Journal of Electronics and Informatics (ISSN: 2582-3825)
www.irojournals.com/iroei/

Implementation of Message Service Queue Using Rabbit MQ

Deepti Ravi Kumar[1], Nandha Kishore[2], D. Rahul Raj[3], E. A. Raswanth[4], Samyuktha Sreekanth[5], S. Sruthi[6], T. Anusha[7]

Department of Computer Science and Engineering, PSG College of Technology, Coimbatore, India

E-mail: [1]19z210@psgtech.ac.in, [2]19z228@psgtech.ac.in, [3]19z234@psgtech.ac.in, [4]19z236@psgtech.ac.in, [5]19z240@psgtech.ac.in, [6]20z434@psgtech.ac.in, [7]anu.cse@psgtech.ac.in

Abstract

A distributed system is a software application that makes use of a collection of protocols to manage the activities of numerous processes running on a communication network, so that each part cooperates to finish a single or a condensed number of related tasks. Representational State Transfer (REST) API is also integrated into this, making it simple to access online services without the need for additional processing. The server sends a client-side representation of the requested resource whenever a RESTful API is used. It is also integrated with microservices through the open-source message-broker programme RabbitMQ. With little developer involvement, the configuration file must automatically network new messaging services that join the distributed system.

Keywords: Distributed computing , MOM , microservices

1. Introduction

The subject of this research uses distributed computing, which splits a problem into numerous tasks and uses Message-Oriented Middleware (MOM), to complete each task. MOM is used in order for the client and the service provider to interact asynchronously. It must also be coupled with message queues for software architectural style known as Representational State Transfer Calls (REST API). Microservices, a variation of the service-oriented architecture design using RabbitMQ, incorporate REST API requests. The Open Telecom Platform clustering and failover framework serves as the foundation for the RabbitMQ server application. For all widely used programming languages, client libraries are available to interface with the broker. The REST requirements may not be strictly adhered by http-based APIs.

DOI: https://doi.org/10.36548/jei.2023.1.001
Received: 11.12.2022, received in revised form: 12.01.2023, accepted: 29.01.2023, published: 09.02.2023

2. Literature Review

Paper [1] focused on the technological fields that are expanding, and the fastest is computer vision. The paper provided a comparative analysis of numerous research efforts completed by several researchers and delivered a thorough study on the topic of object detection. The article also serves as a straightforward dictionary for numerous studies in the area of 3D object identification. It largely focused on the well-known applications of real-world object detection in the fields of medicine, the military, and crowd control. The report presented a comprehensive comparison of convolution neural network-based methods. The paper also recommended an effective multi item detection method.

Paper [3] focused on the AMQP protocol and the various RabbitMQ exchange types. The open-source messaging platform RabbitMQ facilitates the integration of numerous applications via messages and queues. The RabbitMQ server, a message broker, acts as the message coordinator for the applications intended to integrate. Thus, a platform for transmitting and receiving messages between the systems may exist. AMQP allows clients to talk to middle wares. Direct exchange, Fanout exchange, and Topic exchange are the many exchange kinds covered.

Paper [4] focused on the applications that can share data and communicate with one another by sending and receiving messages, thanks to MOM, with benefits of loose connection between players, asynchronous and multi-point transmission, etc. In this backdrop, the work examined the potential and difficulties that MOM may encounter as they adjust to new needs and a more complex environment. Paper [5] focused on RESTful web services that offer an architectural framework for creating web services and a method for clients to use those APIs. The http-based APIs might not adhere to all of the REST restrictions. Designing a solution for API validation was the driving force behind this work. The method determined whether the implementation was created in accordance with the specifications outlined in the relevant API’s specification paper. The work also investigated difficulties in REST API analysis and implementation validation. The technique took into account the RESTful web API implementation's Open API Specification.

Paper [6] focused on the architecture of microservices that are gaining interest in academia and business, and is frequently contrasted with monolithic design. Regarding the performance of various architectures, many of the findings of these research publications are at odds with one another. As a result, these two architectural designs were contrasted in this article, and several particular microservices application configurations were also assessed in terms of service discovery. In concurrency testing, monolithic architecture outperformed microservices design by 6% in terms of throughput. The two architectures did not significantly differ in the load testing scenario. Additionally, a third test comparing microservices systems created with other service discovery platforms, such as Consul and Eureka, revealed that apps using Consul displayed improved throughput results.

