



**Roble Austral**



# Manual RabbitMQ

Handel Venegas Diocares

<b>Resumen</b>	<b>2</b>
<b>1. Introducción a RabbitMQ</b>	<b>4</b>
1.1 Qué son las colas de mensajería	4
1.2 Nociones de RabbitMQ	5
1.2.1 Elementos principales	5
1.2.2 Manejo de recepción de mensajes	6
1.3 Patrón competing consumers en RabbitMQ	7
<b>3. Ejemplo de RabbitMQ: Haciendo un auto-escalador de consumidores</b>	<b>8</b>
3.1 Necesidad de un auto-escalador	8
3.2 Instalación de RabbitMQ	9
3.2.1 En Ubuntu	9
3.2.2 En Windows	10
3.3 Arquitectura de la implementación	10
3.4 Implementación en código	11
3.4.1 Productor	11
3.4.2 Consumidor	15
3.4.3 Autoescalador	19
3.4.4 Herramienta visual de monitoreo	23

# Resumen

En este manual se proporciona una guía para entender y utilizar RabbitMQ de forma básica, un broker de mensajería ampliamente utilizado en la industria para gestionar la comunicación entre componentes de software de manera eficiente y confiable.

Como ejemplo práctico, se implementó un autoescalador de consumidores en Python, utilizando la biblioteca Pika. Este autoescalador ajusta dinámicamente el número de consumidores en función de la carga de trabajo y el uso de CPU, optimizando el uso de recursos y garantizando un procesamiento eficiente de los mensajes. También se incluyeron instrucciones detalladas para la instalación de RabbitMQ en Ubuntu y Windows, así como el uso de la interfaz web de monitoreo para visualizar métricas en tiempo real.

# 1. Introducción a RabbitMQ

## 1.1 Qué son las colas de mensajería

Una cola de mensajería es una forma de comunicación asíncrona entre componentes de software. En esencia, una cola de mensajería actúa como un intermediario donde los mensajes enviados por un componente emisor (*productor*) son almacenados temporalmente hasta que otro componente receptor (*consumidor*) los recoge.

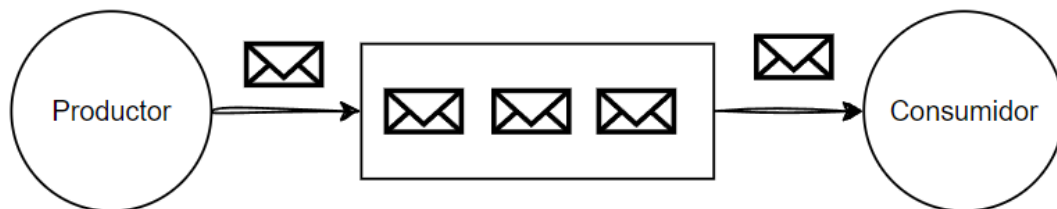


Figura 1. Esquema básico de una cola.

Así se asegura que los mensajes no se pierdan si el consumidor no está disponible en el momento en que el mensaje se envía, esto lo hace especialmente útil en arquitecturas en donde productores y consumidores trabajan de forma independiente y desacoplada tales como microservicios o en procesamiento por lotes.

En este manual se abordará RabbitMQ, un software para la gestión de colas y transmisión de mensajes ampliamente utilizado en la industria.

## 1.2 Nociones de RabbitMQ

RabbitMQ es un *broker* de mensajería basado en el protocolo AMQP. Se usa para gestionar la comunicación entre componentes de software o aplicaciones.

### 1.2.1 Elementos principales

Para entender cómo funciona RabbitMQ, primero conozcamos algunos de sus elementos:

- **Productor** - es el componente o aplicación que envía los mensajes.
- **Consumidor** - recibe los mensajes desde la cola y los procesa.
- **Colas** - buffer liviano en donde el productor inserta los mensajes para que uno o más consumidores puedan procesarlos. Funcionan con la misma idea que las colas en la programación en general, llega un elemento por un lado y sale por el otro, siempre de forma ordenada de uno en uno.
- **Exchange** - los productores, en lugar de insertar los mensajes directamente a la cola, se lo pasan a un *exchange* y este decide a qué cola, o colas debe enviarlo basándose en reglas específicas. Por ejemplo, un exchange puede decidir enviar copias de un mensaje a múltiples colas, o puede usar información del mensaje para mandarlo a una cola específica. Es como un enrutador. Existen varios tipos de exchanges que varían según la configuración que se requiera. Más adelante se explicarán.
- **Binding** - es el enlace entre las colas y el exchange. Cuando
- **Broker** - el exchange y sus colas asociadas forman en conjunto el broker de mensajería.

