FACULDADE DE CIÊNCIAS E TECNOLOGIA DA UNIVERSIDADE DE COIMBRA

Agricultura de Precisão

Internet das Coisas

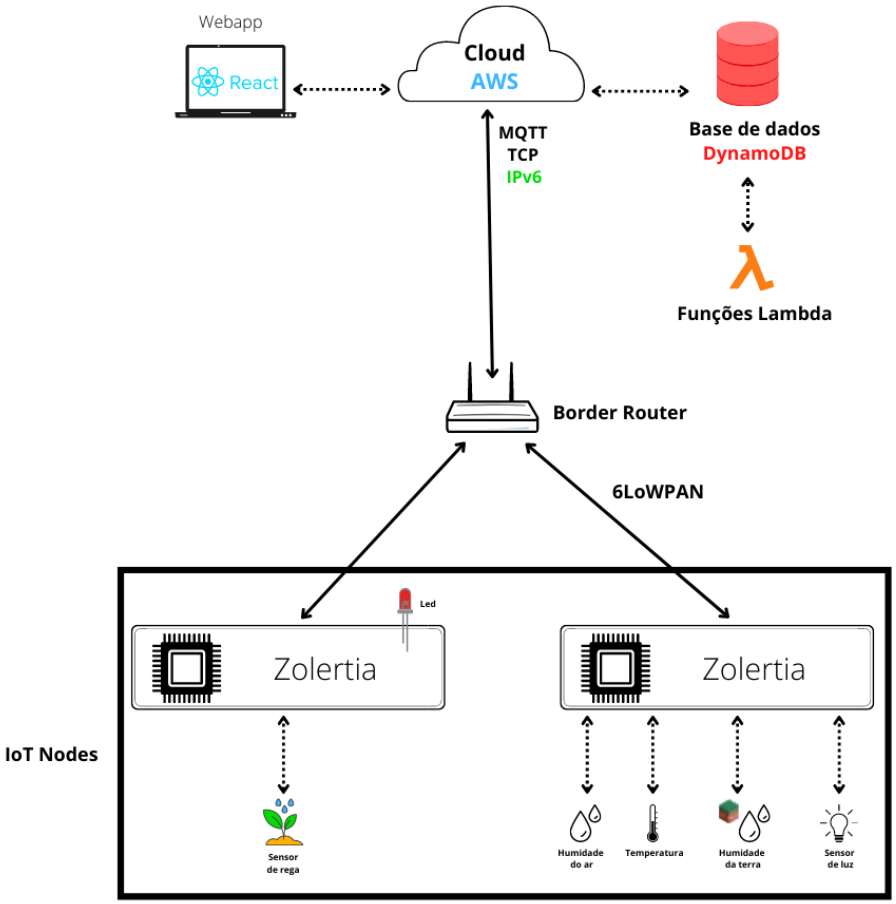
2021/2022

Duarte Cruz - 2017264087

Afonso Martingo - 2018299655

Índice

1. Diagrama da Arquitetura do projeto



1. Introdução

Este projeto tem um modo automático, em que os módulos comunicam entre si e definem o seu estado de ligado ou desligado (por exemplo, a rega), consoante os valores recebidos pelos outros e tem um modo manual, em que o utilizador através da web app pode dar comandos para os módulos.

Para conseguir esta ideia, achamos melhor e por termos mais base das aulas o uso da arquitetura MQTT, assim vai permitir uma maior flexibilidade entre comunicações e pode ser usado em ambientes menos capacitados de internet e energia.

A Arquitetura MQTT é um protocolo baseado na comunicação de dispositivos de publish-subscribe. Este método de comunicação é semelhante a uma rede social, uma placa

publica alguma mensagem e para o destinatário receber essa mensagem tem que subscrever ao tópico. Nesta arquitetura as placas vão se encontrar num meio local (mosquitto), a comunicar via rádio por 6LoWPAN. Como não temos acesso a um broker, utilizamos uma plataforma disponibilizada nas aulas práticas, o tunslip, que serve para simular a conexão via usb da placa e enviar para o mosquitto. Depois essa conexão vai ser transmitida em uma ligação encriptada TLS do mesmo protocolo. O tipo de IP usado é o IPv6, transmitido pelo 6LoWPAN.

