**Sistemas Distribuídos**

Licenciatura em Engenharia Informática

# 2º Trabalho

Sistema de Monitorização Ambiental para a Universidade de Évora


**André Gonçalves, 58392 | André Zhan, 58762**

Departamento de Informática
Universidade de Évora
Janeiro 2026

# Índice

# 1 Introdução

Este relatório apresenta a implementação de um sistema distribuído de monitorização ambiental desenvolvido para a Universidade de Évora na cadeira de Sistemas Distribuídos. O sistema tem como objetivo monitorizar a temperatura e humidade em diversas áreas da universidade, simulando dispositivos IoT que comunicam através de três protocolos distintos: MQTT, gRPC e REST.

O trabalho foi desenvolvido utilizando Java com Spring Boot, PostgreSQL como base de dados, e inclui múltiplos clientes independentes que simulam diferentes tipos de sensores e um cliente de administração via linha de comandos (CLI).

## 1.1 Objetivos do Sistema

O sistema implementado permite:

- Receber métricas de temperatura e humidade através de três protocolos diferentes (MQTT, gRPC, REST)
- Registar, consultar, atualizar e eliminar dispositivos IoT
- Armazenar métricas de forma organizada
- Consultar dados agregados por diferentes níveis
- Validar dispositivos antes de aceitar métricas
- Gerir todo o sistema através de uma interface CLI

# 2 Justificação das Escolhas Técnicas

## 2.1 Base de Dados

### 2.1.1 PostgreSQL

**Escolha:** Utilização de PostgreSQL como Sistema de Gestão de Base de Dados Relacional.

**Justificação:**

- **Requisito obrigatório do enunciado** (Secção 5): "Base de Dados: PostgreSQL ou H2".
- PostgreSQL oferece **robustez e fiabilidade** para sistemas em produção.
- Suporte nativo a **tipos de dados temporais** (timestamp, LocalDateTime) essenciais para métricas com dimensão temporal.
- **Queries complexas de agregação** são eficientes com SQL (médias, agrupamentos por localização).

### 2.1.2 Tabela `devices`

A tabela de dispositivos foi estruturada com os seguintes campos:

| Campo | Descrição | Tipo |
|---|---|---|
| id | Identificador único do dispositivo (PK) | VARCHAR(255) |
| protocol | Protocolo utilizado (MQTT/GRPC/REST) | ENUM |
| room | Sala onde está localizado | VARCHAR(100) |
| department | Departamento | VARCHAR(100) |
| floor | Piso/Andar | VARCHAR(50) |
| building | Edifício | VARCHAR(100) |
| active | Estado do dispositivo (ativo/inativo) | BOOLEAN |
| created_at | Data de criação do registo | TIMESTAMP |
| updated_at | Data da última atualização | TIMESTAMP |

Tabela 1: Estrutura da tabela `devices`

**Justificação das escolhas:**

- **ID como String**: Permite identificadores descritivos (ex: `sensor-XXX`) em vez de números sequenciais.
- **Protocol como ENUM**: Garante integridade referencial e facilita queries por tipo de protocolo.
- **Campos de localização**: Permitem filtros por (sala, departamento, piso, edifício).
- **Campo active**: Permite desativar dispositivos sem perder histórico de métricas.
- **Timestamps de auditoria**: Rastreiam criação.

### 2.1.3 Tabela `metrics`

A tabela de métricas armazena os dados recebidos dos sensores:

| Campo | Descrição | Tipo |
|---|---|---|
| id | Identificador sequencial (PK) | BIGINT (AUTO) |
| device_id | Referência ao dispositivo (FK) | VARCHAR(255) |
| temperature | Temperatura registada (°C) | DOUBLE |
| humidity | Humidade registada (%) | DOUBLE |
| timestamp | Data/hora da leitura no sensor | TIMESTAMP |
| received_at | Data/hora de recepção no servidor | TIMESTAMP |

Tabela 2: Estrutura da tabela `metrics`

**Justificação das escolhas:**

- **ID auto-incrementado**: Otimiza inserções e indexação em tabela.
- **Relação Many-to-One com Device**: Permite rastreabilidade e queries agregadas eficientes.
- **Dois timestamps**: Distingue momento da criação das métricas (`timestamp`) do momento de recepção (`received_at`) para análise de latências e detecção de atrasos na rede.
- **Tipos DOUBLE**: Suportam valores decimais com precisão adequada para medições ambientais.