- **Routing key** - es un string que funciona como la dirección del mensaje.

Una diferencia importante entre el exchange y las colas es que el primero no almacena el mensaje, solo los transfiere a las colas. Si por ejemplo mandamos un mensaje a un exchange que no existe, este se pierde.

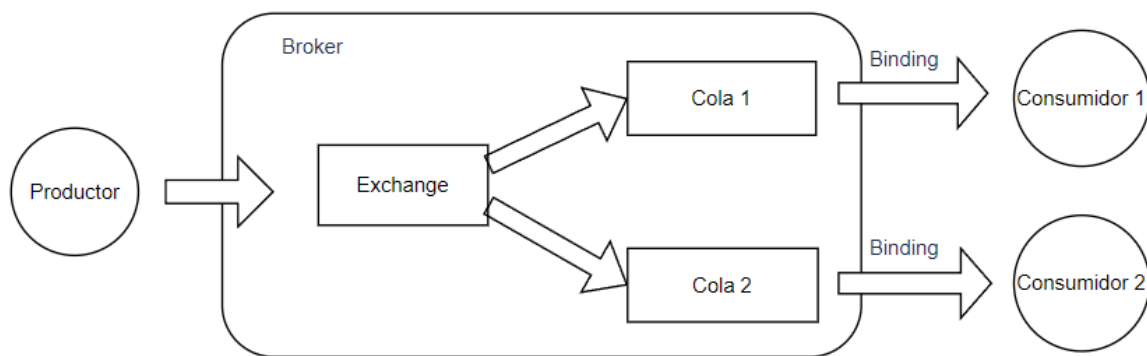


Figura 2. Ejemplo de una arquitectura básica y sus elementos

### 1.2.2 Manejo de recepción de mensajes

Uno de los mejores aspectos de RabbitMQ es la **confiabilidad** en la entrega de mensajes. Con RabbitMQ se puede garantizar la entrega y procesamiento de mensajes por parte de los consumidores aunque estos fallen, esto es gracias al mecanismo de reconocimiento de mensajes, **acknowledgment (ACK)**.

Cuando un consumidor recibe un mensaje, RabbitMQ espera una confirmación explícita de que el mensaje ha sido procesado correctamente. Si el consumidor no envía el ACK (por ejemplo, debido a

un fallo), RabbitMQ puede re-encolar el mensaje o simplemente desecharlo, según se haya configurado.

El flujo de trabajo en RabbitMQ es el siguiente:

1. Uno o varios productores envían un mensaje al broker especificando una routing key.
2. El exchange revisa el routing key y envía el mensaje a una o más colas según corresponda.
3. La cola almacena el mensaje hasta que un consumidor los recibe.
4. El consumidor recibe y procesa el mensaje. Puede mandar un mensaje de confirmación (ACK) para manejar la pérdida de mensajes.

Antes de recabar en código real, explicaremos lo que es el patrón *competing consumers*, que será importante para el ejemplo que mostraremos más adelante.

### 1.3 Patrón competing consumers en RabbitMQ

El patrón Competing Consumers es una estrategia de procesamiento de mensajes en que múltiples consumidores compiten por los mensajes de una misma cola.

En lugar de tener un único consumidor procesando todos los mensajes, este patrón permite que varios consumidores trabajen en paralelo, distribuyéndoles la carga de trabajo y acelerando el procesamiento.

En RabbitMQ, este patrón se implementa fácilmente: simplemente conectas múltiples consumidores a la misma cola, y RabbitMQ se encarga de distribuir los mensajes entre ellos de manera equitativa.

