priority backup node to assume the role of master.

During the configuration of the Keepalived node, it was necessary to select a virtual IP address that would be assigned to the active node within the VRRP group. To locate an available IP address that has not already been assigned to other running containers, it is necessary to examine the IP addresses of the Docker containers in the specific network. To do this, we can use the command:

```
docker inspect -f '{{.Name}} - {{range .NetworkSettings.Networks}}{{.
    IPAddress}}{{end}}' $(docker ps -q)
```

This command allows us to view the IP addresses assigned to the various containers in the Docker network, including their names and associated IP addresses, as shown in Figure 4.10.

```
iwdroot@border:~$ docker inspect -f '{{.Name}} - {{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' $(docker ps -q)
/LoadBalancing2 - 172.18.0.4
/LoadBalancing1 - 172.18.0.5
/WebServer2 - 172.18.0.3
/WebServer1 - 172.18.0.2
```

Figure 4.10: List of Docker containers and their IP addresses within the network

We can now start the Keepalived service and verify that the Virtual IP address has been successfully assigned checking the network interface information and see which node the virtual IP has been bound to.

```
root@loadbalancer1:/etc/keepalived# ip addr show eth0
eth0@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether 02:42:ac:12:00:04 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 172.18.0.4/16 brd 172.18.255.255 scope global eth0
   valid_lft forever preferred_lft forever
inet 172.18.0.6/32 scope global eth0
   valid_lft forever preferred_lft forever
```

Figure 4.11: Ethernet interface

On the master node we can see that this is exactly the VIP address we set just now, while on the backup node there is no such information, which means that this node is still in the backup state. When the master node is unavailable, the virtual IP will be bound to the backup node.

To test our architecture we may stop one of the web servers to see how the load balancer manages the requests, e.g.:

```
docker stop WebServer1
```

Figure 4.12 shows the HA Proxy interface already discussed in Chapter 4.4. As expected, if one of the back-end servers (e.g., WebServer1) is down, all incoming requests will be routed to the running WebServer2. This is accomplished by load balancing configured through HA Proxy, which automatically detects the status of the server and routes traffic to the working instance. In this way, despite service interruption on a node, communication remains active thanks to load distribution and duplication of back-end servers.

app

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
WebServer1	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	2m48s DOWN	* L4CON in 3054ms	1	Y	-	6	2	2m48s	-
WebServer2	0	0	-	0	4		0	1	-	135	135	7m14s	34523	237962		0		0	0	0	0	1h18m UP	L7OK/200 in 0ms	1	Y	-	0	0	0s	-
Backend	0	0		0	1		0	0	50000	151	135	0s	76838	846257	0	0		0	0	0	0	1h18m UP		1	1	0		0	0s	

Figure 4.12: HA Proxy statistics with WebServer1 in down state

Now we can proceed by testing the case in which the load balancer in master mode is idle, which is the main reason behind this analysis and implementation of Keepalived. To simulate this situation, we interrupt the service on the LoadBalancer1, causing an interruption in the flow of traffic handled by this node. This test will allow us to evaluate the effectiveness of the failover system implemented with Keepalived in ensuring service accessibility even in the event of a failure of the main load balancer.
Once the service on the LoadBalancer1 is interrupted, we can proceed by analyzing the logs of the container in which the LoadBalancer2 is installed, as shown in Figure 4.13.
From the logs, we can see that at the time of the interruption of the master load balancer, the backup load balancer took over the role of master, thus ensuring the continuity of the load balancing service.

```
root@loadbalancing2:/etc/keepalived# tail -f /var/log/syslog
Keepalived_vrrp[62]: VRRP_Instance(VI_1) forcing a new MASTER election
Keepalived_vrrp[62]: VRRP_Instance(VI_1) Transition to MASTER STATE
Keepalived_vrrp[62]: VRRP_Instance(VI_1) Entering MASTER STATE
Keepalived_vrrp[62]: VRRP_Instance(VI_1) setting protocol VIPs.
Keepalived_vrrp[62]: Sending gratuitous ARP on eth0 for 172.18.0.6
Keepalived_vrrp[62]: VRRP_Instance(VI_1) Sending/queueing gratuitous ARPs on eth0 for 172.18.0.6
Keepalived_vrrp[62]: Sending gratuitous ARP on eth0 for 172.18.0.6
Keepalived_vrrp[62]: message repeated 4 times: [ Sending gratuitous ARP on eth0 for 172.18.0.6]
Keepalived_vrrp[62]: VRRP_Instance(VI_1) Sending/queueing gratuitous ARPs on eth0 for 172.18.0.6
Keepalived_vrrp[62]: Sending gratuitous ARP on eth0 for 172.18.0.6
```

Figure 4.13: Server logs for LoadBalancer2

Thus, we can say that the implementation of Keepalived in combination with HA Proxy has proven to provide increased reliability and availability of load balancing services in the context of the Docker infrastructure. Through the use of Keepalived, we are able to effectively manage any outages or failures of the main nodes, allowing the system to continue to operate without significant disruption. The configuration of the master and backup nodes, along with service state monitoring, ensures a smooth transition of the master role in the event of unexpected failures.

