Taller de comunicación Asíncrona: RabbitMQ

Andrés Felipe Muñoz Aguilar - 2210087

Marisol Osma Llanes – 2211466

En el ámbito de la informática y la ingeniería de software, la comunicación asincrónica es un concepto fundamental que juega un papel crucial en el diseño y desarrollo de sistemas distribuidos y aplicaciones escalables. Este paradigma de comunicación permite que los sistemas interactúen entre sí de manera independiente en el tiempo, lo que proporciona una mayor flexibilidad, capacidad de respuesta y eficiencia en el intercambio de información.

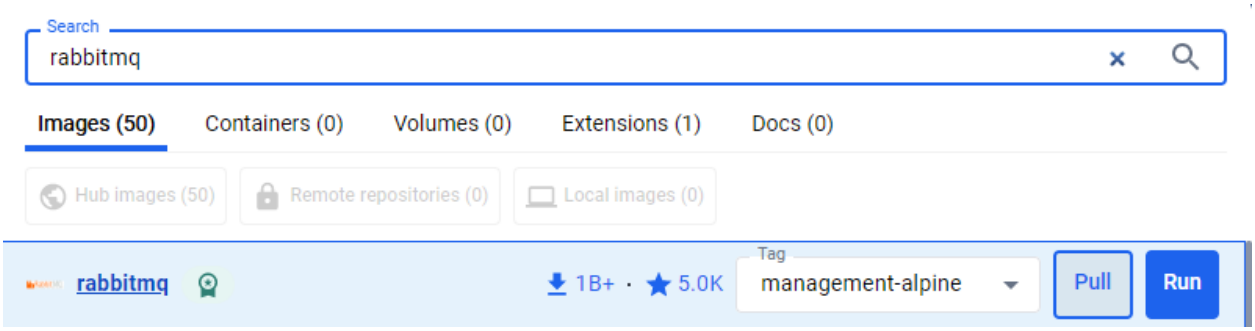
Con el fin de explorar y comprender en profundidad los principios y prácticas relacionados con la comunicación asincrónica, se realiza un taller que se centra en el uso de Docker y RabbitMQ. Docker, una plataforma de contenerización ampliamente utilizada, permite empaquetar, distribuir y ejecutar aplicaciones de manera eficiente y consistente en diferentes entornos. Por otro lado, RabbitMQ, un sistema de mensajería de código abierto, proporciona un mecanismo robusto y escalable para la implementación de colas de mensajes y la gestión de la comunicación asincrónica.

Durante el desarrollo de este taller, se realizará la configuración y uso de Docker para crear entornos de desarrollo aislados y reproducibles. A través de ejercicios prácticos, se implementan dos esquemas de comunicación asincrónica, cada uno compuesto por un productor y un consumidor, usando RabbitMQ como infraestructura subyacente.

DESARROLLO

PARTE 1

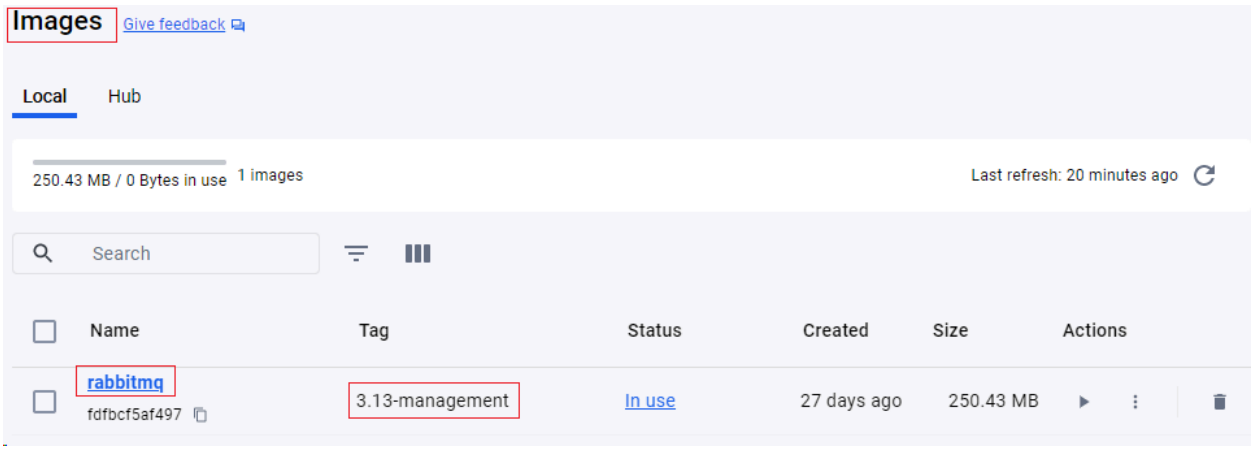
El primer paso es instalar docker en el PC que se va a ejecutar. Tras instalar docker, se instalan los siguientes dos contenedores: Portainer y RabbitMQ, aunque, debido a que se está usando la aplicación “Docker Desktop” para “Windows Subsystem for Linux”, se puede omitir la instalación de portainer, pues, “Docker Desktop” ya ofrece herramientas para administrar los contenedores en ejecución

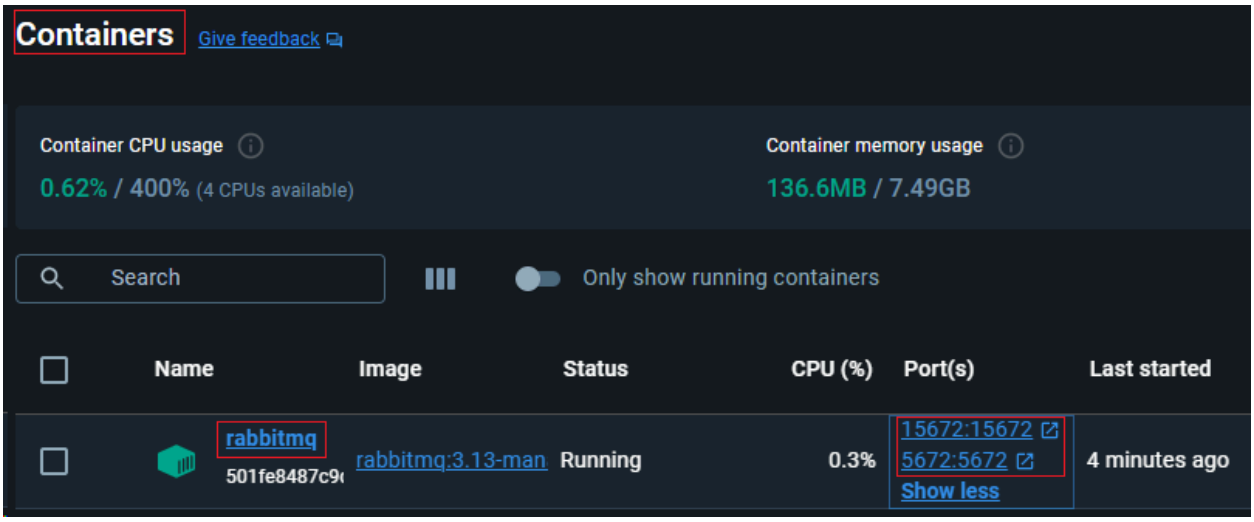


Para instalar RabbitMQ, primero, se debe hacer pull de la imagen que servirá como "template" o "plano" para construir un contenedor a partir de ella. Y, luego, se procede a crear un contenedor en base a esta imagen. El comando para ejecutar la creación de este contenedor (Y sus resultados) es el siguiente:

```
C:\Users\munoz>docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.13-management
Unable to find image 'rabbitmq:3.13-management' locally
3.13-management: Pulling from library/rabbitmq
bccd10f490ab: Pull complete
f1000ca5c91d: Pull complete
15fd8d52bc6c: Pull complete
d9b381d4c87a: Pull complete
497a9eb5f435: Pull complete
dd2bd0cede52: Pull complete
bed3197826ba: Pull complete
0c1d09ec487d: Pull complete
a89de7acba35: Pull complete
74d11b8768c5: Pull complete
b8a34a89caa8: Pull complete
853ab4a97c91: Pull complete
Digest: sha256:18d7104751b66c882c109349f537108c7cd979d87fe9020ef4dc4d773d37691e
Status: Downloaded newer image for rabbitmq:3.13-management
```

