Relatório do Sistema de Monitorização Ambiental

1. Identificação dos Alunos

- **Miguel Pinto**, I58122
- **Miguel Lopes**, I58540

2. Justificação das Escolhas

Estrutura da Base de Dados

A persistência de dados é feita através do **PostgreSQL**, com o esquema gerado automaticamente pelo **Hibernate (JPA)** a partir das entidades definidas no pacote `pt.ue.ambiente.server.data.entity`.

Entidades e Tabelas

1. **Metricas**
    - Responsável por armazenar o histórico de leituras ambientais.
    - **Atributos Principais:**
        - `id` (Long): Identificador.
        - `temperatura` (float) e `humidade` (int): Os valores medidos.
        - `protocolo` (Enum): O protocolo de origem (MQTT, gRPC, REST).
        - `tempoDispositivo` (Timestamp): O momento da leitura no sensor.
        - `tempoRegisto` (Timestamp): O momento da ingestão no servidor.
    - **Restrições:** Chave única composta por (`dispositivo_id`, `tempoDispositivo`, `protocolo`) para garantir a idempotência e evitar registos duplicados (caso que pode acontecer no MQTT pois a mesma mensagem pode ser entregada várias vezes pelo Broker).
2. **Dispositivo**
    - Representa os sensores instalados na universidade.
    - **Atributos:**
        - `id` (Long): Identificador.
        - `nome` (String): Nome escolhido pelo administrador para o dispositivos
        - `ativo` (boolean): Flag que informa o estado do dispositivo
        - `protocolos` (): Lista de protocolos que o dispositivo suporta.
    - **Relacionamentos:** Possui chaves estrangeiras obrigatórias para `Edificio`, `Piso`, `Sala` e `Departamento`, definindo a sua localização física atual.
3. **Localização**
    - Definem como os dispositivos estão localizados na universidade
        - **Edificio:** `nome` (String).
        - **Departamento:** `nome` (String).
        - **Sala:** `sala` (String).
        - **Piso:** `numero` (int).

A localização que está no dispositivo define onde são registadas as novas métricas e a localização das métricas informam onde o dispositivo estava quando essa métrica foi registada.

Métodos de Comunicação

A escolha de múltiplos protocolos reflete a diversidade de dispositivos num ambiente IoT real:

- **MQTT (Message Queuing Telemetry Transport):** Implementado para sensores simples e de baixa potência (simulados pelo `client-mqtt`). O modelo Publish/Subscribe permite que os dispositivos enviem dados assincronamente sem manter conexões HTTP pesadas, poupando bateria e largura de banda, além de não exigir um sistema com muitos recursos.
- **gRPC (Google Remote Procedure Call):** Implementado para gateways ou dispositivos com maior capacidade de processamento (simulados pelo `client-grpc`). O uso de **Protobuf** (Protocol Buffers) garante uma serialização binária compacta e eficiente, ideal para tráfego intenso e baixa latência.
- **REST (Representational State Transfer):** Implementado para integração com dispositivos simples com suporte HTTP/JSON enviam dados através de endpoints REST (simulados pelo `client-rest`).

Arquitetura do Sistema

O sistema usa uma arquitetura distribuída baseada em **microserviços**, orquestrada através de **Docker Compose**.

Arquitetura Utilizada no Servidor

1. **Servidor Backend:**
    - Responsável pela validação de dispositivos, ingestão de dados (e persistência dos mesmos na base de dados), agregação de métricas e exposição de APIs (REST e gRPC).
2. **Message Broker MQTT**
    - **Eclipse Mosquitto** - Responsável por receber os dados dos dispositivos que comunicam via protocolo MQTT.
3. **Base de Dados:**

- **PostgreSQL**

Clientes (Simuladores e Administração)

- Aplicações independentes (CLI) que atuam como produtores de dados (dispositivos) ou cliente de gestão do serviço (Admin CLI).
- Comunicam com o servidor de ingestão de métricas (através dos stubs gRPC e endpoints REST), ou com o broker (caso usem MQTT)

Usamos o Docker pois garante a homogenidade do sistema, permitindo que façamos deploy em qualquer ambiente que seja possível executar o Docker.

3. Instruções

Pré-requisitos

- **Docker** e **Docker Compose V2** instalados (para o servidor).
- **Java 25** (para os clientes).

Execução do Sistema

Para colocar todo o ambiente em funcionamento (Servidor, Base de Dados e Broker MQTT), executar o seguinte comando no diretório raiz do projeto

```
docker-compose up --build -d
```

Execução dos Clientes

Para interagir com o sistema, utilizam-se os clientes desenvolvidos. Abaixo estão os comandos para execução:

Cliente MQTT

- **Modo Submissão Única:**

  ```
  gradle -p client-mqtt run --args="<host> <porta> <id>"
  ```

- **Modo Submissão Contínua:**

  ```
  gradle -p client-mqtt run --args="<host> <porta> <id> <temperatura> <humidade>"
  ```

Cliente gRPC

- **Modo Submissão Única:**

  ```
  gradle -p client-grpc run --args="<host> <porta> <id>"
  ```

- **Modo Submissão Contínua:**

  ```
  gradle -p client-grpc run --args="<host> <porta> <id> <temperatura> <humidade>"
  ```