Chapter 5

Application of Machine Learning Techniques

5.1 Intelligent Load-Balancing Techniques in Cloud Environments

In this Chapter, we explore intelligent load-balancing techniques that are useful for optimising server resource utilization in cloud environments. These techniques are classified according to various factors that play a significant role in improving overall efficiency. These factors include the availability of server CPU and memory resources, adherence to Service Level Agreements, network congestion prediction, QoS assurance, service response time estimation, and management of storage requests within the cloud [16].

Machine learning is a subset of artificial intelligence and it involves training systems to perform tasks without explicit programming. This is achieved by combining historical data and statistical techniques through a process called training, which enables the creation of models that can predict new, unseen values [17]. Deep learning, a specialised branch of machine learning, uses complex neural network architectures with deeper layers and large data sets. By integrating feature extraction and prediction within deep networks that possess hidden layers, deep learning outperforms traditional approaches to machine learning, achieving superior performance. [18]

In the field of cloud computing, task scheduling and load balancing are very important aspects to consider within data centers. In cloud computing systems, service providers allocate available resources in a way that improves Quality of Service.

We will provide an overview of load balancing using machine learning techniques such as Naive Bayes, Random Forest, Decision Tree and Support Vector Machine (SVM). The latter

will be the one used for our analysis, in order to reduce the processing time (makespan) of requests arriving at the load balancer.

We will now introduce a new dynamic load balancing algorithm adapted to cloud environments. Leveraging various machine learning algorithms, the proposed scheme optimizes the allocation of requests among machines to reduce the overall processing time, also known as makespan.
The main objectives of this study are the following:

- Usage of SVM technology for efficient application scheduling effort to minimize both time and resource utilization.
- Classification with the help of machine learning techniques for the fair and equal distribution of incoming requests along the existing resources.
- Comparison with the existing and valid algorithms.

Figure 5.1 presents the flowchart of the proposed architecture.

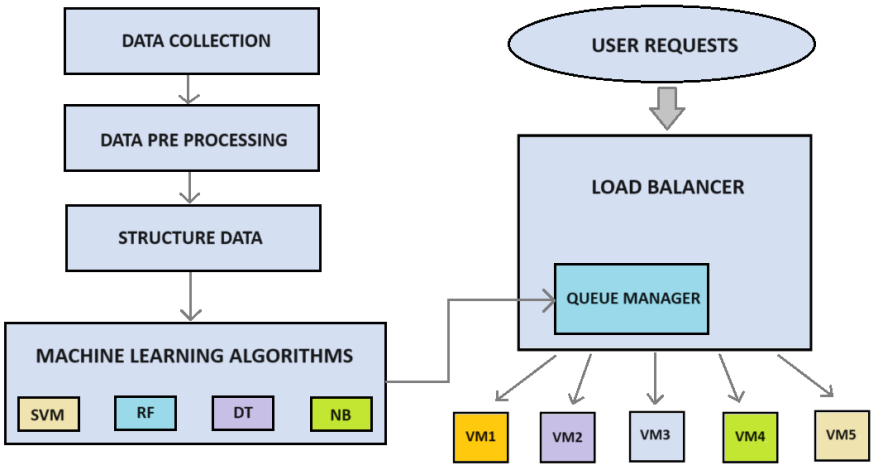


Figure 5.1: Scheme of the proposed architecture

Initially, the collected data undergoes a preprocessing phase to eliminate unrelated information and to obtain a well-defined data structure. Next, a machine learning model is applied to classify the data into three classes, depending on different performance levels:
The dynamic scheduling algorithm, based on the machine learning model, is used to classify the data according to different VM conditions to provide an equal distribution of load among servers. The task scheduling system and servers are all executed in real-time based on the VM's current and previous state. The Virtual Machine Manager is responsible for the complete management

of operations within the virtual environment and provides detailed information on the status, availability and performance of virtual machines.
Until now we have dealt with load balancing with HA Proxy using the Round Robin algorithm, without considering the memory or CPU usage of individual servers. However, we now want to design an algorithm in which the load distribution decision is made at runtime, using current process information and system state. To do this, several aspects of load balancing need to be considered:

