An Overview of High Availability: From Theory to Practice

Nicholas Miazzo
Department of Mathematics
University of Padova, Italy
nicholas.miazzo@studenti.unipd.it

***Index Terms*—availability, ha, replication**

I. Introduction

In recent years, more and more companies have implemented various technologies into their business processes, optimizing and automating their workflows, and innovating how they deliver their services to customers. An example of this innovation is the business model introduced by Deliveroo, in which the interactions between customers, riders, and restaurants are entirely orchestrated using just applications on mobile devices.

However, the increase in responsibilities assigned to IT systems highlights the critical need to maintain their operation. Service disruptions can produce considerable financial losses for companies, but also image damage that could compromise consumer trust. In addition, system outages in critical areas can result in extreme consequences, even loss of human life.

This report explores the concept of availability as a metric for evaluating a system's ability to deliver a given application service level to customers. We also discuss the concept of high availability as an advanced standard that ensures near-continuous operation with minimal downtime [5]. To address the complexity of the topic, we decided to define a set of key questions (KQ) to guide our survey:

- **KQ1**: What are the possible causes of a system outage?
- **kQ2**: What do resilience and availability mean?
- **KQ3**: What is the meaning of High Availability? Who determines the guaranteed availability level of a specific service?
- **KQ4**: What is service replication and how does it improve the availability of the replicated service?
- **KQ5**: How to estimate the uptime of a service?

However, due to the broad nature of the topic, a purely theoretical study risks getting lost in the vast content without providing practical experiments. For this reason, we implemented some theoretical aspects into practical experiments, called Proofs of Concept (PoC). After finishing each proof of concept, we analyzed the results and asked ourselves what was missing to define the next one. We produced a total of five proofs of concept:

- **PoC1**: How can load balancing be implemented?
- **PoC2**: How can updates to services currently used by clients be deployed?
- **PoC3**: How to handle failed replicas?
- **PoC4**: How to manage the replication of data persistence services?
- **PoC5**: How can the workload be distributed across multiple nodes?

II. What are the possible causes of a system outage?

According to [1], real-world systems consist of multiple modules, each with an internal structure of sub-modules. Instead, we can view an entire system as a single module and apply our reasoning recursively to the internal modules and sub-modules. A module is designed to have a certain expected behavior, but at certain periods this behavior may be different from the actual observed behavior. When the expected behavior differs from the observed behavior, it implies that an error has become effective and caused a failure.

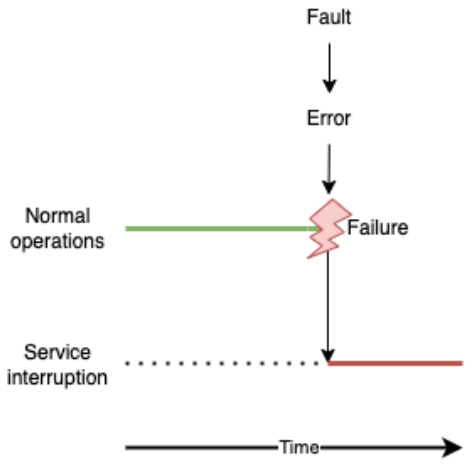

Fig. 1. Fault causing an error and making the module fail.

The cause of an error is called a fault. Four categories can be defined to identify faults [1] [2]:

- **Hardware faults**: issues within physical components, such as disk failures;
- **Design faults**: flaws at the design level of software or hardware;
- **Operations faults**: errors made by maintenance staff (e.g. network misconfiguration);

- **Environmental faults**: natural disasters (e.g., fires, earthquakes), war attacks, power or network failures.

A. Software failure types

Based on [3] and [4], software failures, and software failures that impact availability can be grouped into two groups: Bohbug and Heisenbug.

1) Bohbug: issues that arise consistently and predictably under specific conditions.

