Universidade de Aveiro

Sistemas Flexíveis de Produção

Tiago Marques 98459

11 de Janeiro de 2024

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

O presente relatório visa descrever um cenário real, da ejeção de um *payload* de um foguetão, e o cenário de teste, utilizado para testar algumas das funcionalidades propostas.
Este projeto final é realizado no âmbito da unidade curricular de Sistemas Flexíveis de Produção.

Capítulo 2

Descrição do Problema

2.1 Cenário Real

Num projeto de lançamento de foguete, um *payload*, que contém um ESP32 e vários sensores, é ejetado, quando o foguete atinge o seu apogeu. Ao ser lançado o *payload* começa a recolher diversas métricas sobre o ambiente e o ar ao seu redor até chegar ao chão.

Para mitigar possíveis problemas, como a perda total do *payload* devido a circunstâncias imprevistas, o *payload* adota uma estratégia preventiva. Utilizando um módulo LoRa, ele transmite os dados coletados em tempo real para um ESP32 na estação terrestre.

Após o ESP32 na estação terrestre receber os dados do *payload* por meio da transmissão LoRa, ele realiza o processamento interno dessas informações. Em seguida, o ESP32 inicia um pedido HTTP direcionado para a REST API hospedada no computador. Esse pedido contém as métricas recolhidas pelo *payload*.

A REST API, ao receber esse pedido, entra em ação. Ela valida e verifica os dados antes de proceder à atualização da base de dados.

Posteriormente os dados recebidos são mostrados na *Dashboard*, que realiza pedidos *HTTP* regulares à API em busca das informações mais recentes. A API, por sua vez, responde com os dados atualizados, e a *Dashboard* os exibe em tempo real.

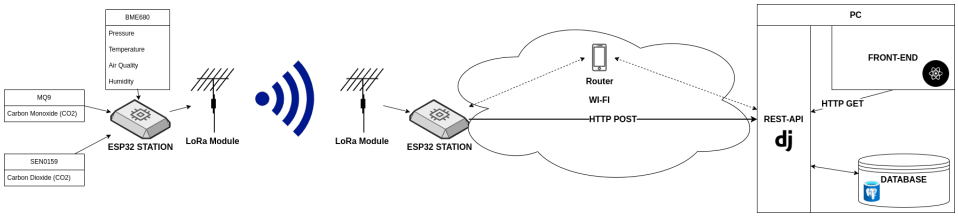

Figura 2.1: Cenário Real

2.2 Cenário de Teste

Neste cenário de teste, será testado todo o *setup* com a exceção da comunicação por LoRa. Assim sendo, será utilizado apenas um ESP32, que faz a recolha de dados dos sensores e que transmite, os dados recém recolhidos, através de tráfego *HTTP* para a API que os processa e atualiza a base de dados com os novos dados. Para ser possível haver comunicação *HTTP* ambos o ESP32 e o PC se encontram na mesma rede, que neste caso é um hotspot de telemóvel.
A imagem abaixo demonstra os diversos componentes usados e tecnologias e como estes comunicam entre si.

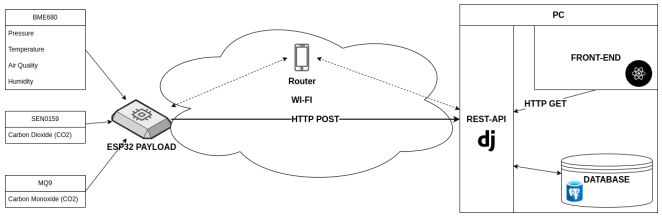

Figura 2.2: Cenário de Teste

2.2.1 Tecnologias

- REST API: Faz uso de Django, que é uma *web framework* em Python, e da sua extensão REST API que simplifica a criação de interfaces (APIs) para aplicações web.
  No contexto do projeto, a API Django REST tem três funções:

  1. Receber, processar as informações vindas do *POST HTTP* vindas do ESP32 e introduzir essas informações, se corretas, na base de dados. E consequentemente responder ao *POST HTTP* do ESP32.
  2. Receber os pedidos *HTTP GET* do *FrontEnd (Dashboard)* e responder com os dados pedidos.
  3. A API é a única que comunica com a base de dados pelo que quaisquer novos dados terão de passar pela API e posteriormente adicionados

à base de dados. Também qualquer pedido de informação é primeiro enviado à API, que por sua vez pede os dados à base de dados que responde, e só depois é que a API envia os dados pedidos.