- **Identification of user task requirements**: it is the first step in load balancing. It involves understanding what tasks are coming to the system and their needs in terms of the resource.
- **VM resource identification information**: it involves collecting information about the present state of resources related to VMs in terms of usage and availability.
- **Task scheduling**: itplays an important role in cloud computing, and several scheduling policies are used to distribute tasks to computing nodes. This methodology helps in proper utilization of resource and also improves performance in mission-critical environments.
- **Resource allocation**: enabling a clear resource allocation strategy is vital because it ensures that the most effective approaches to resource management are used, thus also enhancing general resource use.
- **Migration**: it is specifically fundamental in cloud load balancing as it is vital in ensuring resource optimization. Two cloud solutions could be put properly, with VM migration and task migration being instances. Cloud load balancing utilizes the flexibility of cloud solutions to surf the changing nature of work and also general applicability. Moreover, it also offers efficient cloud app availability monitoring through proper load and traffic allocation in the cloud infrastructure offering efficient health checks.

A viable solution can be to implement the classification of virtual machines performance according to the use of the RAM memory. Since the hosts have different RAM memory, we can analysing the ratio of the used RAM memory to the total RAM capacity. The current state of use of the virtual machines can be classified into three classes based on this data:

- **Class 1 - Low Performance:** Virtual machines that are characterized by long-time processing of requests. In this regard, the VMs require more time to process the tasks . This may be due to the lack of resources or inefficiency in their allocation.
- **Class 2 - Medium Performance:** Virtual machines with moderate processing times for requests could be considered medium performance. These VMs handle tasks within a reasonable amount of time, indicating efficient resource usage without excessive delays.

- **Class 3 - High Performance:** VMs responds quickly since the time for a request is short. Thus, this class is distinguished in that the processing time is so quick that it can be considered very effective.

5.2 Machine learning algorithms for classification

In this Section, we introduce machine learning techniques for classifying tasks under different load conditions, utilizing various classifiers to analyze VM performance.

1. **Decision Tree (DT)**
   Decision Tree is a supervised learning algorithm for classification task. As the name implies, it builds a tree to represent possible decisions based on a set of decision rules. To this end, the dataset is consecutively divided by subsets until it reaches the decision nodes. Similarly, information on leaf nodes combines different outcomes into classes . In turn, split decision nodes can present features or attributes that can later be used for prediction tasks. In general, the goal is to maximize the entropy and reach the maximum information gain . However, making too many splits can lead to model overfitting. Indeed, decision trees are easy to understand and interpret . Yet, the model may be unstable for certain instances due to impurity within leaf nodes.

2. **Random Forest (RF)**
   Random Forest is a collective learning method consisting of many small decision trees. By aggregating tree decisions, it corrects weaknesses that individual trees may have. However, in the case of complex data distributions, random forests can include poorly performing tree classifiers and thus produce incorrect classification results. The crux of optimising the performance of random forests is to identify and filter out faulty trees that would otherwise have a negative impact on the overall classification accuracy.

3. **Naïve Bayesian**
   Naïve Bayesian is a statistical classification tool known for its simplicity and efficiency. It assumes that features are conditionally independent of the class label, simplifying the classification process. Despite the naïve assumption, research suggests that Naïve Bayesian classifiers often achieve comparable or higher accuracy and speed than other classification methods such as decision trees or neural networks. In this approach, each tuple is assigned to the class with the highest posterior probability, based on its characteristics. Suppose we have m classes, labelled as $\{y_1, y_2, ..., y_m\}$, and a dataset represented by tuple-X. The Naïve Bayes classifier predicts that X belongs to the class with the highest backward probability, i.e. class Y_i. In our case, we have several virtual machines $\{VM_1, VM_2, ..., VM_m\}$

available to execute a set of tasks $\{T_1, T_2, ..., T_n\}$. We can use the Naïve Bayes classifier to determine on which virtual machine to execute each task, taking into account the characteristics of the virtual machines and the task requirements. The classifier analyses this information and estimates the probability that a task belongs to each of the available virtual machines. It then assigns the task to the virtual machine with the highest probability, thus enabling optimised workload balancing between the available virtual machines.

4. **Support Vector Machine**

   The SVM algorithm is widely used in machine learning for classification and regression. It is known for its robustness and ability to handle both binary and multiclass classification problems. In order to design efficient multiclass classifiers of low complexity, sophisticated techniques are often employed, such as decomposing the multiclass problem into a set of efficiently solvable binary classification problems.

   In practice, the SVM finds an optimal hyperplane that separates the training data so as to maximize the margin between different classes. Hyperplanes are represented as lines in high-dimensional spaces, and the classification of a new data point is done by determining on which side of the hyperplane the point is located. In the case of a multiclass classification problem, the SVM uses a formulation that optimizes the separation between multiple classes, often by training several binary classifiers and combining their predictions to achieve multiclass classification. Support vectors are the data points closest to the hyperplane and play a key role in defining the optimal hyperplane. Maximizing the margin between support vectors is important to ensure accurate classification of future data points.