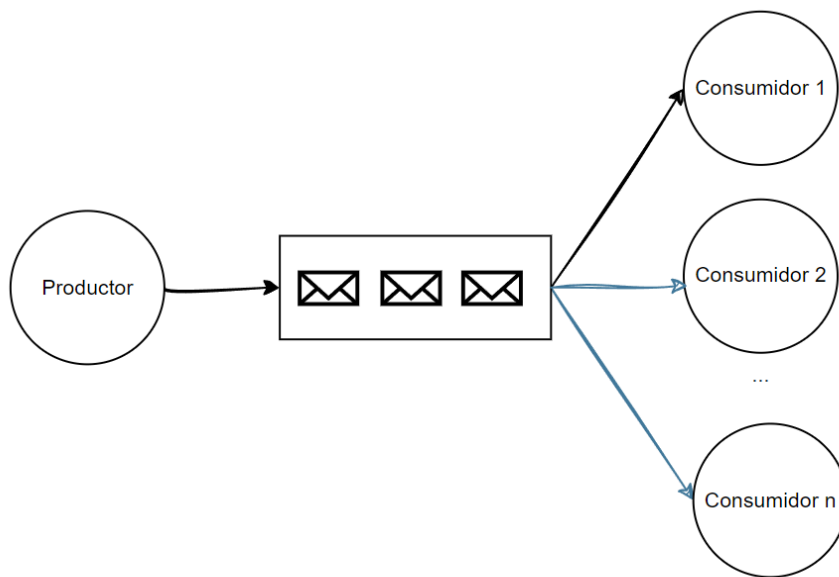


Figura 3. Cola de trabajo usando el patrón Competing Consumers

### 3. Ejemplo de RabbitMQ: Haciendo un auto-escalador de consumidores

Incursionaremos en el código de RabbitMQ a través de un ejemplo. Se mostrará el funcionamiento de un auto-escalador de consumidores que están asociados a una cola.

#### 3.1 Necesidad de un auto-escalador

En sistemas de alta demanda, la cantidad de mensajes en la cola puede variar de manera drástica. Si la carga de trabajo de una cola aumenta repentinamente, nos veremos en la necesidad de aumentar el número de consumidores para apaciguar la demanda, y en el caso de que baje la



carga, tendremos que bajar los consumidores para que no se usen recursos innecesariamente. Para no hacer esta tarea manualmente, podemos hacer un programa que añada (escale hacia arriba) o disminuya (escale hacia abajo) consumidores en base a la demanda de trabajo.

Y eso es lo que haremos. Implementaremos un autoescalador que reaccione y ajuste dinámicamente los consumidores según la demanda de esa cola. Usaremos Python mediante una librería llamada *Pika*.

## 3.2 Instalación de RabbitMQ

### 3.2.1 En Ubuntu

1. Instalar los repositorios requeridos:

```
sudo apt install curl gnupg -y  
curl -fsSL https://packages.rabbitmq.com/gpg | sudo apt-key add -
```

```
sudo add-apt-repository 'deb https://dl.bintray.com/rabbitmq/debian  
focal main'
```

2. Instalar RabbitMQ Server:

```
sudo apt update && sudo apt install rabbitmq-server -y
```

3. RabbitMQ se instalará como servicio. Después de instalar, activarlo:

```
sudo systemctl enable rabbitmq-server  
sudo systemctl start rabbitmq-server
```

4. Verificar si está corriendo:

```
sudo systemctl status rabbitmq-server
```

5. Activar herramientas adicionales:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

### 3.2.2 En Windows

En windows se puede instalar a través de Chocolatey.

Si no se tiene este instalador, hacerlo en el CMD vía este comando:

```
Set-ExecutionPolicy Bypass -Scope Process -Force;  
[System.Net.ServicePointManager]::SecurityProtocol =  
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072;  
iex ((New-Object  
System.Net.WebClient).DownloadString('https://community.choco  
latey.org/install.ps1'))
```

Luego:

```
choco install rabbitmq
```

Se puede verificar la instalación escribiendo:

```
rabbitmqctl status
```

Activar el plugin de gestión:

```
rabbitmq-plugins enable rabbitmq_management
```

## 3.3 Arquitectura de la implementación

La arquitectura de esta implementación es simple. Constará de 3 componentes:

1. Un **productor**, que enviará tareas en la cola.

2. Un programa que será el **autoescalador**. Aquí se definirán las reglas de escalado y el consumidor que escalará, basándose en métricas en tiempo real de la cola.
3. El **consumidor**. Este será solo ejecutado por el autoescalador según sus reglas predefinidas de escalado.
4. Una **broker** que actuará como intermediario en la comunicación entre productores y consumidores. Se compondrá solo de una cola que servirá de puente entre la comunicación entre el productor y los consumidores.

## 3.4 Implementación en código

### 3.4.1 Productor

Como mencionamos antes, los mensajes en lugar de publicarlos directamente en las colas, primero pasan por **exchanges**, que básicamente son enrutadores de los mensajes hacia las colas.

