Attention! Due to Courses website experiencing technical difficulties, files and courses uploaded before the 2023 spring semester are not accessible. We are working on fixing the problem. In case of further problems, write to ati.error@ut.ee

Cloud Computing 2022/23 spring

Practice 10 - Introduction to Message Broker - RabbitMQ

The purpose of this practice session is to provide an overview and working of the message queues and the transmission of the messages using publish/subscribe mechanism. For this experience, we use the open-source distributed message broker RabbitMQ. You will learn how to set up RabbitMQ in the cloud, and how to create an exchange, queues, and set up routing rules. Further, you will work with distributed processing by creating multiple processes which subscribe to and publish messages in parallel.

References

1. More about RabbitMQ concepts https://www.rabbitmq.com/tutorials/amqp-concepts.html
2. RabbitMQ Python Library, Long Documentation: https://pika.readthedocs.io/en/latest/intro.html
3. Long examples: https://pika.readthedocs.io/en/latest/examples.html

If you have problems, check:

1. Possible problems and their potential solutions at the end of this guide.
2. Ask in the Zulip channels

Exercise 10.1. Setting up RabbitMQ in OpenStack

In this task, we set up RabbitMQ in OpenStack VM using Docker CE. Further, we use the RabbitMQ administration interface to manage RabbitMQ entities and simpler steps for publishing and subscribing to data streams. This interface also helps us investigate whether data arrives at the Broker and what are the currently active queues, connections, and relationships.

Create an OpenStack VM instance:

- **Source**: Instead of an *Image*, use **Volume Snapshot**, choose `Ubuntu22+Docker`
  - In this Ubuntu VM snapshot, the installation of Docker (as we did in Lab 2) has already been done for us.
- **Flavour**: should be `m2.tiny`
- Don't forget to enable "Delete volume on instance deletion"!

Setup RabbitMQ:

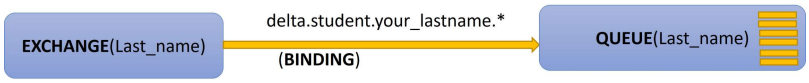
- Create a RabbitMQ Container
  - We configure it to run in background mode (-d)
  - Set environment variables **RABBITMQ_DEFAULT_USER** and **RABBITMQ_DEFAULT_PASS**
  - Port forward RabbitMQ container ports to host ports:
    - `5672:5672` (broker)
    - `80:15672` (RabbitMQ admin user interface)
  - RabbitMQ container image: `rabbitmq:3-management`
  - In the command, replace <**LASTNAME**> with your last name and set **CUSTOM_PASSWORD**
  - The complete command should look like this:

- `docker run -d --hostname <LAST_NAME>-rabbit -p 5672:5672 -p 80:15672 -e RABBITMQ_DEFAULT_USER=admin -e RABBITMQ_DEFAULT_PASS=CUSTOM_PASSWORD --name <LAST_NAME>rabbit rabbitmq:3-management`

Accessing RabbitMQ and working with message publishing and routing:

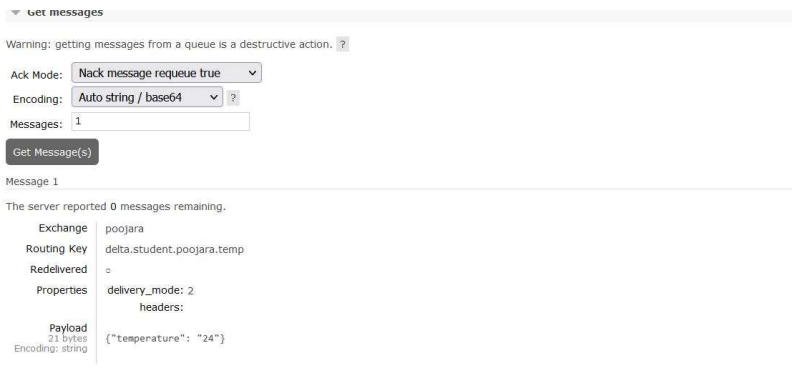
- You can access the RabbitMQ administration interface at http://<VM IP>:80 and log in using the credentials set while creating the container.
- Get familiar with the different tabs available, such as:
  - **Connections**: It shows the connections established to the RabbitMQ server.
  - **Channels**: Information about all current channels between clients and the RabbitMQ server.
  - **Exchanges**: It lists the available exchanges. An exchange receives messages from producers and users, or applications can define bindings between exchanges and queues, which define which data is delivered to a queue.
  - **Queues**: It shows the queues which are already defined in the RabbitMQ server.