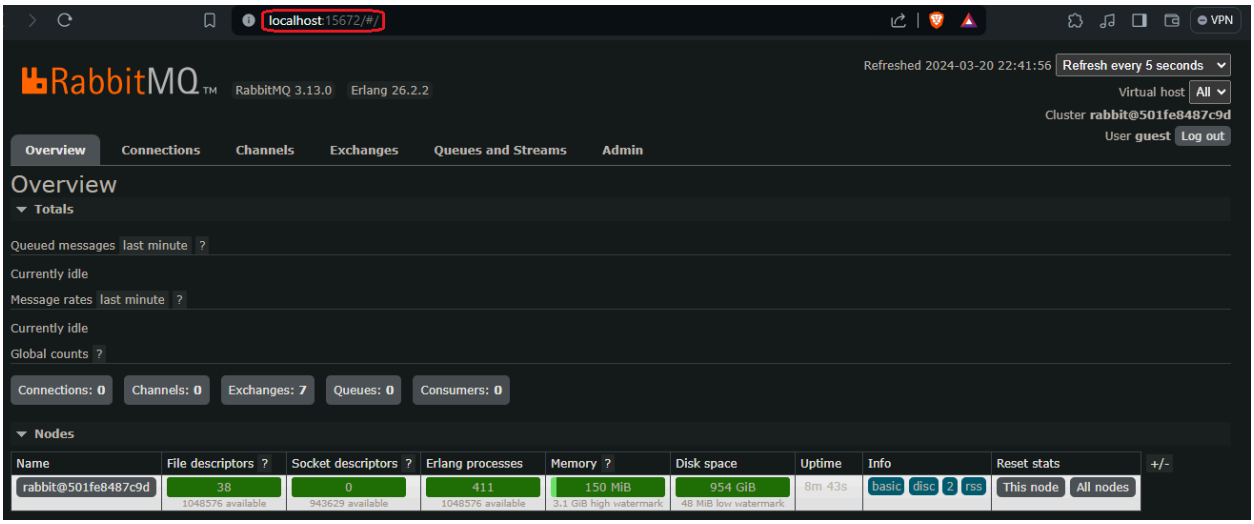
Si se retorna a Docker Desktop, se puede observar la creación de una nueva imagen y un nuevo contenedor:





Con esto, se encuentra levantado RabbitMQ y, además, se encuentra corriendo un contenedor a partir de esta imagen. Esto, incluso, se puede comprobar yendo a la dirección de localhost e inspeccionando puerto 15672:

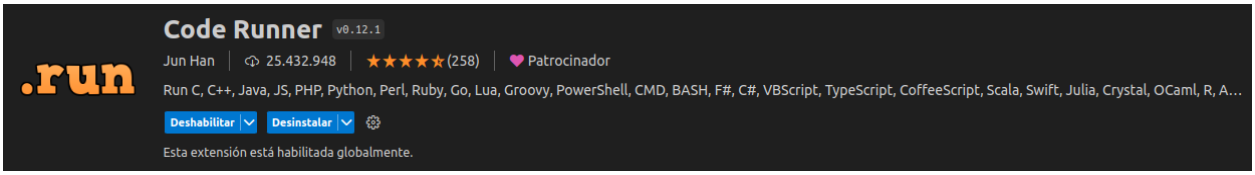
Por defecto, RabbitMQ proporciona el usuario “guest” y la contraseña “guest”



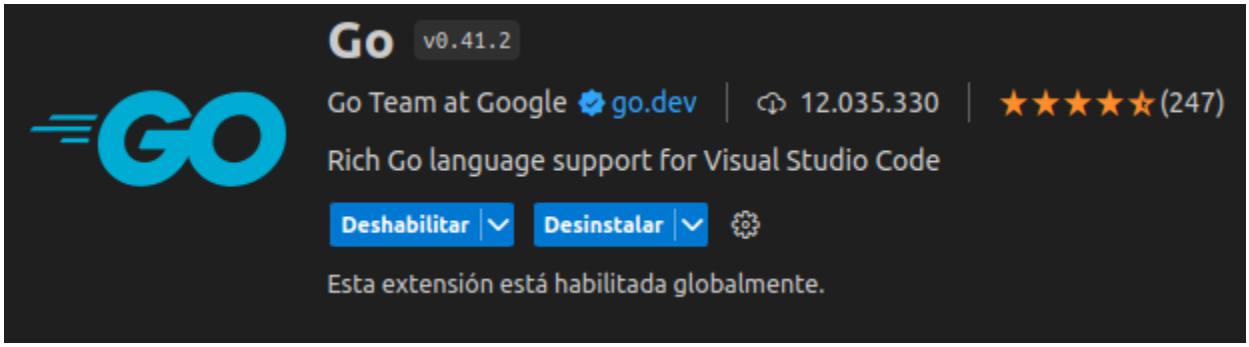
Como se puede observar, el puerto 15672 se encuentra levantado y escuchando los paquetes que envía RabbitMQ.

Lo siguiente es instalar y ejecutar GoLang en el editor de código de mayor conveniencia. En este caso, se escoge "Visual Studio Code" debido a comodidad y experiencia en el uso del editor. Para Windows hay ejecutables que instalan GoLang de manera eficiente, mientras que en Linux se debe extraer el archivo descargado en la carpeta "usr/local" e incluir la variable de entorno en "HOME/user/.profile", sin embargo, esto no se detallará en este taller, pues, no es el propósito que tiene.

Tras instalar GoLang, se usarán las siguientes dos extensiones para "Visual Studio Code":



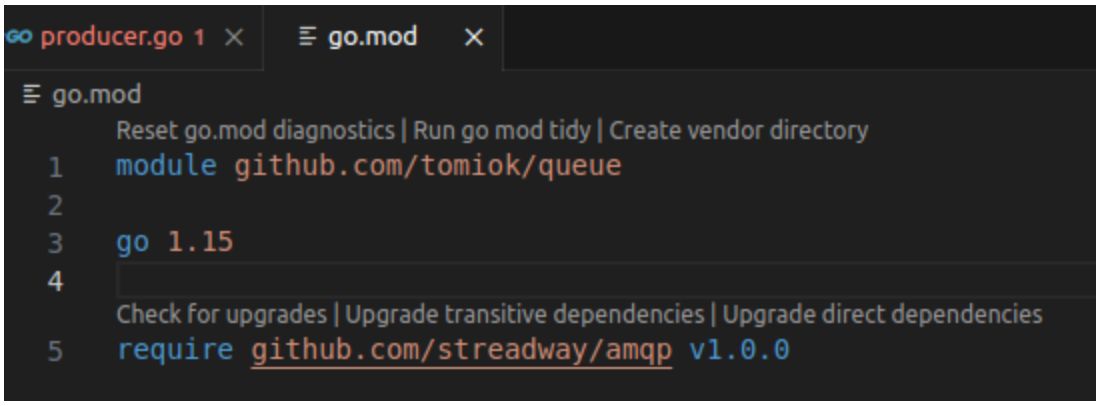
Para correr el código



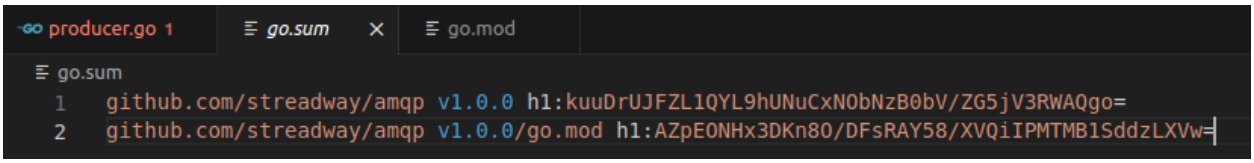
Para soporte de programación en GoLang

Código

Debido a que se necesitan usar "features" del repositorio "github.com/streadway/amqp", primero, se crea el archivo "go.mod" que permite a la aplicación usar el módulo "github.com/tomiok/queue", (Especificando la necesidad de una versión mínima de go igual o superior a la versión 1.15) y, de este, extraer la dependencia del proyecto que incluye la biblioteca "github.com/streadway/amqp" en su versión 1.0.0. A través del uso de este módulo se podrán usar funciones para la publicación de mensajes en RabbitMQ, para el establecimiento de conexiones TCP, declaración de colas, entre otras.



Además de este archivo, se debe crear un archivo "go.sum" en el cual se especifica que el proyecto depende de la librería depende de la biblioteca "github.com/streadway/amq" y almacenar su dependencia con un resumen que usa un hash criptográfico que sirve como firma de la versión de esa dependencia.



