An Overview of High Availability: From Theory to Practice

Nicholas Miazzo
Department of Math
University of Padua
Via Trieste, 63, Padua, Italy
nicholas.miazzo@studenti.unipd.it

***Index Terms*—ha, docker, swarm**

I. Introduction

In recent years, companies have implemented various technologies into their business processes. These implementations have resulted in clear benefits for both companies and their customers who, through internet-connected smartphones, can access services from anywhere in the world. However, the pre-digitization era's processes and services are increasingly being discontinued in favor of the new technology-based ones, leaving no choice on what to rely on. In many areas, the availability of digital services has become a critical concern, with little room for error or interruption. This report aims to explore High Availability. We propose an initial theoretical study of its main concepts. However, due to the broad nature of the topic, a purely theoretical study risks getting lost in the vast content without providing practical experiments. Instead, we will implement proofs of concept and analyze the results to understand what may be missing before proceeding to the next experiment.

II. Understanding High Availability

Reference [1] gave us an excellent starting point for understanding the concept of High Availability. High Availability is a characteristic of a system that enables continuous operation over some time without interruption. Availability is usually represented as a percentage that shows the amount of time a system is operational within a given timeframe, typically a year. To qualify for High Availability, the system must ensure an uptime of 99.999&, meaning that there should be no downtime exceeding 5.26 minutes over that period. Accordingly, Availability can be defined using the following formula.

$$Availability = \frac{uptime}{uptime + downtime} \tag{1}$$

To fully understand (1), it is necessary to define the following indicators:

- **MTTD** (Mean Time To Detection) denotes the time taken for a failure to be detected;
- **MTTR** (Mean Time To Repair) denotes the time taken between identifying a failure and its repair;
- **MTBF** (Mean Time Between Failure): which indicates the length of time the system operates, i.e. the duration between two instances of failure.

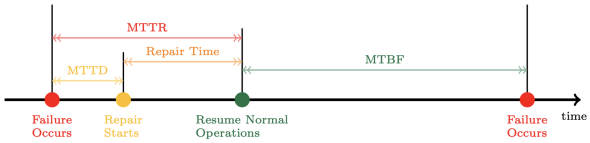

Fig. 1. MTTD, MTTR, MTBF in a timeline [1].

Using these indicators, we can reformulate (1) as:

$$A = \frac{MTBF}{MTBF + MTTR} \tag{2}$$

From this, we can see that a high MTBF, a low MTTD, and a low MTTR contribute to increased system availability.

However, we have to consider that the systems whose availability we want to study are composed of several subsystems, each with its availability. Consequently, the overall availability of the system can be calculated using:

$$A = a_1 * a_2 \cdots * a_n \tag{3}$$

Where a_n is the availability of sub-system n.

III. A peek at service replication

Imagine a post office with only one employee. They guarantee to serve customers at the counter during office hours. However, there is a small probability p that the employee may need to take a break, leaving the counter unmanned. As a result, customers will find the office empty, walk away, and leave a negative review on Google. On average, it can be assumed that the employee will work for 98% of their 8-hour shift, taking around 10 minutes of break time. Furthermore, the postal company cannot prevent the employee from taking a break. In this scenario, it is clear that the post office needs to hire another employee if they want to avoid bad reviews. In this scenario, the post office will only be uncovered if both employees take their breaks simultaneously. Therefore, the overall availability of the post office can be calculated using the following formula.

$$A = 1 - (1 - a)^2 \tag{4}$$

$A = 1 - (1 - 0.98)^2 = 99.96\%$ which means 0.192 minutes of their 8-hour shift. The duration of disruption has been significantly decreased. Let us assume that the postal service recruits a third employee. Doing the maths, with 3 people, the office will be uncovered for 0.00384 minutes for the 8-hour shift. Though it is objectively good news for the office to have further reduced the downtime, in a broader perspective it has to pay a third salary for reducing the downtime by 0.18816 minutes.