Los exchanges pueden ser de diferentes tipos, pero en este caso usaremos el más básico en que los mensajes pasarán directamente a la cola.

En primer lugar crearemos un script *productor.py* que funciona como emisor de mensajes. Usaremos *pika*, una biblioteca que funciona como driver de RabbitMQ para Python.

```
pip install pika
```

```
import pika
```

Se abre primero una conexión con el servicio de RabbitMQ que tenemos en nuestra máquina local:

```
connection =
```

```
pika.BlockingConnection(pika.ConnectionParameters('localhost'))
```

Ahora abriremos un canal. Un canal es una conexión TCP ligera hacia el servicio de RabbitMQ que tenemos en nuestra máquina local.

```
channel = connection.channel()
```

Declaramos nuestra cola y la identificaremos como *cola\_auto*:

```
channel.queue_declare(queue='cola_auto',durable=True, arguments={  
    'x-max-length': 1000  
})
```

El método `queue_declare` se utiliza para declarar una nueva cola en el canal donde se llama. Si la cola ya existe, esta instrucción no realiza ningún cambio (con los mismos argumentos). Esto quiere decir que podemos ejecutar el mismo código reiteradas veces sin temor de que se redeclare la cola ni se pierda.

Como se aprecia, además de definir el nombre, podemos establecer una serie de parámetros adicionales que determinan el comportamiento de la cola.

En este ejemplo, el argumento *durable=True* asegura que la cola será persistente, es decir, sobrevivirá a reinicios del broker.

Por otro lado, el parámetro `arguments` incluye la opción `'x-max-length': 1000`, que establece el número máximo de mensajes que puede contener la cola. Si se excede este número, los mensajes más antiguos serán eliminados automáticamente. Para ver las opciones disponibles, consultar [Queues | RabbitMQ](#).

Hasta este punto solo hemos declarado la cola y hecho algunas configuraciones básicas. Ahora veremos cómo publicar mensajes en la cola.

Para esto, usaremos el siguiente método:

```
channel.basic_publish(  
    exchange="",  
    routing_key='cola_auto',  
    body='message'  
)
```

El método `basic_publish` se utiliza para enviar un mensaje desde el productor a una exchange.

Posee como parámetros principales *exchange*, *routing key* y el mensaje en sí.

También se pueden agregar propiedades opcionales como *mandatory*, para que el mensaje sea entregado obligatoriamente a una cola), o *persistent* (para asegurar que el mensaje persista en disco (por defecto permanece en memoria principal) y sobreviva a un reinicio del broker.

En nuestro ejemplo usamos *exchange* con string vacío y un *routing\_key* con el nombre de la cola. Esto le dirá a RabbitMQ que inserte el mensaje directamente a la cola. El contenido del mensaje será un string “message”.

Finalmente, se cierra la conexión:

```
connection.close()
```

El código completo se ve así

```

import pika
import time
MESSAGES_PER_SECOND = 3


connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='cola_auto', arguments={
    'x-max-length': 1000
})

channel.basic_qos(prefetch_count=1)
message = "Message"

print(f"Enviando mensajes a una tasa de {MESSAGES_PER_SECOND}
mensajes/segundo...")

try:
    while True:
        start_time = time.time()

        for _ in range(MESSAGES_PER_SECOND):
            channel.basic_publish(
                exchange='',
                routing_key='cola_auto',
                body=message
            )

        elapsed_time = time.time() - start_time
        sleep_time = max(0, 1 - elapsed_time)
        time.sleep(sleep_time)

except KeyboardInterrupt:
    print("Deteniendo el productor...")

```

```
connection.close()
```

Creamos un bucle infinito para enviar mensajes a cierta tasa de mensajes por segundo definido por `MESSAGES_PER_SECOND`, esto con el fin de probar nuestro autoescalador y así, no ejecutar el productor cada vez que queremos mandar un mensaje.

### 3.4.2 Consumidor

Crearemos el script `consumidor.py` que permita conectarse a la cola “cola\_auto” y pueda consumir los mensajes en ella.

