COMP.SE.140 – Docker-compose hands on

Version history

V0.1 28.09.2023	Initial version for course staff only
V0.2 01.10.2023	First set of bug fixed
V1.0 03.10.2023	First published version

Synopsis

The purpose of this exercise is to learn (or recap) how several microservices can communicate with each other. You now need have three (+one) microservices that communicate with each other with in three different ways. The learnings from Web Development 2 and large scale SW design will help you. Code from the previous exercise can be re-used.

Learning goals

- How to implement the different communication methods in practice.
- Understand the differences, benefits and disadvantages of the different approaches.

Task definition

This exercise includes optional (bonus) functionality.

In this exercise we will build a simple system composed of three small services implemented in addition to messaging broker. The services are small programs running in separate containers. Thus, the target is the following:

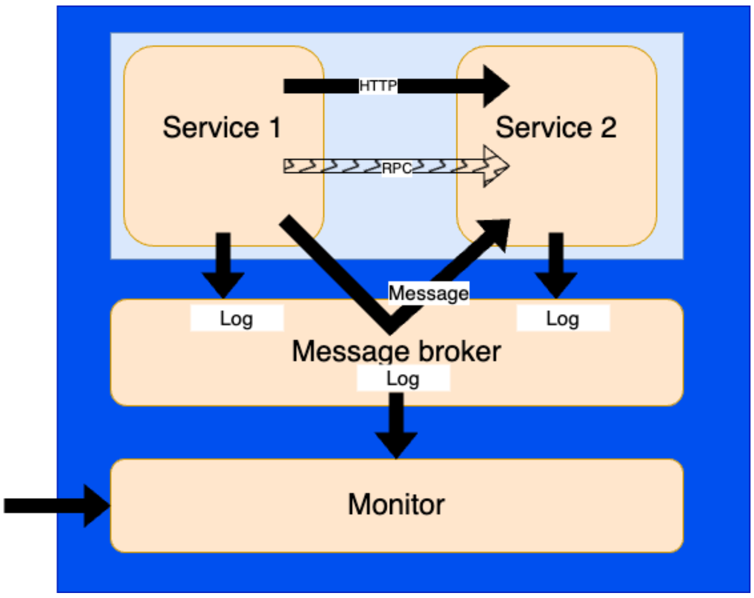


Figure 1. Architecture of the target system.

Note: RPC communication is optional and successful implementation gives 1-5 bonus points.

Compared to the previous exercise, the logs are not stored in files. Instead, the monitor application keeps them. The content can be read with an HTTP-request.

Service/application 1 should repeatedly 20 times in 2 second intervals:

- Compose a **text** from a counter (initialized to 1) current time and address+port of service2. Follow this example syntax precisely:
  ```
  SND 1 2022-10-01T06:35:01.373Z 192.168.2.22:8000
  ```
- Send the above text to message broker (RabbitMQ - topic "message") for Service2
- Send the text with HTTP protocol to service 2. (i.e., the same text is sent through another channel)

- Send the response code with a time stamp of the above topic “log” . Example string:
  `200 2022-10-01T06:35:01.973Z`
- *Optional: Send an RPC (gRPC recommended) request to service 2. Parameter of the called service is the string that was sent over HTTP and RabbitMQ already.*
- *Optional: after the call returns the following string is formed*
  `RPC X`
  where “X” is the returned value. The formed string is sent to message broker (topic “message”).
- If either sending fails, catch the exception and send the error message to Message Broker (topic “log”)
- Increase the counter with 1

After the 20 rounds

- Send “SND STOP” to message broker (topic “log”)
- Wait for the operator to issue command “docker compose down”

Service/application 2 should

- Wait for 2 seconds
- Establish an HTTP server that listens in port 8000
- Listen topic “message” of the message broker.
- *Optional: establish an RPC (gRPC) service “count0”.*
- *Optional: when the RCP call arrives, the service counts the number of zero-characters (0) in the received string and returns the result as a number.*
- As a response to incoming HTTP message create a new text that adds the remote address (address of service 1). An example:
  `SND 1 2022-10-01T06:35:01.373Z 192.168.2.22:8000 192.168.2.21:78390`
  Send the above text to message broker (RabbitMQ - topic “log”)
- As a response to incoming message through message broker, create a new text that adds the “MSG” to the string. An example:
  `SND 1 2022-10-01T06:35:01.373Z 192.168.2.22:8000 MSG`
  Send the above text to message broker (RabbitMQ - topic “log”)
- Do not stop, but wait for the operator to issue command “docker compose down”.