O 6LoWPAN define mecanismos de encapsulamento e compressão de cabeçalho que permitem que pacotes IPv6 sejam enviados e recebidos em redes baseadas em IEEE 802.15.4.

Esta arquitetura utiliza o protocolo de comunicação TCP que fornece entrega confiável, ordenada e com verificação de erros, de um fluxo de octetos (bytes) entre aplicativos executados em hosts que se comunicam por meio de uma rede IP. Para uma conexão ocorrer o servidor tem que estar em modo escuta, senão não consegue comunicar.

O formato das mensagens são em JSON, como usado nas aulas práticas.

2. Aquisição de Dados e Processamento *Zolertia*

Começámos por programar o *firmware* e a bridge para a *cloud*. Para isso usámos como uma boa base o código disponibilizado pelos docentes.

No *firmware* tivemos que adicionar as drives e o processamento dos dados dos sensores e modificar a mensagem *JSON* transmitida para o tópico local do *mosquitto*. Como as *drivers* já estão presentes no *Contiki*, vamos ativá-las no *PROCESS_THREAD*.

```
1.  //configure sensors (analog and digital)
2.    adc_zoul.configure(SENSORS_HW_INIT, ZOUL_SENSORS_ADC_ALL);
3.
4.    /* Use pin number not mask, for example if using the PA5 pin then use 5 */
5.    //adc_sensors.configure(ANALOG_GROVE_LIGHT, ADC_PIN);
6.
7.    SENSORS_ACTIVATE(dht22);
8.
9.    SENSORS_ACTIVATE(tsl256x);
10.
11.
12.     /* Enable the interrupt source for values over the threshold.  The sensor
13.     * compares against the value of CH0, one way to find out the required
14.     * threshold for a given lux quantity is to enable the DEBUG flag and see
15.     * the CH0 value for a given measurement.  The other is to reverse the
16.     * calculations done in the calculate_lux() function.  The below value roughly
17.     * represents a 2500 lux threshold, same as pointing a flashlight directly
18.     */
19. //0x15B8
20.   tsl256x.configure(TSL256X_INT_OVER, 0x15B8);
21.
22.   TSL256X_REGISTER_INT(light_interrupt_callback);
23.
```

Agora na função *Publish*, vamos alterar a mensagem *JSON* para o que queremos alterar. No nosso caso escolhemos transmitir o nome da placa, para a identificar no caso de haver mais do que uma, o número da iteração e o tempo que o sensor está ativo.

```
1.    seq_nr_value++;
2.    buf_ptr = app_buffer;
3.
4.    len = snprintf(buf_ptr, remaining,
5.                   "{"
6.                  "\"Board\":\"%s\","
7.                  "\"iteracao\":%d,"
8.                  "\"Uptime\":%lu",
9.                   BOARD_ID_STRING, seq_nr_value, clock_seconds());
10.
11.   if(len < 0 || len >= remaining) {
12.     printf("Buffer too short. Have %d, need %d + \\0\n", remaining, len);
13.     return;
14.   }
```

Feito isto, vamos adicionar os dados extraídos dos sensores à mensagem anterior. Para isto, vamos dar uso a uma função integrada no *Contiki* para extrair os valores dos sensores e no caso do *Soil Moisture Sensor*, formatar os valores para percentagem.

Cada vez que queremos aumentar a mensagem, temos de aumentar o *buffer* da mesma.

```
1.    remaining -= len;
2.    buf_ptr += len;
3.
4.    aux = adc_zoul.value(ZOUL_SENSORS_ADC1);
5.    int percentage = (aux*100) / 18400;//percentage valor
6.    len = snprintf(buf_ptr, remaining, ",\"Soil\":\"%u\"", percentage);
7.
8.    remaining -= len;
9.    buf_ptr += len;
10.
11.   aux = adc_zoul.value(ZOUL_SENSORS_ADC3);
12.   len = snprintf(buf_ptr, remaining, ",\"Light\":\"%u\"", aux);
13.
14.   remaining -= len;
15.   buf_ptr += len;
16.
17.   len = snprintf(buf_ptr, remaining, "}");
```

Por último, vamos fazer um pequeno processamento dos dados recebidos e averiguar se devemos ativar o estado de emergência e ligar a rega automaticamente. Para isso vamos proceder a uma calibração, que pode variar consoante o ambiente em que o sistema se encontrar.