Cliente REST

- **Modo Submissão Única:**

  ```
  gradle -p client-rest bootRun --args="<id> --ambiente.server.url=<endpoint>"
  ```

- **Modo Submissão Contínua:**

  ```
  gradle -p client-rest bootRun --args="<id> <temperatura> <humidade> --ambiente.server.url=<endpoint>"
  ```

Cliente Admin

- **Execução:**

  ```
  gradle -p admin-cli bootRun --args="--ambiente.server.url=<endpoint>"
  ```

4. Observações de Desenvolvimento

Durante a realização deste projeto, deparámos-nos com alguns desafios, tais como:

1. **Integração de Múltiplos Protocolos no Spring Boot:**

- Coordenar três interfaces de comunicação distintas (REST, gRPC e MQTT) numa única aplicação Spring Boot foi difícil.
- Tivemos de garantir que o servidor gRPC e o cliente MQTT (Paho) iniciassem corretamente junto com a framework Spring sem bloquear a thread principal, permitindo que a API REST continuasse responsiva.(inicialmente tivemos dificuldade em colocar os três protocolos a funcionar em simultâneo)

2. **Consistência e Validação de Dados:**

   - Lidar com a assincronia do MQTT implicou cuidados extra para evitar métricas duplicadas, mitigados pela restrição de chave única na tabela de métricas. (usámos o identificador do dispositivo, o protocolo por onde foi recebido a métrica e o tempo que marcava no dispositivo de modo a evitar que tivessemos métricas duplicadas pois o MQTT apesar de garantir que a mensagem é entregue ele pode duplicá-la)

5. Análise de Performance e Conclusões

Para finalizar, apresentamos uma comparação detalhada do desempenho dos três protocolos implementados (REST, MQTT, gRPC), baseada nos dados experimentais recolhidos durante a execução do projeto.

5.1 Metodologia de Teste

Os testes consistiram na emissão sequencial de 10 pedidos de envio de métricas para cada protocolo usando cada um dos clientes desenvolvidos. Os tempos registados correspondem à latência (desde que a métrica foi recolhida no dispositivo até à receção por parte do servidor de ingestão de métricas).

- O ambiente de testes utilizado foi um computador em que executamos localmente o servidor e os clientes de modo a reduzir o impacto da latência do canal de transporte e perdas de pacotes.

5.2 Resultados Obtidos

A tabela abaixo apresenta os tempos de resposta em milissegundos (ms) para cada um dos 10 pedidos efetuados.

Nº Pedido	REST (ms)	MQTT (ms)	gRPC (ms)
1 (Warm-up)	176	17	179
2	5	42	4
3	6	44	4
4	5	13	4
5	5	42	4
6	5	10	4
7	5	42	4
8	6	42	4
9	5	11	4
10	5	43	3
Média	22,3	30,6	21,4

5.3 Análise e Comparação

1. gRPC (Google Remote Procedure Call)

- **Desempenho:** Foi o protocolo consistentemente mais rápido após a fase de aquecimento (warm-up), com uma média de **3.9 ms**.
- **Análise:** O excelente desempenho deve-se ao uso de **Protocol Buffers** (formato binário compacto) e HTTP/2. A serialização/deserialização é extremamente eficiente, exigindo menos CPU e largura de banda que o JSON.
- **Adequação:** Ideal para **dispositivos de alto desempenho** (como Gateways) com capacidade de processamento robusta e conexão estável, onde a baixa latência é crítica.
- **Warm-up:** O pico inicial (179ms) é justificado pela inicialização do canal HTTP/2.

2. REST (Representational State Transfer)

- **Desempenho:** Apresentou uma performance muito sólida, com média de **5.2 ms**, apenas ligeiramente superior ao gRPC.
- **Análise:** Embora utilize JSON (texto), que é mais pesado e lento de processar que o binário, o overhead em redes locais ou ambientes Docker é mínimo para cargas pequenas.
- **Adequação:** Adequado para **dispositivos com capacidade média** onde a facilidade de implementação e a interoperabilidade são mais importantes que a performance pura.
- **Warm-up:** Semelhante ao gRPC (176ms), devido ao estabelecimento da conexão TCP/HTTP.

3. MQTT (Message Queuing Telemetry Transport)

- **Desempenho:** Apresentou a maior latência média (**32.1 ms**) e maior variabilidade (oscilando entre ~10ms e ~43ms).
- **Análise:**
  - **Assincronia:** Ao contrário do REST e gRPC, o MQTT não é desenhado para resposta imediata síncrona. A mensagem viaja do Publisher para o Broker e depois para o Subscriber (Servidor). Este "salto" extra introduz latência natural.
  - **Variabilidade:** A oscilação deve-se provavelmente ao agendamento de mensagens no Broker (Mosquitto) e ao mecanismo de polling ou callback do cliente.
  - **Eficiência:** Apesar da latência, é o protocolo com **menor overhead de cabeçalho** (apenas 2 bytes mínimos), tornando-o ideal para redes instáveis ou com largura de banda limitada, onde a velocidade de entrega imediata é menos crítica que a garantia de entrega.
- **Adequação:** Perfeito para **dispositivos de baixa potência e recursos limitados** , pois minimiza consumo de energia e não requer tantos recursos como os outros protocolos.