   Figure 5.2 shows the classification of three classes. The objective is to find a plan that maximises the margin, i.e. the maximum distance between the data points of the three classes. This increase provides greater confidence in the classification of future data points between virtual machines.

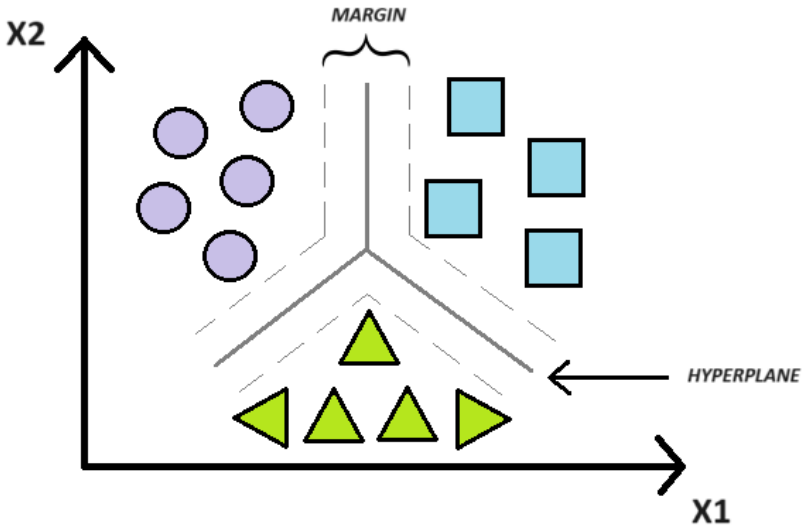


Figure 5.2: SVM Algorithm with three classes

5.3 SVM Algorithm applied to load balancing

In this section, we will explore the application of the SVM algorithm to load balancing, focusing on how this approach can be implemented and the benefits it can offer. Using historical data on server performance and request characteristics, the SVM algorithm can predict which virtual server is best suited to handle a given request in real time. [19]
The SVM model identifies historical processing times on the basis of requests. It also performs clustering techniques so that the virtual machines (hosts) are divided into high-performance, medium-performance and low-performance groups of instances. The process is executed in K-means. Finally, it assigns the requests with the highest processing load to the virtual machines with the lowest activity level
The classification of incoming requests according to the algorithm is mainly based on processing time and uses the SVM algorithm. In this way, we can determine to which group of virtual machines a request will be assigned. For virtual machines, we divided them into clusters with K-Means to obtain: three closest clusters (k=3). The VMs in cluster 3, which is the cluster with the most available resources, are prioritised for handling complex requests with longer processing times. This algorithm will reduce the communication load between the virtual machine and existing resources, increasing throughput and, consequently, service requirements for users.

For better organisation and clarity, we have divided the algorithm into three distinct modules, each of which is described in a dedicated subsection as follows.

5.3.1 Request classification using SVM algorithm

The SVM algorithm, in this module, uses the attributes of the request to compute the processing time of the request and then classifies the requests. These parameters, such as the size of the request, length of response, maximum length, etc., help to determine the processing time group. The formulation for classifying the processing time of a request is as follows:

$$\text{Processing Time Group} = \text{SVM}(X_1, X_2, ..., X_n) \tag{5.1}$$

Where each X_i is an attribute of the request sent to the cloud.
The groups for the processing time classification can differ. They usually range from 4 to 8-10 or even more, depending on the variability of the requests. The SVM obtains information about the last requests. This information is gathered over a period from 1 to 10 minutes. It enables training dataset to always be updated and real-time.

Suppose we have *m* service requests with *n* attributes each.
The training data set is represented as an X-matrix of size $m \times n$, where each row represents a request and each column represents an attribute of the request.

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix} \tag{5.2}$$

Suppose we also have a vector y of size $m \times 1$ that contains the class labels corresponding to each request in the training dataset.

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{5.3}$$

Where y_i represents the processing time class of i-th request.
The SVM algorithm builds a model on the training dataset X and class labels y, looking for an optimal hyperplane to separate the classes of processing time. This model is used to classify new requests into the appropriate groups of processing time.