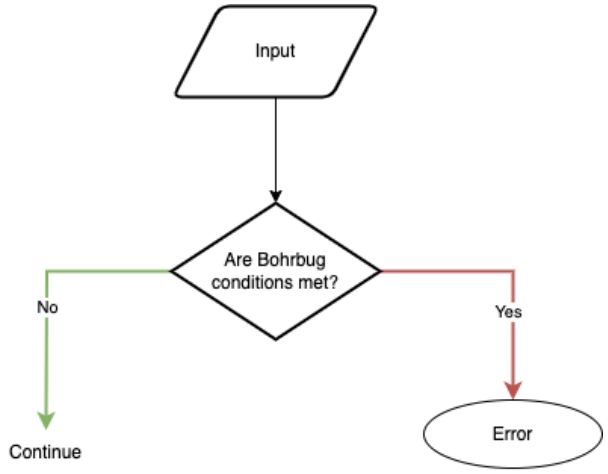

Fig. 2. High-level Bohrbug flow.

2) Heisenbug: issues that occur only under specific but uncommon, such as compiler optimizations error or race conditions. The pre-conditions required for this type of bug to appear are unknown, making it challenging to reproduce the bug in a monitored environment. Most software failures fall into this category.

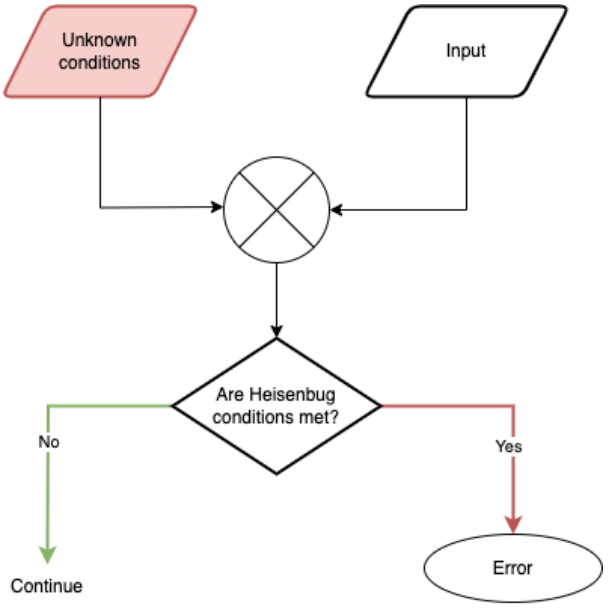

Fig. 3. High-level Heisenbug flow.

III. What do resilience and availability mean?

References [5] and [3] define system reliability as the probability of the system performing the intended task continuously over a defined period. For instance, let's consider a system that ensures 99% reliability in one year. This means there is a 99% probability that the system will operate without outages throughout the year.

On the other hand, availability refers to the percentage of time that a system is operational within a given time frame. Availability takes into account the actual time of functionality, called uptime, and the time required for fault detection and recovery, called downtime. With these concepts in mind, the availability of a system can be calculated using the following equation:

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

However, it is possible to interpret availability differently than in Equation (1) by introducing the following terms:

- **MTBF** (Mean Time Between Failure): the average duration of continuous system operation. It represents the average time between the end of one outage recovery and the start of the next outage.
- **MTTD** (Mean Time To Detection): the mean time taken for a failure to be detected;
- **MTTR** (Mean Time To Repair): the mean time taken between identifying a failure and its recovery.

The figure below provides a graphical representation of the previously introduced terms on a time axis:

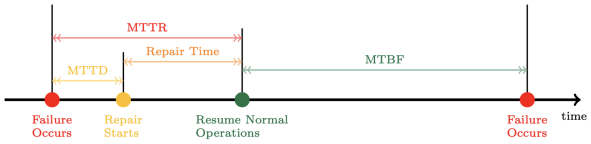

Fig. 4. MTTD, MTTR, MTBF in a timeline. Source: [3].

Equation (1) can be rewritten using these definitions as:

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

Therefore, to increase the availability of a given system, we should aim for a high MTBF and a minimum MTTR. One way to increase MTBF is by fixing software bugs. Tools such as unit testing, load testing, peer review, and automated verification software can facilitate bug detection before they enter the production environment.

A. Opening the black box

So far, we considered the study of availability on systems as a single, atomic module. However, a real-world system is composed of multiple modules that have functional dependencies on each other. If we consider a system made of n modules, each with a_i $i \in \{1 \dots n\}$ availability, we can define the overall system availability [3] as:

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

IV. What is the meaning of High Availability? Who determines the guaranteed availability level of a specific service?

Referencing to [1], availability can be classified into several classes:

System type	Unavailability (minutes/year)	Availability (%)	Availability class
Unmanaged	50.000	90	1
Managed	5.000	99	2
Well-managed	500	99.9	3
Fault-tolerant	50	99.99	4
High-availability	5	99.999	5
Very-high-availability	0.5	99.9999	6
Ultra-availability	0.05	99.99999	7

TABLE I
AVAILABILITY CLASSES

This means that a system is considered high availability if it achieves 99.999% availability. For instance, if we consider the time frame of one year, it is acceptable for a high availability system to experience 5 minutes of cumulative downtime.