- Base de Dados: usa PostgreSQL que é um sistema de gestão de base de dados relacional. É usado para armazenar os dados enviados pelo ESP32 do *payload* que foram recebidos pela API.
- FrontEnd: Usa React que é uma biblioteca JavaScript utilizada para construir *User Interfaces* interativas e dinâmicas. Na *Dashboard* do projeto, o React facilita a exibição em tempo real dos dados provenientes da API/-base de dados.

2.2.2 Componentes

- REST API: Possui uma serie de *endpoints* que permite a atualização dos dados recolhidos pelos sensores, assim como *endpoints* que permitem ao *FrontEnd* pedir os dados mais recentes recolhidos.
  - **mq9/**: No caso do pedido *HTTP* ser um *POST*, o pedido deve conter um *JSON* com o seguinte formato {'co':<float>}, este pedido serve para adicionar uma nova medição de CO (Carbon Monoxide).
    Se o pedido *HTTP* for um *GET* a API retorna o valor mais recente de CO (Carbon Monoxide) na base de dados em formato *JSON*, {'id': <int>, 'co':<float>, 'time':<datetime>}.
  - **mq9/history/<int:last_of>/:** O pedido precisa de ser um *HTTP GET*. A API responde com um arrray *JSON*, de tamanho last_of, com o seguinte formato: [{'id': <int>, 'co':<float>, 'time':<datetime>}, ...]. Este array contém as ultimas last_of medições de CO (Carbon Monoxide) presentes na base de dados. A variável last_of tem de ser maior ou igual que 0, e no caso de não ser atribuído um valor à variável ou se for atribuído o valor 0, a API responde com todas as medidas de CO que existem na base de dados.
  - **sen0159/**: No caso do pedido *HTTP* ser um *POST*, o pedido deve conter um *JSON* com o seguinte formato {'co2':<float>}, este pedido serve para adicionar uma nova medição de CO2 (Carbon Dioxide).
    Se o pedido *HTTP* for um *GET* a API retorna o valor mais recente de CO2 (Carbon Dioxide) na base de dados em formato *JSON*, {'id': <int>, 'co2':<float>, 'time':<datetime>}.
  - **sen0159/history/<int:last_of>/:** O pedido precisa de ser um *HTTP GET*. A API responde com um arrray *JSON*, de tamanho last_of, com o seguinte formato: [{'id': <int>, 'co2':<float>, 'time':<datetime>}, ...]. Este array contém as ultimas last_of medições de CO2 (Carbon Dioxide) presentes na base de dados. A variável last_of tem de ser maior ou igual que 0, e no caso de não ser atribuído um valor à variável ou se for atribuído o valor 0, a API responde com todas as medidas de CO2 que existem na base de dados.

- **bme680/**: No caso do pedido *HTTP* ser um *POST*, o pedido deve conter um *JSON* com o seguinte formato {'temperature':<float>, 'humidity':<float>, 'pressure':<float>, 'gas':<float>}, este pedido serve para adicionar uma novas medições da temperatura, pressão, humidade e qualidade do ar.
  Se o pedido *HTTP* for um *GET* a API retorna os valores mais recente da temperatura (ºC), humidade (%), pressão (hPa) e índex de qualidade do ar (entre 0 e 500) na base de dados em formato *JSON*, {'id': <int>, 'temperature':<float>, 'humidity':<float>, 'pressure':<float>, 'gas':<float>, 'time':<datetime>}.
- **bme680/history/<int:last_of>/:** O pedido precisa de ser um *HTTP GET*. A API responde com um arrray *JSON*, de tamanho last_of, com o seguinte formato: [{'id': <int>, 'temperature':<float>, 'humidity':<float>, 'pressure':<float>, 'gas':<float>, 'time':<datetime>}, ...]. Este array contém as ultimas last_of medições de temperatura, humidade, pressão e qualidade de ar presentes na base de dados. A variável last_of tem de ser maior ou igual que 0, e no caso de não ser atribuído um valor à variável ou se for atribuído o valor 0, a API responde com todas as medidas de temperatura, humidade, pressão e qualidade de ar que existem na base de dados.
- **logs/:** O pedido precisa de ser um *HTTP GET*. A API responde os *logs* da API no formato [logs: "linha de log1", "linha log2", ...].