Como não temos acesso a um sistema de rega inteligente, simulámos com a ativação do led integrado da placa.

```
1.  if (percentage > 60 && aux > 15000)
2.    {
3.      printf("Auto Rega on!\n");
4.      leds_on(LEDS_BLUE);
5.    }
6.    else{
```

```
7.      printf("Auto Rega off!\n");
8.      leds_off(LEDS_BLUE);
9.    }
```

3. Transmissão de Dados e Armazenamento na *Cloud*

Como não temos acesso a um broker, temos que usar um ferramenta utilizada nas aulas práticas, cujo nome é *tunslip*. Esta ferramenta tem como objetivo simular um broker e ligar a placa a um tópico local no *mosquitto*, usando a porta *usb* do computador.

Surge outro problema, enviar os valores localmente para a *cloud*. Para resolver este problema vamos usar outra ferramenta disponibilizada pelos docentes, o *Bridge*. Este é um script em *python* que vai transmitir os valores recebidos localmente para a *cloud* por *mqtt*.

Na *cloud* criamos uma *rule* com um comando em *SQL*, na secção de IoT, para quando chegar a mensagem *JSON* a um tópico *MQTT*, armazenar a mesma numa base de dados. Esta base de dados foi criada à priori, na plataforma *DynamoDB* incluída na *AWS*.

4. Interface gráfica

A interface para o utilizador vamos usar uma plataforma dentro da *AWS*, chamada de *Amplify*. É um conjunto de ferramentas necessárias para desenvolver vários tipos de aplicações, que no nosso caso escolhemos fazer uma *web app*.

A *Amplify* vai possibilitar o uso de dados da *DynamoDB*, para interagir com os valores guardados dos sensores e a integração de uma autenticação e gerenciador de usuários na app. Toda a parte gráfica foi desenvolvida usando a libraria *javascript React*.

Para a criação da interface propriamente dita, usámos o *visual studio code* ligado à *Amplify CLI* e como *framework* a libraria *React* em *javascript*.

Na Amplify começámos por criar os métodos de autenticação e habilitar uma ferramenta muito útil para a aplicação conseguir comunicar com as base de dados existentes, o *GraphQL API*. Na autenticação definimos usar apenas por email.

Depois passando para o puro código, começámos por criar os botões da interface. Para isso temos de criar funções que consigam comunicar com a DynamoDB, seja para atualizar os dados da aplicação com valores novos ou criar valores noutra base de dados.

7.1. Fetch Data

O primeiro botão vai servir para atualizar os dados lidos pela aplicação. Isto é para simular a atualização de dados em tempo real, visto que, não conseguimos arranjar outra

forma de os atualizar devido a não estarmos muito familiarizados com a linguagem de desenvolvimento e com limitações nas funções criadas pela *API.*

Esta função vai pegar nos valores lidos na linha 3 e guardá-los nos arrays Soil e Light, para serem usadas mais tarde.

```
1.        async function fetchTodos() {
2.      try {
3.          const todoData = await API.graphql(graphqlOperation(listBoardZolertia1S ))
4.          const todos = todoData.data.listBoardZolertia1S.items
5.          setTodos(todos)
6.          todos.map((todo, index) => (
7.              Soil.push(todo.Soil),
8.              Light.push(todo.Light)
9.          ))
10.
11.     } catch (err) { console.log('error fetching todos') }
12.   }
```

```
1.  <button style={styles.button} onClick={fetchTodos}>Fetch DB</button>
```

7.2. Irrigation ON

Este botão vai criar e atualizar um elemento numa base de dados diferente à usada anteriormente. O valor da descrição 1 é para definir que quer ligar a rega, quando for processado posteriormente.

```
1.    async function RegaON() {
2.      try {
3.        const regaon = { id: 'f9660efe-c831-4088-929f-2ebc1eb9b260', name: 'Rega',
    description: 1 }
4.        await API.graphql(graphqlOperation(updateTodo, {input: regaon}))
5.      } catch (err) {
6.        console.log('error creating todo:', err)
7.      }
8.    }
9.
```

```
1.  <button style={styles.button} onClick={RegaON}>Irrigation ON</button>
```

7.3. Irrigation OFF

Este botão tem a mesma função que o anterior só que muda a descrição para o valor 0.