Productor

Este programa, al estar escrito en "Go" (Google Programming Language) comienza con la instrucción "package main" (Un paquete principal llamado main). El paquete principal es el punto de entrada del programa, y debe contener una función main que servirá como punto de partida para la ejecución del programa. Al escribir "package main" se está indicando que ese archivo forma parte del paquete principal del programa y que, al ejecutarlo, se debe entrar por la función main. Tras esto, se realizan distintos import que sirven a distintas funciones:

- **Fmt**: El paquete fmt proporciona funciones para el formateo de entrada y salida en Go. Permite imprimir en la consola, leer y escribir en streams de datos, etc.
- **Log**: El paquete log proporciona funciones para la gestión de registros (logs) en Go. Permite imprimir mensajes de registro en la consola o en otros destinos de registro, como archivos, sistemas de monitorización o servicios de registro remotos.
- **Time**: El paquete time proporciona funciones para trabajar con el tiempo y las fechas en Go. Permite la creación, manipulación y formato de objetos de tiempo. En este caso, se usa para calcular el tiempo que el programa temporalmente pausa

- **Amqp**: El paquete amqp es una biblioteca en Go para interactuar con el protocolo AMQP (Advanced Message Queuing Protocol). Permite la creación, envío y recepción de mensajes en sistemas de mensajería basados en AMQP, como RabbitMQ.

Ulteriormente, se establece una conexión con un servidor RabbitMQ que se ejecuta en el servidor virtual local "localhost" en el puerto 5672, usando las credenciales de usuario guest y contraseña guest ("//guest:guest@"). La función "amqp.Dial()" devuelve un objeto de conexión (conn) y un posible error (err). Si la conexión se establece correctamente, este error será nil (Valor no inicializado), mientras que "conn" contendrá la conexión establecida. Tras esto, el condicional comprueba si sucedió un error, en caso de ser cierto, el programa usa el paquete "log" para imprimir el error y finalizar la ejecución del programa. Tras esto, sin importar que haya ocurrido un error o no, se ejecuta el código "defer conn.Close()" que programa el cierre de la conexión una vez que la función actual (main) haya terminado su ejecución. Esto garantiza que la conexión se cierre correctamente cuando ya no sea necesaria, evitando posibles fugas de recursos y liberando los recursos utilizados por la conexión.

Este modo de operar se aplica, también, a la creación de un canal de comunicación con el servidor RabbitMQ utilizando la conexión "conn" creada previamente. Del mismo modo, posteriormente, se comprueba si la creación del canal ha generado un error; en caso de ser cierto, se usa el paquete "log" para imprimir el error por consola y detener la ejecución del programa. Finalmente, sin importar si ha sucedido un error o no, se ejecuta el código "defer ch.Close()" que programa el cierre del canal una vez que la función actual (main) haya terminado su ejecución. Este se hace para evitar posibles fugas de recursos y para liberar los recursos utilizados por el canal.

```go
package main

import (
    "fmt"
    "log"
    "time"

    "github.com/streadway/amqp"
)

func main() {
    conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")

    if err != nil {
        log.Fatal(err)
    }

    defer conn.Close()

    ch, err := conn.Channel()

    if err != nil {
        log.Fatal(err)
    }

    defer ch.Close()
```

Tras crear la conexión y el canal, finalmente, se puede usar el canal anteriormente creado para crear una cola de mensajería con RabbitMQ. Los parámetros que se le pasan a la función son:

- **Name**: Se declara el nombre "gophers" para la cola
- **Durable**: False. Es decir, no es una cola duradera
- **Autodelete**: False. Es decir, la cola no se elimina automáticamente
- **Exclusive**: False. Es decir, la cola no es exclusiva
- **Nowait**: False. Es decir, la cola no espera por la confirmación del servidor (Por este motivo, es comunicación asíncrona)
- **Args**: nil. Es decir, no se añaden argumentos adicionales

Tras declarar la creación de la cola asincrónica, usando el condicional, se revisa si hubo algún error al crear la cola; de ser cierto, se usa se usa el paquete "log" para imprimir el error por consola y detener la ejecución del programa.

De manera ulterior, se imprime el objeto "q", es decir, la cola recientemente creada, lo que permitirá ver el nombre, mensajes encolados. etc.

Finalmente, se ejecuta un bucle for infinito que publicará, de manera indefinida, mensajes a la cola. Para esto, se hace uso de la función “ch.Publish()” con argumentos:

- exchange: “”. Es el intercambio al que se enviará el mensaje (en este caso, se utiliza un intercambio vacío "" para enviar directamente a la cola)
- key: q.Name. Es el nombre de la cola en la cual se hará la publicación del mensaje. Debido a que se había creado una cola anteriormente, la key será el nombre de la cola previamente creada (gophers)
- mandatory: false. El mensaje no se devolverá si no se encamina a una cola
- immediate: false. Indica al servidor que no devuelva el mensaje si no puede enviarlo inmediatamente a un consumidor
- content: amqp.Publishing()

Esto quiere decir que, usando el canal anteriormente creado, se publican mensajes con una cabecera (Cabecera del mensaje TCP) con un valor no asignado, que contiene un tipo de contenido (ContentType) de texto plano y en cuyo cuerpo del mensaje se envía una cadena de bytes que representan el siguiente mensaje: “Sent at 00:00:00 ms=0”. Este mensaje quiere decir: “Enviado a las 0 horas, 0 minutos, 0 segundos con 0 milisegundos. Es simplemente un mensaje para mostrar la hora a la cual se envió el mensaje. Tras hacer la publicación del mensaje en el canal previamente creado, dentro de la cola “gophers”, se verifica si hubo un error; en caso de ser cierto, se sale del bucle para que se finalice la ejecución, de lo contrario, fuerza a la función a esperar 2 segundos y, luego, volver a publicar un mensaje en la cola “gophers” con la hora a la cual se envió el mensaje.

```go
q, err := ch.QueueDeclare("gophers", false, false, false, false, nil)

if err != nil {
    log.Fatal(err)
}

//debug only
fmt.Println(q)

for {
    err := ch.Publish("", q.Name, false, false,
        amqp.Publishing{
            Headers:     nil,
            ContentType: "text/plain",
            Body:        []byte("sent at " + time.Now().String()),
        })

    if err != nil {
        break
    }

    //wait 2 seconds until send another message
    time.Sleep(2 * time.Second)
}
```

Consumidor

El código del consumidor comienza de la misma manera que el código del productor: Indicando el paquete main e importando los paquetes necesarios para la impresión de datos por consola, el manejo de errores por consola y la comunicación con la cola de RabbitMQ. Asimismo, inicia declarando una conexión (conn) con el servidor virtual local (localhost) a través del puerto 5672, usando el usuario "guest" y la contraseña "guest" ("//guest:guest@"). Tras declarar la conexión, usa el condicional para verificar si hubo algún error al intentar conectarse al localhost y, en caso de ser verdad, se usa el paquete "log" para imprimir el error por consola y detener la ejecución del programa. A través del uso de esta conexión previamente creada, se crea un canal para obtener los mensajes de una cola determinada (Esta canal se nombre "ch"). Tras declarar el canal, se vuelve a usar el paquete "log" para imprimir el error por consola y detener la ejecución del programa. Para administrar estas conexiones, tanto para la conexión, como para el canal derivado de esta conexión se usa la instrucción "defer <variable_name>.Close()", la cual se asegura de programar el cierre de la conexión una vez que la función actual (main) haya terminado su ejecución.

```go
package main

import (
    "fmt"
    "log"

    "github.com/streadway/amqp"
)

func main() {
    conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")

    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    ch, err := conn.Channel()

    if err != nil {
        log.Fatal(err)
    }

    defer ch.Close()
```

Ulteriormente, se crea una lista que contiene todos los mensajes que se encuentran en una cola determinada determinada por sus argumentos. Estos son:

- **queue**: "gophers". El nombre de la cola de la cual se obtendrán los mensajes
- **consumer**: "". El nombre del consumidor (en este caso, se deja vacío para que RabbitMQ lo genere automáticamente)
- **autoAck**: true. El broker de mensajería esperará automáticamente un acuse de recibo de los mensajes entregados a este consumidor
- **exclusive**: false. Determina si la cola se exclusiva y se asigna la cola a la conexión
- **noLocal**: false. El broker de mensajería entregará mensajes al consumidor si también se publicaron en esta conexión
- **noWait**: false. Determina que se debe esperar una confirmación de entrega del servido
- **args**: nil. Es decir, no se añaden argumentos adicionales

Haciendo uso de estos argumentos, la función "ch.Consume()" configura un consumidor para leer mensajes de la cola "gophers" utilizando el canal de comunicación "ch" previamente creado. Esta función devuelve un canal

(chDelivery) a través del cual se enviarán los mensajes consumidos y un posible error. En caso de haber un error se usa el paquete “log” para imprimir el error por consola y detener la ejecución del programa.