- Now, let us create a new **exchange**. Click on **Exchanges--> Add a new Exchange**.
  - Name: `LASTNAME` Give your last name. It should not contain punctuation or special letters.
  - Type: `topic`
  - Leave other details as default.
- Let us publish a new message manually using the admin interface. For this, click on your newly created exchange and got to publish a message
  - Routing key: `delta.student.your_lastname.temperature`
  - Payload: `{"temperature": "24"}`
  - Leave other details as default.
- Create a new queue. Click on **Queues-->Add a new queue**
  - Clients will be able to subscribe to this queue to listen to messages.
  - Name: `your_lastname_queue`
  - Leave other details as default.
- Create a new Binding. Click on the newly created queue and then go to **Bindings**
  - Binding defines which data (based on routing key value/pattern) is routed from an exchange to a specific queue.
  - From exchange: Your (previously created) Exchange name
  - Routing key: `delta.student.your_lastname.*`
  - Leave other details as default.
- The overall **Exchange**, **Binding** and **Queue** looks like



- Now, let us test by Manually republishing the same message you created in Exchange through the administration interface as before. (NB! do not do this through the Queue page but through the Echanges page)

- You should NOT see a message that message was "not routed"
- Check for the messages that arrived in the queue
  - Go to **Queues--> Get Messages**. You should see the output as below



- Take a screenshot of the situation where you can see that the queue was successfully retrieved. (NB! Please make sure that your rabbitMQ server IP is visible on the screenshot)

Exercise 10.2. Publishing and Consuming the Messages using RabbitMQ

In this task, we are going to learn to publish and subscribe to messages to/from RabbitMQ using the Python library (Pika).

Publish Messages:

We create a new Python program (Same as in the previous practice session. For example, using the PyCharm IDE), or you create it in the VM using nano. The program will use the following instructions to publish a message. Before writing the Python program, Please install the Pika library using pip (pip3 install pika) or in PyCharm IDE, follow the Practice Session 8 instructions to add a Python library.

- Import the libraries

```
import pika
import random
import datetime
import json
```

- Set up the details of the connection:
  - Broker location: `broker_host = VM_IP` (replace VM_IP to Your VM IP address)
  - Broker port: `broker_port = 5672`
- Configuring authentication. Here, we use a username and password while creating the rabbitmq container to create a Long PlainCredentials object:
  -

```
username = "..."
password = "..."
credentials = pika.PlainCredentials (username, password)
```

- Create a connection string
  -

```
connection = pika.BlockingConnection (
        pika.ConnectionParameters (host = broker_host, port = broker_port, credentials = credentials))
channel = connection.channel()
```

  - Here, channel () creates a new channel through which we can send and listen to data to the RabbitMQ broker.
  - Here, we use BlockingConnection: It is a blocking, or synchronous, connection to RabbitMQ. This means that our program will wait for confirmation that the data sent has been entered into the RabbitMQ **Exchange**. You can also use **SelectConnection** for asynchronous communication.
- Define a sample message to send using Python dictionaries
  -

```
 message = {
        "device_name": "your_last_name_sensor",
        "temperature": random.randint (20, 40),
        "time": str (datetime.datetime.now())
    }
```

