Task 1: Communication models and middleware

Distributed Systems

Intro

In this first assignment you will build a **distributed meteorological data processing system** based on the client-server model. The final implementation will have all the components connected, on and working, as a fire-and-forget implementation.

As seen during the course, the components of a distributed system can communicate with each other through different patterns and technologies. Thus, you will implement two versions of the same system: one based on **direct communication** between processes, and the other based on **indirect communication**. Your goal is to compare the different protocols used, analyze their perks and disadvantages, and discuss on their suitability for each type of communication.

To ease the development of the assignment, you will implement a local, standalone version of the system, substituting **machines** with **processes**.

System design

The system is divided into two modules, the server side, and the client side. Client-side machines will generate raw data or read the processed results, whereas server-side machines will process raw data and temporarily store the processed results. The system will comprise the following components.

Client side	**Server side**
Air quality sensors (n)	Compute servers (n)
Pollution sensors (n)	Intermediate proxy (1), load balancer (indirect version, 1)
Monitoring terminals (n)	In-memory storage server (Redis, 1)

Sensors

Sensors generate data records periodically (every *X* seconds) and transfer them to the servers. You will implement **two types of sensors**: air wellness sensors and pollution sensors.

Each type of sensor produces a different kind of record: `RawMeteoData` (encapsulating air wellness as temperature and humidity) and `RawPollutionData` (encapsulating air pollution as co2 concentration). Records must be accompanied by their detection timestamp.

```
RawMeteoData {
    temperature: float
    humidity: float
    timestamp: datetime
}

RawPollutionData {
    co2: int
    timestamp: datetime
}
```

We provide the logic for data generation in the attached file “meteo_utils.py”. Each sensor should instantiate the class `MeteoDataDetector` and call its corresponding record generator.

```
detector = MeteoDataDetector()

# Air sensors
meteo_data = detector.analyze_air()
# returns a dictionary; { “temperature”: x, “humidity”: y }

# Pollution sensors
pollution_data = detector.analyze_pollution()
# returns a dictionary; { “co2”: z }
```

As the system is designed to have multiple sensors and multiple servers, records from the sensors must be distributed across the available servers through a **load-balancing** algorithm.

Servers

Servers receive data records and process them for their analysis. They calculate two different coefficients, depending on the type of record. **Each record must be processed separately**.

- `RawMeteoData` records will be converted into an air wellness coefficient.
- `RawPollutionData` records will be converted into a pollution coefficient.

We provide the logic for data processing in the attached file “meteo_utils.py”. Each sensor should instantiate the class `MeteoDataProcessor` and use the correct processor function depending on the type of input.

```
processor = MeteoDataProcessor()
wellness_data = processor.process_meteo_data(meteo_data)
pollution_data = processor.process_pollution_data(pollution_data)

# process_meteo_data expects RawMeteoData: an object
# with the attributes temperature and humidity
# process_pollution_data expects RawPollutionData: an object
# with the attribute co2
```

Servers save their processed data into the storage server, along with their original detection timestamps.

Storage-server

For data persistence and fault-tolerance, you will implement a **Redis**-based storage server. Compute servers put **every processed record** into the Redis server **with its detection timestamp**. Every *Y* seconds, the proxy will read the records corresponding to the last time interval to calculate their mean.

Both types of records must be saved separately, and thus the proxy must read them also separately.

Proxy

The proxy implements a **tumbling window** to calculate the mean value of the coefficients every *Y* seconds. Such means are transferred to **every connected terminal**. Means of both coefficients **must not be aggregated**. Nevertheless, they **can be sent jointly** (in the same message) to the terminals.

Timestamps **must not be aggregated** but means must be accompanied with a timestamp. You can simply use the most recent timestamp available.

Terminals

All terminals receive the mean air wellness and pollution coefficients every *Y* seconds and save all their values locally.

Terminals must represent the data they receive **in real time** through a simple, visual **user interface**:

- Plots updated in real-time (Real time plotting with Matplotlib in Python).

- Some kind of fancy data representation through the terminal (Rich: Generate Rich and Beautiful Text in the Terminal with Python).
- A simple HTTP interface printing the mean values/standard deviation...

- *be creative! :).*

Implementation 1: direct communication

In the first implementation, communication is performed directly through RPC calls. Both gRPC and XML-RPC can be used to implement this part. You can choose which technology you would like to use. However, if you choose XML-RPC you will be required to justify the decision and explain its disadvantages in comparison to gRPC. The diagram below shows an overview of the implementation with direct communication.