Este código en Go crea un goroutine para consumir mensajes del canal de entrega (chDelivery) previamente creado en un bucle continuo. Primero, se crea un canal sin buffer llamado “noStop” utilizando la función “make(chan bool)”. Este canal se utiliza para controlar la finalización del goroutine más adelante en el código. Segundo, se lanza una nueva goroutine utilizando la palabra clave “go”, que ejecutará una función anónima. Dentro de esta función anónima, se inicia un bucle for que itera sobre cada mensaje entregado en el canal “chDelivery” e imprime el contenido del cuerpo del mensaje (delivery.Body) en la consola utilizando la función “fmt.Println()”. Esta goroutine se ejecuta en paralelo con el resto del programa y continuará consumiendo mensajes de “chDelivery” mientras esté en funcionamiento. Lo que implica que estará en funcionamiento mientras haya mensajes en la cola “gophers”.

```go
chDelivery, err := ch.Consume(
    "gophers",
    "",
    true,
    false,
    false,
    false, nil)

if err != nil {
    log.Fatal(err)
}

noStop := make(chan bool)

go func() {
    for delivery := range chDelivery {
        fmt.Println("msg: " + string(delivery.Body))
    }
}()

<-noStop
```

Tras explicar el funcionamiento de ambos códigos, se procede a hacer la ejecución y la revisión en RabbitMQ.

Se revisa que funcione el consumidor

```
[Running] go run "/home/administrador/Música/Taller RabbitMQ/consumer.go"
msg: sent at 2024-03-21 10:54:02.584220174 -0500 -05 m=+0.003625980
msg: sent at 2024-03-21 11:23:01.387518226 -0500 -05 m=+0.003314287
msg: sent at 2024-03-21 11:23:03.388406894 -0500 -05 m=+2.004202956
msg: sent at 2024-03-21 11:23:05.388614002 -0500 -05 m=+4.004410070
msg: sent at 2024-03-21 11:23:07.389563035 -0500 -05 m=+6.005359133
msg: sent at 2024-03-21 11:23:09.391567944 -0500 -05 m=+8.007364057
msg: sent at 2024-03-21 11:23:11.392640224 -0500 -05 m=+10.008436311
msg: sent at 2024-03-21 11:23:13.392967339 -0500 -05 m=+12.008763416
msg: sent at 2024-03-21 11:23:15.394188078 -0500 -05 m=+14.009984183
msg: sent at 2024-03-21 11:23:17.39476452 -0500 -05 m=+16.010560654
msg: sent at 2024-03-21 11:23:19.395606385 -0500 -05 m=+18.011402513
msg: sent at 2024-03-21 11:23:21.396781341 -0500 -05 m=+20.012577426
msg: sent at 2024-03-21 11:23:23.398248004 -0500 -05 m=+22.014044108
msg: sent at 2024-03-21 11:23:25.399547314 -0500 -05 m=+24.015343428
msg: sent at 2024-03-21 11:23:27.399906627 -0500 -05 m=+26.015702761
msg: sent at 2024-03-21 11:23:29.400708236 -0500 -05 m=+28.016504336
msg: sent at 2024-03-21 11:23:31.401875001 -0500 -05 m=+30.017671103
msg: sent at 2024-03-21 11:23:33.40320822 -0500 -05 m=+32.019004318
```

Efectivamente, se puede ver que está recibiendo los mensajes y está imprimiendo por consola la hora a la que recibió el mensaje.

Se revisa que funcione el productor

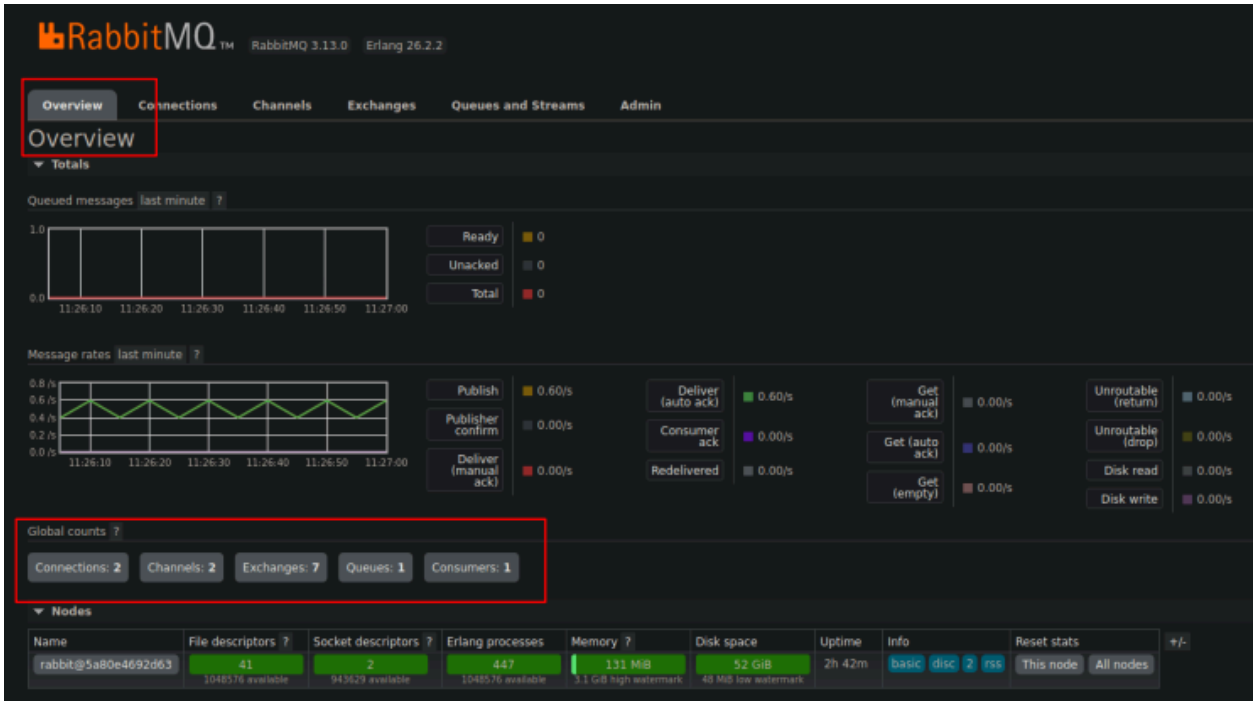
```
[Running] go run "/home/administrador/Música/Taller RabbitMQ/producer.go"
{gophers 1 0}
```

El productor funciona exitosamente, pues, no sólo ejecuta su código, imprimiendo por consola la “queue” que ha creado, sino, también, sus mensajes llegan al consumidor exitosamente

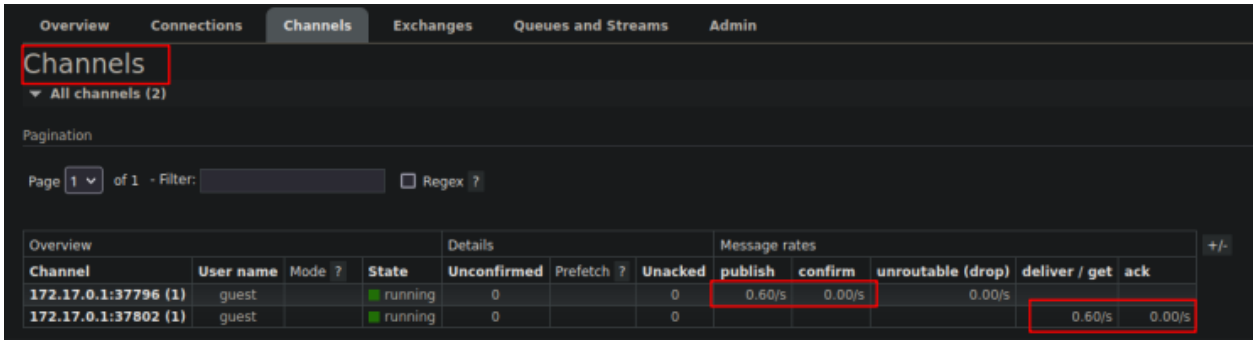
Finalmente, se revisa el funcionamiento en RabbitMQ.

