UNIVERSITY OF TRIESTE
DATA SCIENCE AND ARTIFICIAL INTELLIGENCE

Project Exam March 25, 2024

Cloud Computing Basic Exercise Report

by

Luca Pernice

1 Abstract

This report details the design and implementation of a simulated cloud-based file storage system using MinIO, a high performance, distributed object storage server. The project's core objective was to demonstrate the deployment and functioning of a scalable and manageable cloud storage system within a contained environment, specifically on a personal laptop. The system was architected using Docker Compose to create four separate MinIO server instances, each simulating a node in a distributed storage network. Each container was configured with two local storage volumes, effectively emulating storage disks, to underline the principles of distributed data storage. This setup provided a practical illustration of key cloud computing concepts such as containerization, data distribution, and basic object storage functionalities within a controlled environment.

2 System Design and Architecture

The core of this project revolves around the creation of a cloud-based file storage system using MinIO, an open-source object storage server that is compatible with Amazon S3 cloud storage service. MinIO is known for its high performance and minimal software footprint, making it an ideal candidate for educational and experimental purposes. The system was designed to run on a personal laptop, simulating a distributed cloud environment within a local setting.

2.1 Architecture Overview

- **MinIO Servers**: The architecture includes four Docker containers, each running an instance of MinIO. These containers are configured to communicate with each other, forming a distributed object storage system. This multi-container setup emulates a cloud storage cluster, demonstrating how data can be managed and stored across different nodes in a cloud environment.

- **Storage Volumes**: Each MinIO container is linked to two local storage volumes. These volumes represent the physical storage of the data, akin to disks in a cloud storage server. This configuration allows for the demonstration of data distribution and redundancy within the system.

- **Network Configuration**: The containers are configured to expose their services on different ports (9001 to 9004) on the host machine. This setup mimics the way services are exposed in a real cloud environment, with each node accessible via a unique endpoint.

- **Docker Compose**: The use of Docker compose simplifies the deployment and management of these multiple containers. It allows for defining and running multi-container Docker applications, thereby creating an environment that closely mirrors a real-world cloud setup.

2.2 Design Choices

- **Local Environment Simulation**: The decision to deploy the system on a personal laptop was driven by the need for an accessible, controlled environment for academic purposes. It allows for hands-on experience with cloud storage concepts without the complexities and costs associated with a

full-scale cloud deployment.

- **Distributed Storage Simulation**: By configuring multiple MinIO instances, the system demonstrates key aspects of distributed storage systems, such as data replication, fault tolerance, and load balancing, albeit on a smaller scale.
- **MinIO Selection**: MinIO was chosen due to its high performance, S3 compatibility, and ease of use. It supports a wide range of cloud-native deployment scenarios and is well-suited for educational demonstrations of cloud storage systems.
- **Volume Configuration**: The dual-volume setup for each container is designed to illustrate the concept of data storage and replication in a distributed system. It provides a tangible understanding of how data is managed across different storage units in a cloud infrastructure.

In summary, the system's design and architecture aimed to create a simplified yet effective representation of a cloud-based file storage system. The use of MinIO and Docker containers facilitated a practical approach to understanding the fundamental components and operations of cloud storage in a distributed environment.

3 Implementation Details

This section provides an in-depth look at the practical aspects of implementing the MinIO-based cloud storage system. The implementation involved setting up MinIO instances using Docker, configuring storage volumes, and establishing network communication among the instances.

3.1 MinIO Setup with Docker

The implementation began with the deployment of MinIO servers using Docker containers. Docker provides an efficient, isolated environment for each MinIO instance, allowing for a scalable and manageable setup.

- **Container Configuration**: Each of the four Docker containers was configured to run the latest MinIO image. The Docker Compose file facilitated

this setup, defining the necessary parameters such as image version, ports, volumes, and environment variables.

- **Environment Variables**: For each MinIO instance, environment variables `MINIO_ACCESS_KEY` and `MINIO_SECRET_KEY` were set, providing necessary credentials for access. These were uniformly set across all instances to simplify the configuration.

- **MinIO Command**: The command `server http://minio{1...4}/data{1...2}` was used to initialize each MinIO server. This command enables the distributed mode, allowing each MinIO instance to connect and share data with the others, thereby creating a unified storage pool.

3.2 Storage Volume Configuration

The core of the system's storage capabilities lies in the configuration of the storage volumes.

- **Volume Mapping**: Each container was linked to two volumes, simulating the behavior of storage disks. These volumes were mapped to specific directories on the host machine, effectively creating a local data storage environment.