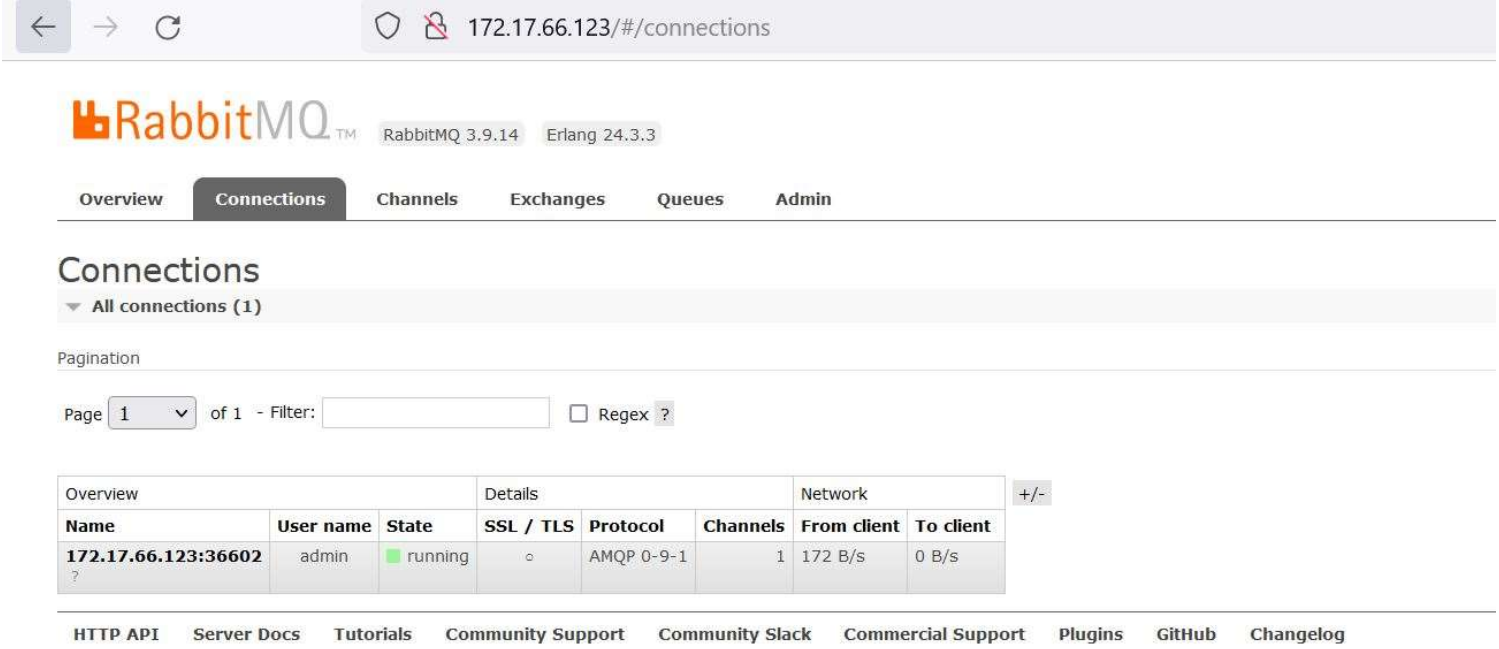
  - Convert the message dictionary to a string value using the JSON library: `message_str = json.dumps(message)`
- Finally, we will publish the data to the broker
  - First of all set rout key: `routing_key = "iotdevice.your_last_name.tempsensor"`
  - Define your exchange `exchange=your_exchange` replace here <your_exchange> that you created in previous task
  - Publish statement every five seconds and should send the messages until you interrupt with the keyboard. Here, use either a sleep function for 5 seconds for each publish or maybe try out some other ideas.

```
channel.basic_publish (
        exchange = exchange,
        routing_key = routing_key,
        body = message_str)
```

- At last, should close the channel connection using

```
channel.close()
connection.close()
```

- Run the Python program to publish the messages; check for the live connection and channel in the admin web interface, and it should look like as below screenshot.
  - It may be necessary to leave the connections open while making the screenshot. Feel free to add some code (e.g. for sleep) to pause the script before you close the RabbitMQ session.