En primera instancia, se revisa el apartado “Overview” de “RabbitMQ”. En este se observan detalles generales de la ejecución del contenedor. Como se puede apreciar, hay dos conexiones abiertas; estas existen debido a que en cada uno de los scripts (Productor y consumidor) se crea una conexión, de donde se derivan un canal para cada uno de estos. Esto último explica, subsecuentemente, la existencia de dos canales que se muestran en el apartado “global counts”. En este mismo apartado se puede apreciar la cantidad de colas que administra “RabbitMQ”; en este caso, sólo se administra una cola. Este comportamiento es acorde a lo programado en los scripts, puesto que sólo un

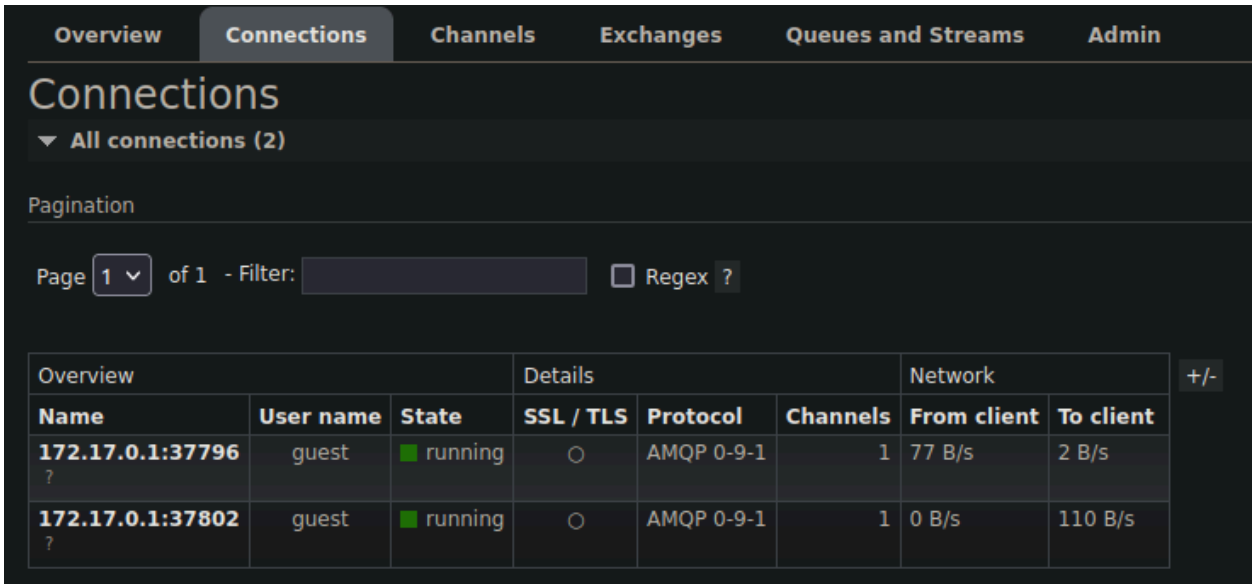
script (“Producer.go”) crea la cola, mientras que el otro (“Consumer.go”) únicamente consume los mensajes que esta cola le proporciona. De hecho, este mismo consumidor también se ve reflejado en el apartado “global counts” al final de la imágen siguiente:



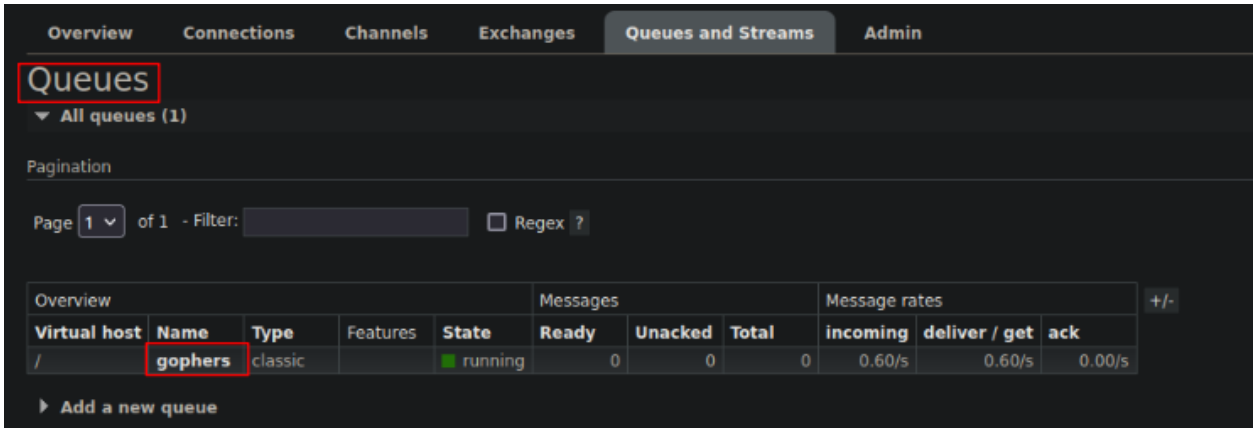
Ulteriormente, se revisa el apartado “Channels”, el cual da un resumen de las estadísticas correspondientes a los canales que manejan las distintas colas de RabbitMQ. Como se puede apreciar en la imagen posterior a este párrafo, existen dos canales: 172.17.0.1 comunicandose en el puerto 33796 y 172.17.0.1 comunicandose en el puerto 37802. En una inspección más profunda, se puede apreciar que el canal que se comunica por el puerto 33796 usa recursos para publicar y confirmar mensajes dentro de la cola creada; esto quiere decir que este puerto y, por consiguiente, su respectiva dirección IP pertenecen al productor quien, constantemente, publica mensajes a la cola. Por otro lado, el canal que se comunica por el puerto 37802 usa recursos para adquirir mensajes y realizar procesos de “ack” dentro de la cola creada; esto quiere decir que este puerto y, por consiguiente, su respectiva dirección IP pertenecen al consumidor quien, constantemente, consume los mensajes que el consumidor publica en la cola.



Tras esto, se revisa el apartado “Connections”. En este se dan detalles sobre las conexiones que se están manejando a través de este broker de mensajería. Se puede apreciar que hay dos conexiones, cada una correspondiente a los canales que se explicaron en el párrafo anterior; esto es así, debido a que, para poder crear un canal se debe, anteriormente, haber creado una conexión activa y, debido a que este canal se deriva de la conexión previamente establecida, el canal y la conexión comparten dirección y puerto



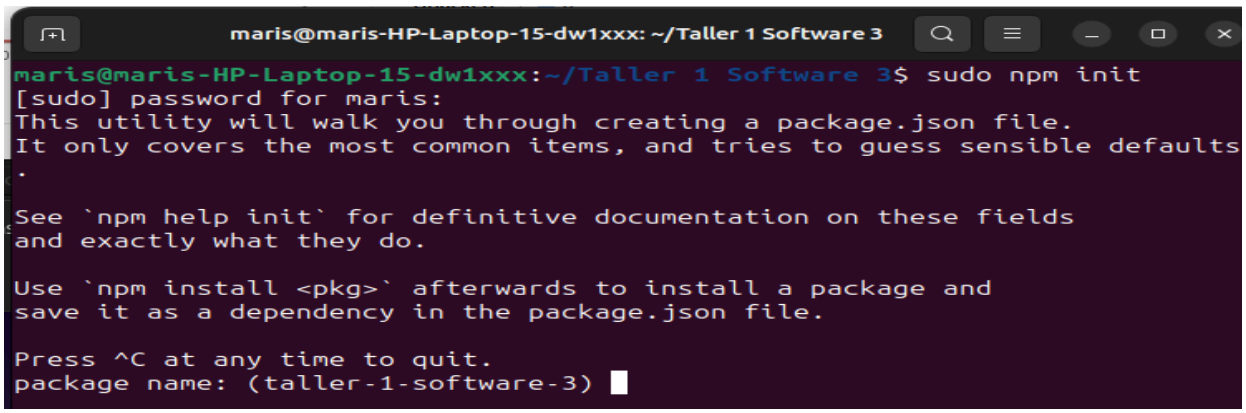
Finalmente, se revisa el apartado “Queues”, que ofrece especificaciones de características y datos estadísticos de las colas que maneja el broker de mensajería “RabbitMQ”. En este caso, se puede observar que se creó correctamente la cola “gophers”, que es la cola que se creó en el script del productor. Se observa, además, que está ocupando ancho de banda en publicar mensajes entrantes y en devolver mensajes que se encuentren dentro de la cola, permitiendo apreciar que el código funciona de manera correcta y que la cola se encuentra activa publicando y devolviendo mensajes.

