Fr. Conceicao Rodrigues College of Engineering

Department of Computer Engineering
Academic Year 2022-23
Distributed Computing Lab (B.E. Computer Engineering)
LAB 4

Aim: To Implement a Message Queueing System

Introduction:

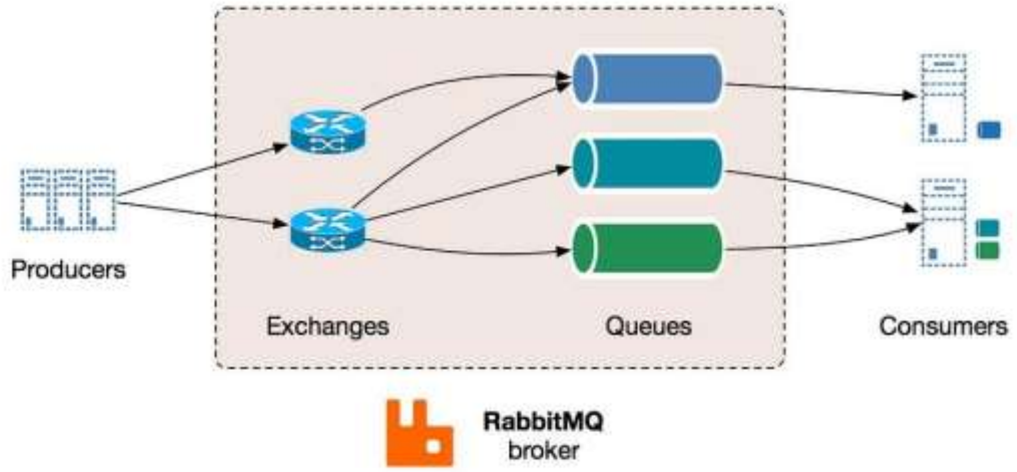
A message queueing system is a software architecture pattern that enables communication between different parts of a distributed system by allowing them to exchange messages. In this system, messages are stored in a queue and are retrieved by consumers when they are ready to process them. This decouples the producers and consumers of messages, allowing them to operate independently and asynchronously.

Message queueing systems can be implemented using various technologies, including opensource solutions like Apache Kafka, RabbitMQ, and ActiveMQ. In this practical we are going to implement a simple message queueing system using RabbitMQ.

RabbitMQ Model:

RabbitMQ is one of the most widely used message brokers, it acts as the message broker, “the mailman”, a microservice architecture needs. RabbitMQ consists of:

1. producer — the client that creates a message
2. consumer — receives a message
3. queue — stores messages
4. exchange — enables to route messages and send them to queues



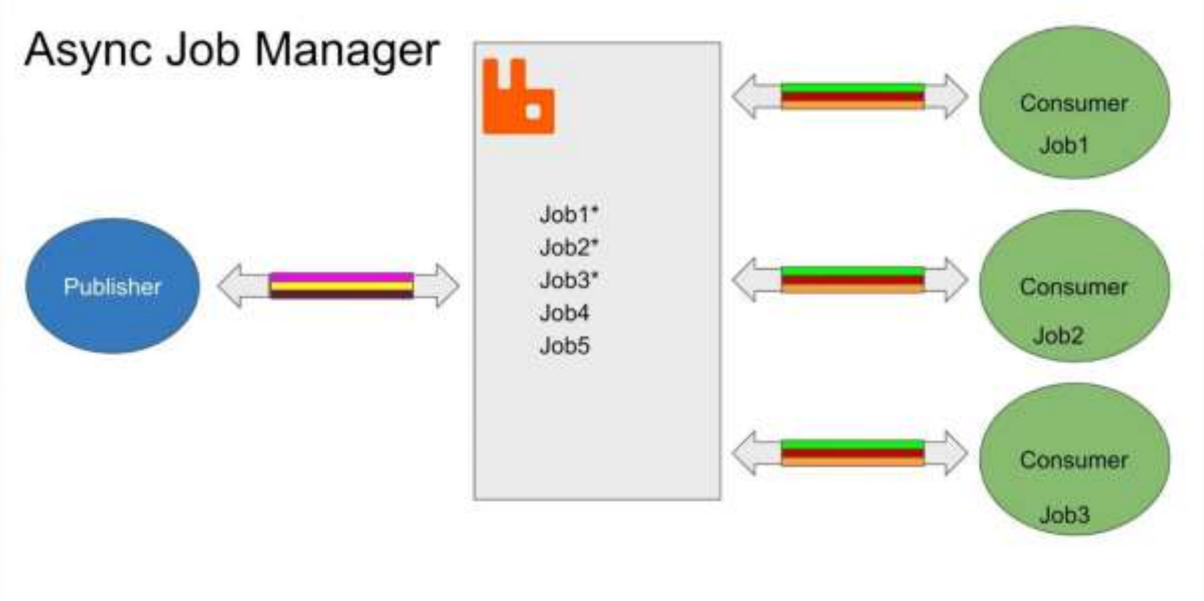
The system functions in the following way:

1. producer creates a message and sends it to an exchange
2. exchange receives a message and routes it to queues subscribed to it
3. consumer receives messages from those queues he/she is subscribed to

Implementation:

We are going to implement a job manager as described in the below figure. Components of our message queueing system are:

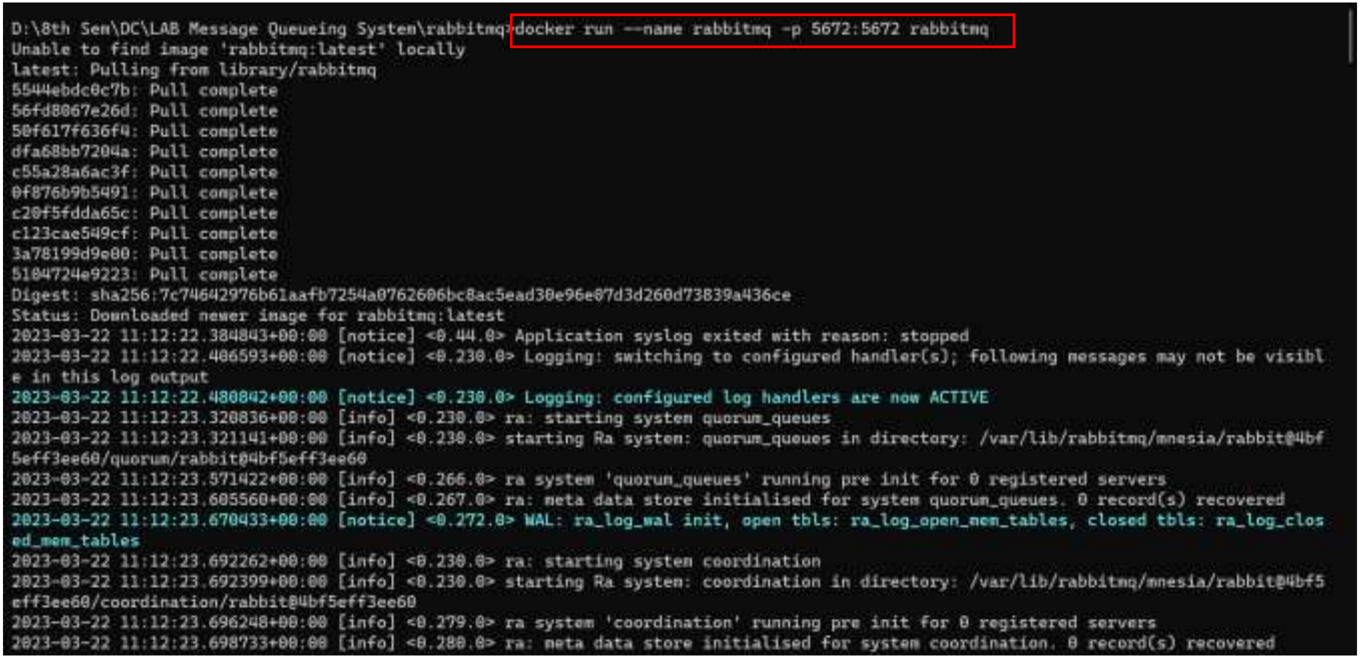
o Publisher – produces jobs/messages into the queue o Consumers – consumes the jobs o RabbitMQ broker – contains the exchange and queue o Connections – denoted by double-sided arrows o Channels – denoted by colourful bands within the connections