Paper [7] focused on the communication middleware, which connects the system's diverse components and controls how they interact, which is the primary element of distributed computing. The client and service provider can communicate synchronously via Remote Procedure Call (RPC) middleware. In order to interact asynchronously between the client and the service provider, MOM makes use of messaging. For the purpose of this study, an analytical M/M/1 model with a First-In, First-Out (FIFO) policy and a priority queuing model were devised and created. The models examined the throughput of various communication paradigms and contrasted how well RPC and MOM perform with priority queuing.

3. Concepts and Technologies Used

3.1 Rabbit MQ

RabbitMQ Message Queuing Software is also known as Message Broker or Queue Manager. This software controls which applications connect to transmit one or more messages and queues them.

Message exchange in RabbitMQ:

- A message is posted to an exchange by the producer. The type must be given when creating an exchange.
- After receiving the message, the exchange is now in charge of routing it. Depending on the exchange type, the exchange considers various message properties, such as the routing key.
- The exchange must have bindings established to queues. In this instance, there are two bindings, each to a distinct exchange queue. Depending on the message's properties, the exchange directs the message into the queues.
- Until a consumer responds to the messages, they remain in the queue.

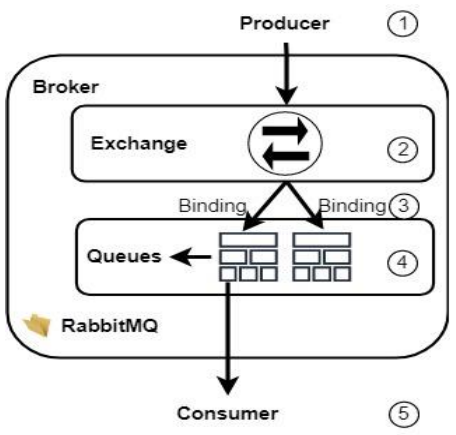


Figure 1. Message Flow using Rabbit MQ

In Fig.1, the message published by the producer goes on to the binding which in turn is bound to the queues and is consumed by the consumer.

3.2 Rest API

The Representational State Transfer (RESTful API) design is built to take advantage of current protocols. Almost any protocol can be used with REST, but HTTP is typically used for web APIs. The REST API design does not require developers to install any additional software or libraries to use it. REST API is a 2000 paper by Dr. Roy Fielding. Its amazing flexibility is one of its key qualities. REST can handle different kinds of calls, return data in different formats, and even change the structure if hypermedia is used correctly. The REST API acts as an interface and adheres to the constraints of the REST proposed architecture. In the separation of client and backend service, if the endpoint is available on the 4300, the backend service doesn't need to worry about implementing the 4000 clients. By simply calling multiple clients, all clients with the necessary permissions to access the resource will have access. A REST implementation allows developers to have different implementations of her for different clients, using different technologies in different code bases. The various clients that access the common REST API are mobile browsers, desktop browsers, tablets, or API testing tools. Introducing a new client type or changing the client code does not affect the functionality of the backend service. This means that the client and backend service are decoupled.

3.3 Microservices

Software is developed using an architectural and organizational strategy known as microservices. These services are owned by a small independent group. Microservices architecture drives' innovation, accelerates application scalability and development, and reduces time to market for new features. In a monolithic environment, all processes are tightly coupled and run as a single service. If the demands on an application's processes increase significantly, the overall architecture must be scaled. The ability of a monolithic application to add or extend functionality becomes increasingly difficult as the code base grows. This complexity limits experimentation and makes it difficult to put new concepts into practice. Due to the large number of interdependent and tightly coupled processes, monolithic architectures increase application availability risks. Applications are built as separate components, running each application process as its service in a microservices architecture. These services use lightweight APIs to communicate through well-defined interfaces. Services are created for business purposes and each service performs only one task. Because they operate independently, each service can be modified, invoked, and extended to meet specific application functional needs.

