Name:Atharva Pawar Roll No:-9427 Exp:4 BE COMPS A

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 open- source solutions like Apache Kafka, RabbitMQ, and ActiveMQ. In this practical we are going to implement a simple message queueing system using Python.

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

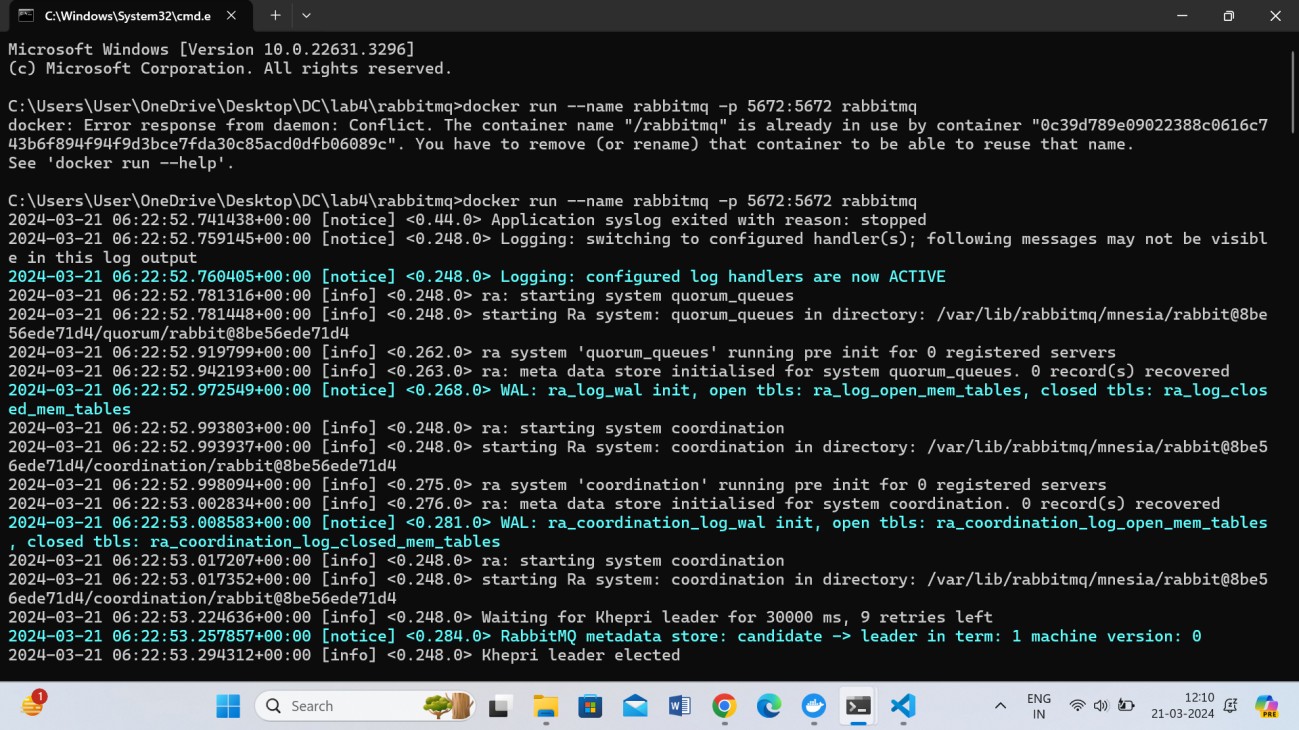
The system functions in the following way:

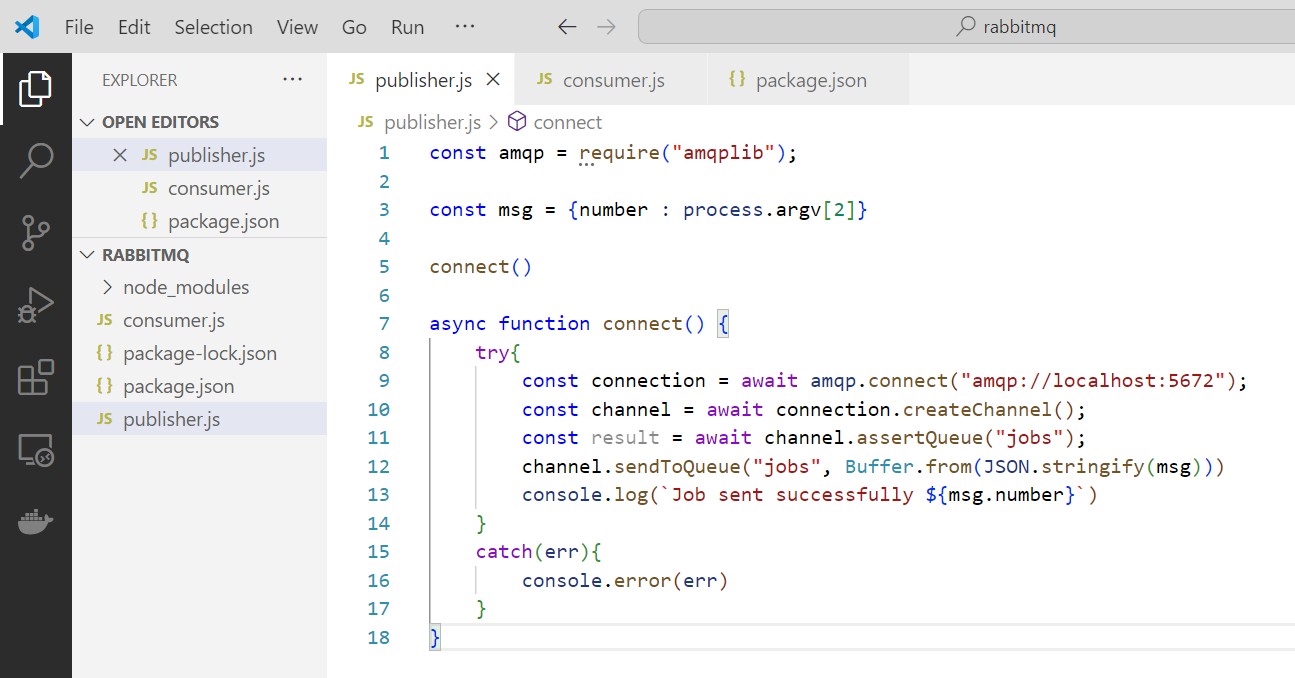
1. producer creates a message and sends it to an exchange
2. exchange receives a message and routes it to queues subscribed to it
3. consumer receives messages from those queues he/she is subscribed to Implementation:

We are going to implement a job manager as described in the below figure.

Components of our message queueing system are:

* Publisher – produces jobs/messages into the queue
* Consumers – consumes the jobs
* RabbitMQ broker – contains the exchange and queue
* Connections – denoted by double-sided arrows
* Channels – denoted by colourful bands within the connections Technologies Used:
* Docker
* RabbitMQ Image
* Node.js
* amqplib Library