Tal como en el código anterior, se debe establecer la conexión con el servicio, abrir un canal que coincida con el del productor, y aunque parezca redundante, se debe ‘declarar’ la cola para asegurarse de que exista.

```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))

channel = connection.channel()

channel.queue_declare(queue='cola_trabajo', arguments={
    'x-max-length': 1000
})
```

Ahora crearemos el manejador del mensaje de modo que se pueda aplicar la lógica correspondiente de su procesamiento. Para esto, declaramos una

función callback que estará en espera y se ejecutará una vez que el consumidor reciba un mensaje.

```
def callback(ch, method, properties, body):  
    try:  
        time.sleep(2)  
        ch.basic_ack(delivery_tag=method.delivery_tag)  
  
    except Exception as e:  
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)  
  
channel.basic_qos(prefetch_count=1)  
  
channel.basic_consume(queue='cola_auto',  
on_message_callback=callback, auto_ack=False)
```

Los parámetros de la función son:

- ch (channel): representa el canal que maneja la conexión. Es el mismo declarado anteriormente.
- method: contiene información sobre el mensaje recibido, como la etiqueta de entrega delivery\_tag que se usa para confirmar el mensaje como procesado, o reencolarlo.
- properties: contiene propiedades del mensaje.
- body: es el contenido del mensaje en sí, generalmente en formato binario, que se puede decodificar para procesarlo.

Desglosamos cada línea a continuación:



Estas dos líneas representan el procesamiento de la tarea en nuestro ejemplo. Se imprime el mensaje decodificado usando el parametro `body`, y simplemente simularemos el tiempo de procesamiento de la tarea con el módulo `time` y estableceremos en este caso que durará 2 segundos.

```
print(f"Recibido: {body.decode()}")  
time.sleep(2)
```

Esta línea le dice a RabbitMQ que el mensaje se ha procesado correctamente. Por obvias razones siempre se pone luego de que la tarea se ha procesado.

```
ch.basic_ack(delivery_tag=method.delivery_tag)
```

En caso de que falle el código del manejador del mensaje, se saltará a la excepción y se enviará un aviso de confirmación negativo dándole indicación a RabbitMQ de que reencole el mensaje (`requeue=True`).

```
ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
```

Esta línea le dirá al consumidor que solo reciba un mensaje a la vez de la cola y no se le asignen más mensajes hasta que haya procesado y enviado un mensaje de confirmación (ACK) para el mensaje actual.

```
channel.basic_qos(prefetch_count=1)
```

Por último se tiene el método `basic_consume`. En esta línea se consume el mensaje y se llama para esto al callback definido anteriormente. Se indica la cola, el callback y la forma de confirmación de mensajes, que es con un

mensaje de confirmación manual.

```
channel.basic_consume(queue='cola_trabajo',  
on_message_callback=callback, auto_ack=False)
```

Se cierra la conexión:

```
channel.start_consuming()
```

El código completo es:

```
import pika
import time

connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost')
)
channel = connection.channel()

channel.queue_declare(queue='cola_auto', arguments={
    'x-max-length': 1000
})

def callback(ch, method, properties, body):
    try:
        time.sleep(2)
        print(f"Mensaje recibido: {body}")

        ch.basic_ack(delivery_tag=method.delivery_tag)

    except Exception as e:
```

```
        ch.basic_nack(delivery_tag=method.delivery_tag,
requeue=True)

channel.basic_qos(prefetch_count=1)

channel.basic_consume(queue='cola_auto',
on_message_callback=callback, auto_ack=False)

print("Worker ejecutado")
channel.start_consuming()
```

### 3.4.3 Autoescalador

El autoescalador que implementaremos monitoreará la cantidad de mensajes en la cola y ajustará dinámicamente el número de consumidores según la demanda. Además, tomaremos en cuenta el uso de CPU como un factor adicional en la estrategia de escalado.

Para ello, definiremos los siguientes parámetros:

- **Umbral de escalado hacia arriba y hacia abajo:** Define los límites de mensajes en la cola que activarán la adición o eliminación de consumidores. Si se supera el umbral superior, se agregará un nuevo consumidor; si se desciende por debajo del umbral inferior, se reducirá el número de consumidores.
- **Umbral de CPU:** Es el límite de uso de CPU que, al ser superado, desencadenará la adición de un nuevo consumidor para distribuir la carga de procesamiento.
- **Intervalo de monitoreo:** Frecuencia en segundos con la que el sistema verificará el estado de la cola y el uso de CPU para tomar decisiones de escalado.
- **Cantidad mínima y máxima de consumidores:** Establece los límites dentro de los cuales puede escalar el sistema, evitando un crecimiento descontrolado en el número de trabajadores.