Take a screenshot of the situation where you can see the live connections in the RabbitMQ web interface. (NB! Please make sure that your IP is visible on the screenshot)

Subscribe to Messages:

Create a Python program that listens for messages from the RabbitMQ server and prints them out.

- Make a copy of the previous Python program
- We will reuse the code until `channel` the object is created.
- Create a queue:
  - Name the queue `queue_name = "last_name_queue"`
  - Create the queue `channel.queue_declare(queue=queue_name, durable=True)`, Here we use durable queues, that data is queued even if there is currently no listener.
- Create a Routing key
  - Let us listen to all temperature sensors under iotdevice, not just yours:
    - `routing_key = "iotdevice.*.tempsensor"`
    - NB! Because students all have separate RabbitMQ servers, you don't actually see each other's data here.
- Create a Binding between the Queue and Exchange:
  `channel.queue_bind(exchange=exchange, queue=queue_name, routing_key=routing_key)`
- Create a separate Python function that will be called later for each message, and The feature should print the contents of the incoming message and the routing key.

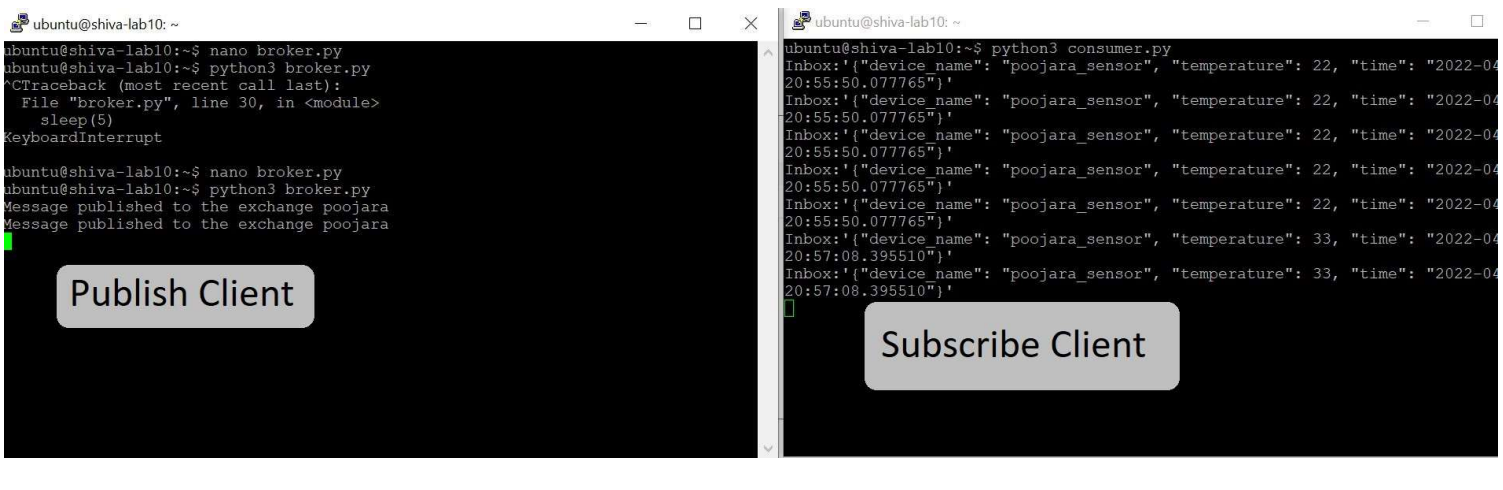
```
def lab_callback (ch, method, properties, body):
    print ("Inbox:% r"% body.decode())
    ch.basic_ack (delivery_tag = method.delivery_tag)
```

- We set the application to listen to the RabbitMQ queue and run our lab_callback function on every incoming message.

```
channel.basic_consume(queue = queue_name, on_message_callback = lab_callback)
channel.start_consuming()
```

Testing publish and consume (subscribe) clients:

Now run both the publish and consume Python codes on different terminals and should show the output below:



Take a screenshot of both publisher and consumer terminals and save both codes for submission.

