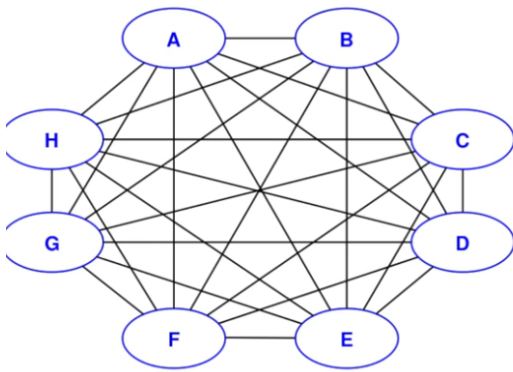
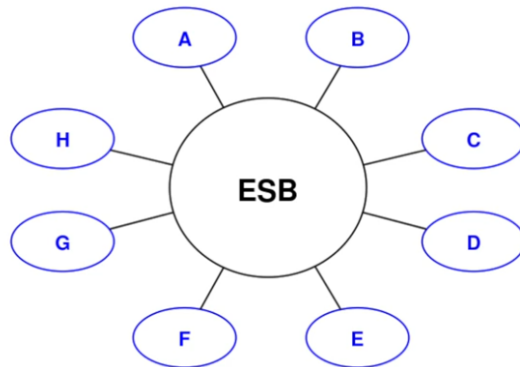


RabbitMQ

-> erlang <developed by>



Impact = N

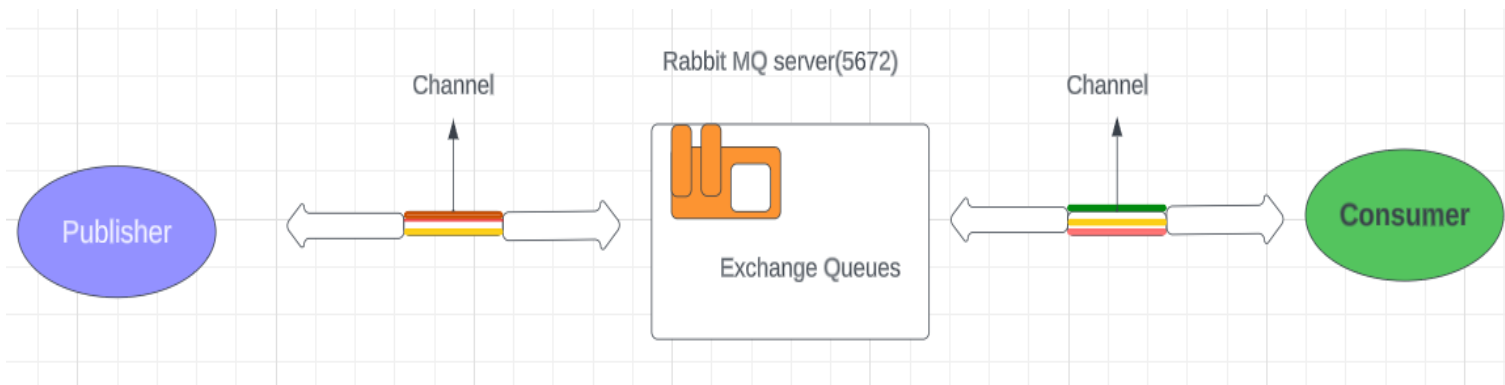


Impact = 1

- spagheety mesh architecture <where many clients wants to comunicate with on another >
- rabbitmq supports many protocols .

Table of Content

- RabbitMQ Components
- Spin RabbitMQ server with Docker
- Write a Publisher client NodeJs
- Write a Consumer client Nodejs
- My Thoughts about this tech
- Summary



=> AMQP <asvance message queuing protocol>

< publisher establishes a stateful tcp connection between itself and server >
- a two way communication
- it is using tcp or rtcp <not http>
- there are other protocols as well but we are intersted in the AMQP
<AMQP> is an open internet <or "wire"> protocol standard for message queuing communications .