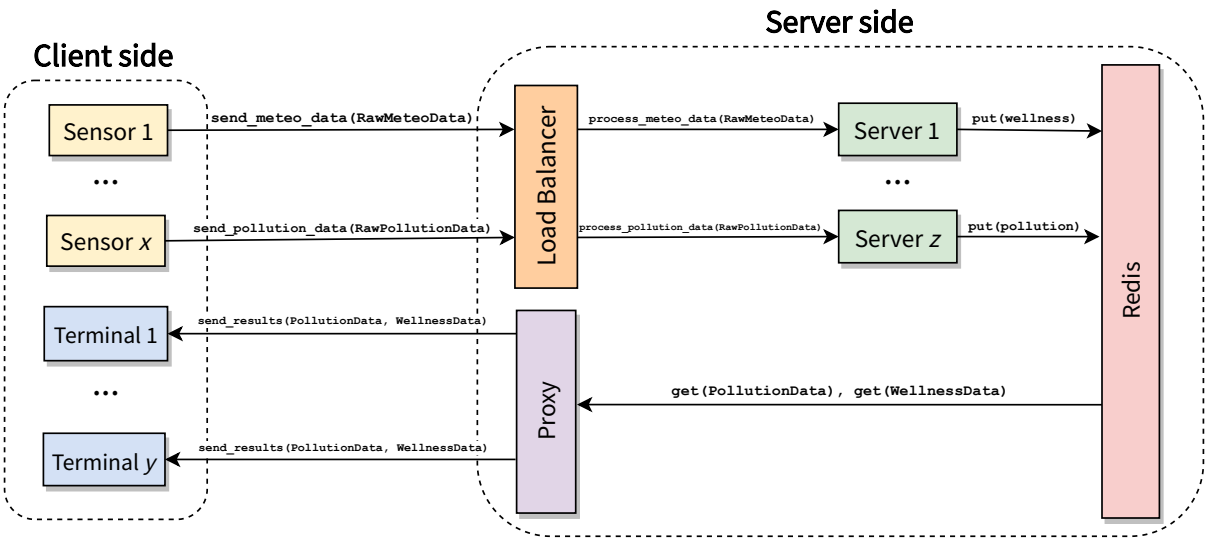

Figure 1. Implementation with direct communication.

Sensor-server communication

You will have to implement a specific load-balancer server (LB) between sensors and servers. Its functioning must be the following.

1. Sensors send **records, along with the sensor ID**, to the **LB** through RPC calls (sensors will be RPC clients, while the LB will be the RPC server).
2. The LB chooses which server will process each record. **Load-balancing** can be simplified by choosing the server in a **round-robin** fashion.
3. The LB redirects records to their corresponding servers through RPC calls (the LB will be the RPC client, whereas servers will be the RPC servers). Servers spend some time processing the data.

The compute servers provide the following RPCs:

- **`process_meteo_data(meteo_data: RawMeteoData):`** processes the raw meteorological data and returns the processed result, that is, the air wellness coefficient.
- **`process_pollution_data(pollution_data: RawPollutionData):`** processes the raw pollution data and returns a processed pollution coefficient.

Simplification: Ideally, compute servers should register themselves with the LB. However, to simplify the implementation, servers and their configuration can be hard coded or loaded from a file directly in the LB.

Server-Terminal communication

Once finished the calculation, servers save the coefficients with their detection timestamps into the storage server. Aggregated results must be transferred to all the terminals. Communication between the proxy and the terminals is also implemented through individual RPC calls.

1. At every tumbling window, the terminal performs an RPC to every single terminal sending the corresponding aggregate data (the proxy will act as the RPC client, whereas terminals will be RPC servers).
2. Every terminal updates its corresponding UI with the new data.

Simplification: Ideally, terminals should register themselves with the proxy. However, to simplify the implementation, terminals can be hard coded directly or loaded from a file in the proxy.

Implementation 2: indirect communication

In the second implementation, communication is performed using queues (**RabbitMQ**). The diagram in Figure 2 shows an overview of the implementation with indirect communication.