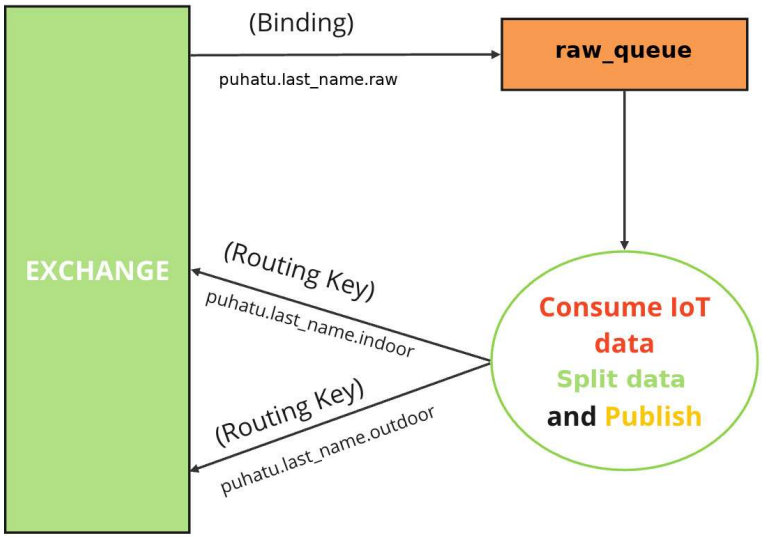
Exercise 10.3. Message Processing and Routing

This task will help you to explore how to subscribe to the messages. process them and publish and route them to the other output queues. For this, we will use real-time IoT data.

Dataset description:

[ ]

In this exercise, we will mimic the sensor behavior and publish the sensor data to the exchange. We will write a new Python application that listens for messages from the input queue, and launches a user-defined function (to parse the raw sensor data, extract the required fields and split the messages based on whether it arrives from an indoor testing device or on the bog device using device id), and publishes the results to other output queues. The overall scenario to be designed in this exercise is as shown below:



Publish the IoT data to Exchange:

Create a python program to publish the IoT data by reading from a CSV file at intervals every 2 seconds.

- Make a copy of the previous publisher Python program
- We will add our code after the line `channel = connection.channel()`
- Read the csv using Pandas data frame.
  - Import and install Pandas Python package
  - Can use pandas.read_csv(file_path) method to directly read in csv file.
- Convert the dataframe row in to json documents like `messages= json.loads(dataframe.to_json(orient="records"))`
- Define rouring_key and exchange
  - `routing_key = "puhatu.last_name.raw"`
  - `exchange = your_exchange` (replace with earlier created exchange)
- Write the code to publish the messages every 2 seconds
  - Iterate over the messages
  - convert the json message to a string using json.dumps method like this: `message = json.dumps(messages[i])`
  - Use basic_publish to publish the message
  - Write the print statement like print("A new Message published to the exchange: ", message)
  - Make the script wait/sleep for few seconds between sending messages
- Close the channel and connection.
- After running the program, each row of the CSV published as a JSON document to RabbitMQ Exchange

To be able to validate that everything worked properly, let's create a queue and check the messages arrive:

- Create a new queue in the RabbitMQ named: `puhatu_queue`
  - You can do this manually through the web interface or from the code.
  - Create a new binding that routes all messages with the routing key "puhatu.last_name.raw" to this queue
- You can see the message in the admin interface under puhatu_queue below:

Take a screenshot of the publisher as shown above (Make sure your VM IP is visible).

Subscribe, Process the message (raw IoT data), and Publish again to exchange:

Here, we consume the IoT data which is published above and process to extract the `dev_id` field, then find that the IoT device is an indoor or outdoor device, and publish the data with another routing key, which marks whether data is from indoor or outdoor device.