Technologies Used:

- Docker
- RabbitMQ Image
- Node.js
- amqplib Library

Step 1: Run RabbitMQ's Docker Image



```
D:\8th Sem\DC\LAB Message Queueing System\rabbitmq>docker run --name rabbitmq -p 5672:5672 rabbitmq
Unable to find image 'rabbitmq:latest' locally
latest: Pulling from library/rabbitmq
5544ebdc0c7b: Pull complete
56fd8067e26d: Pull complete
50f617f636f4: Pull complete
dfa68bb7204a: Pull complete
c55a28a6ac3f: Pull complete
0f876b9b5491: Pull complete
c20f5fdda65c: Pull complete
c123cae549cf: Pull complete
3a78199d9e00: Pull complete
5104724e9223: Pull complete
Digest: sha256:7c74642976b61aafb7254a0762606bc8ac5ead30e96e07d3d260d73839a436ce
Status: Downloaded newer image for rabbitmq:latest
2023-03-22 11:12:22.384843+00:00 [notice] <0.44.0> Application syslog exited with reason: stopped
2023-03-22 11:12:22.406593+00:00 [notice] <0.230.0> Logging: switching to configured handler(s); following messages may not be visibl
e in this log output
2023-03-22 11:12:22.480842+00:00 [notice] <0.230.0> Logging: configured log handlers are now ACTIVE
2023-03-22 11:12:23.320836+00:00 [info] <0.230.0> ra: starting system quorum_queues
2023-03-22 11:12:23.321141+00:00 [info] <0.230.0> starting Ra system: quorum_queues in directory: /var/lib/rabbitmq/mnesia/rabbit@4bf
5eff3ee60/quorum/rabbit@4bf5eff3ee60
2023-03-22 11:12:23.571422+00:00 [info] <0.266.0> ra system 'quorum_queues' running pre init for 0 registered servers
2023-03-22 11:12:23.605560+00:00 [info] <0.267.0> ra: meta data store initialised for system quorum_queues. 0 record(s) recovered
2023-03-22 11:12:23.670433+00:00 [notice] <0.272.0> WAL: ra_log_wal init, open tbls: ra_log_open_mem_tables, closed tbls: ra_log_clos
ed_mem_tables
2023-03-22 11:12:23.692262+00:00 [info] <0.230.0> ra: starting system coordination
2023-03-22 11:12:23.692399+00:00 [info] <0.230.0> starting Ra system: coordination in directory: /var/lib/rabbitmq/mnesia/rabbit@4bf5
eff3ee60/coordination/rabbit@4bf5eff3ee60
2023-03-22 11:12:23.696248+00:00 [info] <0.279.0> ra system 'coordination' running pre init for 0 registered servers
2023-03-22 11:12:23.698733+00:00 [info] <0.280.0> ra: meta data store initialised for system coordination. 0 record(s) recovered
```

Step 2: Write a Producer Program - *publisher.js*

```
const amqp = require("amqplib"); const
msg = {number : process.argv[2]}
connect()
 async function connect() {     try{         const connection = await
amqp.connect("amqp://localhost:5672");         const channel = await
connection.createChannel();         const result = await
channel.assertQueue("jobs");         channel.sendToQueue("jobs",
Buffer.from(JSON.stringify(msg)))         console.log(`Job sent successfully
${msg.number}`)
    }     catch(err){
console.error(err)
    }
}
```

o A Node Library named "amqplib" is used to implement AMQP (Advanced Message Queueing Protocol) o We then create a connection with the RabbitMQ server.