5.3.2 Clustering of virtual machines and resources

In this module, we will use the k-Means clustering algorithm (with k = 3) to divide virtual machines into groups based on activity level, using virtual machine resources including high, medium and low activity clusters. The clustering analysis Considered varies for each parameter, such as CPU utilization, RAM memory and different comparison to evaluate the precise state of VM. As a result of the k-Means algorithm, a valuable virtual machine is assigned to one of the following grouping:

- Cluster 1: represents the classes with low performances
- Cluster 2: represents the classes with medium performances
- Cluster 3: represents the classes with high performances

The choice of k = 3 is based on the division of virtual machines into three distinct activity levels. However, it is possible to extend the value of k to 4, 5 or even more, depending on the specific needs of the environment. The optimal number of clusters to be used remains the subject of further study.

To obtain more accurate results, we use the same data used to train the SVM model, i.e. the most recent information on virtual machines, to form the training dataset for the clustering algorithm. This allows us to keep the model up-to-date with the latest virtual machine activity information and dynamically adapt it to changes in the cloud environment.

In our practical implementation, we have decided on a classification of the performance of virtual machines based on the utilization of RAM memory. Considering that the hosts have different memory capacities, we analyse the percentage of RAM memory used in relation to the total capacity. Based on this data, we can divide the current state of virtual machines into the three previously defined clusters:

- **Low Performance:** Virtual machines with a high utilization of RAM compared to the total capacity can be grouped as low-performance machines. Since they make full use of the resources available on the host, these virtual machines have almost no room to perform additional workloads, thus limiting overall performance.
- **Medium Performance:** VMs with moderate RAM usage for total capacity can be recognized as medium performance. These VMs take a fair amount of resources that are not running at their peak capacity, which enables them to continue their operations at a reasonable performance level.
- **High Performance:** VMs with low RAM usages for total capacity could also be classified according to high performance. Since these VMs use few resources for executing new workloads, they continue to maintain high system performance.

5.3.3 Service allocation

This module manages the allocation of requests to VMs. When a request arrives at the system, Module 1 classifies the request according to its characteristics, e.g. type of request or the size. Module 2 groups VMs according to their current performance. This can be done by considering RAM utilization or other performance metrics, thus dividing the VMs into clusters with different capacities and load levels. Using the results of the request classification from module 1 and the clustering of VMs from module 2, the algorithm determines which VM is best suited to process the current request. This process considers the characteristics of the request and the performance of the VMs to find the optimal match. If the request processing time has already been calculated, the request is routed to the most appropriate VM based on performance and available capacity.
For example, requests with a longer processing time could be routed to VMs with high capacity, while smaller requests could be allocated to VMs with lower workloads.

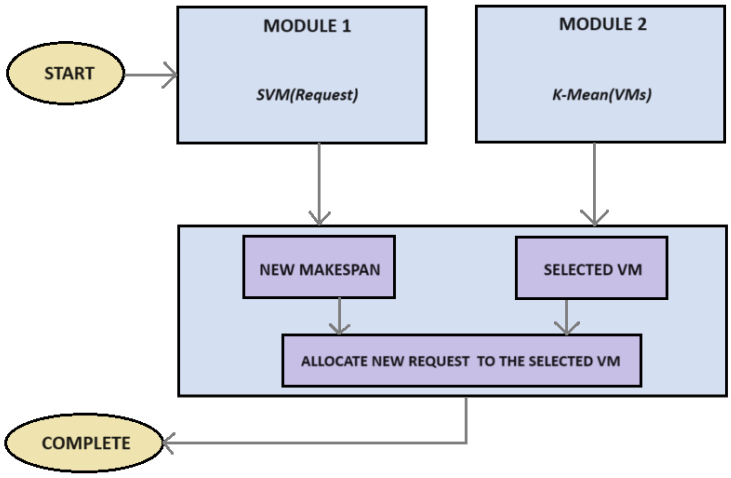


Figure 5.3: Algorithm process flow

5.4 SVM Algorithm Implementation and Simulation

In this Section, we will explore the implementation and simulation of load balancing algorithm based on SVM. The use of SVM to classify incoming requests according to their characteristics will be illustrated, enabling optimal distribution of available virtual resources. For training the SVM algorithm, we used a training set that included a number of request examples, each characterized by three key attributes: the size of the request, the length of the response and the maximum length. These features are used to calculate the processing time of requests. The algorithm is shown below:

```python
from sklearn.svm import SVC
import numpy as np

training_data = np.array([
    [1500, 500, 10000],     # [size, response length, max length]
    [2000, 1000, 20000],
    [4000, 1500, 35000],
])
labels = np.array(["low", "medium", "high"])
svm_model = SVC()
svm_model.fit(training_data, labels)
def classify_request(new_request):
    predicted_class = svm_model.predict([new_request])
    return predicted_class[0]
```