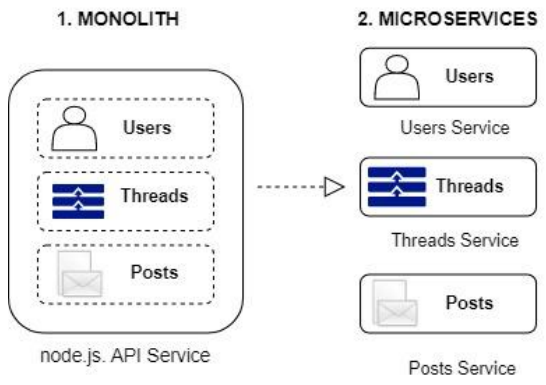


Figure 2. Breaking Monolithic into Microservices

In Fig.2, the entire node.js application runs in a container as a single service under monolithic architecture, which is transformed into microservices, each of which operates as a distinct service within different containers as a part of the application.

3.4 Characteristics of Microservices

1. Autonomous microservices architecture allows for the development, deployment, operation, and scaling of each component service without affecting the functionality of other services. Services are not required to exchange implementation or code with one another. The only means of communication between the various parts is through a clear API.
2. Each specialized service is built for a certain set of features and focused on resolving a particular issue. A service can be divided into smaller services as additional code is added by developers over time and the service grows more complicated.

3. Agility small, unbiased groups that take ownership of their services can grow their business, thanks to microservices. Teams are empowered to work more independently and quickly within a constrained and well-understood context.

3.5 Message Oriented Middleware

MOM refers to the concept of sending data between programs via a communication channel that carries self-contained information. In MOM-based communication systems, messages are transmitted and received asynchronously, and applications are abstractly separated through message-based communication. The sender and receiver are not aware of each other and use APIs to communicate within the messaging system via messaging clients offered by the MOM provider. A MOM system includes a MOM provider that employs various architectures for message delivery and routing and may use a centralized messaging server or distributed routing and delivery operations, or a combination of the two. Performance can be monitored and optimized by adding a management interface into messaging providers that is broker messaging between clients.

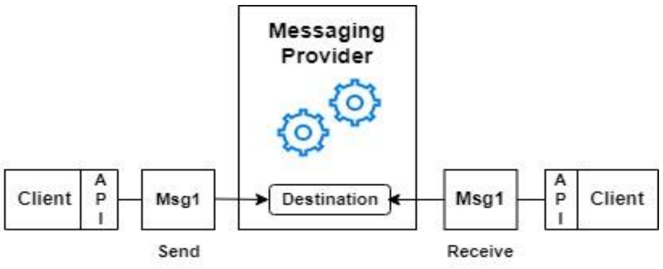


Figure 3. Message based System

In Fig.3, to send a message to a provider-managed destination, the client uses an API request. The call activates provider services to transmit the message and route it.

For its messaging operations, MOM makes use of a message service. The components of a MOM system are, a MOM provider that uses a range of architectures for message delivery and routing, and for communication between client computers, either a centralised messaging server or distributed routing and delivery operations can be employed. Some MOM products integrate the two strategies.

3.6 Distributed Enterprise Computing

Distributed enterprise computing solves computing problems by using distributed systems. The problem can be divided into subsequent tasks, solved using one or more computers which communicate by message passing. In this architecture, components are present on different platforms and several components can work with one another over a communication network to achieve a specific objective.

The most crucial duties are:

- Resource sharing, including the ability to exchange hardware, software, and data.
- Openness: How freely is the programme intended to be created and distributed?
- Concurrency: The simultaneous processing of a single function by several machines.
- Scalability: How can computing and processing power scale when applied to a large number of computers?
- Fault tolerance: How fast and easily can system problems in individual components be found and repaired?

3.7 Architecture

The Distributed Enterprise Connectivity architecture addresses issues such as connectivity, security, end-user performance, visibility, and manageability in enterprise computing. To enhance branch productivity and avoid disruptive service interruptions, businesses require a new approach to branch connectivity. By combining various services, such as security, phone, data, and access servers, into a single, remotely manageable platform, a service gateway can simplify branch networking and reduce the Total Cost of Ownership (TCO). This defines a new category of products known as Enterprise Reference Architecture, which develops a model that considers the company's requirements and the number of users at a particular branch location.