Going back to our central subject, we can apply the same concept to High Availability. If a sub-system fails to provide sufficient High Availability, we can create n independent replicas of the component. Moreover, it may not be trivial to estimate the number of replicas required when the availability of a single sub-system is unknown. Using a low number of replicas does not ensure High Availability while having a high number of them is not justifiable in terms of maintenance costs.

Considering (2), replications can be considered an effective approach for reducing MTTR. Once identified as unhealthy, replicas will be removed from the pool of available servers to receive requests, effectively preventing failures.

IV. Software failure

Reference [1] and [2], software failures that impact availability can be categorized into two groups: Bohbug and Heisenbug. The former refers to bugs that arise every time the same input is provided.

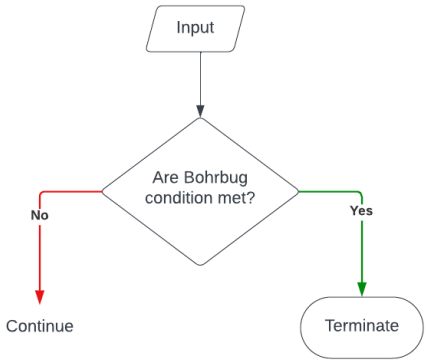

Fig. 2. Bohrbug flow.

The latter occurs only under specific, uncommon conditions, such as compiler errors or race conditions. The pre-conditions required for the bug are unknown, making it challenging to reproduce the bug in a monitored environment. Most software errors fall into this category.

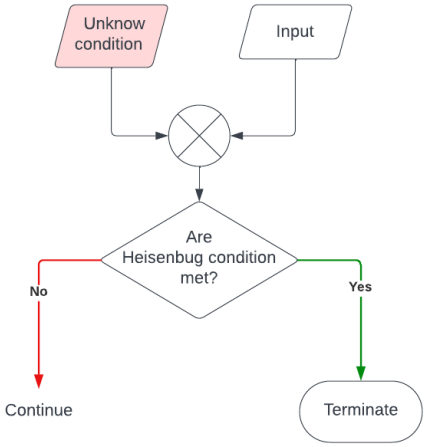

Fig. 3. Heisenbug flow.

Fixing bugs in software can increase the MTBF which, in turn, increases system availability. Detection of bugs before they enter the production environment can be facilitated by tools such as unit testing, load testing, peer review, and automated verification software.

V. Estimating uptime

An empirical approach may be preferable to an analytical study of a system's availability due to potential complexity and lack of reliability in theoretical outcomes. By utilizing suitable software, canary requests can be sent to the system for evaluation of response success in terms of latency, errors, etc. These requests can be simulated to come from outside the system, mimicking client service usage. However, it is essential to consider that a specific service may have dependencies. To evaluate the condition of the entire system and all its subsystems, a deep health check approach is used, while a shallow health check approach inspects the state of the individual service. Once a sufficient number of requests have been gathered over a certain period, availability can be determined as:

$$A = \frac{Successfully\ requests\ counter}{Total\ requests\ counter} \tag{5}$$

VI. Proof of concept introduction

For this report, we adopted a 'learn by doing' approach, implementing the theoretical notions through the proof of concept and questioning the results to find out how to further explore the topic of High Availability. We implemented each proof of concept to be reproducible, using containers and virtual machines, and providing scripts to simulate service interruptions. This methodology allows readers to replicate the experiments conducted on their devices.

VII. Replicas and Load balancer

In Section III, we discovered the importance of replication in increasing the availability of a system. In practice, we

realized that n replicas need to be managed in some way. This duty is assigned to a system component called the load balancer. This component handles requests from clients and loads distribution to replicas while keeping the existence of the replicas hidden from the client.