Step 1: Run RabbitMQ’s Docker Image

Step 2: Write a Producer Program - publisher.js

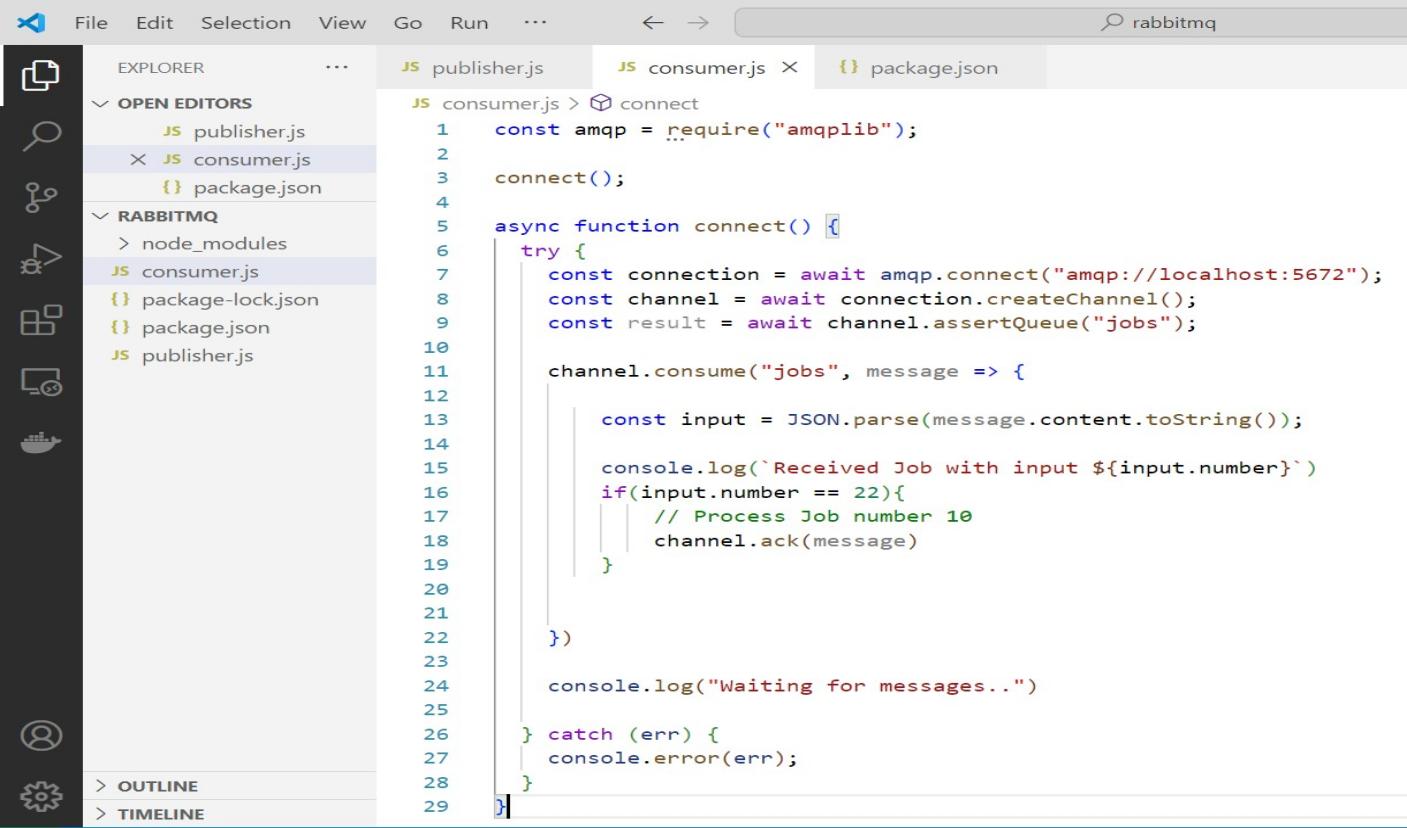
* A Node Library named “amqplib” is used to implement AMQP (Advanced Message

Queueing Protocol)

* We then create a connection with the RabbitMQ server.
* Then a channel is created using connection’s createChannel() function
* This channel is used to create a new queue named “jobs” which resides within our

RabbitMQ broker

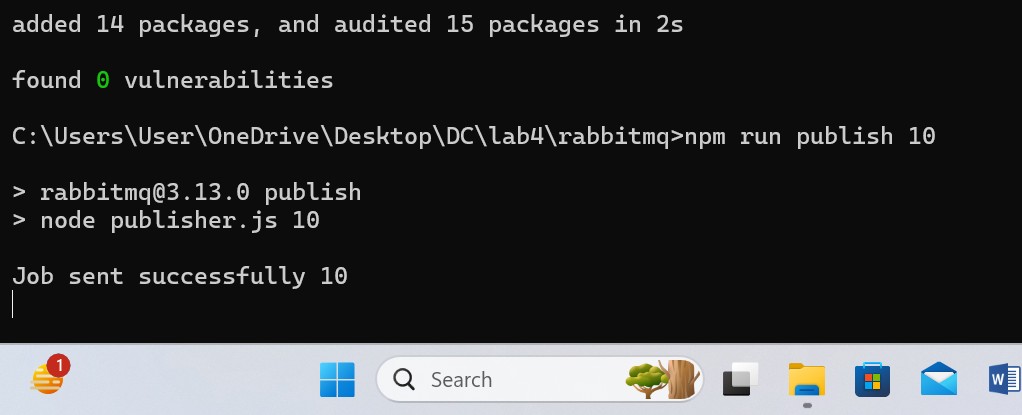
* A new message is enqueued within the queue. In other words, a new job is produced. The content of this message is provided as a command line argument when we run our producer program