- **Data Distribution and Redundancy**: The dual-volume setup for each container allowed for an illustration of data distribution and redundancy in a distributed storage system. This setup highlights how data is replicated and managed across different storage units, ensuring data availability and fault tolerance.

3.3 Network Configuration and Access

- **Port Mapping**: To simulate a real-world scenario where each storage node is accessed via a unique endpoint, the containers were configured with different port mappings. The ports 9001 to 9004 on the host were mapped to the standard MinIO port (9000) on each container, enabling independent access to each MinIO instance.

- **Access and Testing**: The system was tested by accessing the MinIO web

interface through the mapped ports. This provided a straightforward way to interact with each MinIO server, allowing for operations such as file upload, download, and management.

3.4 Monitoring and Health Checks

- **Health Checks**: Docker Compose was configured with health checks for each MinIO container. These checks involved simple curl commands to the MinIO health endpoint. Regular health checks ensure that each instance is operational and accessible, mimicking a basic monitoring mechanism found in production environments.

3.5 Erasure Coding in MinIO

A critical aspect of the MinIO implementation is its use of erasure coding, a method of data protection and fault tolerance.

- **Concept of Erasure Coding**: Erasure coding in MinIO works by breaking down data into chunks and then spreading these chunks across different drives or nodes. Along with the data chunks, parity chunks are also created and stored. This method allows the system to reconstruct the original data in the event of partial data loss, such as a disk failure.

- **Implementation in Dockerized MinIO**: In the Dockerized MinIO setup, each MinIO instance is connected to two simulated storage disks (volumes). When a file is stored in the MinIO cluster, it is split into data and parity blocks, which are then distributed across these volumes.

- **Fault Tolerance**: This setup allows the system to tolerate the failure of any one volume without data loss. For example, if one volume becomes inaccessible, MinIO can still reconstruct the original data using the remaining volumes.

- **Scaling and Performance**: Erasure coding is a scalable method for data protection. It allows MinIO to maintain high levels of performance and efficiency, even as the amount of stored data grows.

- **Configuration and Management**: In the Docker Compose setup, MinIO's

erasure coding feature does not require additional explicit configuration. It is inherently managed by MinIO when the distributed mode is enabled.

In this implementation, MinIO's erasure coding feature plays a vital role in illustrating how modern cloud storage systems ensure data durability and availability. It provides a real-world example of how data is protected in distributed storage environments.

4 Security Measures

In implementing MinIO for my project, I explored various security features relevant to a production-level deployment. While these measures provide a strong foundation for securing a MinIO setup, due to my limited experience in the field at the time, I did not implement all elements of the security checklist in my academic project. They served primarily as a learning guide.

4.1 Overview of Security Measures

The main security considerations included:

- **Access Policies**: Establishing group and individual access policies using MinIO or a third-party Identity Provider.
- **Firewall Configuration**: Correctly configuring firewall access for TCP traffic to the MinIO Server S3 API and Console.
- **Encryption-at-Rest**: Employing external Key Management Services for server-side encryption.
- **Encryption-in-Transit**: Implementing TLS for data transmission security.

4.2 Contextual Application

These measures, while crucial for a production deployment, were more of educational elements in the context of my demonstrative project. They offered a valuable reference point for understanding security in cloud storage systems.

Reference: `https://min.io/docs/minio/container/operations/checklists/security.html`

5 Cost-Efficiency Analysis

In my project, the feasibility of conducting a detailed cost-efficiency analysis for the MinIO implementation on my laptop presents both challenges and limited practicality. The nature of the deployment environment and its scale significantly influences this assessment.

5.1 Challenges of Conducting a Cost-Efficiency Analysis

- **Scale and Environment Limitations**: Running MinIO on a personal laptop within a Dockerized environment does not accurately reflect the resource usage, operational costs, and scalability encountered in a production-level cloud storage environment.
- **Irrelevance of Cloud Cost Factors**: Key factors that impact cost-efficiency in a cloud setting, such as network bandwidth, storage costs per GB, and data transfer costs, are not directly applicable or measurable in a localized laptop setup.
- **Absence of Cloud Infrastructure Elements**: The lack of real-world cloud infrastructure elements, such as maintenance and monitoring overhead costs, diminishes the relevance of a precise cost analysis in my project's context.

5.2 Adopting General Cost-Efficiency Principles

While a detailed cost-efficiency analysis is not practical in the context of my project, adopting general cost-efficiency principles can still provide valuable insights. These principles offer a broader understanding of what to consider in a full-scale MinIO deployment in a cloud environment.