- <consumer also connects a statful tcp connection between itself and server >
* server pushes message to the consumer, when they are ready

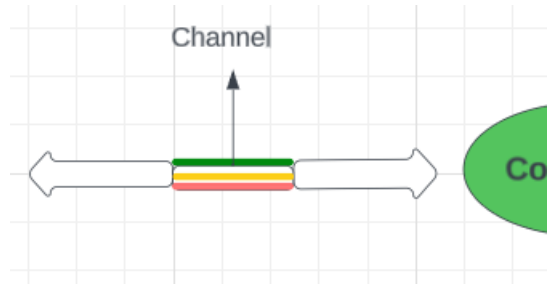
- channel ?
- : a logical connection / a mini connection in your connection

<they want to separate the consumer connection from multiple consumer inside that consumer >

- : like here three channels using the same tcp connection

< so instead of three consumers have three tcp connection there is one consumer with three channels using the same TCP connection >

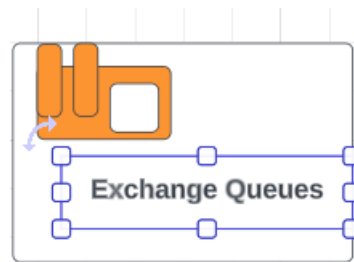
- [it's called multiplexing as well]
- [http2 also uses the multiplexing]



- it's a nice abstraction

=> Exchange Queues :

=> all the info is received and consumed from the queue



< publisher and consumer are not aware of the queues >

< they are aware of exchanges <7th layer of abstraction >

- you send all the stuff to the exchange and the task of the propagating is done by the exchange <there are default exchange >

- we will be using default exchange< exchanges uses different algorithm >

=> instead of installing the rabbitmq we will be using it with the help of the docker

1> once we know we have docker

2> now we will spin up our rabbit mq message container and that container will have rabbitMQ server

```
evans@evans:~$ sudo docker run --name rabbitmq -p 5672:5672 rabbitmq
```

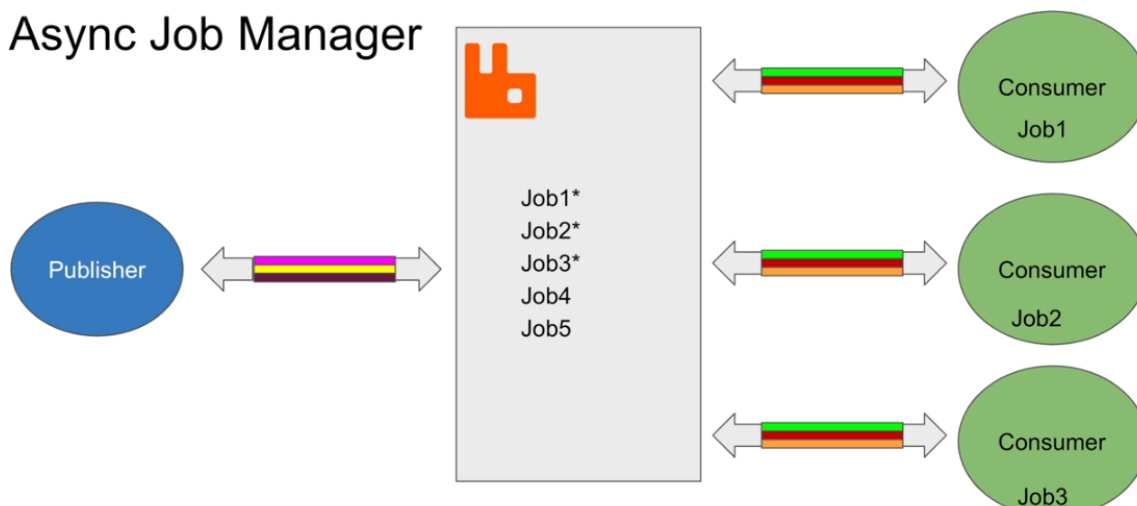
- first we will pull an image of rabbit mq

