Cloud Computing - Mini Project Report

Micro-service communication with RabbitMQ

April 2023

Submitted in partial fulfillment of the requirements for the award of degree of

Bachelor of Technology

in

Computer Science & Engineering

UE20CS352 – CC Mini Project

Submitted by:

TEAM 14

SUJAN RAO R	**PES2UG20CS468**
UJJWAL TIKU	**PES2UG20CS475**
VIDHU NIRANJAN R	**PES2UG20CS480**
TARUN S	**PES2UG20CS474**

Under the guidance of

Prof. Jeny Jijo

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

PES UNIVERSITY

Short Description and Scope of the Project

PROBLEM STATEMENT:

Building and deploying a micro-services architecture where multiple components communicate with each other using RabbitMQ. A message broker is an architectural pattern for message validation, transformation and routing. For the scope of this project, we will build 4 micro-services: A HTTP server that handles incoming requests to perform CRUD operations on a Student Management Database + Check the health of the RabbitMQ connection, a micro-service that acts as the health check endpoint, a micro-service that inserts a single student record, a micro-service that retrieves student records, a micro-service that deletes a student record given the SRN.

PROPOSED SOLUTION:

We used Python and Flask library to receive TCP requests from clients, and Docker to host and run micro-services, including RabbitMQ which runs on a specific port. The Pika library is used to connect from the consumer files to the RabbitMQ queuing system. The producer is responsible for handling TCP requests, and it listens on Localhost port 5000 and directs the requests to their respective clients or consumers. We store data in MongoDB, which all consumers individually connect to on request.

To build and run our system, we define all dependencies in the Docker Compose file, which allocates ports for all producers and consumers. We then run the Docker Compose on the terminal with Docker open in the background to run the micro-services container.

Our system consists of four consumers:

- **Heartbeat:** Sends a TCP response to indicate that it is functioning correctly after verifying that it has received responses from all the consumer services to which it had sent a heartbeat signal.
- **Insert Record:** Connects to the MongoDB instance and inserts a record into the database with the specified credentials.
- **Delete Record:** Connects to the database and deletes the specified record in the request.
- **Read:** Connects to the database and retrieves all records present in it.

Overall, our system is designed to handle TCP requests efficiently using Python, Flask, RabbitMQ, and MongoDB, all packaged in a Docker container for easy deployment and scalability.

Methodology

- **Microservices architecture:** The project is designed using a microservices architecture where each service is developed, deployed, and scaled independently. This allows for greater flexibility and easier maintenance of the overall system.

- **Containerization:** Docker is used to containerize the microservices and database, which allows for consistent deployment across different environments and simplifies the deployment process.

- **Message Queuing:** RabbitMQ is used as a message broker to enable communication between the producer and consumers in a loosely coupled manner.

- **RESTful API:** The producer service exposes RESTful APIs that allow clients to interact with the system, such as sending requests to insert, read or delete data in the database.

- **Agile methodology:** The project is likely developed using an Agile methodology that emphasizes collaboration, iterative development, and continuous delivery. This allows for faster feedback and more rapid development cycles, which is especially important for complex and rapidly evolving projects.

Testing

The instructions to check if the microservices were communicating were as follows:

- The docker-compose file was run using the command “docker-compose up” and it was checked if all the containers were created. The HTTP server started running and the consumers were able to wait for messages through RabbitMQ queues.
- For consumer_one, a GET request was sent to the server with the appropriate path & argument. Requests were sent via Postman or cURL and it was checked if the message was transmitted via the queue. An example URL was "http://{IP-Address}/{health_check route}".
- For consumer_two, a POST request was sent to the server with the appropriate path & data to be sent. It was checked if the data was inserted into the database.
- For consumer_three, a GET database request was sent to the server with the appropriate path & argument. It was checked if the record was deleted in the DB.
- For consumer_four, a GET request was sent to the server with the appropriate path. It was checked the content of the DB.
- To end the process, the command “docker-compose down” was used.

Results and Conclusions

- The above project is a microservices-based application that uses Python, Flask, RabbitMQ, MongoDB, and Docker to create a distributed system that can handle HTTP requests and process them asynchronously through message queues. The system consists of a producer that receives requests and distributes them to four different consumers based on their type: health_check, insert_record, delete_record, and read_database.
- Each consumer listens to its own message queue and performs a specific action such as checking the health of the system, inserting a record into the database, deleting a record from the database, or retrieving all records from the database. The system is designed to be scalable and fault-tolerant, with multiple instances of each microservice running in containers.

- The use of Docker and Docker Compose allows for easy deployment and management of the application across different environments, while RabbitMQ provides a reliable and efficient messaging system for inter-service communication.
- Overall, this project demonstrates the power and flexibility of microservices architecture and how it can be used to create complex distributed systems that are both scalable and reliable.