## 2.2 Métodos de Comunicação Cliente-Servidor

### 2.2.1 MQTT

**Implementação:**

- **Broker**: Eclipse Mosquitto (via Docker)
- **Cliente (Sensor)**: Eclipse Paho MQTT Client
- **Servidor**: Spring Integration MQTT (subscriber automático)
- **Tópicos**: `sensors/#` (wildcards para múltiplos sensores)
- **Stub:** Blocking (síncrono).
- **QoS**: 1 (At Least Once) - garante entrega sem duplicação excessiva

**Justificação da escolha:**

- **Comunicação assíncrona**: Sensores publicam dados sem esperar resposta, reduzindo latência.
- **Baixo overhead**: Protocolo binário leve, ideal para dispositivos IoT com recursos limitados.

### 2.2.2 gRPC

**Implementação:**

- **Protocol Buffers**: Definição de contrato em `metrics.proto`
- **Servidor**: gRPC Spring Boot Starter (porta 9090)
- **Cliente**: gRPC Java stub gerado automaticamente via Maven plugin
- **Comunicação**: Unary RPC (request/response síncrono)

**Definição do contrato (.proto):**

```proto
syntax = "proto3";
package metrics;

service MetricsService {
    rpc SendMetric(MetricRequest) returns (MetricResponse);
}

message MetricRequest {
    string device_id = 1;
    double temperature = 2;
    double humidity = 3;
    string timestamp = 4;
}

message MetricResponse {
    bool success = 1;
    string message = 2;
}
```

Listing 1: metrics.proto

**Justificação da escolha:**

- **Alto desempenho**: Serialização binária (Protocol Buffers) é mais rápida que JSON.
- **Contratos tipados**: .proto garante compatibilidade entre cliente e servidor.
- **Geração automática de código**: Reduz erros e acelera desenvolvimento.

### 2.2.3 REST

**Implementação:**

- **Framework**: Spring Boot REST Controllers
- **Formato**: JSON (Jackson serialization)
- **Endpoints**:
  - `POST /api/metrics/ingest` - Ingestão de métricas
  - `POST /api/devices` - Criar dispositivo
  - `GET /api/devices` - Listar dispositivos
  - `PUT /api/devices/{id}` - Atualizar dispositivo
  - `DELETE /api/devices/{id}` - Remover dispositivo
  - `GET /api/metrics/average` - Métricas agregadas
  - `GET /api/metrics/raw` - Métricas brutas
  - `GET /api/devices/{id}` - Obter dispositivo específico
  - `GET /api/devices/active` - Listar dispositivos ativos
  - `GET /api/devices/active/{protocol}` - Listar dispositivos ativos por protocolo

- `GET /api/metrics/count-by-protocol` - Contagem de métricas por protocolo
- `GET /api/metrics/average-latency-by-protocol` - Latência média por protocolo
- `DELETE /api/metrics` - Apagar todas as métricas

**Retry logic no cliente:**

```
private void sendMetricWithRetry() {
    int retries = 0;
    boolean success = false;
    while (!success && retries < maxRetries) {
        // Backoff fixo de 1s
    }
}
```

**Justificação da escolha:**

- **Universalidade**: HTTP/JSON é suportado por qualquer linguagem e plataforma.
- **Debugabilidade**: Formato legível por humanos facilita troubleshooting.
- **Simplicidade**: Não requer bibliotecas especiais ou brokers intermediários.
- **Stateless**: Cada request é independente, simplificando escalabilidade.