Estos parámetros serán colocados en un archivo `.env`:

```
HOST=localhost
PORT=5672
USER=guest
PASSWORD=guest
QUEUE=cola_auto
VHOST=%2f
API_STATS_PORT=15672

MAX_WORKERS = 20
MIN_WORKERS = 1
MONITORING_INTERVAL = 3
QUEUE_THRESHOLD_MAX = 5
QUEUE_THRESHOLD_MIN = 2
CPU_THRESHOLD = 80
CONSUMER =consumidor
```

Este es el archivo de configuración completo. Se incluyen datos del servicio de nuestra máquina de RabbitMQ y el nombre del proceso del consumidor.

Crearemos el archivo del programa, llamado *autoescaler.py*.

Requeriremos una librería llamada *dotenv*, que servirá para importar las variables de entorno del archivo `.env`.

Como mencionamos, usaremos el número de mensajes que hay en la cola, necesitaremos una forma de extraer ese parámetro. Para esto, RabbitMQ ofrece un plugin que da acceso a una API HTTP con tales métricas. Ya lo hemos activado en la instalación. Para usarlo, se hacen llamadas al endpoint `http://server-name:15672/api/`.

En el código, lo primero que haremos será importar las variables de entorno:

```

HOST = os.getenv("HOST")
PORT = os.getenv("PORT")
USER = os.getenv("USER")
PASSWORD = os.getenv("PASSWORD")
QUEUE = os.getenv("QUEUE")
VHOST = os.getenv("VHOST")
API_STATS_PORT = os.getenv("API_STATS_PORT")

QUEUE_THRESHOLD_MAX = int(os.getenv("QUEUE_THRESHOLD_MAX"))
QUEUE_THRESHOLD_MIN = int(os.getenv("QUEUE_THRESHOLD_MIN"))
CPU_THRESHOLD = int(os.getenv("CPU_THRESHOLD"))
MAX_WORKERS = int(os.getenv("MAX_WORKERS"))
MIN_WORKERS = int(os.getenv("MIN_WORKERS"))
MONITORING_INTERVAL = int(os.getenv("MONITORING_INTERVAL"))
CONSUMER = os.getenv("CONSUMER")

```

Definiremos la URL del servicio donde está nuestra cola:

```

URL =
f"http://{HOST}:{API_STATS_PORT}/api/queues/{VHOST}/{QUEUE}"

```

Tal como hemos hecho con el productor y consumidor, estableceremos conexión con la cola:

```

connection =
pika.BlockingConnection(pika.ConnectionParameters(host=HOST))
channel = connection.channel()
channel.queue_declare(queue=QUEUE, arguments={
    'x-max-length': 1000
})

```

Crearemos una función que extraiga el tamaño en mensajes de la cola. Lo hará a través de una solicitud *get* hacia el endpoint dicho antes.

```

def get_queue_length():

```

```

try:
    response = requests.get(URL, auth=HTTPBasicAuth(USER,
PASSWORD))
    response.raise_for_status()
    data = response.json()
    queue_length = data.get("messages", 0)
    return queue_length
except requests.exceptions.RequestException as e:
    print(f"Error al conectar con RabbitMQ API: {e}")
    return -1

```

Ahora veremos como lanzaremos y pararemos los consumidores. Estos serán procesos del sistema

```

worker_processes = []

def start_worker():
    global worker_processes
    process = subprocess.Popen(['python3', f'{CONSUMER}.py'])
    worker_processes.append(process)

def stop_worker():
    global worker_processes
    process = worker_processes.pop()
    process.terminate()
    process.wait()

```

Cada consumidor activo estará en el array `worker_processes`.

Finalmente, crearemos el bucle principal del programa.

```

while True:
    queue_length = get_queue_length()
    cpu_percentage = psutil.cpu_percent(interval=0)

    print(f"CPU % {cpu_percentage} | Longitud de la cola:
{queue_length} mensajes | Workers activos:

```

```
{len(worker_processes)}")

    if (queue_length > QUEUE_THRESHOLD_MAX or cpu_percentage
> CPU_THRESHOLD) and len(worker_processes) < MAX_WORKERS:
        start_worker()

    elif (queue_length < QUEUE_THRESHOLD_MIN and
cpu_percentage < CPU_THRESHOLD) and len(worker_processes) >
MIN_WORKERS:
        stop_worker()


    time.sleep(MONITORING_INTERVAL)
```

#### 3.4.4 Herramienta visual de monitoreo

RabbitMQ ofrece un visualizador con diferentes gráficos y estadísticas de las colas, conexiones, canales, etc.