- **Storage Optimization and Resource Management**: Effective storage management is key to cost-efficiency. This includes employing data lifecycle policies, where data is moved to cheaper storage options based on its access

frequency. Efficient resource management also plays a crucial role, as it directly impacts operational costs. Monitoring and optimizing the use of CPU, memory, and network resources can lead to significant cost savings, especially in larger-scale deployments.

- **Scalability Implications**: Understanding how scalability affects costs is essential. As storage needs grow, MinIO's architecture allows for scaling horizontally by adding more nodes. This scalability must be balanced with the costs of additional hardware, resources, and potential complexities in management. Evaluating the trade-offs between scalability and cost is crucial for maintaining an efficient system.

- **Data Transfer and Access Costs**: In a cloud environment, data transfer and access can incur significant costs. It's important to consider these costs when designing and scaling the system. Strategies like caching frequently accessed data and optimizing data transfer routines can help in reducing these costs.

- **Redundancy and Backup Costs**: While redundancy is crucial for data availability and protection, it also has cost implications. Strategies like erasure coding, used by MinIO, provide redundancy without the same level of cost as traditional replication methods. Understanding these techniques can help in achieving a balance between cost and reliability.

- **Alternative Solutions Comparison**: Comparing MinIO with other storage solutions such as AWS S3, Google Cloud Storage, or Azure Blob Storage can offer a perspective on cost-effectiveness. This comparison should include direct costs (like storage pricing) and indirect costs (like ease of management and integration capabilities). Such a comparison can inform decisions about choosing the most cost-effective solution for specific needs.

These principles, while explored in a theoretical context in my project, form the foundation of cost-efficiency considerations in cloud storage systems. They highlight the importance of strategic planning and optimization in managing costs effectively in any cloud storage deployment.

6 Deployment

The deployment strategy for the MinIO-based cloud storage system in my project is theoretical and serves as a guide for a real-world scenario. This section outlines the key steps and considerations, including the use of MinIO's SUBNET subscription for streamlined management.

6.1 Deployment in a Containerized Environment

- **Using Docker and Docker Compose**: Docker and Docker Compose are ideal for deploying MinIO, providing scalability, ease of management, and efficient updating of instances.
- **Configuration Management**: Proper configuration, including environment variables, storage mappings, and network settings, is essential for performance and security.
- **Cluster Management**: Deploying MinIO in a clustered configuration with multiple instances in distributed mode ensures data redundancy and high availability.

6.2 Monitoring and Management with SUBNET

- **System Monitoring**: Implementing monitoring solutions like Prometheus and Grafana is crucial. MinIO's SUBNET subscription offers additional support and tools for monitoring and managing MinIO deployments effectively.
- **Performance Tuning**: Regular tuning based on monitoring data is key. SUBNET can provide insights and support for optimizing MinIO performance.

6.3 Deployment in a Cloud Environment

- **Cloud Provider Selection**: The choice of a cloud provider depends on cost, scalability, and compatibility. SUBNET can offer guidance on optimizing MinIO deployment in various cloud environments.
- **Leveraging Cloud Features**: Utilizing features like auto-scaling and load

balancing enhances efficiency. SUBNET can assist in integrating these features with MinIO.

- **Security Considerations**: In addition to standard security measures, SUBNET provides specialized support for ensuring secure deployments in the cloud.

6.4 Backup and Disaster Recovery

- **Regular Backups and Recovery Plans**: SUBNET can play a significant role in advising and assisting with backup strategies and disaster recovery planning for MinIO deployments.

While these deployment strategies are theoretical for my project, they offer a foundational understanding of deploying MinIO in a more complex environment, with SUBNET providing valuable support and management capabilities.

7 Testing and Performance Evaluation

For evaluating the performance and reliability of my MinIO implementation, I utilized Locust, an open-source load testing tool, and conducted a resilience test by manually stopping one of the MinIO containers.

7.1 Load Testing with Locust

- **Locust Setup**: Locust was configured to simulate a number of users performing typical file storage operations, such as uploading, downloading, and accessing files on the MinIO cluster.

- **Performance Metrics**: The key metrics monitored during the load test included response times, throughput, and error rates. This provided insights into how the MinIO cluster handles increased load and concurrent user access.

- **Observations and Analysis**: The results from Locust were analyzed to assess the performance of the MinIO implementation under different load conditions. This helped in identifying any potential bottlenecks or performance

issues.