Monitor listens topic “log” of the message broker and keeps the received messages in the memory. Monitor also listens GET requests in port 8087, and as a response returns the list of received strings from the message broker – MIME-type “text/plain” and each message on a separate line.

As the service should run in separate containers, you should write *Dockerfiles* for the both services and *docker-compose.yaml* to start both containers and connect then with a private network. In docker-compose.yaml make the port 8087 visible to outside, too. (but not other ports)

For fluent testing by teaching staff you should do your best to ensure that that both docker images and running containers have unique names from your other students.

Some notes

The IP address may be IP4 or IP6 address – depending on the system. For example the “:ffff:”-prefix provided by some libraries can be included.

By remote address/port we mean the address of the service that sent the request.

The built images should have the application installed. Do not “install” it by bringing it on a volume.

Start a new GIT branch “exercise2” – do not change files in exercise1!

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.

Submitting for grading

After the system is ready the student should return (in the git repository – in branch "exercise2").

- Content of all Docker and docker-compose.yaml files
- Output of "**`curl localhost:8087`**" (executed when the services are up and running.) in a text file
- Source codes of the applications.
- Information about the host; output of :

```
uname -a; docker --version; docker-compose --version
```

  or equal in the Readme.md
- Report on using AI-based tools in case they were used.

Please do not include extra files in the repository.

These files are returned with some git service like in the previous exercise.

You should prepare your system in a way that the course staff can test the system with the following procedure (on Linux):

```
$ git clone -b exercise2 <the git url you gave>
$ docker-compose up –build
… wait a minute or two
$ curl localhost:8087
$ docker-compose down
```

Grading

The points from this exercise depend on timing and content:

- Maximum 8 standard points are given.
- missing the first deadline (23.10.2023): points reduced by 1 points / starting day.
  The absolute deadline is 30.10.2023.
- how well the requirements (including technical instructions to the submit your project) are met: 6p
- following the good programming and docker practices: 2p
- 1-5 bonus points if (g)RPC-functionality is implemented

On using ChatGPT or similar AI (large language models - LLM) tools

The university-level guidelines say:

> *"If a student uses a language model in an assignment or a thesis, for example, as part of language editing, this must always be mentioned. When individual students describe their use of language models, we can share good practices. The use of a language model for language revision is justified, for example, to produce a grammatically or structurally fluent text (cf. proofreading and translation tools and similar tools)."*

In this exercise we interpret this as follows

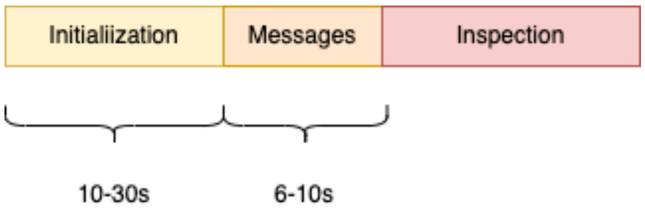
- If language models are used, a separate report (~page) must be written (included in the submission). This report includes:
    - The used LLM tool
    - Motivation/reason to use LLM
    - How and why LLM helped
    - What kind of mistakes LLM did
    - What were things that LLM was not able to provide
- You allow course staff to use this report in grading and use of it (after anonymization) for teaching development/research.
- The course staff will investigate different ways to discover use of LLM – students using LLM without reporting it, will be discontinued from the course.

Hints

Do not provide the link from the browser (the one you see when you access your repo with a GUI) – that does not work with git clone.
If you use Python, note that some libraries do not provide access to remote address. Do not use such library!

Before submitting, remember to test that everything works on top of clean table. (e.g. remove docker images, networks, etc.)

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 listened.

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

More robust solution can be found with special waiting scripts like
https://github.com/vishnubob/wait-for-it#

Useful material:
https://www.rabbitmq.com, https://www.rabbitmq.com/documentation.html (see "Tutorials")

https://grpc.io/docs/languages/node/quickstart/