- Make a copy of the previous subscriber Python program from Excerices 10.2
- First, let's have the script programmatically create a new queue (which we created manually before) that our program will listen to and also define a new binding to route the raw Puhatu data to the queue:
  - ```
    queue_name = "puhatu_queue"
    input_routing_key = "puhatu.last_name.raw"
    # Create a new queue
    channel.queue_declare(queue = queue_name, durable = True)
    # Binding Exchange and queue with the routing key
    channel.queue_bind(exchange = exchange, queue = queue_name, routing_key = input_routing_key)
    ```
- Update the **lab_callback** method with the following things:
  - Get the message and decode `message = json.loads(body.decode())`
  - Check whether message['dev_id'] is one of the following:
    - **indoor_devices** = ['fipy_e1', 'fipy_b1','fipy_b2','fipy_b3']
      - If it is an indoor device, set the `input_routing_key = "puhatu.last_name.indoor"`
    - or *outdoor_devices = ['puhatu_b1','puhatu_b2','puhatu_b3','puhatu_c1','puhatu_c2','puhatu_c3','puhatu_l1']*'
      - If it is an outdoor device, set the `output_routing_key = "puhatu.last_name.outdoor"`
  - This will split the messages into two streams using routing keys - as `outdoor` or `indoor`.
    - Then you can republish the original message string object to RabbitMQ - but now with a different routing key.
    - Publish the modified message to the output queue:
      - `ch.basic_publish(exchange = exchange, routing_key = output_routing_key, body = body.decode())`

To be able to test that the messages are published correctly, let's also create the queues: indoor_queue and outdoor_queue:

- You can do this manually through the admin interface or using the following code:

```
# indoor
channel.queue_declare(queue ="indoor_queue", durable = True)
channel.queue_bind(exchange = exchange, queue ="indoor_queue", routing_key = "puhatu.last_name.indoor")
# Outdoor
channel.queue_declare(queue = "outdoor_queue", durable = True)
channel.queue_bind(exchange = exchange, queue ="outdoor_queue", routing_key="puhatu.last_name.outdoor")
```

Testing the end-to-end system:

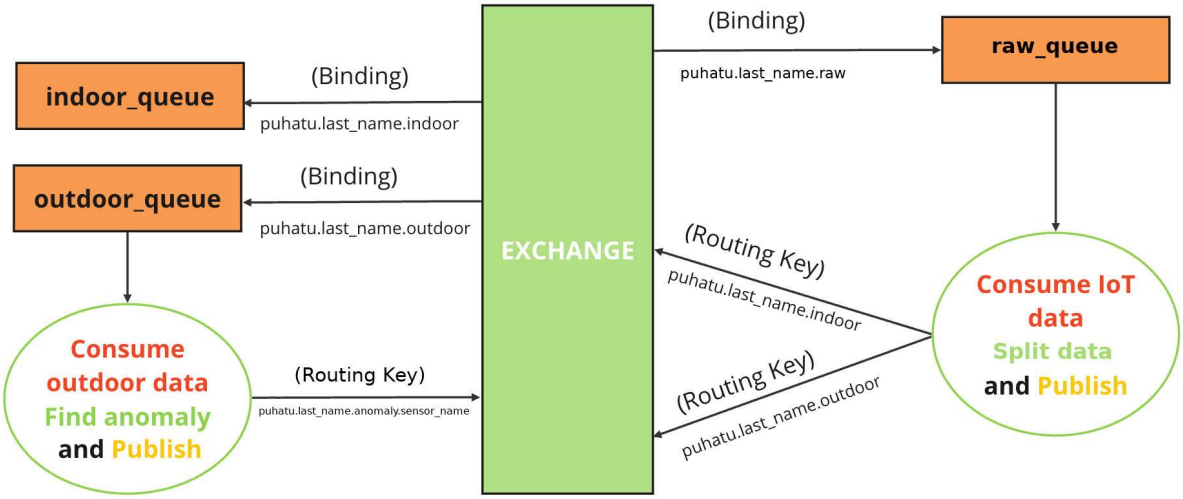
- Run the publisher in one terminal
- Run the consumer in another terminal
- Now, you should see the queued, printed messages in the administrative dashboard under **Queues-->outdoor_queue-->Get Messages**

Take a screenshot of both of the queues in the RabbitMQ admin interface. It should be visible from the screenshot that data has arrived in the queue. (Make sure the IP of your VM is visible)

Exercise 10.4. Anomaly detection

Consume the **outdoor_queue** and process messages arriving in the queue to find anomalies. Write a subscriber program to find anomalies in sensor data and publish the anomalous data using the routing key "puhatu.last_name.anomaly.sensor_name"
```

