Messaging systems

Summary:

- Messaging systems are asynchronous parallel systems
- In the background there is socket communication as well
- The system is asynchronous because we should not wait for the answer, execution flow is continuous, non-blocking.
- Each function call creates a message on the “Message Queue”. Message processing is alway parallel in a different process.
- This indirect method facilitated the loose coupling of different systems.
- Guaranteed message delivery is feasible, because of the intermediate message queue.
- Synchronous function calls can be simulated with a second message queue.

Message Queue implementations

A message queue is a software that enables communication between different software components in a distributed system. It allows components to exchange messages asynchronously, which can improve the overall reliability and scalability of the system. Message queues are commonly used in software integration, where they facilitate the exchange of messages between different applications, services, and systems.

RabbitMQ (https://www.rabbitmq.com/#features) is a popular open-source message broker that implements the Advanced Message Queuing Protocol AMQP (https://www.rabbitmq.com/resources/specs/amqp0-9-1). It allows applications to communicate with each other through a message queue, which can be hosted locally or in the cloud. RabbitMQ supports a wide range of messaging patterns, including point-to-point, publish-subscribe, and request-reply. It also provides features such as message persistence, routing, and priority queuing.

In RabbitMQ, messages are published by producers to a specific exchange, which routes them to one or more queues based on the specified routing key. Consumers then subscribe to the queues and receive messages. RabbitMQ supports multiple programming languages, including Java, Python, .NET, and Node.js, making it a versatile messaging solution for various use cases.

How to set up a queue in RabbitMQ:

- install RabbitMQ (e.g. in a docker container)
- create a connection: You need to establish a connection to RabbitMQ using a client
- create a channel: channel is a lightweight connection to RabbitMQ, which allows you to interact with the message broker. You can use the channel to declare queues, exchanges, and bindings.
- declare a queue: use the channel to declare a queue by specifying its name, durability, and other properties. For example, in Python, you can use the `queue_declare` method to create a queue:

```
channel.queue_declare(queue='my_queue', durable=True)
```

This code creates a durable queue called 'my_queue', which means the queue will survive a RabbitMQ broker restart.

- Publish messages: Now you can publish messages to the queue using the `basic_publish` method. In Python, you can do this as follows:

```
channel.basic_publish(exchange='', routing_key='my_queue', body='Hello,
world!')
```

This code publishes a message with the text “Hello, world!” to the 'my_queue' queue.

- Consume messages: Finally, you can consume messages from the queue using the `basic_consume` method. In Python, you can do this as follows:

```
def callback(ch, method, properties, body):
    print("Received message:", body)

channel.basic_consume(queue='my_queue', on_message_callback=callback,
auto_ack=True)

channel.start_consuming()
```

This code sets up a callback function that will be called every time a message is received from the 'my_queue' queue. The `auto_ack` parameter specifies whether to automatically acknowledge the message after it has been processed. Finally, the `start_consuming` method starts consuming messages from the queue.

Type of "Exchange" in RabbitMQ

An exchange in RabbitMQ is a messaging entity that receives messages from producers and routes them to queues based on some criteria. When a producer sends a message to RabbitMQ, it sends the message to an exchange. The exchange then examines the message's routing key and decides which queue(s) the message should be sent to.

There are four types of exchanges in RabbitMQ:

Direct Exchange: A direct exchange routes messages based on a routing key that is *matched exactly with the routing key* of the queue. When a message is sent to a direct exchange, RabbitMQ will deliver it to the queue(s) whose binding key exactly matches the routing key of the message.

- **For example**, a stock market application may send messages to a direct exchange with the routing key being the stock ticker symbol, and each queue bound to the exchange would represent a different stock. This way, the application can send specific messages to the appropriate queue, where consumers can consume them and perform actions based on the stock data.

```
            [Queue: Stock A]
            [Queue: Stock B]
[Direct Exchange: Stock Market] → [Queue: Stock C]
            [Queue: Stock D]
            [Queue: Stock E]
```

Topic Exchange A topic exchange routes messages based on matching the routing key of the message with one or more binding keys that the queue has specified. A binding key can contain one or more words, separated by dots. The routing key of the message is also a string with words separated by dots. The topic exchange uses a pattern matching algorithm to match the routing key of the message with the binding keys of the queues.

- **For example**, a blog platform may send messages to a topic exchange with the routing key being the topic of the blog post. Queues can then bind to the exchange using a matching pattern to receive messages that match certain criteria. For example, a queue bound to the exchange with the pattern “sports.#” would receive messages about sports topics, while a queue bound to the exchange with the pattern “#.technology” would receive messages about technology topics. This way, the application can route messages to the appropriate queue based on the topic of the blog post.

```
              [Queue: Sports]
[Topic Exchange: Blog Platform] → [Queue: Technology]
              [Queue: Politics]
              [Queue: Entertainment]
```

Fanout Exchange A fanout exchange routes messages to all queues that are bound to it, regardless of the routing key of the message. It is useful for broadcasting messages to multiple queues or multiple consumers.

