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.