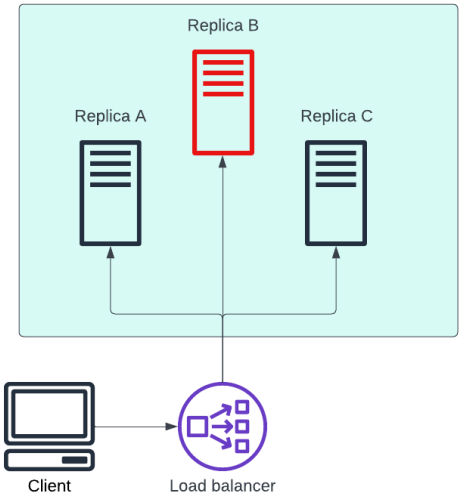

Fig. 4. Replicas behind a load balancer.

A. Load balancing algorithm

The selection of the server to which the client's request will be forwarded is determined by the algorithm chosen during configuration. We will describe six of them:

1) Round Robin: requests are distributed through a circular queue of servers, leading to a very fast routing decision. However, the load of the server to which the request is forwarded is not taken into account;

2) Weighted Round Robin: similar to the previous one, each server is assigned a weight indicating how many requests it can receive before the load balance selects the next server. It can be utilized in situations where various servers possess differing load capacities;

3) Least connections: requests are routed to the server with the fewest active connections. This method is inadequate when servers have varying load capacities, and a weighted-oriented solution is more appropriate;

4) Source IP hash: clients are assigned to servers based on their IP addresses, ensuring that a particular client will consistently connect to the same server. However, this approach may not ensure even load distribution;

5) Least Response Time: requests are forwarded to the server with less response time;

6) Random: clients are randomly assigned to a server upon each request.

B. Proof of concept

In this first proof of concept, we will try to create replicas of a simple web service. The service returns a web page showing the number of times the user has used the service and, for educational purposes, the host that handled the request. Since it is not the responsibility of the client to handle the routing of the request to the replicas, a load balancer is placed to accept incoming requests by routing them through a round-robin algorithm. We intentionally made the latter component unable to check whether the replica is online before forwarding the request, so that we can reason about the effectiveness of replicas under these conditions.

During the execution of this experiment, the following container images will be used:

- **nginx:1.25.2**: reverse proxy and load balancer;
- **php:8.2-apache**: PHP programming language and Apache web server for web service delivery;
- **locustio/locust:2.16.1**: load testing tool.

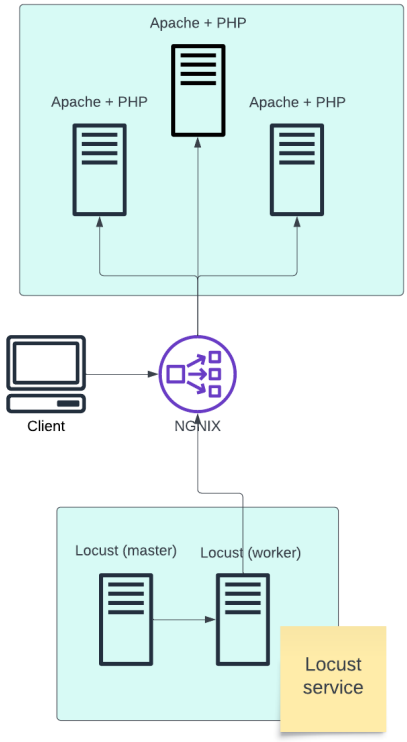

Fig. 5. PoC 1 system workflow.

This experiment can be replicated by following these steps:

1) Execute `1.sh`. It creates three replicas of the web service, and the load balancer, and prepares the load test. The web service can be accessed at `http://localhost`;
2) Start the load test using the web interface at `http://localhost:9098`;
3) Execute `2.sh`. This script will stop one of the three replicas.

C. Observations

The initial observation when using this service is that the visit counter is malfunctioning. This is because the application

retains the value based on an internal state that isn't synchronized with the other replicas. Moreover, it is noticeable that the counter is accurate when viewed in context with the server that processed the request (displayed on the web page). Subsequently, by utilizing the load test, we can analyze the number of requests correctly handled by the replicas, as reported by 6. We can note the exact time when a replica was stopped. Clients had to wait for a long timeout before they could try again, and 33% of subsequent requests failed because they were redirected to the unavailable server. From this initial insight, assuming that sooner or later a replica will fail, the load balancer must be able to remove failed replicas from its routing algorithm. Although it may seem trivial, this is a necessary condition for implementing 4. Let us assume that each replica has a 99% availability. According to the equation, our system should have a 99.99% availability. However, the continued use of a failed replica resulted in improper handling of client requests 33% of the time.