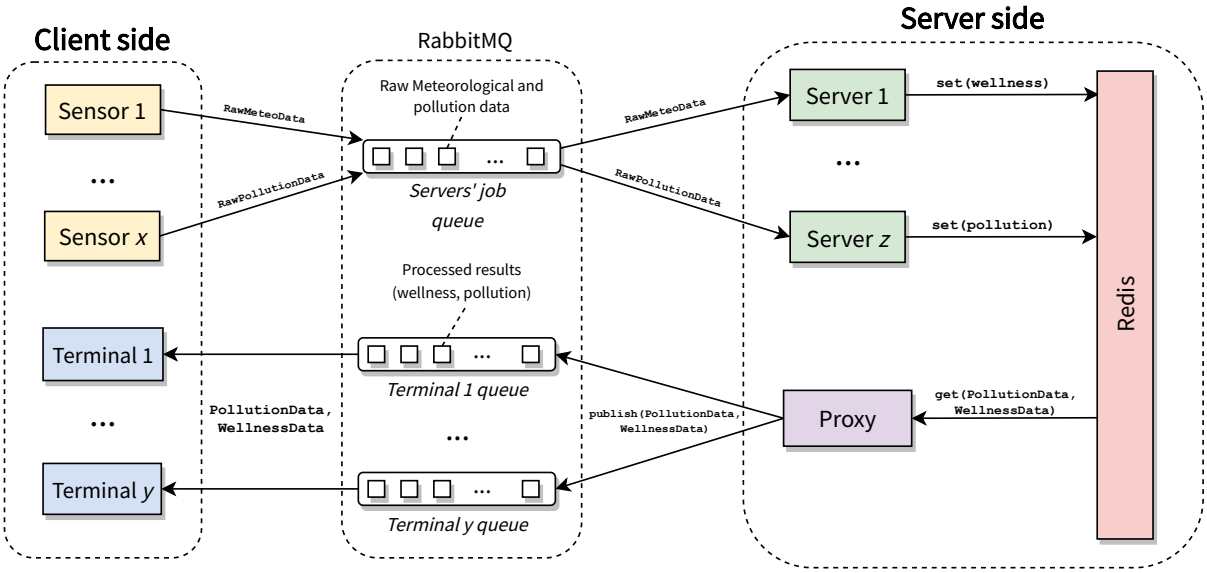

Figure 2. Implementation with indirect communication.

Sensor-server communication

In this case, load-balancing is automatically managed with a **RabbitMQ queue**. All sensors and servers share a **common queue** for exchanging data. The sensor-server side performs as follows:

1. Sensors publish records into the queue as they generate them.
2. Servers continuously process data.
    a. Each server will **pop a record** from the queue.
    b. Servers will decide which processing function to use **based on the data type** of the record.
    c. Processed coefficients and detection timestamps will be saved in the storage server (exactly as in the direct communication version).

Server-terminal communication

The proxy implements the tumbling window exactly as in the direct communication version. To distribute data across all terminals, you will follow a publish-subscribe approach with **RabbitMQ**. The server-terminal side works as follows:

1. At each tumbling window, the proxy publishes the aggregated coefficients and detection timestamps. Data will be broadcasted to the terminal queues by RabbitMQ (there are mechanisms for that!).
2. All terminals will represent updated data in their UIs exactly as in the direct communication version.

Documentation

Aside from the code, you will hand a report that explains all your design decisions, as well as the answers to some specific questions:

1. Frame all communication steps of the system based on the **four types of communication** (synchronous/asynchronous, pull/push, transient/persistent, stateless/stateful). Also include their pattern and cardinality (one-to-one, one-to-all...).
2. Mention which communication type is more appropriate for each step and justify your decision in terms of **scalability** and **fault tolerance**.
3. Are there **single points of failure** in the system? How could you resolve them?
4. Regarding system **decoupling**, what does a Message Oriented Middleware (MOM) such as RabbitMQ provide?
5. Briefly describe Redis' utility as a storage system in this architecture.

The **document** should include the following sections.

1. A brief introduction to the system's features and goals.
2. System implementation, including design choices.
3. A discussion on your final version of the system justifying to which extent the goals have been fulfilled.
4. Coherent and **concise** answers to the previously proposed questions.

Tasks

The assignment is divided into 4 tasks, which will be assessed as follows:

1. 30% – A working version of the implementation with direct communication.
    - 15% – Communication from sensors to servers.
    - 15% – Communication from servers to terminals.
2. 40% - A working version of the implementation with indirect communication.
    - 15% – Communication from sensors to servers.
    - 25% – Communication from servers to terminals.
3. 15% – Implementation of the tumbling window in the proxy with Redis.
4. 15% – Documentation and theoretical questions.

All system components will be implemented in **Python 3.8 or above**.

Note that the assignment asks for a minimum number of 2 components of each type (except for the load balancer, the proxy, and the storage server). However, your implementation should be generic enough to support an indefinite number of them.

Deliverables and assessment

The assignment consists of two deliverables:

1. The source code of your implementation, as well as any scripts, configuration files or test data files that you may use and that are required for execution.
2. The described report, including answers to the questions **in PDF format**.

Create a zip file which includes both deliverables and submit it to the *Task 1* activity in *Campus Virtual.* The deadline is the **15th of April 2023**.

After the evaluation of the task, the professor responsible for your group will ask you about some points of your delivery to assess your degree of knowledge.