< it's good to always give a name to your container like here rabbitmq >

< now we have a container that is running rabbitMQ>

3> now we will build Async job execution engine

Async Job Manager



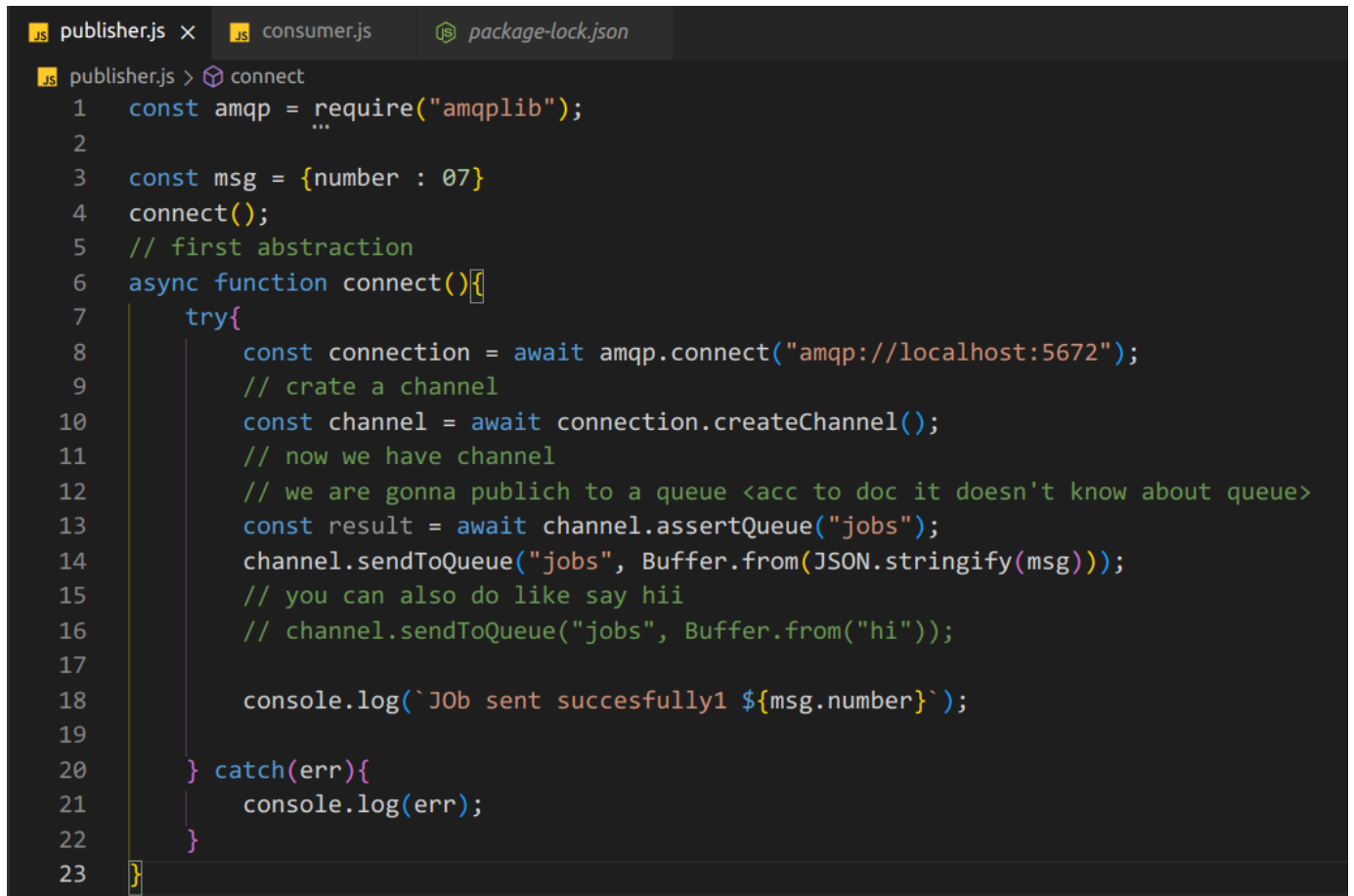
-- we will using Nodejs to build this

=> rabbit MQ uses the AMQP protocol

=> consumer have different channel and tcp connection it is not the same one as publisher is using

=> and once you publish the message then you can kill the connection but

-- for consumer you have to keep the consumer alive



```
publisher.js x consumer.js package-lock.json
publisher.js > connect
1  const amqp = require("amqplib");
2
3  const msg = {number : 07}
4  connect();
5  // first abstraction
6  async function connect(){
7      try{
8          const connection = await amqp.connect("amqp://localhost:5672");
9          // crate a channel
10         const channel = await connection.createChannel();
11         // now we have channel
12         // we are gonna publish to a queue <acc to doc it doesn't know about queue>
13         const result = await channel.assertQueue("jobs");
14         channel.sendToQueue("jobs", Buffer.from(JSON.stringify(msg)));
15         // you can also do like say hii
16         // channel.sendToQueue("jobs", Buffer.from("hi"));
17
18         console.log(`Job sent succesfully1 ${msg.number}`);
19     } catch(err){
20         console.log(err);
21     }
22 }
23 }
```