## 2.3 Validação e Segurança

O servidor implementa validação de dispositivos em três camadas:

1. **Camada de Protocolo**:
   - MQTT: Valida formato JSON
   - gRPC: Validação automática via Protocol Buffers
   - REST: Bean Validation (`@Valid`, `@NotNull`)
2. **Camada de Serviço (`MetricService`)**:

```
public boolean ingestMetric(MetricDTO metricDTO) {
    Optional<Device> deviceOpt =
        deviceRepository.findById(metricDTO.getDeviceId());

    if (deviceOpt.isEmpty() || !deviceOpt.get().isActive()) {
        logger.warn("Rejected metric from unknown/inactive device:
    {}",
            metricDTO.getDeviceId());
        return false;
    }

    Metric metric = new Metric(deviceOpt.get(),
        metricDTO.getTemperature(),
        metricDTO.getHumidade(),
        metricDTO.getTimestamp());

    metricRepository.save(metric);
```

```
17         return true;
18     }
19
```

Listing 2: Validação de dispositivo antes de aceitar métrica

3. **Camada de Persistência**:
   - Constraints NOT NULL em campos obrigatórios
   - Foreign Key `device_id` garante integridade referencial

## 2.4 Gestão dos sensores com Thread Pools

**Escolha:** Implementação de um padrão Supervisor/Worker utilizando `ScheduledExecutorService` para gestão de múltiplos dispositivos simulados concorrentemente.

**Justificações:**

1. **Dois Thread Pools Separados:**
   - **Poller Thread (Thread de Supervisão):** Single thread dedicada para polling periódico do servidor REST (`GET /api/devices/active/{protocol}` para verificação periódica dos dispositivos ativos na base de dados).
   - **Shared Pool (Pool de Dispositivos):** Thread pool configurável (default: 5 threads) para execução em simultâneo de múltiplos simuladores de dispositivos, onde cada thread corresponde a um sensor.
   - **Separação de responsabilidades:** Evita que sobrecarga de dispositivos bloqueie sincronização com servidor.
2. **Escalabilidade:**
   - Um único processo cliente pode simular **dezenas de dispositivos** simultaneamente.
   - Configuração via `numberOfThreads = 5`: permite até 5 dispositivos enviando métricas em paralelo.

# 3 Configuração da Base de Dados PostgreSQL

## 3.1 Visão Geral

O sistema utiliza PostgreSQL 15 como Sistema de Gestão de Base de Dados Relacional (SGBDR), provisionado via Docker Compose para garantir reprodutibilidade e isolamento do ambiente. A criação automática de tabelas é gerida pelo Hibernate (JPA) através da propriedade `spring.jpa.hibernate.ddl-auto=update`, eliminando a necessidade de scripts SQL manuais.

## 3.2 Pré-requisitos

Antes de iniciar a configuração, certifique-se de que os seguintes componentes estão instalados:

- Docker e Docker Compose instalados
- Porta 5432 disponível no sistema

## 3.3 Passo 1: Configuração de Variáveis de Ambiente

O sistema utiliza um ficheiro `.env` na raiz do projeto para centralizar configurações sensíveis. Este ficheiro define credenciais e portas para PostgreSQL e MQTT.

### 3.3.1 Criação do ficheiro `.env`

Crie um ficheiro `.env` na raiz do projeto com o seguinte conteúdo:

```
# Database Configuration
POSTGRES_DB=monitoring
POSTGRES_USER=admin
POSTGRES_PASSWORD=admin
POSTGRES_PORT=5432

# MQTT Broker Configuration
MQTT_PORT=1883
MQTT_WS_PORT=9001
```

Listing 3: Ficheiro .env com configurações de base de dados

## 3.4 Passo 2: Provisionar PostgreSQL com Docker Compose

O ficheiro `docker-compose.yml` define a configuração do container PostgreSQL:

```
version: '3.8'

services:
  mqtt-broker:
    image: eclipse-mosquitto:2
    container_name: mosquitto-broker
    ports:
      - "${MQTT_PORT}:1883"
      - "${MQTT_WS_PORT}:9001"
    volumes:
      - ./mosquitto/config:/mosquitto/config
      - ./mosquitto/data:/mosquitto/data
      - ./mosquitto/log:/mosquitto/log
    restart: unless-stopped

  postgres:
    image: postgres:15-alpine
    container_name: monitoring-postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
```

```
      - "${POSTGRES_PORT}:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: unless-stopped

volumes:
  postgres_data:
```