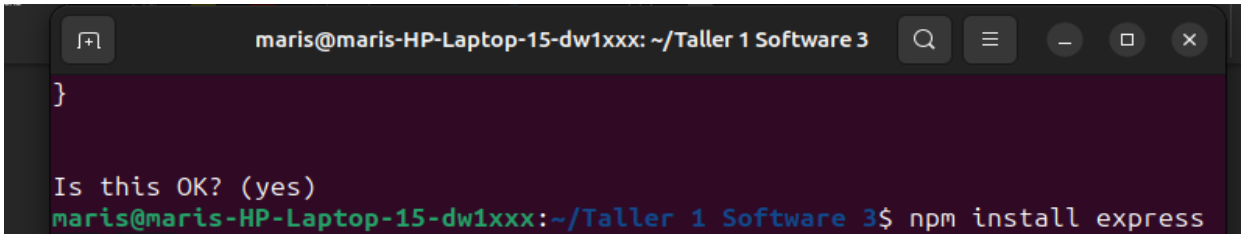


PARTE 2

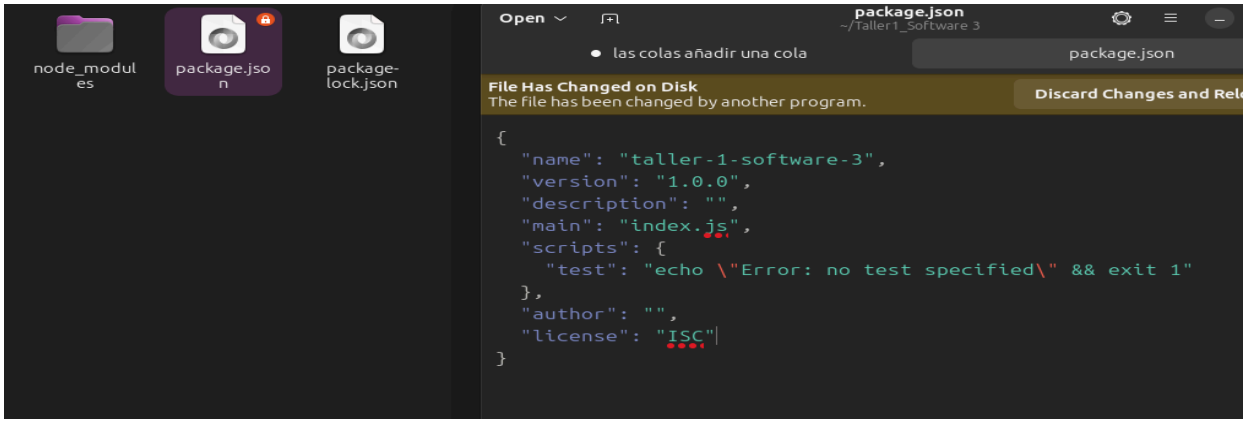
Para esta parte se trabajó el ejercicio por medios de comandos realizados en la terminal de Linux. Primero se asegura tener el Docker instalado para poder crear el contenedor de RabbitMQ que se va a utilizar, luego se instala el RabbitMQ como se ha mencionado anteriormente para observar el comportamiento de este y su ventaja con las colas para recibir y enviar mensajes. En las instalaciones de comando para evitar problemas a futuro no está de más instalar: : `npm install amqplib` para qué Rabbit pueda correr adecuadamente.

Para esta segunda parte es necesario construir un Api Rest, para eso se utilizó el framework de Node.js con Express, para poder construir un ejemplo de mensajería asincrónica. Como se va a utilizar Node.js se requiere su instalación previa, en su instalación viene incluido con el NPM que se iniciará para crear los paquetes de javascript; Una vez instalado se procede con la instalación de Express como se ve en la imagen dos, si no corre de esa forma intente agregar sudo al comienzo.

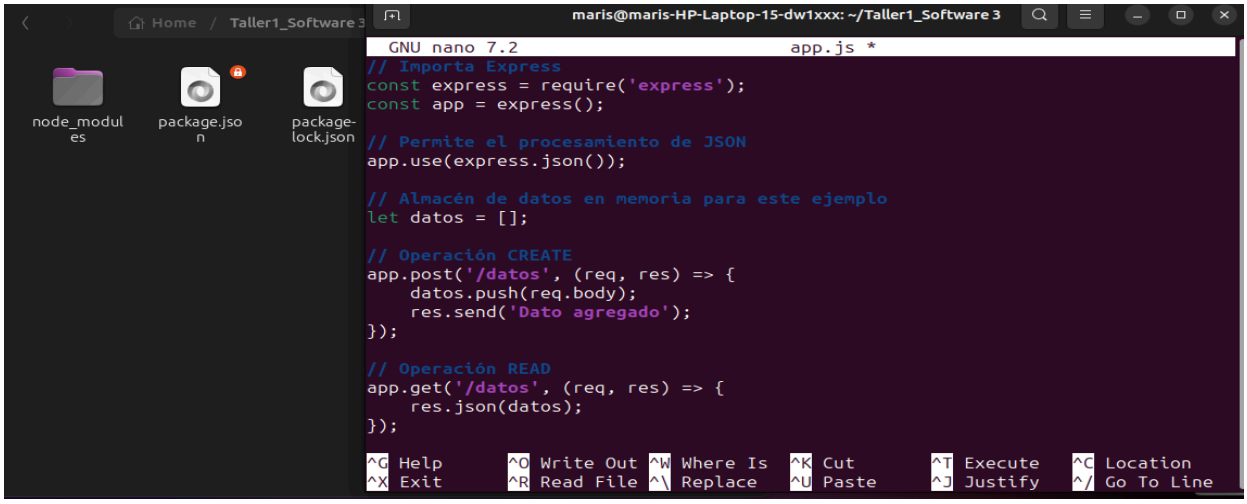




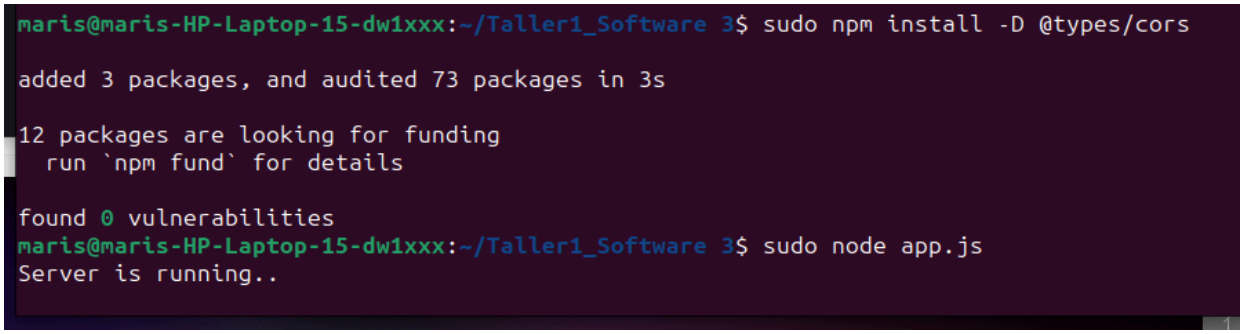
Luego verificamos que se haya instalado correctamente Express para poder manejarlo con operaciones cRUD más adelante, además es necesario ya que se necesita para cargar los módulos necesarios, y crear una aplicación de Express.

