POLYTECHNIQUE MONTRÉAL

LOG8415E – ADVANCED CLOUD COMPUTING

Final Assignment

Cloud Design Patterns : Implementing a DB Cluster

Par :
Tristan, Lachance (2141919)

Submitted to :
Vahid, Majdinasab

December 3rd, 2024, Montreal

Table of Contents

Introduction

The first goal of this project was to design and deploy a MySQL cluster on Amazon AWS, consisting of a manager and two workers. This cluster enables seamless interaction with a database and its two replicas, ensuring data consistency and reliability. Additionally, the project required the implementation of two key cloud design patterns : Gatekeeper and Proxy. The final architecture processes incoming requests by first routing them through the Gatekeeper, which filters and forwards valid requests to the Trusted Host. From there, the requests are directed to the Proxy, which dynamically routes them to the manager or the workers, depending on the operation type. Finally, we conducted performance benchmarks to analyze the cluster's efficiency. This report will detail the MySQL benchmarking process, the implementation of the Proxy and Gatekeeper patterns, the overall system architecture, and a summary of the results, along with instructions for running the code.

Benchmark MySQL with sysbench

Here are our benchmark results of the MySQL cluster with sysbench :

The performance of the MySQL standalone instances was evaluated using the Sysbench tool, which is widely used for database benchmarking. The tests measured the throughput and latency of read and write operations under a workload of 4 concurrent threads over 60 seconds.

0.1 Test Configuration

The benchmarking was performed with the following settings :

— **Database :** Sakila sample database

— **Threads :** 4

— **Duration :** 60 seconds

— **Sysbench Lua Script :** `oltp_read_write.lua`

0.2 Results

The following table summarizes the benchmarking results for the three MySQL standalone instances :

Instance IP	Transactions/sec	Queries/sec	Avg Latency (ms)	95th Percentile (ms)	M
54.86.14.94	228.35	4567.70	17.51	24.83	
54.146.205.145	228.68	4575.56	17.49	24.83	
54.167.59.125	245.54	4912.07	16.29	22.69	

Tableau 1 – Sysbench Results for MySQL Standalone Instances

0.3 Analysis

The results indicate consistent performance across the instances with slight variations :

— The instance at `54.167.59.125` outperformed the others, achieving the highest transaction and query throughput with lower average and maximum latency.

— All instances maintained a 95th percentile latency below 25 milliseconds, ensuring responsive query handling even under load.

— Minor variations in maximum latency could be attributed to network conditions or resource contention during the tests.

0.4 Conclusion

The benchmarking demonstrated that the standalone MySQL instances provided stable performance under moderate concurrent workloads. These results serve as a baseline for comparing the performance of the MySQL cluster and evaluating the effectiveness of the Proxy and Gatekeeper patterns.

Implementation of The Proxy pattern

The Proxy Pattern was implemented to enhance the scalability and performance of the MySQL cluster by routing database requests dynamically based on their type. The Proxy serves as an intermediary, ensuring that write requests are handled by the manager while read requests are distributed among the worker nodes. This approach leverages data replication to optimize read performance and maintain consistency across the cluster.

0.5 Proxy Implementations

Three distinct implementations of the Proxy Pattern were developed, each with unique routing logic :

— **Direct Hit :** All incoming requests, regardless of their type, are directly routed to the manager instance. This configuration acts as a baseline for performance comparison, as it bypasses any load-balancing logic.

— **Random Distribution :** Read requests are randomly routed to one of the worker nodes, while write requests are always sent to the manager. This approach introduces basic load balancing for read operations but does not account for server load or latency.

— **Optimized Distribution (Fastest Response) :** The Proxy pings all worker nodes to measure response times and selects the node with the lowest latency for each read request. Write requests continue to be directed to the manager. This implementation maximizes efficiency by dynamically adapting to network conditions.

0.6 Technical Details

The Proxy was developed using Python with Flask to handle HTTP requests. The logic for routing requests is encapsulated in `proxy_script.py`, which interacts with the cluster nodes defined in `instances_ips.json`. Key components include :

— **Routing Logic :** Read requests (`GET`) are processed by workers. Write requests (`POST`) are handled exclusively by the manager to ensure consistency.

— **Dynamic Worker Selection :** For the optimized implementation, a helper function calculates response times for all workers using `ping` commands. The fastest node is selected dynamically for each request.

— **Resilience :** The Proxy handles connection errors and retries failed requests, ensuring high availability.

0.7 Deployment

The Proxy runs on a `t2.large` EC2 instance to ensure sufficient resources for processing concurrent requests. It is configured to listen on port `5000` and communicates securely with the Manager and Workers over the private network.

0.8 Conclusion

The Proxy Pattern significantly improves the scalability of the MySQL cluster by offloading read requests to worker nodes and reserving write operations for the manager. The three implementations—Direct Hit, Random Distribution, and Optimized Distribution—offer different trade-offs between simplicity, load balancing, and efficiency, which were analyzed further in the benchmarking phase.

