Project Report

1. Introduction

This project discusses the design and implementation of a Java Spring Boot REST service that acts as an orchestration layer between Redis, RabbitMQ, Kafka, and a persistent storage service. Key requirements centered on implementing REST API endpoints to manage the flow of messages through the provided service libraries.

2. Architecture Design

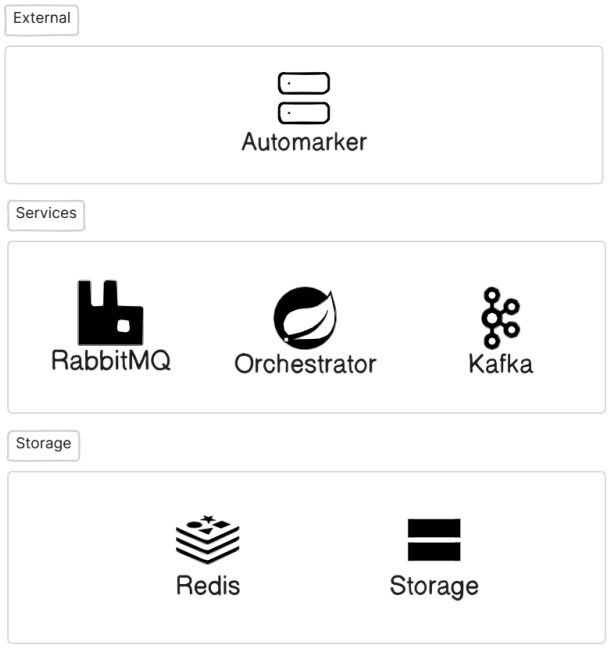

Figure 1. Components

I chose to design my application by leveraging on Spring's service beans to segregate the business logic from the controller. Kafka, RabbitMQ, Storage and Redis form the underlying services that are required for more complex message passing operations in process messages and transform messages.

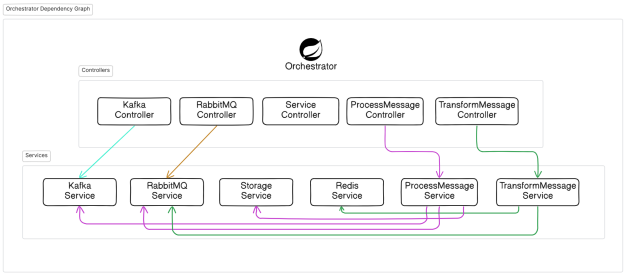

Figure 2. Dependency Graph between Controllers and Services

These components are commonly re-used and each service needs to have their configurations defined. Thus, using Spring's service allowed me to define the underlying services, and use them through Spring's dependency injection mechanism.

3. Native package vs Spring package

For RabbitMQ, Kafka and Redis, Spring's wrapper also offers higher level abstractions include auto configuration, connection thread management, and setup of listeners to stream messages into the application.

Based on the project requirements, parameters are provided during runtime through the REST API endpoints. This dependency on runtime values makes it difficult to utilize Spring's package as it needs to be defined prior (Eg. queue/topic listener). Hence, I made the choice to use the native packages which are more flexible.

4. Tasks

4.1. Setting up RabbitMQ

Before the native RabbitMQ package can be used, a connection with the RabbitMQ broker needs to be setup [2]. There were 2 implementations I have considered:

1. **try-with-resource** This method automatically creates the RabbitMQ channel and connection within the try block, and this connection is closed once the block ends. This prevents any unused opened connections, which can lead to resource leakage.

2. **Abstraction to Service** The connection state is managed in a separate service instead of it being created

within the controller. The connection has to be manually closed.

I chose the second choice because creating a new connection each time can be quite inefficient as mentioned in RabbitMQ documentation [3].

I also intended to consume RabbitMQ messages asynchronously (push) instead of polling (pull). This provides performance advantages as each network request will return messages, while polling may not. This spawns a background thread that waits for messages to arrive. I cannot terminate the connection until all the intended messages arrive. Thus, try-with-resource is not suitable here as it closes the connection as soon as the main thread completes.

Figure 3. Consuming RabbitMQ Messages

4.2. Setting up Kafka

A new Kafka consumer has to be created for each thread that wants to consume records [1]. Due to the increased complexity in managing the list of Kafka consumer instances, I opted to use a single Kafka consumer to poll for records.

However, there are times when Kafka is not able to retrieve any records when given a timeout value. This is because Kafka needs to discover the correct broker that holds the leader partition for the specified topic, leading to a longer delay on the first message polling.

For future code improvements, I intend to implement a heartbeat mechanism where the Kafka consumer periodically polls the topic to let the group coordinator know the consumer is active.

4.3. Process Messages

This task requires the integration of Kafka, Storage service and RabbitMQ and resembles a data pipeline architecture. In my current implementation, I emphasized simplicity and maintainability by using spring dependency injection to pass the required services to the process messages service, and utilize polling to retrieve Kafka records.

Polling is chosen over asynchronous listeners because it is likely that most data will be placed in the same partition if the partition key is constant. The advantage of multiple listeners is limited as each thread belongs to the same consumer group, and only 1 consumer instance can read from the same partition.

