

Microservices architecture for collecting data related to electric consumption

Cloud Computing Technologies project

Federico Garegnani

Università degli Studi di Milano

September 19, 2024



Table of contents

1 View from above

- Domain of use
- General structure
- Docker

2 Detailed view

- Data Generator
- Redis
- Data decoding
- MongoDB Database
- Store connector
- MinIO Object Store

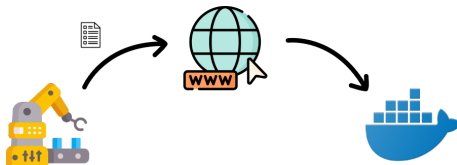
3 Cloud properties

- Health checks
- Restart policies



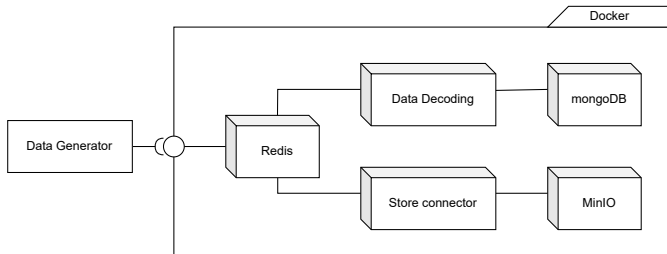
Application domain and working principle

- Industrial environment.
- An energy meter sends data regarding the electric consumption of a machinery through the internet.
- Data are received, analysed and collected by the cloud application.

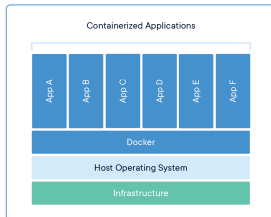


General structure of the system

- 5 microservices.
- Every microservice carry on a specific task.
- Only one entry point.

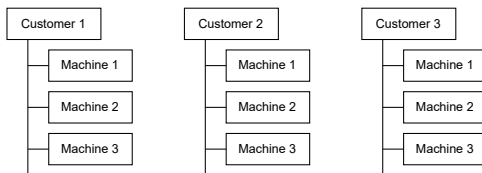


- It is a set of *platform as a service* (PaaS) products.
- It uses OS-level virtualization to execute software inside packages called *containers*.
- A container encapsulates its own software, libraries and configuration files.
- A container can communicate with other services through well defined channels.
- Each container execute the application in an isolated environment.



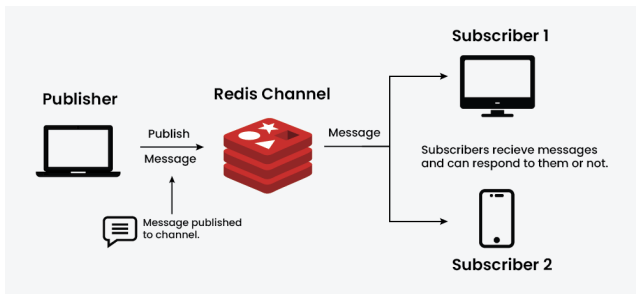
Data Generator

- It simulates data produced by each power meter.
- It publish data on a specific Redis channel.
E.g. Data from machine1 of customer1 are published on `data.customer1.machine1`



Redis

- It is a key-value database that offers the functionalities of a message broker.
- It receives data from all the meters and forward them to two microservices: *DataDecoding* and *Storeroom*.
- It is the entry point of the entire system. It exposes the port 6379.



- Publishers need to authenticate before starting to send data.
- A list of users is defined inside an *Access Control List* (ACL).
- Each user can perform only a subset of commands.

```
user default off
user customer on >customerpass +PUBLISH &data.*
user admin on >admin ~* &* +@all
user decoding on >dec +PSUBSCRIBE &data.*
user storeroom on >store +PSUBSCRIBE &data.*
```

Figure: File `redis.conf` specifying the ACL

```
r = redis.Redis(host='redis', port=6379,
→ username='decoding', password='dec',
→ decode_responses=True)
```

Figure: Connection to Redis server in a Python script



Data decoding

- It authenticates in Redis and subscribes to channel 'data.*'.
- It receives a hexadecimal string that is split to obtain data, hour and numeric value of the measure.

E.g. '2024-09-07T19:53:19.561339' → '1268b41553b7'

Numeric value 68 DEC → 44 HEX.

Then the two strings are concatenated: '1268b41553b744'.

- It prepares a JSON document and writes it into the database.

```
{  
  "customer": "customer1",  
  "machine": "machine1",  
  "date": "2024-07-18T18:05:05Z",  
  "EE": 32  
}
```

Figure: Example of document



- It stores all measurements.
- Clients use a specific user called 'client-1' with only read and write permissions.
- DB and users are defined in a JavaScript file executed at the start-up of container.

```
db = db.getSiblingDB('cct')
db.createCollection('measurements')
db.createUser({
  user: "client-1",
  pwd:  "client-1",
  roles:[{role: "readWrite" , db:"cct"}]}
})
```

Figure: Configuration file of MongoDB

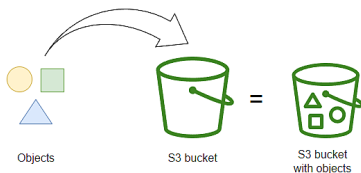


- It subscribes to Redis channel 'data.*'.
- It encapsulates raw data (hexadecimal strings) into text files.
- A unique filename is set to every file, given following this scheme `f"{self.machine_name}_{current_time}_{unique_id}"`, where `unique_id = uuid.uuid4()`.
- Files are sent for permanent storage.



MinIO Object Store

- It is an object store platform, organized in buckets and fully compatible with Amazon S3.
- It takes care of the permanent storage of raw data.
- One bucket for each customer.



Health checks

- It is a Docker instruction to determine the state of a container (*healthy/unhealthy*).
- Health checks are performed at regular intervals.
- They aim to provide *reliability, resilience, self-healing*.

```
healthcheck:  
  test: ["CMD", "redis-cli", "ping"]  
  interval: 30s  
  timeout: 10s  
  retries: 3
```

Figure: Health check definition for Redis container



Health checks aim to provide

- *Reliability*: they monitor that an application is working properly;
- *Resilience* and *Self-healing*: it is possible to detect a fault and reboot the service.

```
healthcheck:  
  test: ["CMD", "curl", "--fail",  
    ↪ "http://localhost:5000/health"]  
  interval: 30s  
  timeout: 10s  
  retries: 3
```

Figure: Health check definition for Decoding container



Restart policies

- They define how Docker should handle containers when they stop or crash.
- These policies are important to guarantee *reliability* and *high availability*.
- For each container the policy `restart`: `always` is set, this means that the service is always restarted regardless the cause stopping it.

```
mongodb:  
  image: mongo:latest  
  container_name: mongodb  
  restart: always
```

Figure: Definition of the restart policy for MongoDB