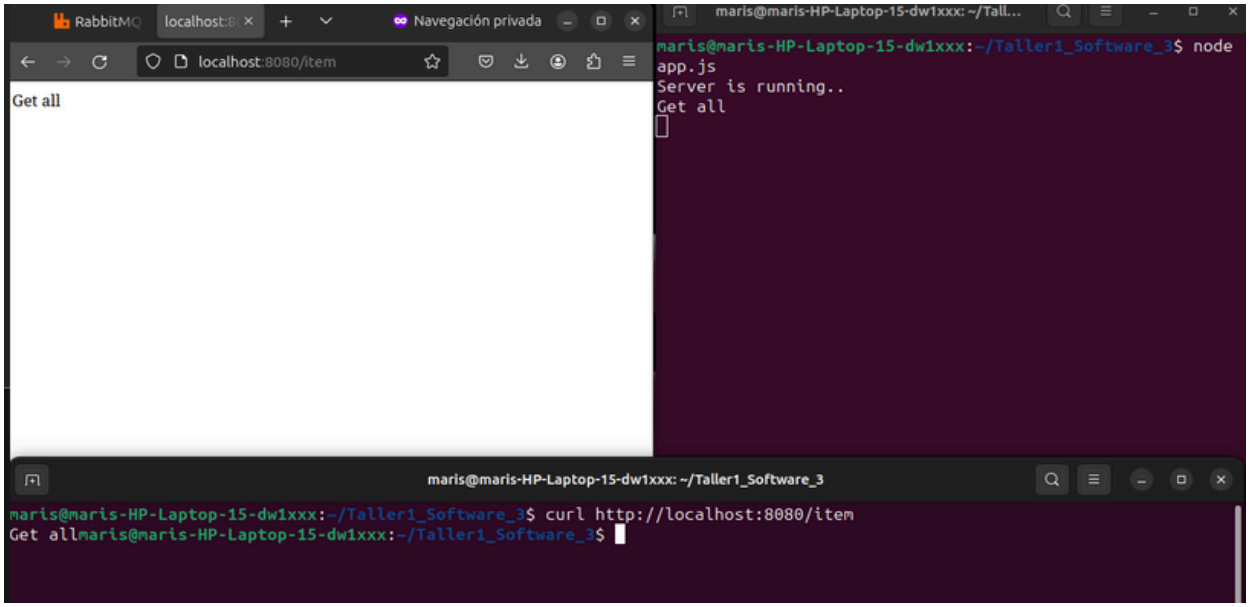


Una vez tenemos la ubicación del proyecto se crea un archivo para la aplicación Express con el comando nano app.js este abrirá el archivo en un editor de texto, para poder configurar una aplicación básica para entender la creación de las API's REST.

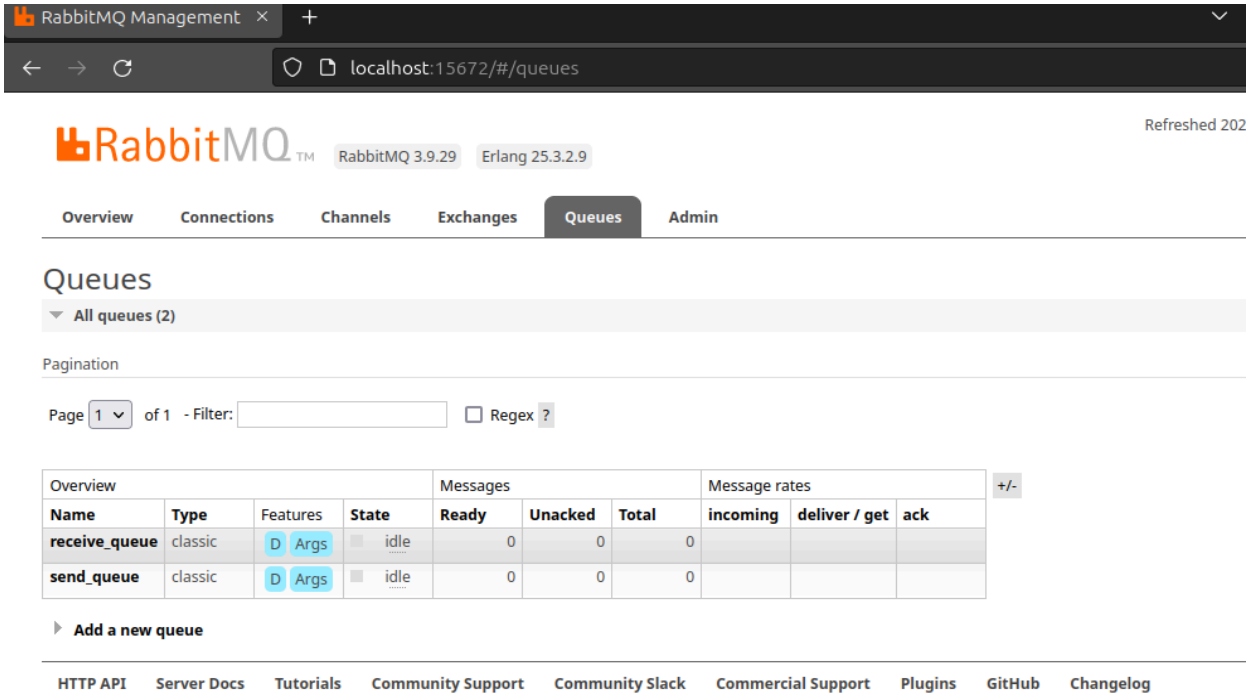


Una vez hecha la app.js se guarda y se comprueba que se ejecuta correctamente, para esto se utiliza el comando CRUD donde cada vez que el usuario interactúe con la página web o se mande el comando de ejecución la terminal responde con el mensaje configurado en el Backend.





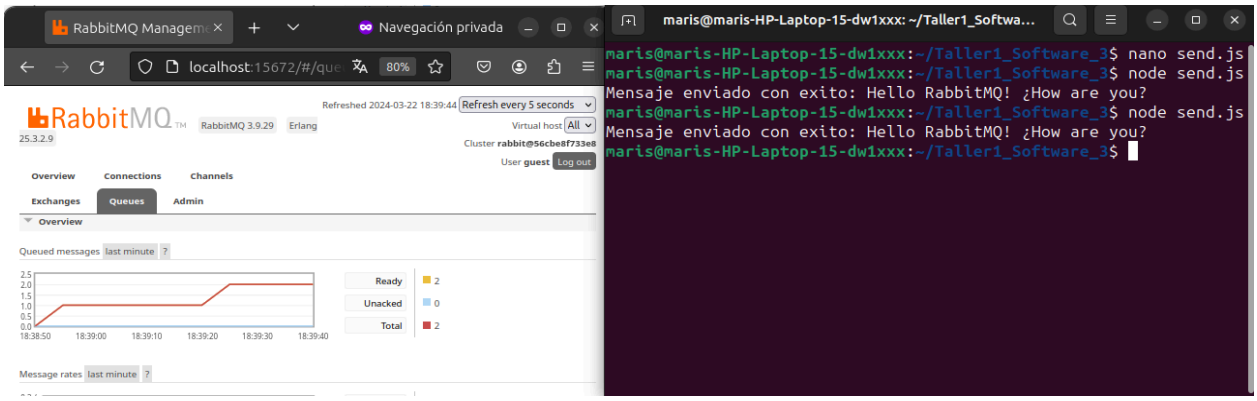
Para poder observar la página hay que entrar con el localhost y el puerto que se haya mapeado, como era una prueba el HTML solo arrojaba el mensaje configurado en el app.js pero está era de prueba para observar que se podía realizar la interacción de las dos formas y como respondía cuando se ponía a correr el servidor.



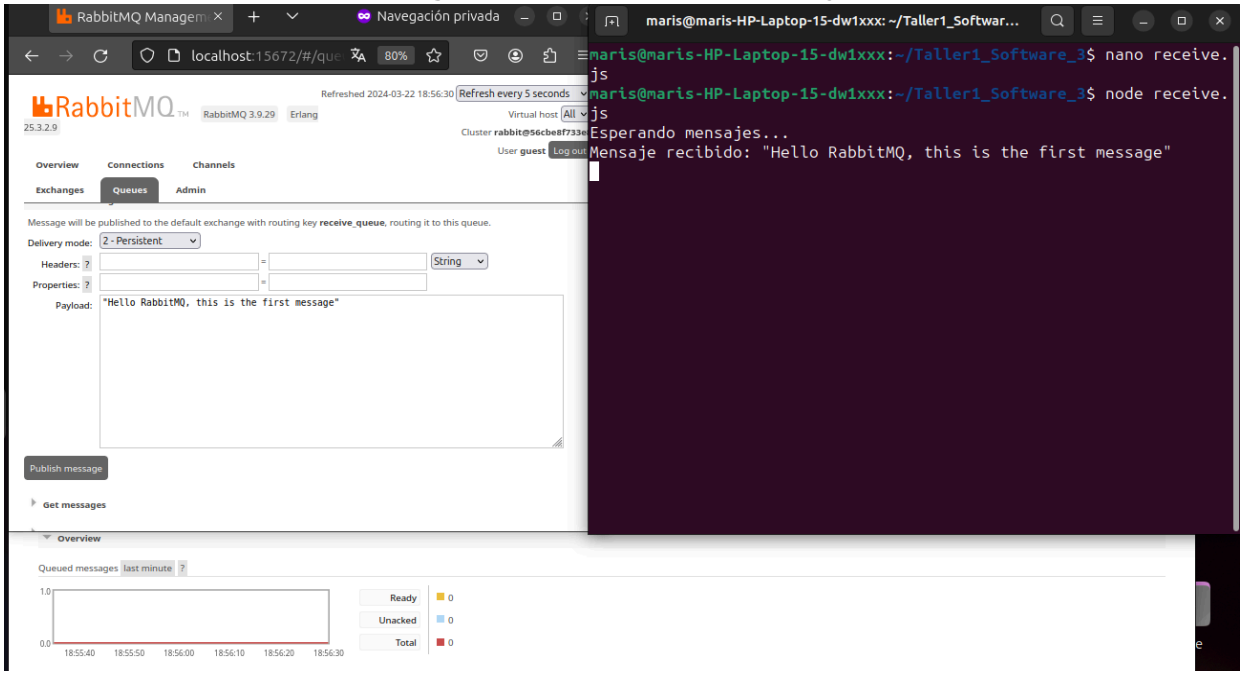
Volviendo al ejercicio de RabbitMQ y ya después de conocer cómo se crea y funcionan las API REST, vamos a la página de RabbitMQ, ahí vamos a la sección de 'Queues' que es donde se crearán dos colas, se decidió dividir la lógica para enviar y recibir mensajes de RabbitMQ en archivos JavaScript para hacer que el código sea más organizado y modular.

```
maris@maris-HP-Laptop-15-dw1xxx:~/Taller1_Software_3$ nano receive.js
maris@maris-HP-Laptop-15-dw1xxx:~/Taller1_Software_3$ nano receive.js
```