The 'training_data' array contains the request data used for training the SVM model. Each row represents a request instance and the values in each row represent the characteristics of the request (size , response length , maximum length). The 'labels' array contains the class labels associated with each request, which indicate the expected processing time (processing time) for that request. We use the SVC class of scikit-learn to initialize an SVM model anf then we train the model using the fit method. To perform the classification we use the 'predict' method of the SVM model, taking as input the characteristics of a new request and returning the predicted class for the request from the trained SVM model.
We can now simulate the classification of a request and observing the prediction of the expected processing time by the SVM model: Let us consider an HTTP POST request as an example. This type of request typically involves a large amount of data transferred through HTTP communication, so we can expect to be classified with "high" processing time label.
The Figure 5.4 demonstrated our expectations.

```
Characteristics of the request:
Request size: 3000 Bytes
Response length: 1400 Bytes
Maximum length: 31480 Bytes
Classification of processing time: high
```

Figure 5.4: Classification of HTTP request

We can now focus on the k-means algorithm, which allows us to group VMs based on their percentage RAM utilization, dividing them into clusters with similar resource utilization characteristics. This allows groups of VMs with similar performance to be identified, facilitating the appropriate allocation of requests based on performance needs.

```
from sklearn.cluster import KMeans
import numpy as np

free_ram_usage_data = np.array([65, 20, 85, 35, 55, 90, 40, 45, 80, 95])  # RAM usage data for each virtual machine (in percentage)
free_ram_usage_data = ram_usage_data.reshape(-1, 1)
kmeans_model = KMeans(n_clusters=3)
kmeans_model.fit(ram_usage_data)
cluster_labels = kmeans_model.labels_
low_performance_vms = [idx for idx, label in enumerate(cluster_labels) if label == 0]
medium_performance_vms = [idx for idx, label in enumerate(cluster_labels) if label == 1]
high_performance_vms = [idx for idx, label in enumerate(cluster_labels) if label == 2]
```

In the example, we considered a set of data representing RAM usage by different VMs within a cloud environment. This data could be collected by monitoring tools or by software agents deployed on the VMs themselves, which periodically send RAM usage data. To get an objective value, we considered the percentage of RAM memory used in each VM. This real-time data is then used as input for the K-Means algorithm, which analyzes RAM utilization patterns and groups VMs based on similarities in their memory utilization behavior. This makes it possible to identify groups of VMs with similar resource utilization characteristics, facilitating the optimal allocation of requests.

Now, we are ready to combine the SVC and K-Means models in order to efficiently allocate our HTTP requests to VMs. When a new HTTP request arrives, it is first classified using an SVM to understand the workload it produces. It is then assigned to the most suitable virtual machine. The selection is based on both its complexity, determined by the SVM label, and the desired optimal performance, obtained from the K-Means cluster labels. To implement this algorithm, we use the `'allocate_requests'` function. The latter creates an empty dictionary called `'allocated_requests'`, which will be used to organize incoming requests according to desired performance. The keys are “low”, “medium” and “high”, which denote the performance level of VM: the function proceeds to iterate through all incoming requests stored in the requests list. For each request, it retrieves the `'processing_time'`. If the processing time is ”low,” the request should be routed to a cluster with VMs of lower performance. Similarly, if the processing time is ”medium”, the current request is added to the requests list, containing those for VMs with medium performance. If the processing time is ”high,” the request is is managed by the high-performance virtual machine.

```python
def allocate_requests(requests, low_perf_vms, medium_perf_vms,
    high_perf_vms):
    allocated_requests = {"low": [], "medium": [], "high": []}
    for request in requests:
        processing_time = request["processing_time"]
        if processing_time == "low":
            allocated_requests["low"].append(request)
        elif processing_time == "medium":
            allocated_requests["medium"].append(request)
        elif processing_time == "high":
            allocated_requests["high"].append(request)
    return allocated_requests
```

At this point we can simulate the arrival of a POST request and see in which virtual machine it is handled.

Consider a request with a size of 3KB, a response length from server to client of 1.5KB, and a maximum allowed length of 30KB. In Figure 5.5, we can see that the request has been ranked with a high processing time, and consequently it has been sent to the cluster with the highest amount of RAM memory available. In addition, the algorithm will select the best virtual machine at that time in terms of memory utilization.

```
Request [3000, 1500, 30000] has been sent to the 'High Performances' cluster
The VMs in the 'High Performances' cluster are:
VM 2: 85%
VM 5: 90%
VM 9: 95%
The request will be handled by VM 9 with the highest RAM usage available
```

Figure 5.5: Results of the algorithm

5.5 Comparison between traditional and SVM-based Load Balancing

In this section, we will compare the execution time of the traditional Round Robin algorithm and the SVM-based model. As mentioned in Chapter 4, the Round Robin algorithm distributes incoming requests sequentially over the available virtual machines, so that all requests are processed evenly and fairly. In contrast, the SVM-based model applies an SVM classification model to classify the processing time of incoming requests. This model is trained with historical data of request parameters and virtual machine performance. New requests are classified using the SVM model and assigned to virtual machines where they can be executed efficiently.