A. Uncovering the Service Level Agreement

The Service Level Agreement (SLA) is a contract between a service provider and customers that establishes the quality standards that the provider agrees to guarantee. The agreement also indicates the penalties the provider faces if it fails to meet the agreed standards[1]. One of the metrics of standards that the document can report is availability. We examined the service level agreements of various services to determine how guaranteed availability would be in a real-world scenario:

Service name	Free/Paid	SLA Availability (%)
AWS DynamoDB (global tables)	Paid	99.999
AWS Key Management Service	Paid	99.999
AWS EC2 (region level)	Paid	99.99
AWS S3	Paid	99.99
DigitalOcean Droplet	Paid	99.99
Google AlloyDB	Paid	99.99
Google Workspace	Paid	99.99
Slack	Paid	99.99
Atlassian Enterprise plan	Paid	99.95
Firebase	Paid	99.95
Atlassian Premium plan	Paid	99.9
AWS SimpleDB	Paid	99.9
GitHub.com	Free/Paid	99.9
OneDrive for Business	Paid	99.9
AWS EC2 (instance level)	Paid	99.5
GitLab.com	Free/Paid	99.5
WordPress.com	Free/Paid	**No**
Shopify	Paid	**No**
Telegram	Free/Paid	**No**

TABLE II
AVAILABILITY OF REAL-WORLD SERVICES

Only two of the analyzed services aim to offer high availability. Due to a lack of experience, we had assumed that many of these services were aiming for high availability. Additionally, some commercial services, such as WordPress.com and Shopify, only offer the best effort to ensure service availability without stating a precise minimum value.

[1]https://en.wikipedia.org/wiki/Service-level_agreement

V. What is service replication and how does it improve the availability of the replicated service?

Imagine a post office with only one employee. During the post office opening hours, the employee needs to go to the bathroom on average every 57 minutes. These breaks have an average duration of 3 minutes. Assuming an 8-hour workday, we can estimate that the employee is operational for 456 minutes and spends 24 minutes on breaks. To calculate post office availability, we can use equation (1):

$$Availability = \frac{uptime}{uptime + downtime} = \frac{456}{456 + 24} = 0.95 \tag{4}$$

This means that the post office has an availability of 95%. The postal company is concerned that customers will come during one of the employee's breaks and, finding the post office unattended, walk away leaving a bad review on Google. In this scenario, it is clear that the post office needs to hire a second employee. Considering that the second employee will also take, on average, a 3-minute break every 57 minutes, how has the availability of the post office changed?

Considering a represents the availability of the individual system and n represents the number of system replicas, the equation used to calculate the total availability of a redundant system [3] is:

$$A = 1 - (1 - a)^n \tag{5}$$

Applying the equation to the post office system with two employees:

$$A = 1 - (1 - 0.95)^2 = 0.9975 \tag{6}$$

The total office availability is 99.75%. During an 8-hour workday, the post office will be unattended for a total of 1.2 (cumulative) minutes.

There are clear benefits to hiring a second employee, but how would availability and downtime be affected by hiring additional employees? To effectively answer this question, we calculated availability and downtime values for one to five employees. These values are shown in the following table and Figures 5 and 6:

Employees	Availability (%)	Cumulative downtime (minutes/8 hours)
1	95	24
2	99.75	1.2
3	99.9875	0.06
4	99.999375	0.003
5	99.99996875	0.00015

TABLE III
POST OFFICE AVAILABILITY AND DAILY CUMULATIVE DOWNTIME.

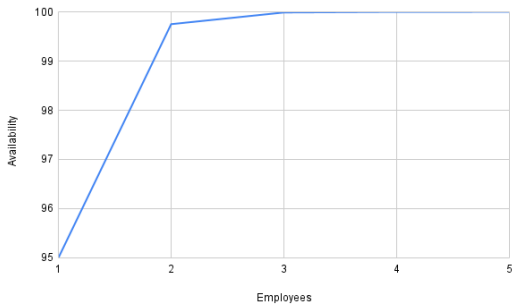

Fig. 5. Post office availability over employees number.