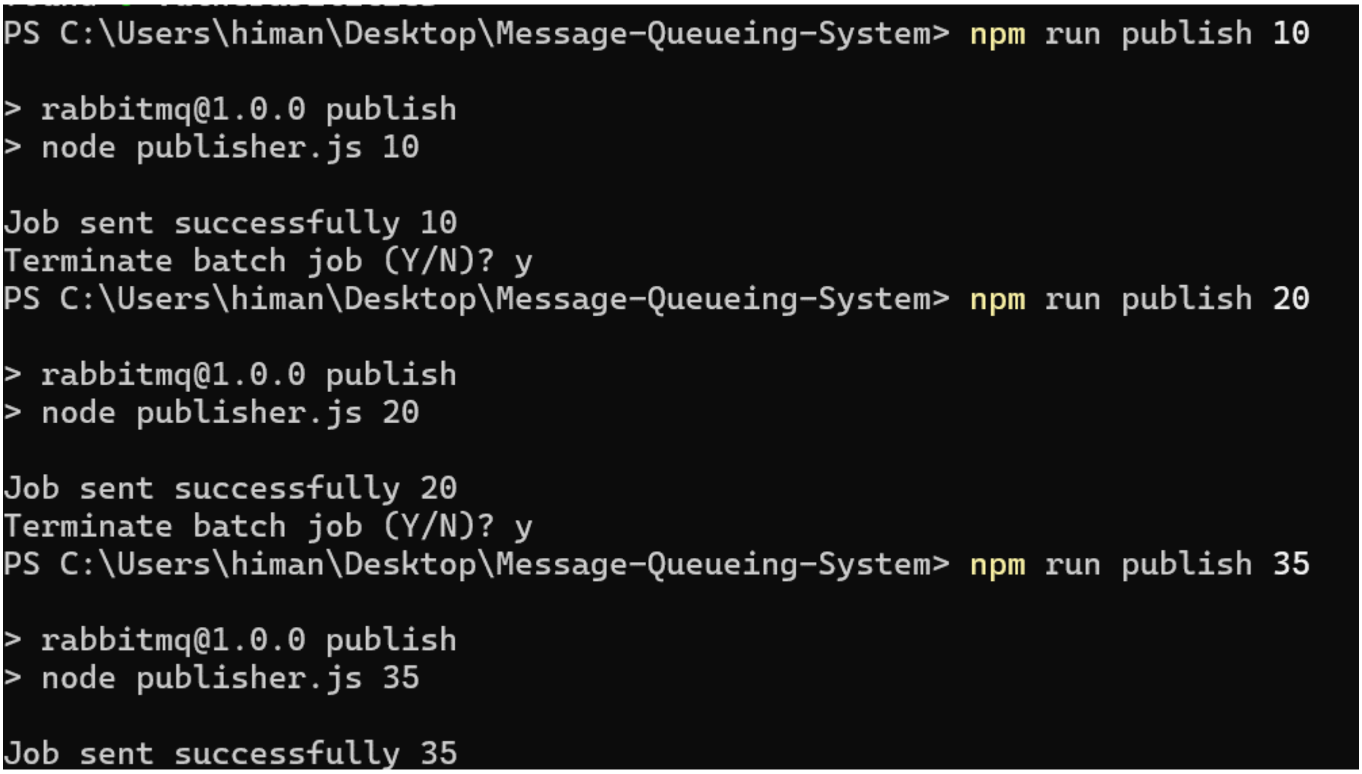
o Then a channel is created using connection's createChannel() function o This channel is used to create a new queue named "jobs" which resides within our RabbitMQ broker o A new message is enqueued within the queue. In other words, a new job is produced. The content of this message is provided as a command line argument when we run our producer program