Para acceder a él, se pone en el navegador la URL con el host del servicio de RabbitMQ y el puerto 15672.

En nuestro caso, la URL es: <http://localhost:15672>



Username:  \*

Password:  \*

Login

Figura 4. Login de interfaz.

Por defecto, el usuario y contraseña de localhost son ambos 'guest'.

Al iniciar sesión se desplegará un panel como en la figura 5.

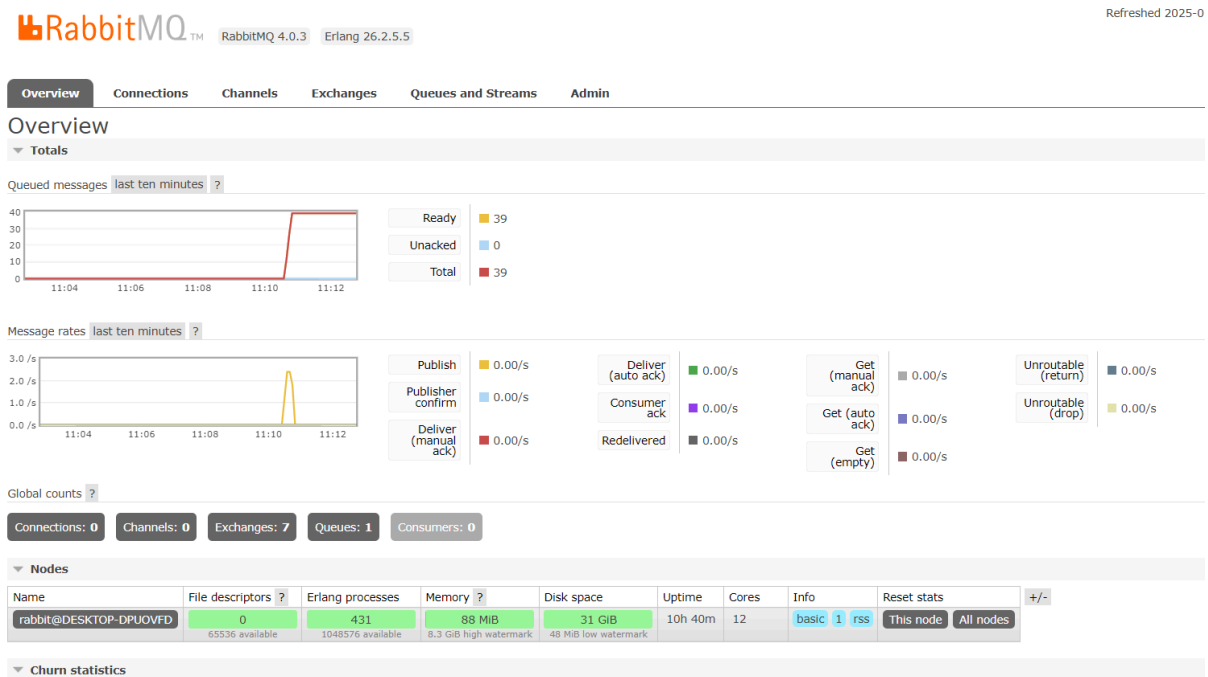


Figura 5. Panel de overview

Desde aquí se pueden visualizar los parámetros de canales, exchanges y colas, así como administrar usuarios y sus permisos. También se facilita la gestión de configuraciones del cluster y permite realizar operaciones como crear, modificar o eliminar colas y exchanges, o cerrar conexiones o canales.

En la pestaña *Queues and Streams*, podemos visualizar nuestra cola cuando ha sido creada.



## Queue cola\_auto

▼ Overview

Queued messages last ten minutes ?



Figura 5. Gráfica de la cola

Para ejemplificar el uso del programa creado, se hizo que el productor enviase constantemente 3 mensajes por segundo hacia la cola. Se activó el programa del autoescalador con un umbral de máximo 5 mensajes, un intervalo de monitoreo de 3 segundos y un límite de 20 consumidores. En la *figura 5*, se puede ver la gráfica con el número de mensajes a través del tiempo. Al principio la cola no tenía mensajes, y cuando se activa el productor, y sube repentinamente la carga de trabajo, el autoescalador reacciona y ejecuta productores hasta que la carga de mensajes se estabiliza en el tiempo.