Implementation of The Gatekeeper pattern

The Gatekeeper Pattern was implemented to ensure secure access to the MySQL cluster by filtering and validating incoming requests. This pattern introduces an additional layer of security by separating roles and restricting direct access to sensitive components. The architecture consists of two main components : the Gatekeeper and the Trusted Host.

0.9 Architecture

The Gatekeeper Pattern was designed to manage incoming requests as follows :

— **Gatekeeper :** This component acts as the entry point for all external traffic. It validates requests to ensure they adhere to predefined rules. Malformed or suspicious requests are rejected immediately. Valid requests are then forwarded to the Trusted Host for further processing.

— **Trusted Host :** Operating on a private network, the Trusted Host receives validated requests from the Gatekeeper. It processes these requests and forwards them to the

Proxy, which determines the appropriate MySQL node (manager or worker) to handle the operation.

0.10 Technical Details

The Gatekeeper and Trusted Host were implemented using Python and Flask to manage HTTP traffic. Security and communication protocols were designed with the following considerations :

— **Validation :** The Gatekeeper checks all incoming requests for proper formatting, valid parameters, and adherence to security policies.

— **Private Communication :** The Trusted Host resides on a private subnet, isolated from external access. It only communicates with the Gatekeeper and the Proxy over secure, internal channels.

— **Firewall Configuration :** Security groups and firewall rules restrict access to the Trusted Host, allowing only verified traffic from the Gatekeeper instance.

0.11 Code Implementation

Here is an excerpt of the Gatekeeper logic from `gatekeeper_script.py` :

```
@app.route('/gatekeeper', methods=['POST'])
def validate_and_forward():
    data = request.get_json()
    if not is_valid_request(data):  # Custom validation logic
        return jsonify({"error": "Invalid request"}), 400

    # Forward valid requests to the Trusted Host
    response = requests.post(TRUSTED_HOST_URL, json=data)
    return jsonify(response.json()), response.status_code
```

0.12 Deployment

The Gatekeeper runs on a `t2.large` EC2 instance configured to handle traffic on port `5000`. It uses security groups to allow traffic only from trusted sources and forwards validated requests to the Trusted Host, which also runs on a `t2.large` instance within a private subnet. This separation ensures that no sensitive operations occur on internet-facing components.

0.13 Conclusion

The Gatekeeper Pattern strengthens the security of the MySQL cluster by filtering malicious traffic and isolating sensitive operations. By introducing the Gatekeeper and Trusted Host, the architecture minimizes the attack surface and enforces strict request validation. This pattern ensures that only verified traffic reaches the database, maintaining both security and operational integrity.

Benchmarking the Clusters

This section presents the results of benchmarking the MySQL cluster under three configurations of the Proxy Pattern : Direct Hit, Random Distribution, and Customized Distribution (Fastest Response). Each configuration was tested using 1000 read and 1000 write requests to evaluate the average response times for both operations.

Results Summary

The following benchmark results were obtained :

- ```
  === Benchmark results summary ===
  Mode: direct_hit
    Average Read Time: 0.1070 seconds
    Average Write Time: 0.1467 seconds
  Mode: random
    Average Read Time: 0.1060 seconds
    Average Write Time: 0.1473 seconds
  Mode: customized
    Average Read Time: 0.1164 seconds
    Average Write Time: 0.1570 seconds
  ```

Analysis of the Results

— **Direct Hit :** This mode routed all requests directly to the manager node. While the results demonstrate stable response times, this approach does not utilize the worker nodes, leading to underutilized resources and potential bottlenecks under heavier workloads.

— **Random Distribution :** Read requests were distributed randomly among the worker nodes, while write requests were still sent to the manager. This mode exhibited slightly better read performance compared to Direct Hit due to load balancing. However, the randomness introduced small inconsistencies in average response times.

— **Customized Distribution (Fastest Response) :** In this mode, the Proxy selected the worker node with the fastest response time for each read request. While this mode theoretically offers the best performance, the overhead of measuring response times slightly impacted the average latency for both read and write operations.

Discussion

The benchmarking results indicate that :

— The Direct Hit mode is straightforward but not scalable, making it less suitable for environments with high read demand.

— Random Distribution provides basic load balancing, leveraging worker nodes to improve read scalability.