The performance of the two load-balancing methods is compared by analysing the change in execution time as requests to the load balancer increase, using an increasing queue of requests. The average time to process an increasing queue of requests is measured with the round-robin and the SVM-based balancing algorithms.

In the experiment setting, we have used five virtual machines. The performed requests are generated with the lengths and sizes randomly there varies from 5 to 1000. Our analysis results of the two balancing algorithms, Round-Robin and SVM, can be shown in the table below:

Requests	Round Robin	SVM
5	140.35	141.06
10	286.73	285.28
20	571.46	566.45
50	1430.93	1416.14
100	2860.37	2782.03
200	5714.19	5425.65
500	14286.63	13343.63
800	22857.85	18391.09
1000	28571.82	22526.67

Table 5.1: Execution time (in ms)

For better understanding, Figure 5.6 shows the trend of execution time as the number of requests increases:

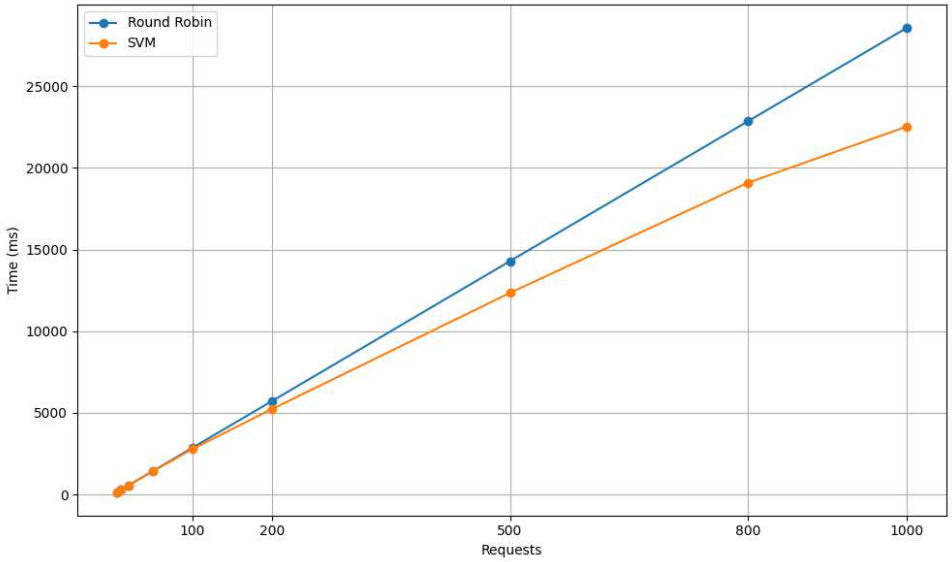


Figure 5.6: Execution time as the number of requests increases

From the graph, we can see that when the number of requests is low, load balancing algorithms such as Round Robin and SVM turn out to behave similarly in terms of execution time. However, when the number of requests increases, SVM tends to become more efficient than Round Robin. SVM uses a classification model to assign requests to virtual machines, based on the characteristics of those requests. This approach can be more efficient in terms of resource utilization than Round Robin, which does not consider differences in requests. In addition, the algorithm dynamically adapts to changes in workload and request characteristics, enabling better distribution of requests to virtual machines. Last but not least, with a of training data, the SVM algorithm has more information on which its classification decisions can be based. This means it can learn more complex and accurate patterns to distinguish between different types of requests and assign them to the most appropriate virtual machines.

We can therefore say that use of the SVM algorithm for load balancing can lead to faster response times, better resource utilization, and greater system scalability in response to varying workloads.

Chapter 6

Conclusions

The research conducted explored a range of issues concerning critical infrastructure and SLA analysis in the context of communications and the Internet of Things. The general archetype of the IoT was studied, along with a real-world case study, in order to better understand the architecture and operational challenges in this area.

Next, methods and tools used to ensure reliability, optimization and infrastructure management were examined, including theoretical models of reliability, node interconnection, container virtualization and orchestration, node replication and load balancing.

A particular focus was placed on observability data analysis and data collection, with the implementation of reliability and system availability models in a multi-node architecture. Specific approaches for load balancing were also explored, including the use of HA Proxy and Keepalived.

Finally, the application of machine learning techniques was examined, with a focus on the use of the SVM algorithm for load balancing. The steps of classifying requests, clustering virtual machines and resources, and allocating services were explored. Comparisons between traditional and SVM-based load balancing methods were also conducted, with the SVM algorithm being implemented and simulated in a test environment.