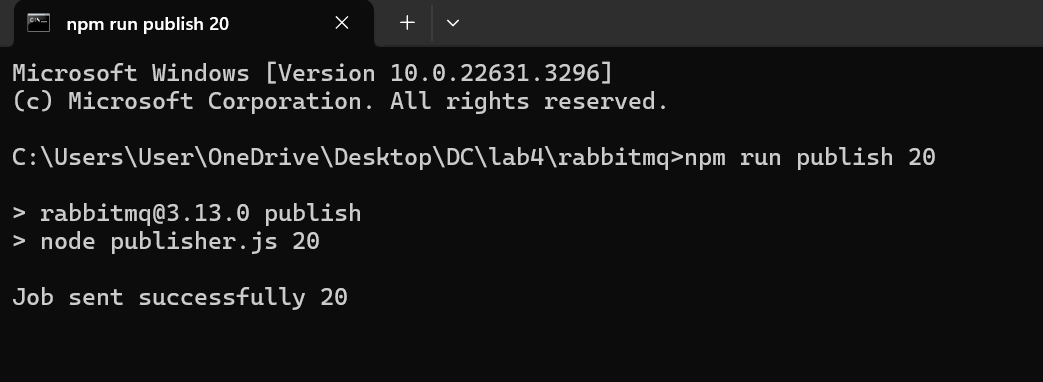
Step 3: Write a Consumer Program – consumer.js

Here too, we create connection and channel the same way as in our publisher.js program

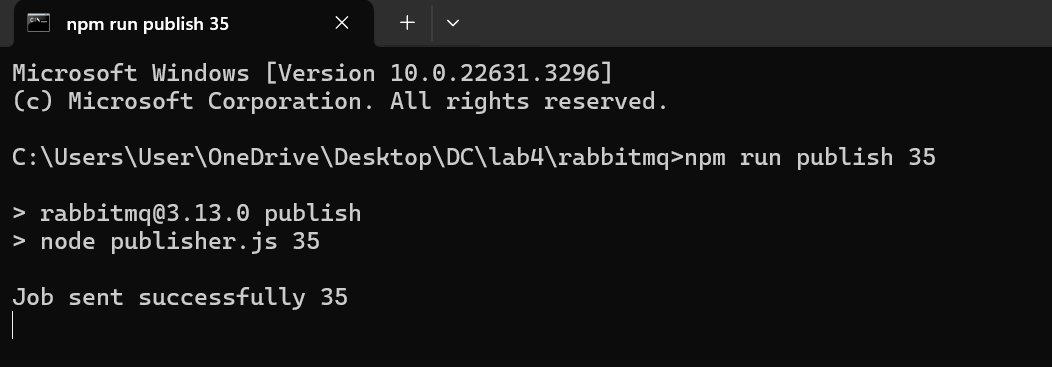
* Then we write functionality to consume the messages already present in the queue
* Let us say that our consumer only consumes message number 22. Hence, if the queue has a message number 22, it will be consumed by the consumer and an acknowledgement will be passed to the RabbitMQ server. Subsequently the message

number 22 will be dequeued Step 4: Testing our system Running Producer – publisher.js

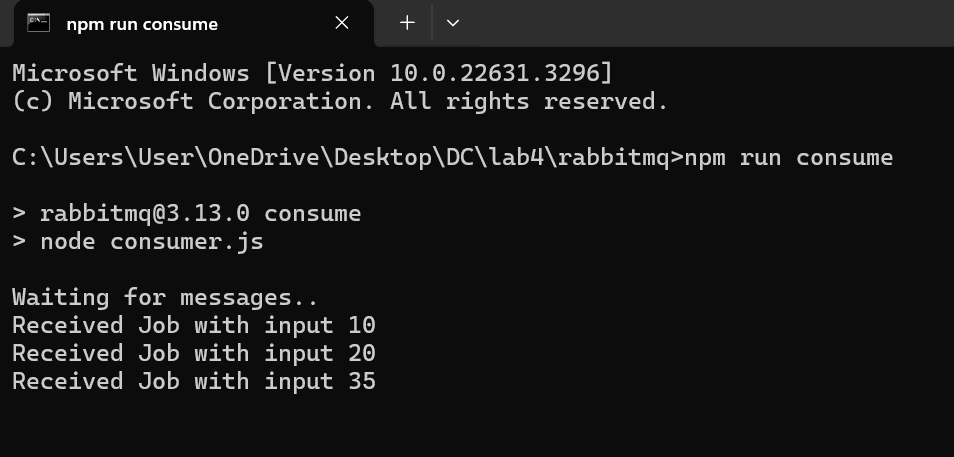
Publish job 10

Publish job 20

Publish job 35



Running Consumer – consumer.js All the jobs displayed:



Conclusion:

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

Postlab Questions:

1. What is message Queueing?
2. What are the benefits of message Queueing?