— Customized Distribution optimizes resource utilization by dynamically routing requests, though the added complexity may introduce slight overhead.
  ```

Conclusion

The Proxy Pattern's Random and Customized modes demonstrate clear advantages in improving the cluster's read performance and scalability. These results validate the effectiveness of the Proxy implementation in distributing load and highlight the trade-offs between simplicity and optimization. Future improvements could focus on refining the measurement mechanism in the Customized mode to reduce overhead and further enhance performance.

Description of the implementation

The implementation of the MySQL cluster with Gatekeeper and Proxy patterns integrates several components, each with distinct responsibilities, to achieve scalability, security, and performance. This section outlines the workflow and interaction between the components.

0.14 Infrastructure Setup

The infrastructure was deployed using Amazon AWS with the following roles :

— **Manager instance :** A `t2.micro` instance that handles all write requests and ensures data replication to the workers.

— **Worker instance :** Two `t2.micro` instances configured to replicate the manager's data and handle read requests.

— **Proxy :** A `t2.large` instance that routes requests dynamically to the manager or workers based on request type.

— **Gatekeeper :** A `t2.large` instance that acts as the public-facing entry point, validating and forwarding requests to the Trusted Host.

— **Trusted Host :** A `t2.large` instance on a private subnet that processes validated requests from the Gatekeeper and communicates with the Proxy.

All instances were configured using an Infrastructure as Code (IaC) script written in Python, which automates the creation, configuration, and initialization of resources.

0.15 Workflow of a Request

The implementation processes requests in the following sequence :

1. **Gatekeeper Validation :**
   — A request is sent to the Gatekeeper, which validates its structure and parameters.
   — Invalid requests are rejected with an error response, while valid requests are forwarded to the Trusted Host over a secure internal channel.
2. **Trusted Host Processing :**
   — The Trusted Host receives validated requests and determines whether the operation is a read or write request.
   — It then forwards the request to the Proxy for further processing.
3. **Proxy Routing :**

- The Proxy inspects the request type :
  - **Write Requests :** Sent to the manager node, which updates the database and propagates changes to the workers.
  - **Read Requests :** Routed to one of the worker nodes. The selection mechanism varies based on the Proxy implementation (Direct Hit, Random Distribution, or Optimized Distribution).

4. **Database Interaction :**
   - The MySQL instances (manager and workers) process the requests and return responses.
   - The Proxy aggregates the responses (if necessary) and forwards them back through the chain to the client.

0.16 Security and Performance Features

The architecture integrates the following security and performance enhancements :

- **Security :**
  - The Gatekeeper prevents malicious traffic from reaching sensitive components.
  - The Trusted Host operates in a private subnet, isolating it from direct internet access.
  - Security groups enforce strict traffic rules for each role.
- **Performance :**
  - The Proxy balances read requests across workers, reducing load on the manager node.
  - Optimized worker selection minimizes response times for read operations.

0.17 Automation and Deployment

The entire infrastructure is automated using the `iac.py` script, which :

- Creates and configures EC2 instances for each role.
- Sets up security groups and firewall rules for secure communication.
- Deploys role-specific scripts on the appropriate instances.

Additionally, the `start.sh` script simplifies the deployment process by verifying the environment and launching the automation pipeline.

0.18 Conclusion

This implementation integrates the Gatekeeper and Proxy patterns to ensure secure and efficient handling of database requests. By automating the infrastructure setup and dynamically routing requests, the architecture achieves a balance between security, scalability, and performance.

Summary of Results and Instructions to Run the Code

The implementation of the MySQL cluster using Gatekeeper and Proxy patterns demonstrated significant improvements in security, scalability, and performance. This section summarizes the benchmarking results and provides instructions to reproduce the experiments.

0.19 Benchmarking Results

— **MySQL Standalone Performance :** The Sysbench benchmarks on standalone instances provided baseline metrics :
  — Transactions per second ranged between 228 and 246 across instances.
  — Average query throughput was consistent, peaking at approximately 4912 queries per second.
  — Latency remained stable, with 95th percentile latencies below 25ms.
  — These results established a foundation for evaluating the impact of clustering and pattern-based improvements.

— **Proxy Pattern :** The Proxy pattern significantly enhanced the scalability of the cluster :
  — **Direct Hit :** Performance was bottlenecked at the manager node, leading to higher response times under load.
  — **Random Distribution :** Load balancing of read requests across worker nodes reduced the load on the manager, improving overall read throughput.
  — **Optimized Distribution :** Selecting the fastest responding worker node minimized latency and maximized resource utilization, though with slightly higher computational overhead.

— **Gatekeeper Pattern :** The Gatekeeper successfully validated incoming requests, rejecting invalid traffic and isolating sensitive components from direct exposure. This additional security layer reduced the system's attack surface and improved reliability.

0.20 Key Takeaways

— The Proxy pattern provided scalable solutions to handle high volumes of concurrent read requests, demonstrating the effectiveness of intelligent request routing.

— The Gatekeeper pattern enhanced the security of the cluster by filtering traffic and preventing unauthorized access to sensitive components.

— The automated deployment pipeline streamlined the setup process, allowing rapid provisioning of resources and reproducibility of results.

0.21 Instructions to Reproduce the Results

Follow these steps to replicate the experiments :

1. **AWS Credentials :** Ensure your AWS CLI credentials are configured in `~/.aws/credentials`.
2. **Install Dependencies :** Install the required Python packages by running :

```
pip install -r requirements.txt
```

3. **Set Up the Infrastructure :** Run the setup script to deploy the MySQL cluster and associated components :

```
./start.sh
```

0.22 Conclusion

The results underline the importance of integrating cloud design patterns like Proxy and Gatekeeper into distributed systems. The Proxy pattern optimized scalability through intelligent request routing, while the Gatekeeper enhanced security by isolating and validating traffic. This implementation achieved a secure, scalable, and high-performing MySQL cluster, showcasing the potential of cloud-native design principles.