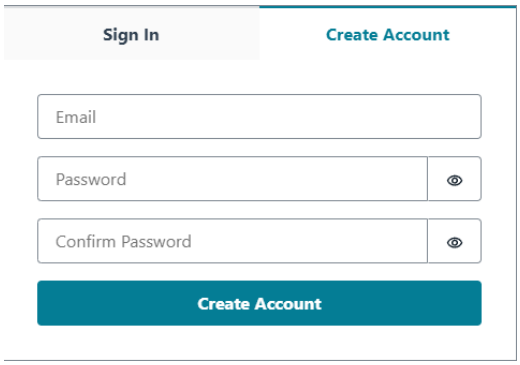
```
1.    async function RegaOFF() {
2.      try {
3.        const regaoff = { id: 'f9660efe-c831-4088-929f-2ebc1eb9b260', name: 'Rega',
    description: 0 }
4.        await API.graphql(graphqlOperation(updateTodo, {input: regaoff}))
5.      } catch (err) {
6.        console.log('error creating todo:', err)
7.      }
8.    }
9.
```

```
1.  <button style={styles.button} onClick={RegaOFF}>Irrigation OFF</button>
```

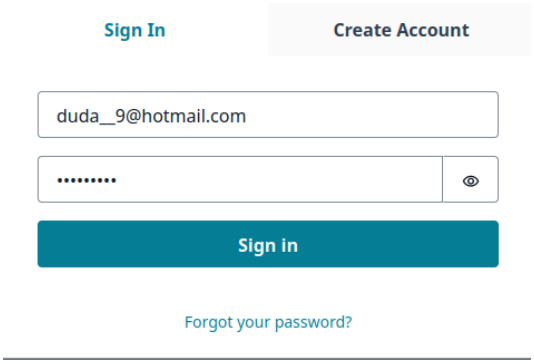
7.4. Gráfico

Este gráfico foi implementado de um template feito por outra pessoa, disponibilizado no github. Tivemos que incorporar no nossa aplicação e passar os valores dos arrays, anteriormente criados com os valores da base de dados, para as coordenadas em y do gráfico.

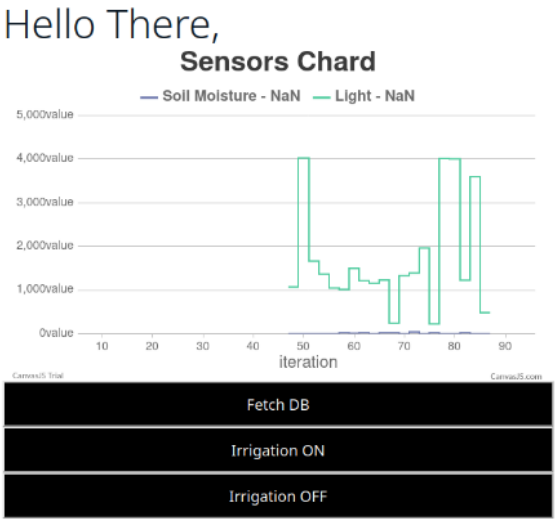
```
1.  for (var i = 0; i < count; i++) {
2.    xValue += 2;
3.    yValue1 = parseInt(Soil.pop());
4.    yValue2 = parseInt(Light.pop());
5.    dataPoints1.push({
6.      x: xValue,
7.      y: yValue1
8.    });
9.    dataPoints2.push({
10.     x: xValue,
11.     y: yValue2
12.   });
13. }
```



1. Criar Conta



2. Página de Login



3. Dashboard

8. Interação Aplicação -> *Zolertia*

A interação da aplicação com a *zolertia* vai ser feito pela base de dados criada anteriormente com os botões da app. Essa base de dados vai ter apenas um elemento que vai variar a sua descrição pelos valores 1 e 0, dependendo da interação com a app.

Para esses valores chegarem à placa, temos que criar um função *lambda*, também integrada na *AWS,* escrita em *python*.