7.2 Resilience Testing

- **Container Shutdown Test**: To evaluate the resilience of the MinIO cluster, I conducted a test by manually stopping one of the MinIO containers. This simulated a node failure scenario.
- **Cluster Functionality Assessment**: Following the shutdown of one container, I monitored the functionality and performance of the remaining MinIO cluster. This was crucial to assess the fault tolerance and data redundancy capabilities of MinIO.
- **Recovery and Stability**: The focus was also on observing the recovery process and stability of the cluster after the container was restarted. This test provided valuable insights into the self-healing and recovery mechanisms of MinIO in a distributed setup.

These tests were vital in understanding the performance and reliability of the MinIO implementation in a simulated, distributed environment. They provided practical insights into how the system would behave under stress and failure conditions, essential for any robust cloud storage solution.

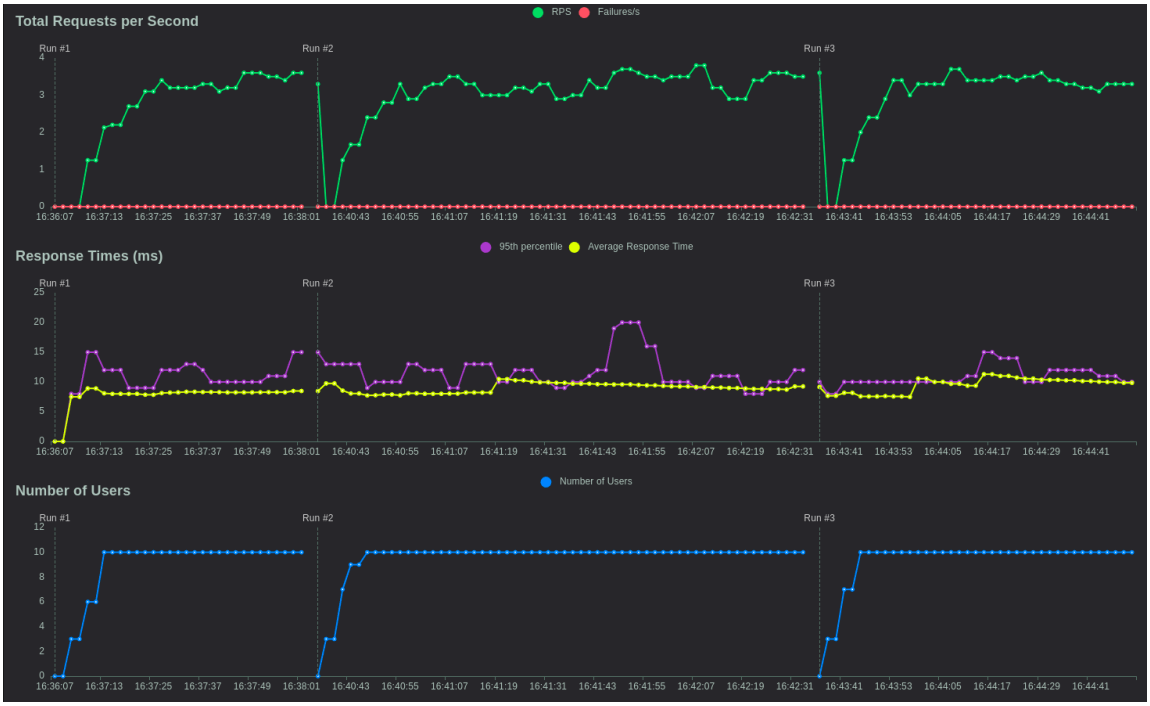

Figure 1: During the first run, the 'Total Requests per Second' graph indicates a steady load with no recorded failures, signifying that the cluster is managing the load efficiently while all four nodes are operational. In the second run, even as a node is stopped, there is no visible dip in the 'Total Requests per Second'. This absence of a dip suggests that the MinIO cluster seamlessly handled the node outage without affecting the overall request handling capacity. The cluster's performance remains consistent, as indicated by the stable 'Response Times' and the lack of failures. Finally, during the third run, the previously stopped node is brought back online. The metrics before and after this event show consistent performance, implying that the MinIO cluster's rebalancing and recovery mechanisms are effective, and the cluster maintains its resilience throughout the node restart process.

