**CS404**

**Distributed System**

**Spring 2023**

**Project MESALIN**

**(A Distributed Message Queue System Solution)**

**By:**

**-Mehdi M’sallem**

**-Ala Boussarsar**

**-Sarra Ben Rjeb**

**-Emir Aissa**

**-Ines Ktiti**

A picture containing text

Description automatically generated

1. **Project Summary**

Project MESALIN is a distributed message queue system that efficiently manages load balancing across multiple nodes. The system is implemented using multiple technologies to demonstrate the principles of high availability, fault tolerance, consistency, and scalability, which are key in distributed computing.

The system supports some important design aspects such as :

* Scalability
* Fault Tolerance
* High Availability
* Real-time Processing
* Performance Tracking
* Isolation & Portability
* Concurrency
* Decoupling

The following user services are currently supported:

* Message Production
* Message Consumption
* Load Balancing
* Monitoring
* Fault Tolerance
* Scalability

The following system allows clients to generate & send messages to a specific Kafka topic where the rate of message can be adjusted to suit the clients requirements. To consume & read messages from the Kafka topic where this could be used to process data or trigger certain event actions based on the content of the message. To allow clients to scale up or down the number of producers, consumers & Kafka brokers based on their current needs. To Monitor various metrics related to the system’s performance, such as the rate of message production, consumption and the number of messages in Kafka topics. Therefore, this could identify the potential issues and optimize their use of the system.

1. **Project Architecture:**

A picture containing text, screenshot, diagram, line

Description automatically generated

Figure 1 - MESALIN Architectural Diagram

The architectural Diagram (Figure 1) used for project MESALIN involves different components – (1) Producers will send generated data to (2) Load-Balancer (HAProxy) that ensures the flow of data and connected on the other side to the (3) Kafka Brokers within a Kafka Cluster indicating that the Load-Balancer is distributing the incoming data among all the brokers that are connected to (4) Zookeeper that manages the Kafka Brokers mainly. On the other hand, (3) Kafka Brokers are connected to 2 separate (5) consumers to receive the incoming data. Also, the (3) Kafka Brokers are grouped within a (6) Kafka Cluster connected to (7) Prometheus that serves as a data source signifying that is scraping metrics from each Broker. Finally, a (8) Grafana component is directly connected to the (7) Prometheus indicating that it is visualizing the metrics scraped. To wrap up, each component in the diagram is represented in a (9) Docker Container to ensure the availability of the system overall.

1. **Cluster Architecture:**

In This architecture, Kafka brokers work together as a cluster to handle high volumes of reads and writes, distribute load, and provide fault tolerance and durability.

1. **Broker Cluster:**

The Kafka brokers form a cluster. A Kafka cluster consists of one or more servers (Kafka brokers), each of which is a node in the cluster. In this case, there are three Kafka brokers, namely kafka1, kafka2, and kafka3. These brokers collaborate to handle data streams efficiently.

1. **Zookeeper:**

Kafka brokers use ZooKeeper for maintaining the status of the cluster, configuring the topics, and synchronizing the state between brokers. ZooKeeper helps in managing and coordinating Kafka brokers. It is a centralized service for managing configuration information, naming, providing distributed synchronization, and group services.

1. **HAProxy:**

HAProxy instance acts as a load balancer. It distributes incoming messages from producers to the Kafka brokers based on its load balancing algorithm. In this case, a round-robin strategy is used, which means each broker in sequence receives a message, circling back to the first broker once all others have received a message. This ensures the load is evenly distributed across all brokers.

1. **Replication:**

Kafka provides built-in replication, which makes the data resilient to broker failures. Each topic can have multiple replicas spread across the Kafka cluster, so even if a broker fails, another broker can serve the data. This is not explicitly mentioned in the initial setup but is a crucial part of the Kafka broker cluster design.

1. **Producers & Consumers:**

Producers send messages to Kafka topics, and consumers read these messages. In the context of the cluster, each producer or consumer can connect to any broker in the cluster, and the connected broker will redirect the producer/consumer to the correct broker if necessary. This is called the bootstrap process.

1. **System Design Functionalities:**
2. **Real-Time Data Processing:**

The system can process real-time data streams using Kafka. Producers send messages to Kafka topics, and consumers read these messages.

1. **Load Balancing:**

The system uses HAProxy to distribute incoming messages from producers among the Kafka brokers evenly, ensuring that no single broker becomes a bottleneck.

1. **High Availability & Fault Tolerance:**

By clustering multiple Kafka brokers and using ZooKeeper for managing and coordinating them, the system ensures high availability and fault tolerance. If one Kafka broker goes down, others in the cluster can continue processing messages.

1. **Scalability:**

The system can handle increasing data volumes by adding more Kafka brokers or scaling up the existing ones.

1. **Monitoring:**

Prometheus continuously scrapes metrics from Kafka, providing real-time monitoring of the system's health and performance.

1. **Visualization:**

Grafana visualizes the metrics scraped by Prometheus, providing useful insights into the system's performance over time.

1. **Containerization & Easy Deployment:**

All the services are containerized using Docker, making it easy to deploy the entire system on any platform that supports Docker.

1. **Multi-Producers & Consumers:**

The system can handle multiple producers sending messages and multiple consumers reading messages concurrently, demonstrating Kafka's capability to support high-concurrency scenarios.

1. **Data Persistence:**

Kafka persists all data to disk, providing durable storage for the messages being processed.

1. **Automatic Recovery:**

In case of a broker failure, Kafka supports automatic recovery. Using its replication factors, it can recover data from the replicated copies of other brokers.

1. **Technology Stack:**

|  |  |
| --- | --- |
| Area | Technology |
| Data Handling | Apache Kafka |
| Synchronization | Apache Zookeeper |
| Data Distribution | Prometheus |
| Data Analysis | Grafana |
| Backend | Python |
| Deployment | Docker |

1. **Conclusion:**

The Kafka-Prometheus-Grafana-HAProxy architecture is a powerful and flexible solution for real-time data processing, monitoring, and visualization. It effectively handles high volumes of data and provides useful insights through metrics visualization. It also ensures high availability and fault tolerance through the use of Kafka and HAProxy. This system is scalable and adaptable to different data processing and analytics needs.