The overall scenario to be extended in this exercise is as shown below:



- Consder two sensors **dist** and **air_Temp_float**.
- Consider **outdoor_queue** as input queue.
- The fields can be accessed from the message as message['dist']
- The **dist** > 70 considered as anomaly and so is **air_Temp_float** > 25.
- When an anomaly is found:
  - Publish the message back to exchange with the routing key "puhatu.last_name.anomaly.dist" or "puhatu.last_name.anomaly.air_Temp_float" based on which anomaly was found
    - Now, a different client/app could listen to different types of anomalies.
  - Also print out a message to the console: "The IoT device puhatu_b1 has anomaly data"
    - Modify the device name in the message accordingly to the real device name inside the message)

NB! If you restart the data generator, it may take some time before anomalies start appearing. Feel free to speed up the data generation to speed up testing.

Take a screenshot of the anomaly data displayed in the terminal.

Exercise 10.5. Distributed system testing

Let's put all the previous components of Puhatu IoT data use case together.

- Run the 10.3 Publish Python program to generate the data (generate one message every second)
  - Generated data routing key pattern: `puhatu.last_name.raw`
- Run the 10.3 Python subscribing program, and split the data (Run two processes in parallel)
  - Input routing key pattern: `puhatu.last_name.raw` from queue `puhatu_queue`
  - Output data routing key patterns: `puhatu.last_name.indoor` or `puhatu.last_name.outdoor`
  - NB! You must deploy Two processes in parallel, both listening to the same input queue!
- Run the 10.4 anomaly data identification for only the outdoor devices queue.
  - Input routing key pattern: `puhatu.last_name.outdoor` from queue `outdoor_queue`
  - Output data routing key patterns: `puhatu.last_name.anomaly.sensor_name` (where the sensor name matches the data type for which anomaly was found)
  - Output should also be displayed in the terminal

Although we currently use the same computer to run all of these programs, it doesn't really matter which computer they run on; they would work exactly the same way. In order to increase the speed of data processing, we can scale the number of processes created in Exercises 10.3 and 10.4.

Take screenshots of the final result. Try to make sure that all the started processes (and their outputs) are visible.

- You can stack small windows next to each other in a bigger screenshot.

Bonus task

Create another RabbitMQ subscribed/publishing application that:

1. Listens to data arriving with "puhatu.last_name.raw" key
   - create a new queue for this app
2. Splits the data into different smaller JSON messages, which contain only one sensor value (e.g. wat_Temp_float, air_Temp_float).
   - Each should have a different routing key, which matches the sensor value type.
3. Design the routing key in a way that we could still subscribe to all raw data messages (by using some wildcard)

    - The main difference is that the JSON messages are smaller and contain only a single sensor value
4. other necessary info should also be kept in the smaller message (device id, timestamp, etc.)

Bonus task deliverables:

1. Python code for the extra RabbitMQ Python program/script
2. Make one screenshot that displays at least one of the JSON messages (its content should be visible) published to RabbitMQ. Either from the web interface or from a subscriber output.

Deliverables:

- Terminate your VM Instance
- Code of the Python programs
    - Submit the versions you used in 10.5.
- Codes and screenshots from the following tasks:
    1. Screenshot from 10.1
    2. From 10.2 - 2 screenshots and code (producer and consumer).
    3. From 10.3 - 2 screenshots and codes.
    4. From 10.4 - 1 screenshot
    5. From 10.5 - 1 screenshot

Task lab 10 - rabbitMQ

Current submission ZIP

(00:29 12.04.2023)

If your homework consists of multiple files (for example HTML + CSS + JS) it should be archived before submitting.

File Choose File No file chosen

Comment

Submit