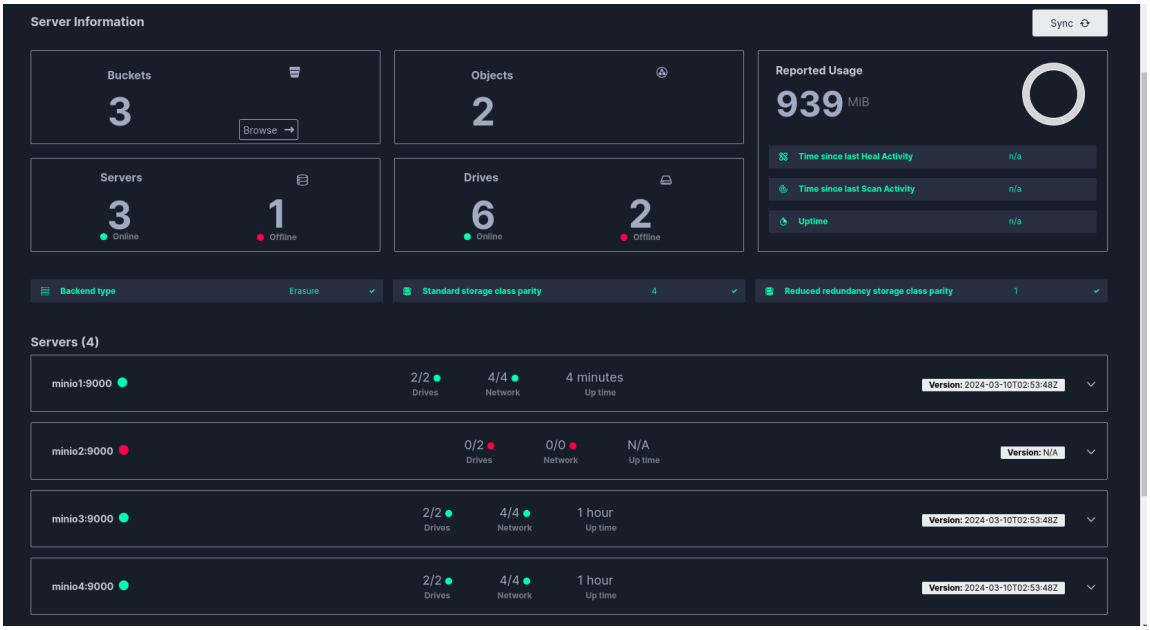

Figure 2: Node 2 offline.

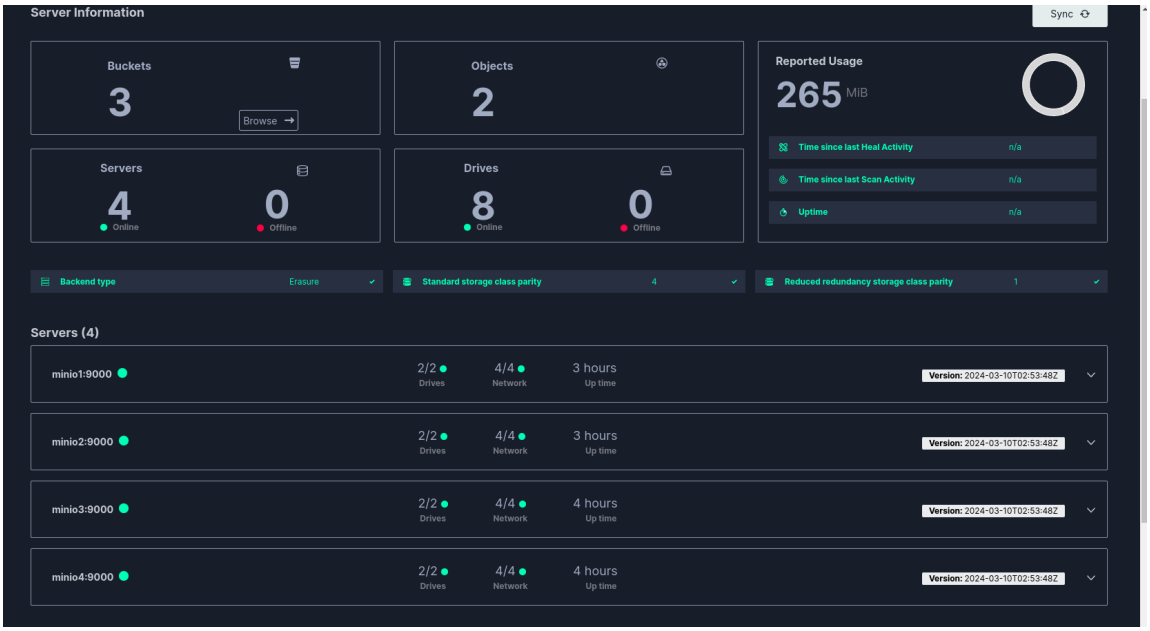

Figure 3: All nodes working