Step 3: Write a Consumer Program – *consumer.js*

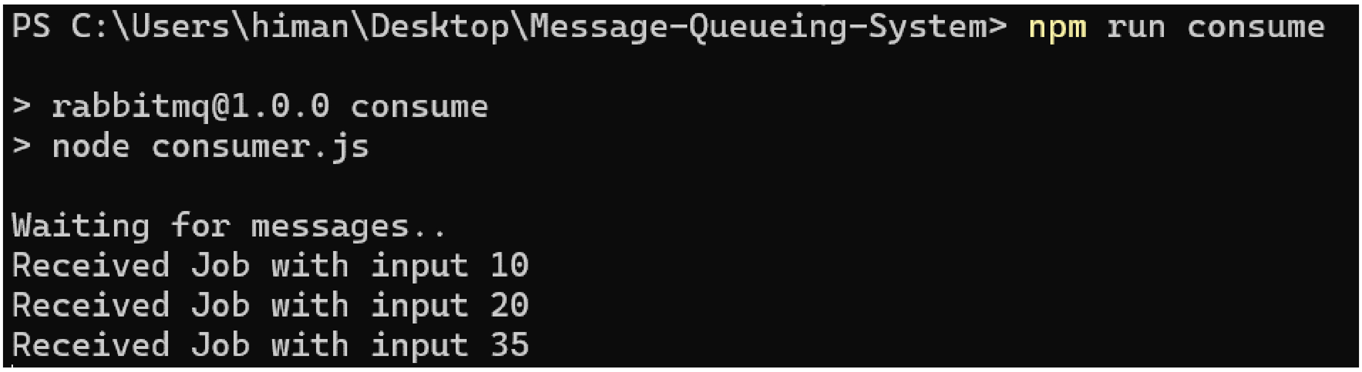
```
const amqp = require("amqplib");

connect();
 async function connect() {   try {     const connection = await
amqp.connect("amqp://localhost:5672");     const channel = await
connection.createChannel();     const result = await
channel.assertQueue("jobs");
     channel.consume("jobs", message => {
         const input = JSON.parse(message.content.toString());
         console.log(`Received Job with input ${input.number}`)
if(input.number == 22){             // Process Job number 10
channel.ack(message)
        }


    })     console.log("Waiting for messages..")
  } catch (err) {     console.error(err);
  }
}
```

- Here too, we create connection and channel the same way as in our publisher.js program
- Then we write functionality to consume the messages already present in the queue o Let us say that our consumer only consumes message number 22. Hence, if the queue has a message number 22, it will be consumed by the consumer and an acknowledgement will be passed to the RabbitMQ server. Subsequently the message number 22 will be dequeued

Step 4: Testing our system
Running Producer – publisher.js



```
PS C:\Users\himan\Desktop\Message-Queueing-System> npm run publish 10

> rabbitmq@1.0.0 publish
> node publisher.js 10

Job sent successfully 10
Terminate batch job (Y/N)? y
PS C:\Users\himan\Desktop\Message-Queueing-System> npm run publish 20

> rabbitmq@1.0.0 publish
> node publisher.js 20

Job sent successfully 20
Terminate batch job (Y/N)? y
PS C:\Users\himan\Desktop\Message-Queueing-System> npm run publish 35

> rabbitmq@1.0.0 publish
> node publisher.js 35

Job sent successfully 35
```



```
PS C:\Users\himan\Desktop\Message-Queueing-System> npm run consume

> rabbitmq@1.0.0 consume
> node consumer.js

Waiting for messages..
Received Job with input 10
Received Job with input 20
Received Job with input 35
```

Conclusion:

- Message queueing systems, its need, architecture, and implementation were understood
- A simple message queueing system was designed and executed using RabbitMQ message broker.

Github Link: https://github.com/ravisinghk/Message-Queueing-System

References:

- *What is a Message Queue and When should you use Messaging Queue Systems Like RabbitMQ and Kafka*. (2020, May 2). YouTube. https://www.youtube.com/watch?v=W4_aGb_MOls
- *What is RabbitMQ?* (2020, November 10). YouTube. https://www.youtube.com/watch?v=7rkeORD4jSw

- *RabbitMQ Crash Course*. (2019, October 18). YouTube. https://www.youtube.com/watch?v=Cie5v59mrTg
- *8 Basic Docker Commands || Docker Tutorial 4*. (2019, October 28). YouTube. https://www.youtube.com/watch?v=xGn7cFR3ARU
- Peng Yang, L. (2022, December 4). *System Design—Message Queues*. Medium. https://medium.com/must-know-computer-science/system-design-message-queues245612428a22

Postlab Questions:

1.What is message Queueing?

2.What are the benefits of message Queueing?