Listing 4: Configuração do serviço PostgreSQL no Docker Compose

**Parâmetros de configuração:**

- **Serviço MQTT (Mosquitto):**
  - **Image:** `eclipse-mosquitto:2` - Versão estável do broker.
  - **Ports:** Expõe as portas do container (1883 para TCP, 9001 para WebSockets) vinculando-as às portas do host definidas no ficheiro `.env`.
- **Serviço PostgreSQL:**
  - **Image:** `postgres:15-alpine` - Versão baseada em Alpine Linux, otimizada para menor tamanho e maior segurança.
  - **Environment Variables:** Define credenciais e nome da base de dados através de interpolação de variáveis para segurança.
  - **Volume:** `postgres_data` - Utiliza um *Named Volume* gerido pelo Docker. Isto garante a persistência dos dados em `/var/lib/postgresql/data` mesmo que o container seja removido, evitando problemas de permissões do sistema operativo.
  - **Restart Policy:** `unless-stopped` - O container reinicia automaticamente em caso de falha ou reinício do sistema, exceto se parado manualmente.

### 3.4.1 Iniciar o serviço PostgreSQL

Execute o seguinte comando na raiz do projeto para provisionar a base de dados:

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

Listing 5: Comando para iniciar PostgreSQL via Docker Compose

**Opções do comando:**

- `-d` (detached): Executa containers em background.

## 3.5 Passo 3: Criação Automática de Tabelas via Hibernate

O sistema utiliza **JPA/Hibernate DDL Auto-Generation** para criar e atualizar o schema da base de dados automaticamente. Esta configuração está definida em `server/application.pr`

```
# PostgreSQL Database
spring.datasource.url=jdbc:postgresql://localhost:5432/monitoring
spring.datasource.username=admin
spring.datasource.password=admin
spring.datasource.driverClassName=org.postgresql.Driver

# JPA/Hibernate
```

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=false
```

Listing 6: Configuração Spring Boot para PostgreSQL

**Significado de `ddl-auto=update`:**

- Na primeira execução, Hibernate **cria** todas as tabelas baseado nas entidades JPA.
- Em execuções subsequentes, **altera** schema apenas se houver mudanças nas entidades.
- Preserva dados existentes (não executa `DROP TABLE`).

### 3.6 Passo 4: Gestão de Utilizadores e Permissões

O sistema utiliza um único utilizador `admin` com privilégios totais:

- **Username:** admin
- **Password:** admin
- **Database:** monitoring
- **Privilégios:** SUPERUSER (criação de tabelas, índices, leitura/escrita)

### 3.7 Passo 5: Iniciar Servidor Spring Boot

Com PostgreSQL em execução e configurado, inicie o servidor:

```
cd server
mvn clean compile
mvn spring-boot:run
```

Listing 7: Compilar e executar servidor Spring Boot

**Verificação de sucesso:**
Procure nos logs pela mensagem:

```
Hibernate: create table devices (...)
Hibernate: create table metrics (...)
Tomcat started on port(s): 8080 (http)
Started ServerApplication in X.XXX seconds
```

## 4 Observações sobre o Desenvolvimento

### 4.1 Metodologia de Desenvolvimento

O projeto foi desenvolvido seguindo a seguinte estrutura:

1. **Fase 1 - Infraestrutura Base**:
   - Configuração de Docker Compose (PostgreSQL + Mosquitto)
   - Estrutura de entidades JPA (Device, Metric)

- API REST básica para gestão de dispositivos

2. **Fase 2 - Implementação dos Protocolos**:
   - Servidor gRPC e definição .proto
   - Integração MQTT com Spring Integration
   - Endpoint REST de ingestão de métricas
3. **Fase 3 - Clientes Simuladores**:
   - Cliente MQTT
   - Cliente gRPC
   - Cliente REST
4. **Fase 4 - Cliente CLI e Refinamentos**:
   - Interface de administração
   - Queries agregadas
   - Logging e tratamento de erros

## 4.2 Desafios Encontrados

### 4.2.1 Desafio 1: Sincronização de Timestamps entre Protocolos

**Problema:** Cada protocolo trata timestamps de forma diferente:

- MQTT: String JSON (formato ISO 8601)
- gRPC: String protobuf
- REST: String JSON

A desserialização inconsistente causava erros de parsing e timestamps incorretos.
**Solução implementada:**