In conclusion, this research contributed to a better understanding of the challenges and solutions in the area of critical infrastructure and SLAs, with a special focus on load balancing using SVM machine learning algorithm.

Bibliography

[1] S. Rangero, *5g: What it means for iot*, https://www.zdnet.com/article/what-is-the-internet-of-things-everything-you-need-to-know-about-the-iot-right-now, 2020.

[2] A. Akbari-Moghanjoughi, J. R. de Almeida Amazonas, G. Santos-Boada, and J. Solé-Pareta, *Service Level Agreements for Communication Networks: A Survey*. Sep. 2023, arXiv:2309.07272, arXiv:2309.07272. doi: 10.48550/arXiv.2309.07272. arXiv: 2309.07272 [cs.NI].

[3] *What is a Service Level Agreement?*, https://getvoip.com/blog/service-level-agreement/, Accessed: 23/01/2024.

[4] E. Marilly, O. Martinot, S. Betge-Brezetz, and G. Delegue, "Requirements for service level agreement management", in *IEEE Workshop on IP Operations and Management*, 2002, pp. 57–62. doi: 10.1109/IPOM.2002.1045756.

[5] *Inkwell Data*, https://inkwelldata.com/, Accessed: 27/01/2024.

[6] *RabbitMQ*, https://www.rabbitmq.com/, Accessed: 18/02/2024.

[7] P. A. Rahman and E. Y. Bobkova, "The reliability model of the fault-tolerant computing system with triple-modular redundancy based on the independent nodes", *Journal of Physics: Conference Series*, vol. 803, no. 1, p. 012 125, 2017. doi: 10.1088/1742-6596/803/1/012125. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/803/1/012125.

[8] *Implementation and evaluation of a container-based software architecture*, https://hdl.handle.net/20.500.12608/24158, Accessed: 25/02/2024.

[9] *Container and containerization*, https://www.serverwatch.com/guides/what-is-container-and-containerization/, Accessed: 03/03/2024.

[10] A. Alakeel, "A guide to dynamic load balancing in distributed computer systems", *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 10, Nov. 2009.

[11] *HA Proxy*, https://www.haproxy.org/, Accessed: 06/03/2024.

[12] *HA Proxy configuration*, https://reintech.io/blog/configuring-load-balancer-haproxy-ubuntu/, Accessed: 08/03/2024.

[13] *How to create a Dockerfile*, https://www.buildpiper.io/blogs/how-to-create-a-dockerfile/, Accessed: 24/02/2024.

[14] O. Tannous and L. Xing, "Efficient analysis of warm standby systems using central limit theorem", in *2012 Proceedings Annual Reliability and Maintainability Symposium*, 2012, pp. 1–6. doi: 10.1109/RAMS.2012.6175433.

[15] *Keepalived*, https://www.keepalived.org/, Accessed: 12/03/2024.

[16] *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 7, 2022, issn: 1319-1578.

[17] *Ibm cloud education, "what is machine learning?," ibm cloud, 15 july 2020.* https://www.ibm.com/topics/machine-learning, Accessed: 17/03/2024.

[18] *Ibm cloud education, "what is deep learning?," ibm cloud, 1 may 2020.* https:/https://www.ibm.com/topics/deep-learning, Accessed: 17/03/2024.

[19] H. Le, T. Huyen, N. Phi, and C. Tran, "Mccva: A new approach using svm and kmeans for load balancing on cloud", *International Journal on Cloud Computing: Services and Architecture*, vol. 10, Jun. 2020. doi: 10.5121/ijccsa.2020.10301.

Acknowledgements

I would like to sincerely thank everyone who has contributed to the achievement of this important milestone in my life. I know that I am an ambitious person and this success is only the beginning of a new journey.

First of all, I want to thank Professor Tomasin for his invaluable guidance during the writing of my thesis and for his patience with corrections.

All my gratitude goes to my parents, Luigi and Antonella, for their love and support at every single moment. I have never shown enough how grateful I am to have two such amazing parents.

To my grandparents, who with their unconditional love and knowledge inspired me to chase my dreams.

To all my family, who have always believed in me and offered me valuable advice along the way.

The company Inkwell Data for giving me the opportunity to grow in a stimulating and positive working environment, and to my colleagues who have contributed to my professional development.

To my friends Denz, Fryghesh, Benny, Zarro, Tommy, Sufy, Pola, Sugo and Stion, who were always by my side and made every moment special.

To my university friends for all the advice and moments together over the years.

Finally, a special dedication goes to Venada, for being my source of inspiration and support at all times. Thank you for always being by my side, for your precious advice and for your infinite sweetness.

To all of you, thank you from the bottom of my heart for making this important achievement in my life possible.

– Nicola Greggio