Lab Report: Final assignement Cloud Design Patterns

Author: Zabiullah Shair Zaie

Submitted to: Vahid Majdinasab

Institution: Polytechnique Montréal

GitHub Repository: GitHub Link

Demo Video: Demo Video Link

Contents

1 Introduction

This lab focuses on implementing the Proxy and Gatekeeper patterns to address scalability and security issues in a cloud-based environment.

Objectives The specific objectives of this lab include:

- Benchmarking MySQL using sysbench.
- Implementing the Proxy and Gatekeeper patterns.
- Benchmarking the performance of the Proxy and MySQL cluster.
- Demonstrating how the implementation works with detailed explanations.
- Providing a summary of results and instructions to run the code.

2 Infrastructure and Automation

This section describes the setup process, automated using a Python script (‘main.py‘), and the purpose of each component.

Key Components

- **Bastion Host**: Provides secure access to private instances.
- **MySQL Cluster**: Includes a master and two worker nodes.
- **Proxy Instance**: Distributes database requests.
- **Gatekeeper Instance**: Ensures secure access.
- **Trusted Host**: Manages critical operations.

Automation Workflow

2.0.1 Key Pair Creation

A key pair named `tp3-key-pair` was generated to enable SSH access, ensuring secure access to all instances.

2.0.2 Security Groups and iptables

- **Public Security Group**: Allows traffic on SSH (22), HTTP (80), HTTPS (443), and Flask application (5000).
- **Private Security Group**: Restricts access within the private subnet.
- **iptables Rules**: Tailored rules were applied at each instance level to allow traffic only from designated sources. For MySQL instances, port 3306 was used, while 5000 was reserved for other services.

2.0.3 NAT Gateway

A NAT Gateway was set up to allow private instances to download necessary packages securely without exposing them to external traffic.

2.0.4 Instance Initialization

User data scripts were employed to:

- Install required packages such as MySQL, Flask, and benchmarking tools.
- Initialize services automatically on instance startup.

2.0.5 Sleep Interval

A 4-minute wait was introduced to ensure all instances were fully operational before proceeding with configurations and benchmarking.

3 Implementation of Cloud Design Patterns

This section provides a concise overview of the Proxy and Gatekeeper patterns implemented in the system.

3.1 Proxy Pattern

The Proxy instance is responsible for managing all database requests and supports three modes:

- **Direct Hit**: All requests are routed to the master database node, ensuring consistency for write-heavy workloads. This mode is ideal for scenarios where data integrity is critical, as all operations are centralized.

- **Random**: Requests are distributed randomly across worker nodes, balancing read operations without additional computational overhead. This mode helps evenly distribute the load among workers, improving scalability.

- **Customized**: The Proxy selects the fastest worker node based on ping response times, optimizing performance for read-intensive scenarios. By dynamically identifying the fastest node, this mode minimizes latency and enhances query throughput.

Key aspects of the Proxy implementation include:

- **Routing Logic**: The Proxy processes JSON requests that specify the query, type (read/write), and mode (direct_hit/random/customized). Depending on the mode:

  - For **write** requests, the Proxy always routes queries to the master node.
  - For **read** requests, the target database is selected based on the specified mode.

  After determining the target node, the Proxy executes the query and returns the result to the requester.

- **Metrics Collection**: The Proxy collects detailed metrics, including:

  - Total requests handled for each mode.
  - Average response times for read and write operations.
  - Cluster request times logged for benchmarking purposes.

  These metrics are saved to a file for performance analysis and system evaluation.

- **Worker Synchronization**: The Proxy does not handle replication. Worker nodes are configured to synchronize with the master node autonomously using MySQL replication. This ensures data consistency while simplifying the Proxy's responsibilities.

- **Custom Query Optimization**: In the **Customized** mode, the Proxy dynamically determines the fastest worker by evaluating ping times to the available nodes. This reduces query execution delays, particularly in high-latency environments.

3.2 Gate Keeper

The Gatekeeper serves as the first line of defense, validating and forwarding client requests to the internal network. It is a critical component in ensuring the system's security and operational integrity.

Key functionalities of the Gatekeeper include:

- **Request Validation**: The Gatekeeper enforces strict validation rules, ensuring every request contains the required fields (`type`, `mode`, `query`). Requests with invalid or missing fields are rejected immediately, reducing unnecessary processing.

- **Forwarding to Trusted Host**: Once a request is validated, the Gatekeeper forwards it to the Trusted Host for further processing. This separation ensures modularity and maintains the integrity of the internal network.