```
// ObjectMapper com modulo JavaTimeModule
ObjectMapper objectMapper = new ObjectMapper();
objectMapper.registerModule(new JavaTimeModule());

// Fallback para timestamp atual se parsing falhar
LocalDateTime timestamp = request.getTimestamp().isEmpty()
    ? LocalDateTime.now()
    : LocalDateTime.parse(request.getTimestamp());
```

### 4.2.2 Desafio 2: Geração de Dados Sintéticos Realistas

**Problema:** Valores aleatórios puros geravam saltos bruscos (15°C → 30°C), não realistas.

**Solução implementada:** Variação gradual com persistência temporal, gera um valor gradual dentro de um intervalo com variação controlada simulando comportamento realista com mudanças suaves.

```
    currentTemperature = generateGradualValue(currentTemperature, 15.0,
    30.0, 0.5); // +-0.5
    currentHumidity = generateGradualValue(currentHumidity, 30.0, 80.0,
    2.0); // +- 2.0

    private double generateGradualValue(double current, double min,
   double max, double maxChange) {
        double change = (random.nextDouble() - 0.5) * 2 * maxChange;
        double newValue = current + change;
        return Math.max(min, Math.min(max, newValue));
    }
```

### 4.2.3 Desafio 3: Sincronização dinâmica dos dispositivos

**Problema:** Clientes precisam descobrir novos dispositivos sem reiniciar. **Solução implementada:** Polling periódico através de GET /api/devices/active/{protocol} onde verifica os dispositivos ativos na base de dados.

```
private void syncWithRegistry() {
    List<DeviceDTO> devices = fetchDevices();
    Set<String> activeIds = new HashSet<>();

    for (DeviceDTO device : devices) {
        if (device.isActive()) {
            activeIds.add(device.getId());
            // computeIfAbsent: atomicamente cria novo se não existe
            simulators.computeIfAbsent(deviceId, missingId -> {
                MqttSensorSimulator sim = new MqttSensorSimulator(...);
                sim.start();
                return sim;
            });
        }
    }