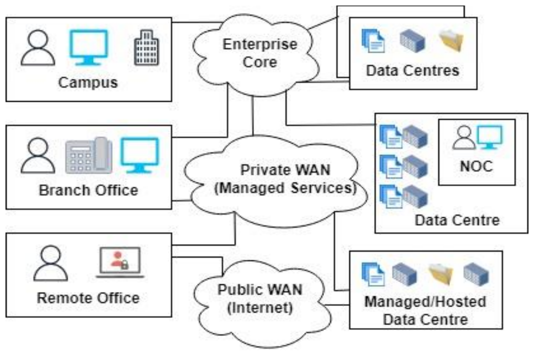


Figure 4. Distributed Enterprise Computing

In Fig.4, a high-level perspective of the distributed enterprise connectivity architecture is shown. It outlines the architectural specifics for each redundancy level and technological solutions for all branch office profiles.

4. Overview

A problem is broken into numerous tasks, each of which is addressed by MOM, which is a system that facilitates data interchange and communication (messages). It entails the transfer of data across apps via a communication channel that sends independent informational units (messages). This will be combined with a message queue for REST API, a software architectural style established to assist the design and development of the World Wide Web architecture. The rest API calls are included amongst microservices, a type of service-oriented architecture that employs RabbitMQ, an opensource message broker. Messages are sent to a queue by the publisher. Using the TCP/IP Protocol, the queue is used to store all messages in a FIFO structure. One or more users can be linked together.

The messages are dequeued one by one for a single user, the information is analysed, and the user takes the necessary action. Until a consumer is connected to dequeue the messages, they remain in the queue. To communicate between the producer and the client, a REST API gateway is employed. To minimize developer interaction, a configuration file is produced. The purpose is to automatically configure and add each new messaging service that enters the network, whether it's a producer or a consumer.

5. Advances over Earlier Techniques

Many real-time streaming data processing applications use distributed message systems as the foundation of their communication infrastructure. Distributed message systems offer a sizable number of parameters for configuration in order to support the enormous variety of such applications. Properly configuring these parameters for better performance, however, overwhelms the majority of users. Therefore, the process of configuring distributed message systems needs to be made simpler.

Auto-config files represents a novel idea. To maintain minimal developer intervention, an auto-configuration file must be created that takes the specifics of the new message service as parameters and configures them for the network. The file automatically configures any new message service that joins the network. By lowering the likelihood of human error during the configuration process, this auto-configuration file will help reduce the amount of manual labour required to set these parameters. Any new messaging service that joins the network can have its settings adjusted by the auto-configuration file to ensure peak performance. A new message service's parameters can also be optimized using the auto-configuration file based on the particular message service's unique requirements (point-to-point message participant, broadcast message participant, etc.).

6. Methodology

6.1 publish.js

Publisher is a messaging user application. The editor has different targets for each protocol. Posts must be forwarded to queues (topics, etc.). Queues can contain online "consumers". Consumers that successfully queued up can accept more deliveries. The consumer receives the message. A failed attempt will occur if tried to publish to an empty queue and throws an exception at the channel level with the error message "404 Not Found" and destroys the channel attempted to close. Publishers frequently lead long lives. Publishers usually open a connection. Publishers make sure to provide the mechanism to application developers to track messages successfully accepted by RabbitMQ. In Publish Wait, a message is posted and immediately waited for a pending message. A confirmation will be received. It is considered as the batch publishing strategy, where the stack size is equal to 1.

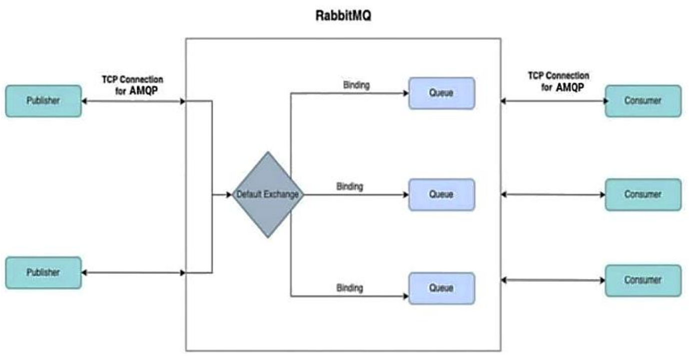


Figure 5. Publisher-Consumer Connection using RabbitMQ

6.2 consumer.js

An application or instance of an application that consumes messages is referred to as a consumer. The consumer consumes from the queue. To use messages, there must be a tail. When new consumers are added, it is assumed that there are existing messages in queue; the delivery will occur when new mail is requested. A channel level exception with the error code (404 Not Found) is raised if a non existent queue is tried to be used, and the channel that tried to terminate is returned. Consumers are expected long-term, which means that during the lifetime of the consumer, he receives a number of deliveries. General consumer is saved when the application starts.