- BME680: Inicialmente é configurada no ESP32 a interface I2C (ficheiro payload/main/payload.c função bme680_init()). De seguida, são configurados vários parâmetros no sensor como o *Oversampling* para as diversas métricas, o tamanho do filtro IIR e o *Heater Profile* (ficheiro payload/-main/payload.c função bme680_config_msr()). Após este processo de configuração o sensor está preparado para recolher as diversas métricas (ficheiro payload/main/payload.c função bme680_work()).

- MQ9: É configurada a ADC no ESP32, (ficheiro payload/main/payload.c função mq9_init()), e o fator R0 é calculado através de 100 leituras. Depois destes processos iniciais a *task* mq9_work (ficheiro payload/main/-payload.c) lê os valores da ADC periodicamente e transforma valores da voltagem para valores reais de CO por PPM.

- SEN0159: À semelhança do mq9, é configurada a ADC no ESP32, (ficheiro payload/main/payload.c função sen0159_init()),. Para diminuir erros e variações das leituras por parte do sensor, são feitas algumas leituras à ADC e calculado a media dessas leituras (ficheiro payload/components/sen0159/sen0159.c função sen0159Read()), esse valor é posteriormente convertido para CO2 por PPM.

2.2.3 Fluxo de Operação

1. Ambos o ESP32 e o PC se conectam à rede móvel do telemóvel, adicionalmente o ESP32 configura os seus sensores.

2. O ESP32 *payload* recolhe os dados dos seus sensores e envia esses dados através de um *POST HTTP* para a API.

3. A REST API recebe os dados do *POST HTTP*, valida os dados recebidos e atualiza a informação na base de dados. Por último envia a resposta *HTTP* para o ESP32.

4. A *Dashboard* faz *HTTP GET*s regulares à API para obter os dados mais recentes armazenados na base de dados. A API responde com os dados que a *Dashboard* pediu, consequentemente a *Dashboard* atualiza as informações exibidas na ecrã.

Capítulo 3

Deployment

3.0.1 Django REST API

Para o *deployment* da API é preciso ter linguagem python3 no sistema e instalar os requisitos que se encontram no ficheiro rest_api/requirements.txt. Para fazer o *deploy* da API basta abrir um terminal, entrar dentro da pasta rest_api e fazer o comando 'python3 manage.py runserver 0.0.0.0:8000'.

Nota: Antes de inicializar a API precisa de ter o serviço da base de dados presente.

3.0.2 Base de Dados PostgreSQL

Para instalar a base de dados recomenda-se que o faça pelas fontes oficiais, site oficial do PostgreSQL. Durante a instalação da base de dados o nome, o utilizador e a palavra passe devem ter todos o valor 'uart' e a base de dados deverá usar o endereço IP 127.0.0.1:5432.

3.0.3 React FrontEnd

Para instalar as dependências deve ter a ferramenta yarn instalada e dentro da pasta UART_Dashboard_UI, num terminal, correr o comando 'yarn install'. Depois de estarem as dependências satisfeitas, pode fazer o comando 'yarn dev', se desejar executar em modo *developer*, ou então usar o comando 'yarn build' que cria o website estático.

3.0.4 Programa ESP32

Para compilar e executar o código do ESP32, precisa de ter a framework esp-idf instalada. Para compilar o código, através de um terminal, entrar na pasta payload e fazer o comando 'idf.py build'. De seguida, se tiver um ESP32 ligado ao PC por USB, basta fazer o comando 'idf.py flash monitor' e o ESP32 ira receber o código compilado e executar o programa.