- **Security Enforcement**: In conjunction with `iptables` rules and AWS Security Groups, the Gatekeeper ensures that only whitelisted IP addresses and legitimate requests can reach the system. Unauthorized traffic is blocked at the network level, reducing potential attack vectors.

The Trusted Host simplifies request handling by forwarding validated requests directly to the Proxy, maintaining a streamlined path for internal communication. This architecture ensures that all requests passing through the Gatekeeper are secure and adhere to the expected format.

The Proxy and Gatekeeper work together to create a secure and scalable architecture, where the Gatekeeper handles external validation and the Proxy optimizes internal query handling.

4 Benchmarking and Results

4.1 MySQL Benchmark with Sysbench

The MySQL benchmarking was performed using Sysbench 1.0.20 to evaluate the performance of the database system under a read-heavy workload. The test was run with a single thread for a duration of 10 seconds. Below are the key results:

- **Total Queries Performed:**
  - Read Queries: 119,140
  - Write Queries: 0
  - Other Queries: 17,020
  - Total Queries: 136,160
- **Transactions:**
  - Total Transactions: 8,510 (850.67 transactions per second)
  - Total Queries: 13,610.72 queries per second
- **Latency (ms):**
  - Minimum Latency: 0.82 ms
  - Average Latency: 1.17 ms
  - Maximum Latency: 4.91 ms
  - 95th Percentile: 1.96 ms
- **Threads Fairness:**
  - Average Events: 8,510 events
  - Execution Time: 9.973 seconds

These results demonstrate the high query throughput and low latency of the MySQL database under the tested workload.

4.2 Cluster Benchmarking

The cluster's performance was evaluated through benchmarking different modes of the Proxy. Each mode was tested with 2,000 requests, split evenly between read and write operations. The results for each mode are summarized below:

- **Direct Hit Mode:**
  - Total Requests: 2,000
  - Total Read Requests: 1,000
  - Total Write Requests: 1,000
  - Total Time Taken: 89.13116 seconds
  - Average Time per Request: 0.04457 seconds
  - Average Time per Read Request: 0.04096 seconds
  - Average Time per Write Request: 0.04817 seconds
- **Random Mode:**
  - Total Requests: 2,000
  - Total Read Requests: 1,000
  - Total Write Requests: 1,000
  - Total Time Taken: 88.40694 seconds
  - Average Time per Request: 0.04420 seconds
  - Average Time per Read Request: 0.03974 seconds
  - Average Time per Write Request: 0.04867 seconds
- **Customized Mode:**
  - Total Requests: 2,000
  - Total Read Requests: 1,000
  - Total Write Requests: 1,000
  - Total Time Taken: 88.33737 seconds
  - Average Time per Request: 0.04417 seconds
  - Average Time per Read Request: 0.03969 seconds
  - Average Time per Write Request: 0.04864 seconds

5 Results and Analysis

5.1 MySQL Benchmark with Sysbench

The Sysbench results demonstrated high query throughput and low latency, with an average latency of 1.17 milliseconds and over 850 transactions per second. These outcomes align with expectations for a well-configured database in a controlled, read-heavy workload. While the results confirm efficient database performance, the absence of write queries leaves mixed workload performance unexplored.

5.2 Cluster Benchmarking

The cluster benchmarking revealed expected performance for Direct Hit and Random modes, with Direct Hit benefiting from its simplicity and Random mode showing comparable results by distributing requests across worker nodes. Unexpectedly, the Customized mode, which uses ping times for optimization, offered only marginal improvements, suggesting the ping-based logic may not fully reflect server conditions. Overall, the results validated the cluster's robustness while identifying areas for improving the Customized mode's efficiency.

6 Instructions to Run the Code

Follow these steps to deploy and benchmark the system:

1. Clone the repository and navigate to the project directory.
2. Set up your AWS credentials in a `.env` file.
3. Run `main.py` to launch the infrastructure and configure instances.
4. Access logs and benchmark results from the `benchmarks_and_logs` directory.

7 Conclusion

The benchmarking demonstrated the efficiency of both the MySQL database and the cluster setup. Sysbench results highlighted high throughput and low latency for

a read-heavy workload, confirming the database's performance under optimal conditions. Cluster benchmarking showed that Direct Hit mode provided the fastest average request times, while Random and Customized modes performed similarly, with only marginal improvements from the latter's optimization. These results validate the system's overall performance and scalability, while also revealing areas for potential refinement in request distribution strategies.

**You can find the source code on GitHub repository and watch the demo video here.