6.3 config.js

This configuration file is designed to maintain developer interference and automatically register any new message services that join the network. This auto configuration file must be constructed in a manner that allows the network to be configured with the mail service's specifics as a parameter.

6.4 Exception Handling

Consumers are expected to manage any exceptions that come up when processing deliveries or performing other consumer-related tasks. These exceptions need to be documented and recorded. If there are no dependencies and the consumer is unable to accept delivery, or in similar situations, it won't be taken into consideration. This must be expressly stated and a temporary break is taken from communication. As a result, it does not employ the RabbitMQ monitoring system. If this condition is breached, for example if the method was called with an unusual set of parameters, an exception handling mechanism enables the function to throw the exception and deal with it. The mechanism for managing exceptions then handles the exception. A prerequisite's and an exception's specification are both arbitrary. Programmers generally determine what constitutes "typical" situations; for example, they may decide that division by zero is improper and hence an exception, or they may build a behaviour like returning zero or a particular "ZERO" result.

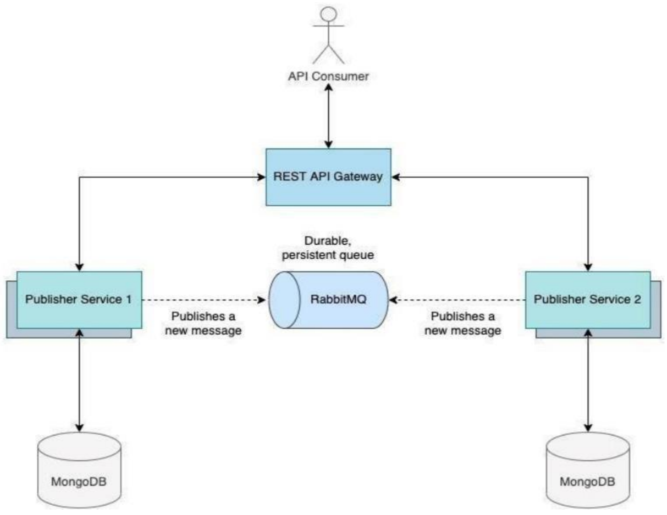


Figure 6. Connection of REST API

In Fig.5, Advanced Message Queueing Protocol is used to create a TCP connection between the producer and consumer, so that messages may be exchanged via the binding queue.

In Fig.6, by sending HTTP queries to the broker, the REST API connects to RabbitMQ which in turn enables the publishers and the consumers to send and receive messages via exchanges and queues, respectively.

7. Implementation and Results

7.1 Program modules

a) publish.js:

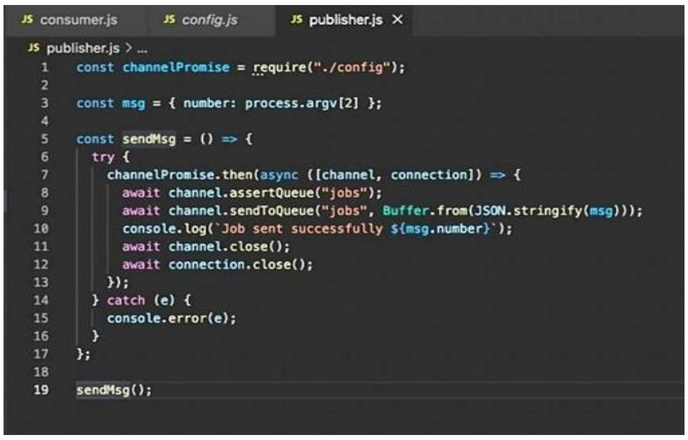
```
constchannelPromise = require("./config");
constmsg = { number: process.argv[2] };
constsendMsg = () => { try {
channelPromise.then(async ([channel, connection]) => { await channel.assertQueue("jobs");
await channel.sendToQueue("jobs", Buffer.from(JSON.stringify(msg)));
console.log(`Job sent successfully ${msg.number}`);
await channel.close();
await connection.close();
});
} catch (e) { console.error(e);
}
};
sendMsg();
```

b) consumer.js:

```
constchannelPromise= require("./config");
constreceiveMsg = () => { try { channelPromise.then(async ([channel, connection]) => { 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 == 7) channel.ack(message);
});
 console.log("Waiting for messages...");
});
} catch (e) { console.error(e);
} };
 receiveMsg();
```

c) config.js:

```
constamqp = require("amqplib");
require("dotenv").config(); const connect = async () => { try { const connection = await
amqp.connect(process.env.AMQPSERVER_API);
const channel = await connection.createChannel(); r
eturn [channel, connection]; } catch (e) { console.error(e);
} };
module.exports = connect()
```