- **For example**, a notification system may send messages to a fanout exchange when a new event is created. Each queue bound to the exchange would represent a different user, and all users should receive the notification. This way, the application can broadcast the message to all connected queues.

```
[Notification System] → [Fanout Exchange] → [Queue: User A]
                                            [Queue: User B]
                                            [Queue: User C]
                                            [Queue: User D]
```

Headers Exchange A headers exchange routes messages based on header values, instead of the routing key. The headers exchange examines the headers of the message and performs a match against the headers specified in the binding. If a match is found, the message is delivered to the corresponding queue.

- **For example**, a logistics system may send messages to a headers exchange with headers such as “destination” and “delivery_method”. Queues can then bind to the exchange using a matching set of headers to receive messages that match certain criteria. For example, a queue bound to the exchange with headers “destination=New York” and “delivery_method=air” would receive messages about shipments that are being delivered to New York by air. This way, the application can route messages to the appropriate queue based on the specific headers of the message.

```
[Logistics System] → [Headers Exchange] → [Queue: New York Air]
                                          [Queue: New York Sea]
                                          [Queue: Los Angeles Air]
                                          [Queue: Los Angeles Sea]
```

Each exchange type has its own routing algorithm and is used in different messaging scenarios.

Understanding the exchange types is important when designing RabbitMQ architectures that meet specific business requirements.

MQTT example

Clone repository into docker playground:

```
  git clone https://github.com/knehez/isi.git
```

docker-compose.yml defines a multi-container application with three services: mqtt, consumer, and producer.

The mqtt service is an instance of the *toke/mosquitto* Docker image, which is a popular open-source MQTT broker. The service is configured to automatically restart unless it is explicitly stopped by the user. Additionally, three volumes are defined for the service: “/mosquitto/conf”, “/mosquitto/data”, and “/mosquitto/log”. These volumes are used to persist the configuration, data, and log files for the MQTT broker respectively.

The consumer and producer services are both custom-built Docker images, which are defined using the build key. The context key specifies the build context, which in this case is the current directory (.), and the dockerfile key specifies the Dockerfile to use for the build. Additionally, a volume is defined for each service that maps the current directory to the “/app” directory in the container.

Finally, the depends_on key is used to specify that both the consumer and producer services depend on the mqtt service. This means that the mqtt service will be started before the other services, and will be available for use by those services.

docker-compose.yml

```
version: '3.7'
services:
  mqtt:
    image: toke/mosquitto
    restart: unless-stopped
    volumes:
      - ./conf:/mosquitto/conf
      - ./data:/mosquitto/data
      - ./log:/mosquitto/log

  consumer:
    build:
      context: .
      dockerfile: Dockerfile-consumer
    volumes:
        - .:/app
    depends_on:
      - mqtt
```

```
  producer:
    build:
      context: .
      dockerfile: Dockerfile-producer
    volumes:
        - .:/app
    depends_on:
      - mqtt
```

Dockerfile-consumer

This Dockerfile defines a simple containerized Python application that can be used as a consumer for a message broker. The dependencies are installed in the container, and the consumer code is copied into the container's /app directory. When the container is started, the consumer.py script is executed to consume messages from the broker.

```
FROM python:3.9-slim-buster

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY consumer.py .

CMD ["python", "-u", "consumer.py"]
```

Here is a breakdown of the different parts of the Dockerfile:

- FROM python:3.9-slim-buster: This line specifies the base image for the Docker image. In this case, the image is based on Python 3.9 running on a slimmed-down version of the Debian Buster Linux distribution.
- WORKDIR /app: This line sets the working directory for the container to /app. This is where the consumer code and other related files will be located.
- COPY requirements.txt .: This line copies the requirements.txt file from the current directory on the host machine to the /app directory in the container. The requirements.txt file lists the dependencies that the consumer requires to run.
- RUN pip install –no-cache-dir -r requirements.txt: This line installs the dependencies listed in the requirements.txt file using pip. The –no-cache-dir option is used to ensure that pip does not cache the downloaded packages, which can help to reduce the size of the Docker image.
- COPY consumer.py .: This line copies the consumer.py file from the current directory on the host machine to the /app directory in the container. This is the main code file for the consumer.
- CMD [“python”, “-u”, “consumer.py”]: This line specifies the command to run when the container is started. In this case, it runs the consumer.py script using the Python interpreter (python). The -u flag is used to enable unbuffered output, which ensures that log messages are immediately visible in the console.