Em primeiro lugar vamos fazer com que essa função consiga ler os valores da *DynamoDB,* com a ajuda da libraria *boto3* do *python* e ligar ao MQTT da AWS.

```
1.  import boto3
2.  dynamodb = boto3.resource('dynamodb')
3.  mqtt = boto3.client('iot-data', region_name='eu-west-1')
4.
```

Com o acesso à base de dados, vamos ler o valores e guardar na variável *item.*

```
1.      table = dynamodb.Table('Todo-wqohswa37zamfnu4ine2bdq3f4-dev')
2.      body = table.scan()
3.      items = body['Items']
4.
```

Agora só resta mandar uma mensagem por MQTT para a placa, consoante o valor seja 1 ou 0. Essa mensagem para o mesmo tópico que o script Bridge subscreveu anteriormente, fazendo com que vá automaticamente para a placa, percorrendo assim o caminho inverso da mensagem para chegar a cloud.

```
1.      for aux in items:
2.
3.          if (aux['description'] == "1"):
4.
5.              response = mqtt.publish(
6.                  topic = 'cloud/action',
7.                  qos=1,
8.                  payload = json.dumps({"Rega": "Sim"})
9.              )
10.
11.         else:
12.             response = mqtt.publish(
13.                 topic = 'cloud/action',
14.                 qos=1,
15.                 payload = json.dumps({"Rega": "Nao"})
16.             )
17.
```

9. Processamento de dados Recebidos *Zolertia*

Para processar a mensagem recebida pelo *Bridge* adicionamos umas restrições para ler os caracteres recebidos e traduzir para comandos de ligar e desligar a rega.

```
1.        const char needle[1] = "Sim";
2.    char *ret;
3.
4.    ret = strstr(chunk, needle);
5.
6.    if(ret != NULL) {
7.      leds_on(LEDS_RED);
8.      printf("Rega on!\n");
9.    } else{
10.     leds_off(LEDS_RED);
11.     printf("Rega off!\n");
12.   }
13.
```

Este processamento vai ser adicionado no *pub_handler. A função strstr, vai servir para verificar se existe a palavra "Sim" na mensagem em string que chega da Bridge. Caso se verifique vai ligar o led, para simular a rega e vise versa.*

10. Lista de Material

1. Grove Light Analog Sensor
2. Moisture Sensor
3. Zolertia RE-Mote

11. Resultados e produto final

Este projeto resolve problemas existentes de manutenção na agricultura, tomando partido da tecnologia para oferecer uma forma fácil e remota de a controlar.

Para mantermos uma agricultura saudável e com qualidade é preciso usarmos muito do nosso tempo apenas para essa tarefa. O nosso trabalho visa melhorar isso com a introdução da manutenção remota, fazendo com que a pessoa possa estar a realizar outras tarefas e, em simultâneo, receber avisos e dar instruções por via do telemóvel.

Com a medição da humidade do solo e do ar, conseguimos precaver um dos maiores problemas ambientais hoje registados, a seca. A seca tem atingido valores muito elevados e pretendemos contribuir para a melhorar.

Em termos de produto final, está preparado para ser escalado a mais placas e sensores, apenas precisa de uns ajustes. A parte gráfica tem a raiz implementada, mas precisa de muito melhoramento para ser usado em escala maior.

Não conseguimos utilizar todos os sensores previsto, devido a problemas inesperados na quantidade de sensores disponíveis no departamento. O que causou estarmos a trabalhar com três sensores analógicos para duas portas digitais e dois deles diferentes do previsto. Com isto, escolhemos usar apenas o sensor de luz e o sensor de humidade do solo, pois o sensor DHT11 não tem drivers para o contiki.

Em termos da realização da aplicação web app, as coisas não correram como esperado, devido a alguns problemas que foram surgindo desistimos do design inicial definido, visto que, apesar de teoricamente ser compatível com *AWS Amplify*, na prática provocou alguns bugs visuais e era de muito complexa implementação. Como solução optámos por usar uma plataforma que também tem integração que é a libraria javascript React.

12. Bibliografia

https://github.com/Zolertia/Resources/wiki/Sensors

https://github.com/Zolertia/Resources/wiki/RE-Mote

https://marketingagricola.pt/como-a-automacao-esta-a-revolucionar-a-agricultura/

https://seeeddoc.github.io/Grove-Light_Sensor/

https://www.waveshare.com/moisture-sensor.htm

https://docs.amplify.aws/start/q/integration/react/

https://github.com/HishmatRai/React-Js-canvasjs-chart-samples