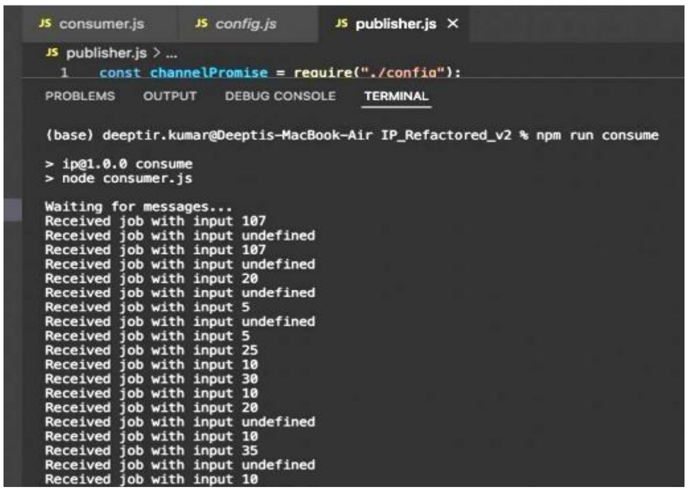
```
const channelPromise = require("./config");

const msg = { number: process.argv[2] };

const sendMsg = () => {
  try {
    channelPromise.then(async ([channel, connection]) => {
      await channel.assertQueue("jobs");
      await channel.sendToQueue("jobs", Buffer.from(JSON.stringify(msg)));
      console.log(`Job sent successfully ${msg.number}`);
      await channel.close();
      await connection.close();
    });
  } catch (e) {
    console.error(e);
  }
};

sendMsg();
```

Figure 7. Publishing message into queue



```
const channelPromise = require("./config");
```

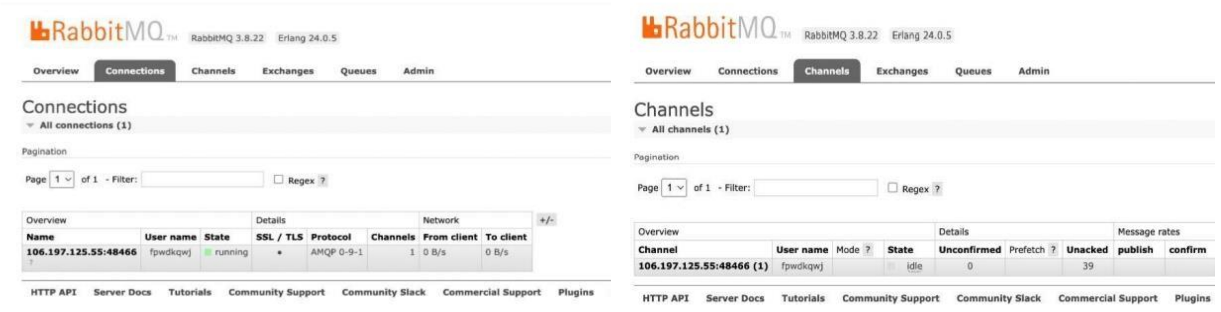
```
(base) deeptir.kumar@Deeptis-MacBook-Air IP_Refactored_v2 % npm run consume

> ip@1.0.0 consume
> node consumer.js

Waiting for messages...
Received job with input 107
Received job with input undefined
Received job with input 107
Received job with input undefined
Received job with input 20
Received job with input undefined
Received job with input 5
Received job with input undefined
Received job with input 5
Received job with input 25
Received job with input 10
Received job with input 30
Received job with input 10
Received job with input 20
Received job with input undefined
Received job with input 10
Received job with input 35
Received job with input undefined
Received job with input 10
```

Figure 8. Consuming messages from queue

7.2 Rabbit- MQ Platform Usage



Overview			Details			Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
106.197.125.55:48466 ?	fpwdkqwj	🟩 running	●	AMQP 0-9-1	1	0 B/s	0 B/s

Overview				Details			Message rates	
Channel	User name	Mode ?	State	Unconfirmed	Prefetch ?	Unacked	publish	confirm
106.197.125.55:48466 (1)	fpwdkqwj		idle	0		39		

Figure 9a and 9b. Rabbit MQ platform Components

In Fig.9a and 9b, the connections and channels tabs display the different connections made to the RabbitMQ server, and the information about all the present channels, respectively. Further by clicking on any connection or channel, a detailed summary of the respective connection or channel is provided.