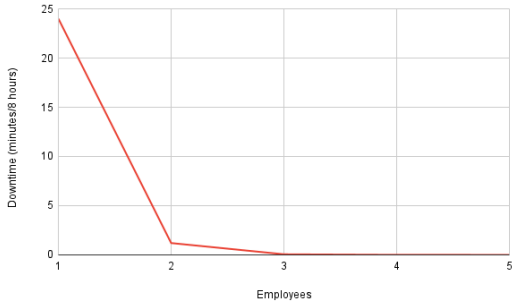

Fig. 6. Daily cumulative downtime over employees number.

When analyzing these values, the benefit of hiring a second employee becomes even clearer. However, it is important to keep in account that each employee receives a salary. While hiring three or more employees may be ideal for reducing downtime, this choice comes with a cost that is not justified by the benefits for the post office.

Returning to the concept of high availability, we can apply the same reasoning to a system that cannot guarantee the required availability for the application domain. If a system is unable to provide sufficient availability, n independent system replicas can be created. Estimating the number of replicas required when the availability of the system is unknown is also not trivial. Using a small number of replicas does not guarantee sufficient availability while having a large number of replicas is not sustainable in terms of maintenance costs.

VI. How to estimate the uptime of a service?

As suggested by [3], an empirical approach may be preferable to an analytical study of a system's availability due to potential complexity and lack of confidence in theoretical outcomes. One methodology for collecting information is to use analytic tools installed on the service to automatically collect information from incoming client requests. However, if the client cannot access the service, the metrics will not be collected, making it appear as if the service is not being used instead of an outage. In such cases, a service with constant and frequent client usage can infer a failure if no requests are logged within a given timeout. An alternative method is to simulate client requests using software called canaries, which record and report various metrics and errors. Examples of such software include Puppeteer[2] and Selenium[3]. The final service availability can be estimated using the following formula after recording a sufficient number of requests within a defined time frame based on applied quality metrics:

$$A = \frac{Successfully\ requests\ counter}{Total\ requests\ counter} \tag{7}$$

VII. Proof of concept introduction

The following sections describe our practical experiments, the information we gathered, and related observations. Each proof of concept was designed to be reproducible using containers and virtual machines. This methodology enables readers to replicate the experiments on their own devices.

VIII. PoC 1: How can load balancing be implemented?

In Section V, 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 distributes them to the replicas.

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. Load balancing algorithms can be either dynamic or static. Dynamic algorithms take performance metrics into account to distribute the load, while static algorithms are based on rules that do not consider the state of replicas. We report seven of them[4].

- **Static load balancing algorithms**:
  - **Round Robin**: requests are distributed through a circular queue of replicas, leading to a very fast routing decision. However, the load of the server to which the request is forwarded is not taken into account;
  - **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 have different load capacities;
  - **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.
- **Dynamic load balancing algorithms**:
  - **Least connections**: requests are routed to the server with the fewest active connections;

[2]https://pptr.dev/
[3]https://www.selenium.dev/
[4]https://www.cloudflare.com/it-it/learning/performance/types-of-load-balancing-algorithms/

- **Weighted least connection**: similar to the previous version but this algorithm also allows for prioritization of replicas. This feature can be particularly useful when replicas have varying load capacities;
- **Weighted response time**: this approach prioritizes replications with the lowest average response time to improve response time for clients;
- **Resource-based**: forwards requests based on the available resources, such as CPU and memory.

Having replicas with their internal state could cause inconsistencies in the use of the service. Our exploration has found methods to handle these 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 latter 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.

This proof of concept implemented a sticky session between clients and replicas of a certain service. We developed a web service using PHP and delivered it with an Apache web server. The choice of these technologies was motivated by our pre-existing knowledge of developing with these technologies. The results and experiments presented in this report can be replicated using equivalent technologies, such as Java with Tomcat. For the load balancer implementation, we looked at several available technologies, including HAProxy[5], NGINX[6], and Traefik[7]. We concluded that the latter was the best fit for our case, providing support and auto-discovery functionalities for services deployed in Docker while offering simple configuration and usage.