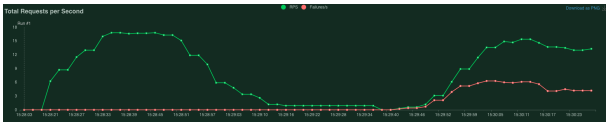

Fig. 6. Request during PoC 1 execution. Green for successful request. Red for failed requests

VIII. Sticky sessions

This section aims to answer the first problem that arose in the previous proof of concept. Having replicas with their internal state could cause inconsistencies in the use of the service, so we need to handle it somehow. From our research, we found methods to handle those situations:

- Move the state out of the service, decoupling the service logic from managing the state between replicas. For example, store user session data in a database that is accessible to all replicas;
- Create stateless services and leave state management to the client;
- Always redirect subsequent requests from a particular client to the previous server. This particular solution could result in an unbalanced load distribution, depending on the number of consecutive client requests and their distribution.

The final technique is referred to as a sticky session. During the first client request, the load balancer routes it according to its routing algorithm (e.g. round-robin). All subsequent requests from the client are automatically forwarded to the server that processed the first request. Cookies, headers, or the IP address of the client can maintain this connection between the client and the replication server.

A. Proof of concept

This proof of concept will implement a sticky session between clients and replicas of a certain service. In addition, we adopted Traefik as a reverse proxy and load balancer. By default, Traefik listens for Docker events, for this reason, it will automatically remove stoppered containers from the server pool.

During the execution of this experiment, the following container images will be used:

- **traefik:v2.10**: reverse proxy and load balancer;
- **php:8.2-apache**: PHP programming language and Apache webserver for web service delivery;

This experiment can be replicated by following these steps:

1) Execute `1.sh`. It creates three replicas of the web service, the load balancer;
2) Go to `http://localhost` and reload the page several times to update the counter;

B. Observations

In this implementation of the sticky session, a persistent connection was established between the client and the replica through a cookie. Our findings suggest that when the server assigned to the user becomes unavailable, the sticky session becomes invalid. In this instance, a different server will be assigned based on the load-balancing algorithm. While this may be a desirable outcome, it is crucial to consider the possible implications for clients, including the loss of their login session. However, we wonder how we could have handled the replication of a service whose state plays a key role, such as a database. We aim to create a database replica in our future experiments.

IX. Updating services

Our study aims to understand how a service can guarantee High Availability. However, it is crucial to update, enhance, and maintain these services before a failure occurs. So, the question now is how to release the new version and retire the old one without the user noticing any disruption.

A. Proof of concept

In this proof of concept, we will deploy a service update and retire the old version. The load balancer is set up to prioritize the replicas of the new service over those of the old version. Consequently, the existing requests to the old services will be completed and we will be sure that they will not receive any new requests. Then we can disable the old service. However, it might be interesting to keep the old version around for a while. In the event of a failure of the updated version, the system can quickly revert to the previous version since it is still operational.

In our proof of concept, a service delivers a web page. However, after deployment, the developers noticed that due to a bug, the response took 5 seconds to be sent to the clients. While the new version has been developed, some clients are already using the service. It is required to let the clients finish their request properly before shutting down the old version, but considering that there won't be a free time window when the service won't be used by new clients.

During the execution of this experiment, the following container images will be used:

- **traefik:v2.10**: reverse proxy and load balancer;
- **php:8.2-apache**: PHP programming language and Apache webserver for web service delivery;
- **locustio/locust:2.16.1**: load testing tool.