    // Remove simuladores de dispositivos inativos
    simulators.entrySet().removeIf(entry -> {
        if (!activeIds.contains(entry.getKey())) {
            entry.getValue().stop();
            return true;
        }
        return false;
    });
}
```

# 5 Análise de Performance

Para efetuar a análise comparativa de performance entre os três protocolos de comunicação implementados (MQTT, gRPC e REST), o foco recaiu sobre a latência de transmissão (desde a criação da métrica até à persistência na base de dados) e a fiabilidade (número total de métricas processadas com sucesso).

Os testes foram realizados variando o número de threads, o intervalo de envio, a quantidade de simuladores concorrentes e a duração total da execução, conforme descrito nos cenários abaixo.

## 5.1 Cenário Base: Configuração Inicial

**Parâmetros:** Threads=3, Intervalo=3s, Simuladores=1, Duração=60s

Este cenário serve como linha de base para verificar o funcionamento dos protocolos sem stress.

| Protocolo | Latência (ms) | Métricas Recebidas |
|---|---|---|
| MQTT | 8.68 | 12 |
| gRPC | 13.41 | 13 |
| REST | 9.54 | 13 |

Tabela 3: Cenário Base: Performance com carga mínima

**Análise:** O MQTT apresentou a melhor latência inicial (8.68ms), confirmando a sua natureza leve para IoT. O gRPC apresentou a latência mais alta (13.41ms), o que é consistente com o overhead inicial de estabelecimento de canais seguros e handshakes HTTP/2 em execuções curtas. O REST posicionou-se no meio termo.

## 5.2 Cenário 2: Teste de Carga de Simuladores

**Parâmetros:** Threads=5, Intervalo=5s, Duração=60s. Variável: Número de Simuladores (10, 50, 100).

| Simuladores | Protocolo | Latência (ms) | Métricas Recebidas |
|---|---|---|---|
| 10 | MQTT | 6.95 | 130 |
| | gRPC | 14.86 | 130 |
| | REST | 10.54 | 125 |
| 50 | MQTT | 5.24 | 704 |
| | gRPC | 5.55 | 571 |
| | REST | **3.85** | 610 |
| 100 | MQTT | 4.44 | 1544 |
| | gRPC | **3.03** | 1300 |
| | REST | 3.47 | 1220 |

Tabela 4: Cenário 2: Impacto do aumento de simuladores concorrentes

**Análise:** Observou-se um comportamento interessante com o aumento de carga:

- **Redução da Latência Média:** À medida que o número de simuladores aumentou, a latência média diminuiu para todos os protocolos (ex: gRPC baixou de 14.86ms para 3.03ms). Isto deve-se ao facto de o sistema entrar em "regime estacionário", onde o custo inicial de conexão é diluído por um grande volume de mensagens processadas rapidamente.

- **REST vs gRPC:** Com 50 simuladores, o REST foi surpreendentemente rápido (3.85ms), mas com 100 simuladores, o gRPC assumiu a liderança (3.03ms), demonstrando a eficiência do Protobuf e HTTP/2 sob carga extrema.
- **Perda de Pacotes:** Com 50 simuladores, nota-se que o gRPC processou menos métricas (571) comparado ao MQTT (704), sugerindo possíveis gargalos no processamento síncrono ou saturação de threads no servidor.

## 5.3 Cenário 3: Impacto do Número de Threads

**Parâmetros:** Intervalo=5s, Simuladores=3, Duração=60s.

| Threads | Protocolo | Latência (ms) | Métricas |
|---|---|---|---|
| 1 | MQTT | 7.83 | 39 |
| | gRPC | **7.35** | 39 |
| | REST | 9.73 | 37 |
| 5 | MQTT | 7.84 | 39 |
| | gRPC | 17.27 | 39 |
| | REST | 9.61 | 39 |
| 10 | MQTT | 7.11 | 39 |
| | gRPC | 12.68 | 39 |
| | REST | 10.43 | 39 |

Tabela 5: Cenário 3: Variação de threads no cliente

**Análise:**

- **Estabilidade do MQTT:** A latência do MQTT permaneceu praticamente inalterada ( 7ms) independentemente do número de threads, provando a eficácia do seu modelo assíncrono (Pub/Sub).
- **Instabilidade do gRPC:** O gRPC sofreu uma degradação significativa ao passar de 1 para 5 threads (de 7.35ms para 17.27ms), recuperando ligeiramente com 10 threads.

## 5.4 Cenário 4: Impacto do Intervalo de Envio

**Parâmetros:** Simuladores=3, Duração=60s, Threads=5.

| Intervalo (s) | Protocolo | Latência (ms) | Métricas |
|---|---|---|---|
| 1 | MQTT | 6.06 | 183 |