For this experiment, we used the following Docker images:

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

[5]https://www.haproxy.org/
[6]https://www.nginx.com/
[7]https://traefik.io/traefik/

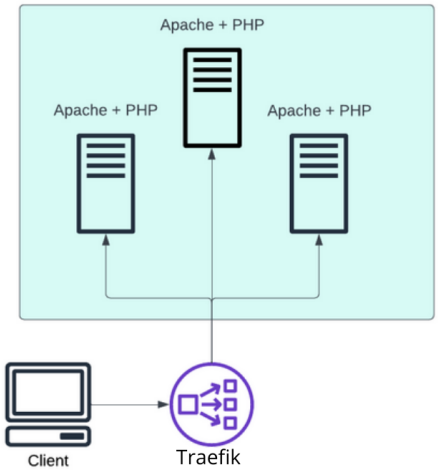

Fig. 7. PoC 1 system architecture.

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;
2) Go to `http://localhost` and reload the page several times to update the counter;

In this implementation of the sticky session, a persistent connection was established between the client and one of the three available replicas using a cookie. Our experiments suggest that when the server assigned to the user becomes unavailable, the sticky session is no longer valid. In this case, another server will be assigned according to the load balancing algorithm. During this experiment, we wondered if it would be possible to release an update to a service that clients are already accessing without causing a disruption.

IX. PoC 2: How can updates to services currently used by clients be deployed?

After a service is released to clients, it may require improvement to optimize performance, fix bugs, or add new features. In cases where high availability is a primary requirement, releasing a new version of the service may need to avoid disruption, especially if frequent updates are planned.

Additionally, it is important to consider the impact on clients currently accessing the service. Even a brief interruption could result in data loss or unexpected system behavior. For example, a client waiting for the outcome of a bank transaction may experience errors or duplicate transactions if the old version of the payment service is interrupted by the release of a new version.

This experiment aimed to implement a V1 service and simulate a graceful release of an update to the V2 service without causing disruption. To achieve this, we released the V2 service without removing the V1 service and gave V2 the highest priority. As a result, new client requests will be directed to V2 replicas, while clients connected to V1 can complete their requests. Once all requests to the old service are fulfilled, it can be decommissioned. Before decommissioning the V1 service, the performance of the V2 service can be

evaluated, and a rollback can be performed immediately if quality degrades.

In our proof of concept, the V1 service delivers a web page. However, due to a bug, the server takes 5 seconds to process the response. Service V2 corrects this issue and should be released without causing errors and outages to connected clients. To simulate requests from clients, we used Locust, which provides load testing on web services. This tool records the outcome of simulated client requests and reports statistics on server response times. In our case, Locust was configured to simulate 500 users making an HTTP request at random intervals between 1 and 5 seconds.

For this experiment, we used the following Docker images:

- **traefik:v2.10**: 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.

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`;
2) Start the load test using the web interface at `http://localhost:9098`;
3) Execute `2.sh`: this script deploys three replicas of web service V2, an updated version of V1;
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 stops all replicas of web service V1.

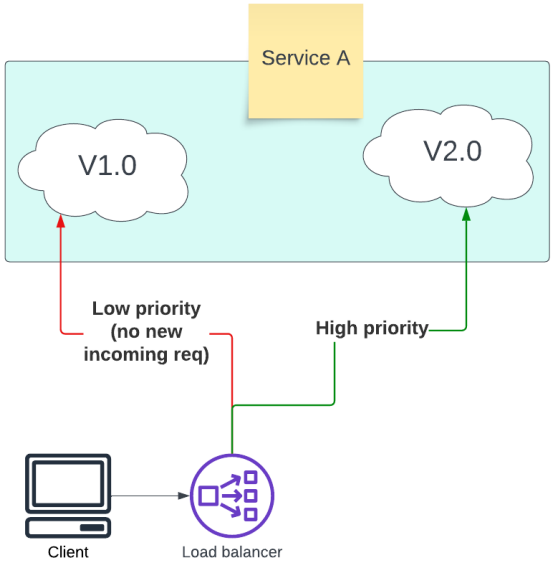

Fig. 8. Workflow of PoC 2 (locust omitted)

The results of a load test with 500 connected users are displayed in Figure 9. Each user had a randomly varying request time between 1 and 5 seconds during the release of the new version. It is clear that after the release of V2 and the decommissioning of V1, no errors are reported by the clients. In addition, there is a significant decrease in response time to clients, indicating that clients are using V2 properly. We have successfully released an updated version of the service. At this point, we considered how to ensure the service's continued operation in the event of an unexpected failure of one of the replicas.