This experiment can be replicated by following these steps:

1) Execute `1.sh`. It creates three replicas of the web service V1, the load balancer, and prepares the load test. The web service can be accessed at `http://localhost`. As previously mentioned, the browser must wait 5 seconds to receive the response;
2) Start the load test using the web interface at `http://localhost:9098`;
3) Execute `2.sh`. This script will deploy three replicas of web service V2;
4) Wait at least five seconds for clients of the old service to receive their response from the server;
5) Execute `3.sh`. This script will stop all replicas of web service V1.

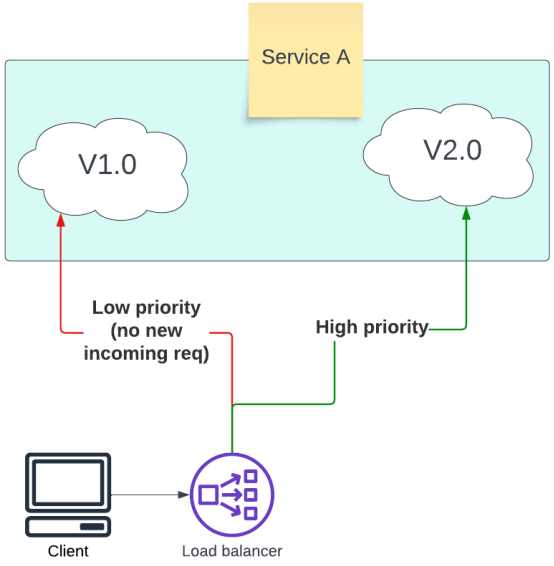

Fig. 7. Workflow of PoC 3 (locust omitted)

B. Observations

Figure 8 illustrates the results of a load test with 500 connected users, each with a request time varying between 1 and 5 seconds. The upper graph shows that no request failed even though the old service was shut down after version 2 was deployed. The lower graph displays the average response time from the server. It is easy to spot the point at which the new version of the service started receiving requests, replacing version 1.

Fig. 8. Request during PoC 3 execution. Green for successful request. Red for failed requests

X. Healthy services & Simple monitoring

In the preceding section, we deactivated a service without triggering any client outages. As we performed these operations on our initiative, we were able to control how and when the servers were turned on and off. The objective of High Availability is to ensure that service remains uninterrupted even if server failures occur under circumstances beyond our control, such as a bug that makes a replica unusable.

A. Proof of concept

In this proof of concept, the system offers three duplicated web servers. Meanwhile, the load balancer checks the health of each server every second. A request will only be considered successful by the load balancer if the response arrives within 5 seconds and if the HTTP code is in class 2XX or 3XX. However, we wondered what should happen if the user encounters an unhealthy server before it is detected by the load balancer. The initial assumption would be to retry the request on another server without disrupting the user's experience. Alternatively, we may inform the client of the failure by sending an HTTP 503 code. The former approach attempts to mask the failure from the user, even though it happened. On the other hand, the second approach leaves the choice up to the client, who can retry the request immediately. Based on our research, users tend to prefer the latter option, particularly in instances where non-idempotent operations may have altered the system state before the error occurred. Finally, the Traefik dashboard provides a basic graphical representation of the service status, although it is limited in scope. More advanced monitoring systems will be explored in the next proofs of concept studies.

During the execution of this experiment, the following container images will be used:

- **traefik:v2.10**: reverse proxy and load balancer;
- **php:8.2-apache**: PHP programming language and Apache webserver for web service delivery;
- **locustio/locust:2.16.1**: load testing tool.

This experiment can be replicated by following these steps:

1) Execute `1.sh`. It creates three replicas of the web service, and the load balancer, and prepares the load test. The web service can be accessed at `http://localhost`;

2) Start the load test using the web interface at `http://localhost:9098`;
3) Execute `2.sh`. This script will make the second replica unavailable;
4) Execute `3.sh`. This script will restore the previous replica to a functioning state.

B. Observations

During the experiments, the load test demonstrated that a total of 3 to 8 clients experienced request errors. The reason for this is that the health check runs every second, and the load test involved 50 users submitting requests at intervals ranging from 1 to 5 seconds. Remember that we are not aiming for 100% availability, so depending on the requirements, this situation may be acceptable. Figure 9 shows the Traefik dashboard after a service health check.