4.4. Transform Messages

I continue to use the asynchronous way to consume RabbitMQ messages as it is more efficient in resource usage as messages are automatically pushed to the application. I chose to implement thread safe variables on the value count to ensure race condition does not happen if I intend to increase the number of consumers in the future.

My Redis implementation uses a Spring service bean to provide automatic closure of Redis resources once it is no longer used. However the Redis connection pool is remained open, improving resource efficiency.

5. Testing

5.1. API Endpoints

For API testing, I manually setup a Postman environment containing the API requests to trigger the endpoint. This allows for a convenient template to see what endpoints are available, and expected response results are also saved in Postman.

5.2. Development

During development, I primarily used the debugger to step through the intermediate steps to check the result. For example, transformMessages requires the Redis value to be overwritten if a newer data version arrives. This change in Redis value cannot be properly observed if I only consider the final value.

5.3. Containerized Environment

I rely on the container logs to observe any error messages that may arise. For example, I had to troubleshoot connection issues caused by incorrect hostnames. I learned that the service name defined in the compose yaml correspond to the hostname which I can use to connect to the service.

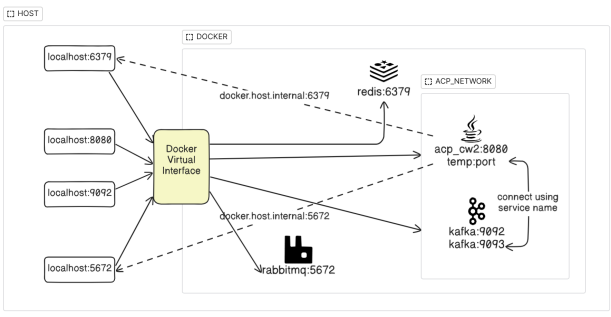

Figure 4. Network Architecture

Furthermore, since my application resides in the acp network while RabbitMQ and Redis are using the default bridge network, my application can only connect to RabbitMQ through the host interface. This is because acp network will not be able to recognize the RabbitMQ hostname as it is outside of it's network.

6. Challenges

6.1. Load

6.1.1 Kafka Consumer Bottleneck

As noted in 4.1 and 4.2, while the decision to use a single Kafka Consumer simplifies code implementation, this inherently limits the output message throughput even though Kafka offers a high message throughput.

6.1.2 Resource Contention

High load can cause contention between shared resources such as thread creation and CPU usage. Even though Spring Boot offers concurrency in it's REST Controller, this is fundamentally limited to the number of concurrent requests which the service can handle and can potentially lead to thread exhaustion. This is exacerbated by the use of polling in consuming from Kafka topics which is inefficient in CPU usage and is considered a blocking process which can lead to long response times.

6.2. Reliability

6.2.1 Data Loss

If a large number of network requests are sent to the application API endpoints, this can potentially overwhelm the application resources, causing it to crash. This can lead to the loss of data where the consumed messages do not get processed but has already moved on in the queue or topic.

In particular, the application uses RabbitMQ auto acknowledgment which identifies message delivery as successful without explicit acknowledgment from the application. If the application crashes after receiving the message without processing, the message is lost forever as RabbitMQ will remove the message from queue. For Kafka, this is less of an issue since it uses offset to manage message delivery. By setting the offset reset to earliest, the missed messages can still be manually retrieved from Kafka.

6.3. Scaling

6.3.1 Application Instance Scaling

The application must be stateless in order to scale. Since our application does not store any state, we can increase the number of application instances and use a load balancer to direct network requests to different application instances, preventing a single application instance from being overwhelmed. However, there is a limitation to this approach due to how Kafka consumers operate. As mentioned in 4.2, only 1 consumer instance in a consumer group can consume from a single partition. This will still cap the speed of handling incoming Kafka messages even with multiple instances.

6.3.2 Dependency Scaling

Kafka, RabbitMQ and Redis are designed to support horizontal scaling, and can scale together with the application. Kafka and RabbitMQ can add more broker nodes into it's cluster while Redis can create read replicas to offload requests. RabbitMQ also supports queue sharding if there are too many messages in a single queue, allowing the messages to be consumed in parallel. However, such parallelism disrupts the ordering of messages, which significantly increases application complexity.

7. Code Improvements

Based on the challenges identified in 6, I intend to improve the following:

1. **Parallelism** A sorting identifier should be appended to messages that are written to a queue or topic. Kafka messages should be evenly split into different partitions to maximize parallelism efficiency. In my code, I will create a thread pool counter, and each thread will own a consumer instance. Then, I will provide a global shared array to aggregate all the messages and sort them based on the identifier.

2. **Explicit Acknowledgment** The application code will send an acknowledgment only after the message has been successfully processed or transformed. This requires the code to turn off auto acknowledgment in RabbitMQ and auto commit in Kafka consumer configuration.

References

[1] AWS. What is apache kafka? https://aws.amazon.com/what-is/apache-kafka/. 2

[2] Lovisa Johansson. What is rabbitmq?, 17 Jan 2025. https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html. 1

[3] RabbitMQ. Networking and rabbitmq. https://www.rabbitmq.com/client-libraries/java-api-guide. 2