Fig. 9. Requests during PoC 2 execution. Green for successful requests, red for failed requests

X. PoC 3: How to handle failed replicas?

We discussed the importance of creating replicas of a service, but how would the system behave if a replica failed to handle client requests?

This proof of concept involves a system with three duplicate web servers. The load balancer can now check the health of each server every second. A request is considered correct by the load balancer only if the response arrives within 5 seconds and the HTTP code is class 2XX or 3XX. If an unhealthy replica is detected, it will be removed from the pool of replicas involved in the load balancing, We wondered what should happen if the user encounters an unhealthy server before it is detected by the load balancer. The first assumption would be to retry the request on another server without interrupting the user's experience. Alternatively, the service can inform the client of the failure by sending an HTTP 503 code. The former approach attempts to hide the failure from the user, even though it has occurred. The latter approach leaves the choice up to the client, who can retry the request immediately. Additionally, Traefik's dashboard offers a simple visual display of the service's status.

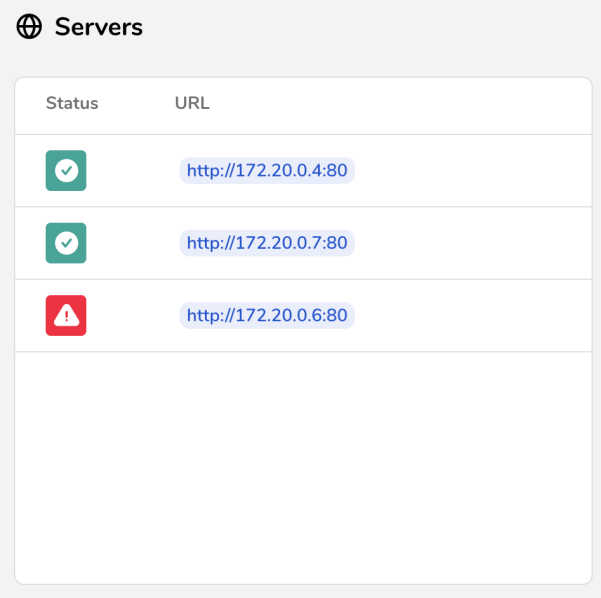

Fig. 10. Replicas monitor in Traefik dashboard.

For this experiment, we used the following Docker images:

- **traefik:v2.10**: 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.

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.

During the experiments, the load test showed that a total of 3 to 8 clients experienced request failures. This was due to the health check running every second, while the load test involved 50 users submitting requests at intervals ranging from 1 to 5 seconds. Depending on the requirements, this situation may be acceptable.

XI. PoC 4: How to manage the replication of data persistence services?

In the context of persistent data systems, such as databases, it is important to understand how to manage stored data. When there is only one data persistence replica, it becomes a single point of failure in the system. However, when there are multiple replicas, the data must be carefully synchronized to ensure consistency of information.

To implement database replication, we analyzed various methodologies [6]. The method called instance-level clustering enables database replicas to read and write to shared storage, such as a network drive, as shown in Figure 11.

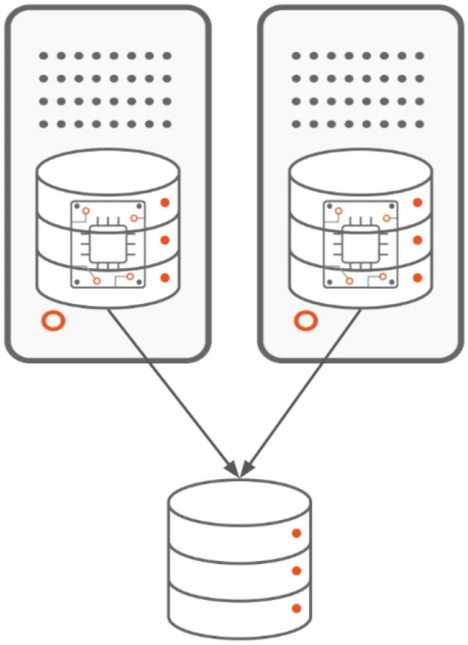

Fig. 11. Instance-level clustering. Image by https://ubuntu.com/blog/database-high-availability

The second approach, called database-level clustering, requires replicas to exchange data to update their local memory. The master replica receives read/write requests from clients and distributes changes to the slave replicas.

