COMP.SE.140 - AMQP Exercise

VERSION HISTORY

1	27.10.2022	Initial version
2	01.11.2022	Host and docker versions need to be documented + other clarifications form 01.11 discussions session.

Note: this will be a component in our project, so make it reusable.

OVERVIEW

In this exercise, you familiarize yourself with message gueue and message-bus-based service architecture and RabbitMQ that implements this paradigm. You implement message routing based on topics and message queues as well as solidify your expertise related to Docker Compose.

RabbitMQ (https://www.rabbitmq.com) implements AMQP 0-9-1 (Advanced Message Queuing Protocol), which specifies a Message-oriented Middleware (MOM) that can route any type of data. It enables message routing and "push" communication, which is more efficient compared to HTTP when data is updated frequently, as there are no repeated requests for updated values.

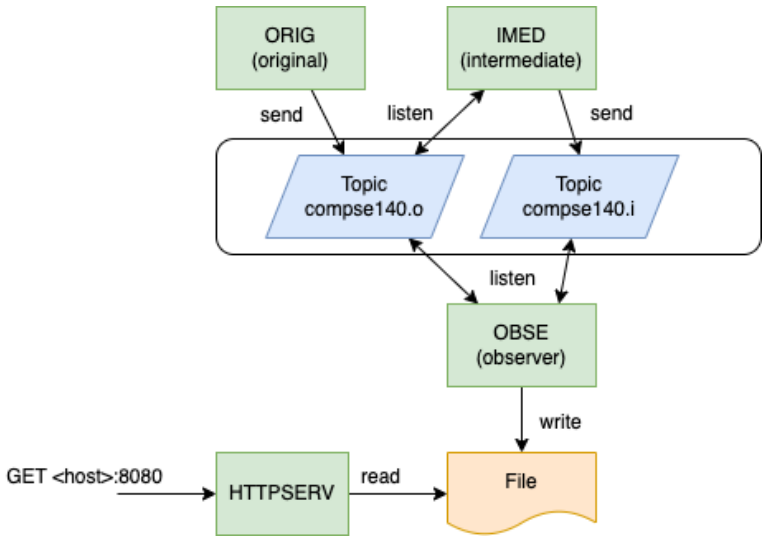
The advantages of topic-based routing are loose coupling and scalability in cloud-native applications. Because each network node directly refers to topics only, the nodes do not know the identity one another. Furthermore, the topics route messages to queues to wait for processing, letting each application process the incoming messages as they can without blocking others. Loose coupling adds flexibility to system updates and facilitates testing, making it straightforward to implement mocks and replace these later with the actual implementation. On the other hand, each topic can deliver messages to an arbitrary number of applications, although the sender publishes each message only once. That is, the Message-Oriented Middleware (MOM) creates as many copies of the message as needed. This adds to scalability in case the initial message source has computational constraints (e.g., a microcontroller with a sensor). On the other hand, if the MOM runs out of resources, load balancing is possible in AMQP.

LEARNING GOALS

- Learn the basics of message-queues and topic-based communication with RabbitMQ and how to implement topic-based publish-subscribe communication for message routing
- Realize the benefits of asynchronous communication
- Learn how to build a more complex systems with Docker Compose and how to use volumes in Docker Compose

COMMUNICATION ARCHITECTURE

The figure below illustrates the system to be implemented.



That is:

- You implement message routing with topics named as
  - compse140.o
  - compse140.i
- You develop the following applications (or Docker images/containers)
  - ORIG (Original)
    - Publishes messages to *compse140.o*
  - IMED (Intermediate)
    - Subscribes for messages from *compse140.o*
    - Publishes message to *compse140.i*
  - OBSE (Observer)
    - Subscribes for all messages within the network, therefore receiving from both *compse140.o* and *compse140.i*
    - Stores the messages into a file
  - HTTPSERV
    - When requested, returns content of the file created by OBSE
- You use RabbitMQ as the broker
  - This is readily available as a Docker image (see for instance https://www.rabbitmq.com/download.html)
- You use Docker Compose to orchestrate ORIG, IMED, OBSE, HTTPSERV and a RabbitMQ broker into one entity

FUNCTIONALITY

ORIG

ORIG publishes 3 messages to topic *compse140.o*. After sending one message, it waits for 3 seconds. The message content shall be “MSG_{n}” without quotes, where {n} is replaced with an incrementing value within [1..3]. That is:

- MSG_1
- (Wait for 3 seconds)
- MSG_2
- (Wait for 3 seconds)
- MSG_3

After that ORIG service stays idle (does not exits).