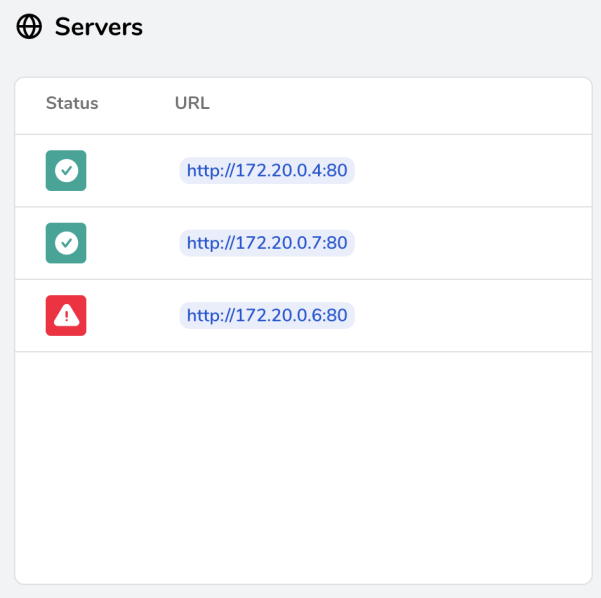

Fig. 9. Replicas monitor in Traefik dashboard.

XI. Data replication

In prior proofs of concept, we exclusively examined identical replicas with an internal state that could be lost if server failure occurred. However, in the context of persistent data systems such as databases, we cannot tolerate any loss of information. To introduce database replication, we researched different methodologies. The first method, instance-level clustering, permits database replicas to read and write to a shared memory, such as a network drive.

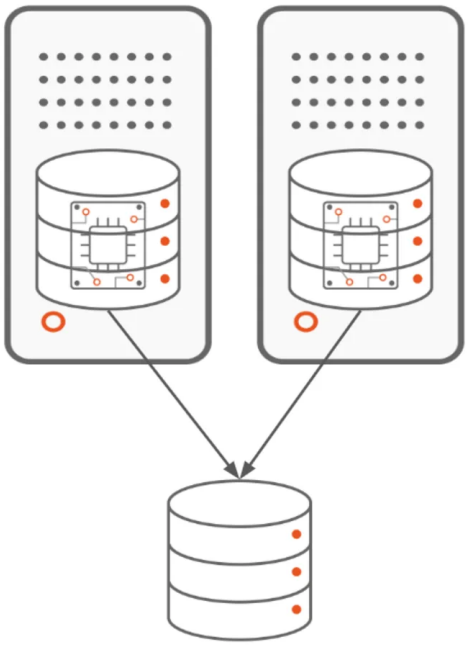

Fig. 10. Instance-level clustering. Image by https://ubuntu.com/blog/database-high-availability

The second approach requires replicas to exchange data to update their data in their local memory, which is called database-level clustering.

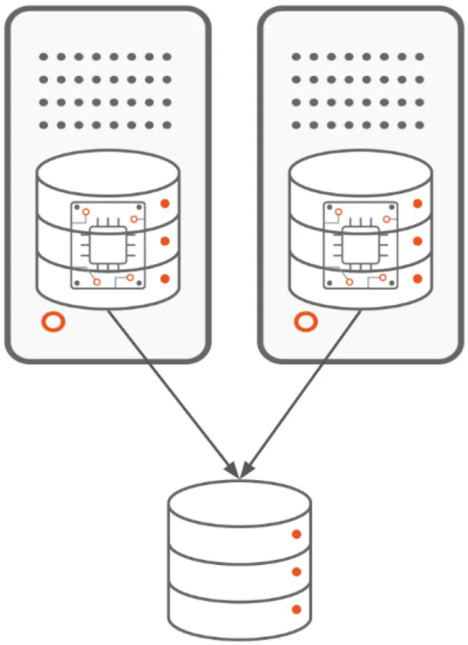

Fig. 11. Database-level clustering. Image by https://ubuntu.com/blog/database-high-availability

A. Proof of concept

For this proof of concept, we replicated a database using database-level clustering and created a web page to show the contents of the instances. During our experiment, we discov-

ered that the master database serves as the main database, and the replicas waiting for data from the master are referred to as slaves. Write operations can only be received by a slave database directly from the master. However, if the master node experiences a failure, the slave can be selected as the new master.