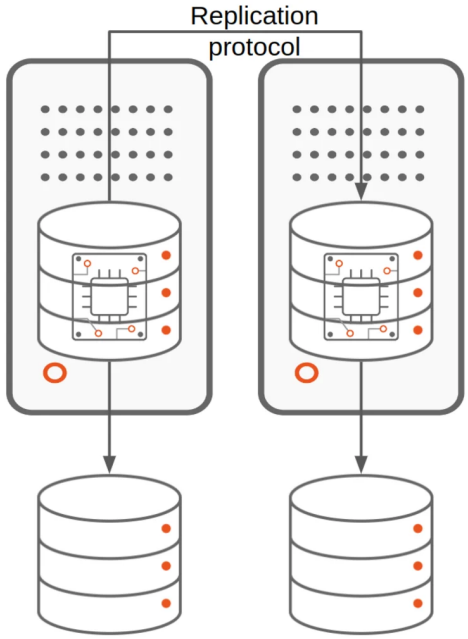

Fig. 12. Database-level clustering. Image by https://ubuntu.com/blog/database-high-availability

In this proof of concept, we replicated a database using database-level clustering and created a web page to display the contents of both the master and slave instances. If the master node experiences a failure, the slave can be selected as the new master.

PostgreSQL technology was used to create replicas of our database due to its support for database-level clustering.

For this experiment, we used the following Docker images:

- **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`: 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.

During the experiment, a simple replica of the database was created. However, designating the slave as the new master resulted in an inconsistent state for the old master node. If the latter node is recovered, it will no longer be synchronized with the new master. Depending on the case and requirements, the system may allow the database to remain in a read-only state until the master node is recovered.

XII. PoC 5: How can the workload be distributed across multiple nodes?

We implemented a container-based system deployed on multiple nodes and orchestrated by Docker Swarm. Docker Swarm was chosen due to its ease of deployment and our familiarity with Docker technology. Although Kubernetes provides more sophisticated and flexible node management, we decided not to use it due to its complexity in installing and managing the cluster. Finally, a sophisticated system for managing and monitoring the infrastructure will be implemented.

To simulate a multi-node cluster, three virtual machines were created to simulate the physical nodes. Next, we installed and configured the Grafana dashboard to visually display performance and error metrics reported by Traefik. Additionally, we configured Portainer to display container status on nodes.

The realized system offers two services. The first, which is addressed to patients of a health service, allows users to view their medical information and send messages to doctors. The second service, addressed to doctors, allows users to view information on all patients and received messages. The first service, considered important but not essential, is replicated in only one node, while the second service is replicated in two nodes.

For this experiment, we used the following Docker images, and distributed them 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.

In our experiment, we implemented a distributed system to improve system availability. Services and replicas are distributed across multiple nodes, preventing any single node from becoming a point of failure. Although the scenario was simplified, we faced several challenges in managing and configuring nodes and containers. In some cases, for example, nodes were unable to communicate with each other due to incorrect network configurations.

XIII. Conclusions

In conclusion, the analysis conducted on high availability strategies, both from a theoretical perspective and through practical implementation on a system, provides an overview of the challenges and solutions to ensure system availability. By combining theory and practice, we realized how crucial it is to consider all related aspects to ensure an available system. This enabled us to identify areas for improvement and develop specific strategies to optimize system availability.

References

[1] J. Gray and D. P. Siewiorek, "High-availability computer systems," in Computer, vol. 24, no. 9, pp. 39-48, Sept. 1991, doi: 10.1109/2.84898.
[2] IBM. High availability overview
[3] Availability and Beyond: Understanding and Improving the Resilience of Distributed Systems on AWS.
[4] Trivedi, K; Ciard, G; Dasarathy, B; Grottke, M; Matias, R; Rindos, A; Vashaw, B. Achieving and assuring High Availability.
[5] Vargas, Enrique. "High availability fundamentals." Sun Blueprints series (2000): 1-17.
[6] Patterns to achieve database High Availability