IMED

Every time IMED receives a message from topic *compse140.o*:

- IMED waits for 1 second
- After waiting, IMED publishes "Got {received message}" without quotes to topic *compse.i*
    - Replace "{received message}" with the content of the received message

OBSE

Every time OBSE receives a message from any of the topics:

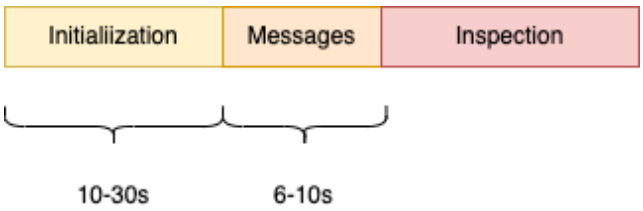
- OBSE builds a string "{timestamp} {n} {message} to {topic}:" without quotes
    - {timestamp} must be in the format *YYYY-MM-DDThh:mm:ss.sssZ* (ISO 8601)
        - Time zone is UTC
    - {n} the running number of the observed message, starting from 1
    - {topic} is the topic that delivered the message
    - {message} is the message body
    - Do not print '{' and '}'
    - For example:
      `2022-10-01T06:35:01.373Z 2 MSG_1 to compse140.o`
- OBSE writes the string into a file in a Docker volume
    - If OBSE is run multiple times, the file must be deleted/cleared on startup

HTTPSERV

- When requested, returns content of the file created by OBSE
    - Nothing else must be returned
- Port: 8080

ADDITIONAL NOTES

The initialization of the system takes time and the messages should not be sent before RabbitMQ has been initialized. Also, the teacher should not test before the messaging phase has been finalized. The assumed timing is:



This means that your services should wait until the RabbitMQ is ready,

On the containers start, the messaging sequence must not take more than 30 seconds to complete! I.e., this is the time from "docker-compose up" to the last printout to the file. Your services may fail, if they try to communicate with RabbitMQ before it is ready. You should implement your services so

that they wait until the RabbitMQ is ready. A straightforward way to just sleep for n seconds does not give full points. There is no sensible upper limit for the sleep time. Pay also attention for not sending before topic is been listened.

If you use a slow virtual machine, it make take even longer than the 30s shown in the above picture.

Port 8080 is the only port that is exposed from the private network built by docker.

Like previously, you can choose the programming language. However, do not use extra frameworks without agreeing with the teaching staff. As an example, for C#/.NET users it is ok to use this approach: https://github.com/seank-com/dotnetcore-console-http-server
(See the email 27.10.2022)

SUBMITTING THE ASSIGNMENT

Return the following files:

- Source code of your application
- Docker Compose file (YAML)
- All Docker files
- Any other files required to build and run the system
- A document (PDF or README.md) in which you cover at least
    - Information about the host; output of :
      `uname -a; docker --version; docker-compose --version`
      or equal.
    - Perceived (in your mind) benefits of the topic-based communication compared to request-response (HTTP)
    - Your main learnings

The files are returned with some Git service – you have a courses gitlab as one alternative. Please prepare your system in a way that the course staff can test the system with the following procedure (on Linux):

```
$ git clone -b messaging <the git url you gave>
$ docker-compose build --no-cache
$ docker-compose up -d
(Wait about 30 seconds, depend on how powerful computer the teacher uses.)
$ curl localhost:8080
<output should follow the requirements>
$ docker-compose down
```

HINTS

Concepts and examples. The RabbitMQ website explains many AMQP-related concepts and provides code examples. This guides you how to implement topic-based communication (please note availability in multiple languages): https://www.rabbitmq.com/tutorials/tutorial-five-python.html or https://www.rabbitmq.com/tutorials/tutorial-five-javascript.html

RabbitMQ image. In case you want a browser interface to look at how your RabbitMQ is doing, you can use an image that comes with the related plugin, such as “rabbitmq:3-management”. For more information, see https://www.rabbitmq.com/management.html

RabbitMQ client libraries. These are many. Not all are listed in RabbitMQ website, although you are probably fine with the ones found there. Python users can try *aio-pika* for a truly asynchronous API.

Timing in applications. You cannot start messaging immediately after startup, as it takes a while from all containers to start up. Therefore, you probably need a delay before messaging.

Wildcard subscription. To subscribe for messages from all topics, the routing key must be "#" (without quotes).

ABOUT GRADING

Again, total 12 points are available.

6 points comes from correct working of the system. Reductions for example from

- Extra ports are exposed
- Synchronization of the services is based just on "sleep"
- The syntax of the messages is wrong
- The content returned by HTTPSERV is not text/plain
- …

3 points from good practices: clean code, sensible comment (but obvious things are not commented)

3 points from the information content of the document.