8. Experimental Analysis

If two or more core modules need to communicate with each other, they should not make direct HTTP calls. This is because they can tightly couple the core layers, and more instances of each core module can become difficult to manage. Also, if the service goes down, the HTTP call pattern will fail because there is no way to track old HTTP request calls after a restart. Therefore, a RabbitMQ that makes a microservice consume data from one queue and publish the result to another queue is required. As a result, a series of microservices handle requests. A message queue consists of two components, the first is a producer responsible for pushing incoming messages onto the queue, and the second component is the consumer used to retrieve these messages. A producer-side microservice sends a message to a queue. If the consumer-side microservice fails, it will be unable to generate and push messages, resulting in complete data loss. But for this queue, each request is persisted. When a consumer microservice becomes active, it starts processing all requests received by its queue of messages. After retrieving data from the queue, an acknowledgment is sent to the producer. If a microservice stops working, the rest of the application won't crash, only part of it will be inaccessible. Whenever a new microservice needs to be added to the network, the configuration created will automatically add the microservice to the network with minimal developer intervention.

9. Future Scope

In the future of microservices, the integration layer that links several microservices will get more and more attention. Microsoft has already investigated the potential for creating applications by fusing different APIs and services. With better tools for swiftly creating and implementing microservices, there will be no longer be a need for a substantial upfront expenditure. Another advantage of microservices over monolithic design, which requires scalability of the entire application, is the ability of separate components to scale at various speeds. In addition, new components can be added without downtime or system redeployment. Microservices consequently allow for rapid system expansion without the requirement for extra resources.

10. Conclusion

In conclusion, this paper aims to demonstrate the implementation of the message service queue using RabbitMQ. The paper further presents the concept of a config file that aims to simplify the process of auto-networking any new message service that joins a distributed network. The proposed solution leverages the principles of microservices communication to create a dynamic and flexible system that can adapt to the ever-changing needs of a distributed network. By introducing the config file, organisations can save valuable time and resources that would otherwise be spent on manual network configuration and integration processes. This research provides a blueprint for the development of an efficient and scalable communication infrastructure that can accommodate new message services without disrupting the existing network. By leveraging this concept, organisations can take a significant step forward in realising the full potential of microservices-based communication systems.

References

[1] Sujan Y M1, Dr. Shashidhara H R2, Dr. Rohini Nagapadma " Framework of REST APIs", Online National Conference On Wireless Communication, Computing And Informatics , 2021

[2] Işıl Karabey Aksakalli , Turgay Çelik, Ahmet Burak Can, Bedir Tekinerdoğan, , "Deployment and communication patterns in microservice architectures: A systematic literature review" , Journal of Systems and Software , 2021

[3] Madhu M P, Sunanda Dixit," Distributing messages using rabbitmq with advanced message exchanges " International Journal of Research Studies in Computer Science and Engineering (IJRSCSE) Volume 6, Issue 2, 2019

[4] Jiang Yongguo , Liu Qiang ,Qin Changshua, Su Jian ,Liu Qianqian "Message-oriented middleware: a review" , International Conference on Big Data Computing and Communications , 2019

[5] Chaitanya Mukund Kulkarni, Prof M.S.Takalikar, "Analysis of REST API implementation" International Journal of Scientific Research in Computer Science Engineering and Information Technology ,Volume 3, Issue 5 , 2018

[6] OmarAl-Debagy, Peter Martinek "A comparative review of microservices and monolithic architectures",18th IEEE International Symposium on Computational Intelligence and Informatics , 2018

[7] S.S Alwakeel, H.M Almansour , "Modelling and performance evaluation of message oriented middleware with priority queuing" , Information Technology Journal , Volume 10 , Issue 1 , 2011