=> this is the publisher code

=> and when we run it it publish the message

: once the message is published you can close the connection

=> similarly we write the code for the consumer as well

: here we have given a name to the queue "jobs"

=> which will be accessed by the consumer with the same name

note : [according to the doc neither publisher nor consumer knows about the queue but in actual when we are implementing it seems and can be seen that they know about the queues anyway]

=> so when we run publisher

```
evans@evans:~/100x/Node/rabbitMQ$ node publisher.js
Job sent succesfully1 7
```

=> now it's time to consume the job

```
JS publisher.js JS consumer.js X package-lock.json
JS consumer.js > connect
1  const amqp = require("amqplib");
2
3  connect();
4
5  // first abstraction
6  async function connect(){
7      try{
8          const connection = await amqp.connect("amqp://localhost:5672");
9          const channel = await connection.createChannel();
10         const result = await channel.assertQueue("jobs");
11
12         channel.consume("jobs", message => {
13             // console.log(message);
14             console.log(message.content.toString());
15             const input = JSON.parse(message.content.toString());
16             console.log("received job with input : ", input)
17         })
18         console.log("waiting for the messages")
19     } catch(err){
20         console.log(err);
21     }
22 }
```

-- and now when we run the consumer <and consumer must be alive>

```
evans@evans:~/100x/Node/rabbitMQ$ node consumer.js
waiting for the messages
{
  fields: {
    consumerTag: 'amq.ctag-WemrFF-MsHV1zNc2zVpe-w',
    deliveryTag: 1,
    redelivered: false,
    exchange: '',
    routingKey: 'jobs'
  },
  properties: {
    contentType: undefined,
    contentEncoding: undefined,
    headers: {},
    deliveryMode: undefined,
    priority: undefined,
    correlationId: undefined,
    replyTo: undefined,
    expiration: undefined,
    messageId: undefined,
    timestamp: undefined,
    type: undefined,
    userId: undefined,
    appId: undefined,
    clusterId: undefined
  },
  content: <Buffer 7b 22 6e 75 6d 62 65 72 22 3a 37 7d>
}
```

-- we get the content as buffer and now we need to stringify this so we get

```
console.log(message.content.toString());
```

- and then we get

```
waiting for the messages
{"number":7}
```

=> to convert a string again in a json we can parse string back to json

```
const input = JSON.parse(message.content.toString());
console.log("received job with input : ", input)
```

- and then we get :

```
waiting for the messages
{"number":7}
received job with input : { number: 7 }
```

i am able to consume <same message as we are getting the same number/msg again and again as many times as now as you can see because till now we don't have told the server that we have consumed the message
-> called acknowledgment

=> instead of the hardcoding we can give user input as

```
// const msg = {number : 07}
const msg = {number : process.argv[2]}
```

- third element is our input
- here you can see that we are publishing the 10 as custom input

```
evans@evans:~/100x/Node/rabbitMQ$ node publisher.js 10
Job sent succesfully1 10
```

- and we can consume as well

```
evans@evans:~/100x/Node/rabbitMQ$ node consumer.js
waiting for the messages
{"number":7}
received job with input : { number: 7 }
{"number":"10"}
received job with input : { number: '10' }
```

- here you can see that we are able to consume previous and current value also

```
console.log(`received job with input : ${input}`);

// for acknowledgment
if(input.number == 10){
  channel.ack(message);
}
```

- now you can see that once it get's consumed and get ack and then it get removed from the queue and next time we didn't get that

```
evans@evans:~/100x/Node/rabbitMQ$ node consumer.js
waiting for the messages
{"number":7}
received job with input : 7
{"number":"10"}
received job with input : 10
^C
evans@evans:~/100x/Node/rabbitMQ$ node consumer.js
waiting for the messages
{"number":7}
```

[in kafka there is certain already defined way to get acknowledged when consumes consumes the message from the queue and in rabbitmq we do it just getting what we was expecting and then we do ack if it get satisfied] but there might be a case when it could be get consumed more than once if conditon of ack was not defined properly <-- explore more about that -->