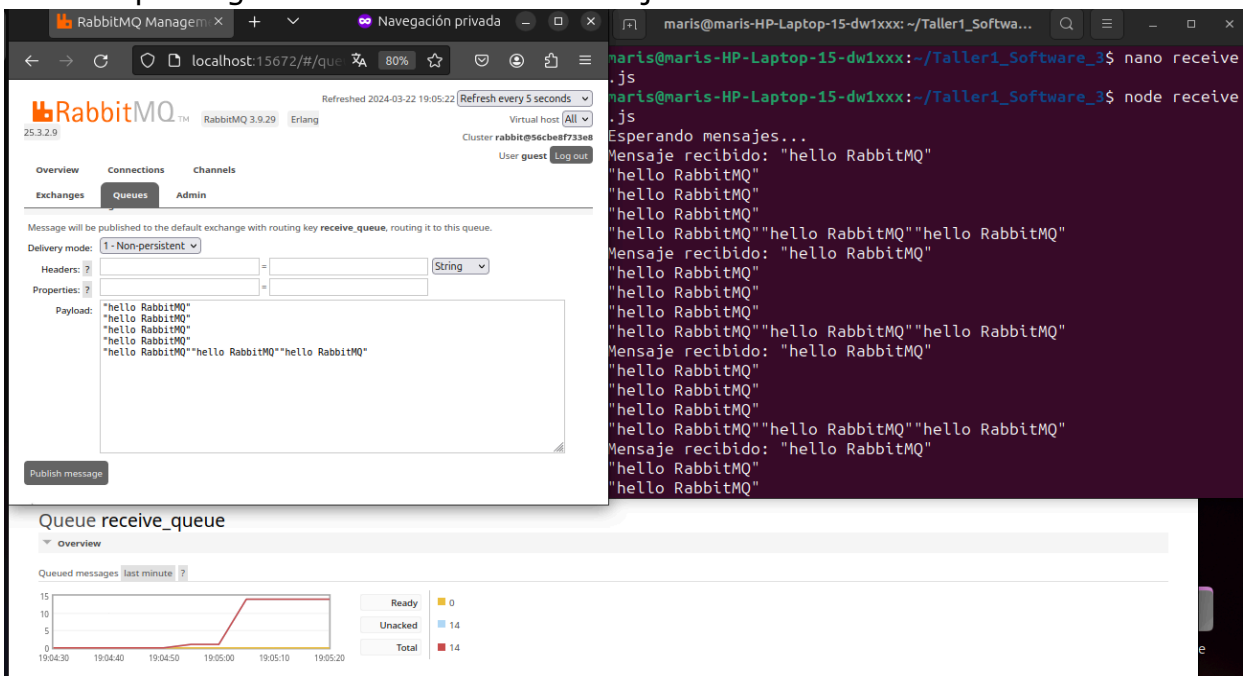
Una vez creadas las aplicaciones intentamos observar el comportamiento de los mensajes enviados, para esto se utilizó el comando node En la gráfica se puede ver como va creciendo la curva a medida que se envía un mensaje por medio de la terminal, en este caso no entraban por tiempo sino manualmente a diferencia de la parte 1 del taller.



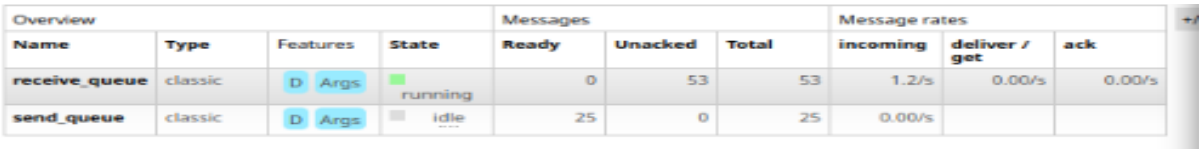
Sin embargo, para la gráfica y la cola de los mensajes que se reciben la terminal es diferente usando la aplicación receive.js, este se conecta por medio de canales del RabbitMQ para hacer las pruebas, ya que son los mensajes que se recibirá en la terminal según la cantidad de mensajes.



La gráfica permanece constante y quieta al principio si se recibe un mensaje a la vez, pero por medio del canal se puede mandar varios mensajes al mismo tiempo, por lo que en la imagen de abajo se puede ver como crece la gráfica a medida que llega desde la cola los mensajes al servidor.

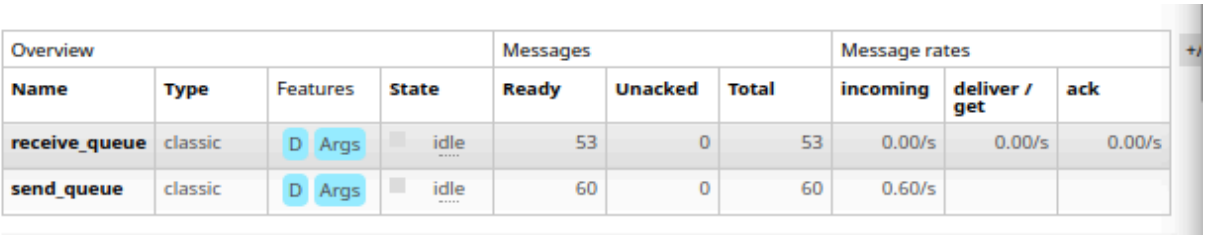


Al enviar y recibir los mensajes manualmente la velocidad de entrada a las colas variaba bastante, como se puede observar en la cola de los mensajes recibidos al tener la ayuda de los canales recibía los mensajes de forma más rápida.



Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
receive_queue	classic	D Args	running	0	53	53	1.2/s	0.00/s	0.00/s
send_queue	classic	D Args	idle	25	0	25	0.00/s		

A diferencia de los mensajes enviados que llegaban a la cola se demoran más en mostrar cambios debido a que se tenían que enviar uno por uno en la consola.



Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
receive_queue	classic	D Args	idle	53	0	53	0.00/s	0.00/s	0.00/s
send_queue	classic	D Args	idle	60	0	60	0.60/s		

CONCLUSIÓN

- En conclusión, como desarrolladores, RabbitMQ juega un papel crucial en la arquitectura de las aplicaciones modernas, ya que actúa como un intermediario para el manejo de mensajes entre diferentes servicios. Como software de gestión de colas de mensajes, RabbitMQ permitió transferir datos en forma de mensajes, y se puede incluir cualquier tipo de información. Esto es especialmente útil en situaciones donde las aplicaciones necesitan comunicarse entre sí, pero no tienen interacciones directas. RabbitMQ puede ayudar a reducir la carga y el tiempo de entrega de los mensajes, lo que puede mejorar significativamente el rendimiento y la eficiencia de las aplicaciones. Por lo tanto, se puede manejar en diferentes partes de un sistema de software.

REFERENCIAS

- Lingotti, T. [@tomaslingotti]. (2021, January 2). RabbitMQ y Golang en Docker - comunicación asincrónica en microservicios. Youtube. https://www.youtube.com/watch?v=GSMH3UapR6s
- Llamas, L. (2017, 21 octubre). Montar un API REST con NodeJs y Express. Luis Llamas. https://www.luisllamas.es/montar-un-api-rest-con-nodejs-y-express/