During the execution of this experiment, the following container images will be used:

- **traefik:v2.10**: reverse proxy and load balancer;
- **php:8.2-apache**: PHP programming language and Apache webserver for web service delivery;
- **bitnami/postgresql:15.4.0**: PostgreSQL database.

This experiment can be replicated by following these steps:

1) Execute `1.sh`. It creates three replicas of the web service, the load balancer, and two database replicas (master/slave). The web service can be accessed at `http://localhost`;
2) Visit `http://localhost` to write some data into the master database;
3) Execute `2.sh`. Stops the master database and elects the slave node to be the new master node;
4) Visit again `http://localhost` to write some data into the new master database.

B. Observations

During this experiment, we were able to create a simple replica of the database. However, by electing the slave as the new master, we put the old master node in an inconsistent state. If the latter node is recovered, it will no longer be synchronized with the new master. Therefore, depending on the cases and requirements, the system may allow the database to remain in a read-only state until the master node is recovered.

XII. Summing up

In this section, we will create a single system that implements many of the aspects discussed in the previous sections. Also, so far we have not been concerned with having all containers running on a single physical node. In this section, we want to implement a container-based system that is distributed across multiple nodes and orchestrated by Docker Swarm. We also want to implement a sophisticated system for managing and monitoring the infrastructure.

A. Proof of concept

To simulate a multi-node cluster, we will create three virtual machines using VirtualBox. To automate the creation of virtual machines, the Vagrant tool must be installed in the execution environment. We will create three nodes: node1 as the cluster manager, and node2 and node3 as cluster workers. Portainer is installed to facilitate the management of nodes and services via a web GUI, and Grafana is used for monitoring.

In terms of application, two services are offered. The first, aimed at patients of a health service, allows users to view their medical information and send messages to doctors. The second service, aimed at doctors, allows users to view information on all patients and messages received. The first service, which is considered important but not essential, is replicated in only one node, while the second service is replicated in two nodes.

During the execution of this experiment, the following container images will be used, and distributed in a Docker Swarm cluster:

- **traefik:v2.10**: reverse proxy and load balancer;
- **php:8.2-apache**: PHP programming language and Apache webserver for web service delivery;
- **bitnami/postgresql:15.4.0**: PostgreSQL database;
- **portainer/portainer-ce:2.19.1**: The Portainer master node to manager active services;
- **portainer/agent:2.19.1**: the Portainer agent to provide information to the main node.
- **grafana/grafana:10.1.2**: a web application to monitor the services status.
- **prom/prometheus:v2.47.0**: the database to receive metrics from Traefik.

B. Observations

Through our proof of concept, we have realized the complexity of distributing the workload across multiple nodes and the importance of having a service management system. It constitutes the final concept in this report, however, our investigation has raised many new questions on the subject of High Availability, which we will answer in future investigations.

XIII. Future work

Further investigation will prioritize the comprehension of systems that recover failed nodes, as well as replicating the load balancer, which currently represents the single point of failure. This case may be resolved through cloud services, such as AWS, which provide High Availability solutions as a service.

XIV. Conclusions

In this report, we investigated the concept of High Availability. Firstly, we undertook a theoretical study before moving on to practical applications to acquire new insights on where and how to proceed. We only recognized the enormity of the subject during the writing of this report, as well as the importance of providing solutions that can guarantee availability.

References

[1] Availability and Beyond: Understanding and Improving the Resilience of Distributed Systems on AWS.

[2] Trivedi, K; Ciard, G; Dasarathy, B; Grottke, M; Matias, R; Rindos, A; Vashaw, B. Achieving and assuring High Availability.