| | gRPC | 7.87 | 183 |
| | REST | 7.22 | 183 |
| 5 | MQTT | 6.55 | 39 |
| | gRPC | 12.95 | 39 |
| | REST | 9.90 | 39 |
| 10 | MQTT | 7.71 | 21 |
| | gRPC | 16.78 | 21 |
| | REST | 11.51 | 21 |

Tabela 6: Cenário 4: Variação da frequência de envio

**Análise:** Confirmou-se que **maior frequência de envio resulta em menor latência média**. Com intervalo de 1s, as conexões TCP mantêm-se ativas e "quentes", evitando o overhead de renegociação. O gRPC foi o mais prejudicado por intervalos longos (10s), atingindo 16.78ms, provavelmente devido à necessidade de gestão de *keep-alive* ou restabelecimento de contexto HTTP/2 entre chamadas espaçadas.

## 5.5 Cenário 5: Teste de Longa Duração (Estabilidade)

**Parâmetros:** Intervalo=1s, Simuladores=3, Threads=5.

| Duração (s) | Protocolo | Latência (ms) | Métricas |
|---|---|---|---|
| 600 (10 min) | MQTT | 4.76 | 1803 |
| | gRPC | 4.79 | 1803 |
| | REST | 4.88 | 1803 |
| 1200 (20 min) | MQTT | 4.83 | 3603 |
| | gRPC | **4.33** | 3603 |
| | REST | 5.02 | 3603 |
| 1800 (30 min) | MQTT | 4.75 | 5403 |
| | gRPC | **4.36** | 5403 |
| | REST | 5.97 | 5403 |

Tabela 7: Cenário 5: Performance em execução prolongada

**Análise:** Este é o cenário mais representativo de um ambiente de produção real.

1. **Convergência:** Inicialmente (600s), os protocolos equivalem-se ( 4.8ms).
2. **Superioridade do gRPC:** A longo prazo (1200s e 1800s), o gRPC torna-se o mais rápido (4.33ms), beneficiando da compactação binária e reutilização eficiente de conexões.

3. **Degradação do REST:** O REST começou a mostrar sinais de ligeira degradação aos 1800s (subiu para 5.97ms), possivelmente devido à acumulação de objetos em memória ou gestão de conexões HTTP menos eficiente que o gRPC.

# 6 Conclusões: Teórica vs. Real

A análise dos resultados permite confrontar a expectativa teórica com a realidade observada na implementação do Sistema de Monitorização Ambiental.

## 6.1 MQTT

**Expectativa Teórica:** Protocolo leve, ideal para redes instáveis, baixo overhead.
**Resultado Real:** Confirmado. O MQTT foi o protocolo mais consistente em quase todos os cenários. Não foi sempre o mais rápido em picos absolutos (perdendo para o gRPC em longa duração), mas a sua latência raramente oscilou de forma drástica, mantendo-se robusto face à variação de threads e intervalos.

## 6.2 gRPC

**Expectativa Teórica:** Alta performance, baixa latência devido a Protobuf e HTTP/2, mas maior complexidade de setup.
**Resultado Real:** Observou-se claramente o custo de inicialização ("cold start"). Em testes curtos e com intervalos longos, teve o pior desempenho. No entanto, em cenários de **longa duração** e **altíssima carga** (100 simuladores), provou ser superior, validando a sua escolha para comunicação entre serviços backend intensivos.

## 6.3 REST

**Expectativa Teórica:** Simplicidade, maior overhead devido ao formato JSON (texto) e falta de streaming nativo.
**Resultado Real:** Surpreendeu pela positiva em cargas médias (50 simuladores), superando o MQTT momentaneamente. Contudo, em testes de longa duração (30 minutos), mostrou a pior tendência de latência, confirmando que para fluxos contínuos de dados, o overhead do HTTP/1.1 e do parsing de JSON acaba por pesar no sistema.

# 7 Referências

1. Spring Boot Documentation. https://spring.io/projects/spring-boot
2. PostgreSQL 15 Documentation. https://www.postgresql.org/docs/15/
3. Eclipse Mosquitto MQTT Broker. https://mosquitto.org/
4. gRPC Documentation. https://grpc.io/docs/
5. Protocol Buffers. https://protobuf.dev/
6. MQTT Version 3.1.1 Specification. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
7. REST API. https://restfulapi.net/
8. Docker Documentation. https://docs.docker.com/
9. Maven Documentation. https://maven.apache.org/guides/
10. JPA/Hibernate Documentation. https://hibernate.org/orm/documentation/
11. GitHub Copilot. https://github.com/copilot