cose

- Sulla seriale c'e' un limite di 64 bytes e tronca i messaggi

App

Per la comunicazione tra l'app e arduino era necessario l'utilizzo del HC-05, un dispositivo che permette la comunicazione tra un microprocesore e un qualsiasi dispositivo dotato di comunicazione Bluetooth.

Una funzionalità importante dell'app era quella di rimanere costantemente aggiornata sullo stato dell'arduino e dei suoi componenti anche quando in modalità automatica, e per fare ciò abbiamo adoperato la classe astratta AsyncTask. Questa classe consente i eseguire determinate operazioni in background senza dover manipolare thread.

```
AsyncTask.execute(()->{
    String stringa = executeRequest("http://localhost:3000/garden/app/getData", "");
    this.runOnUiThread(()->{

        try {
            json = new JSONObject(stringa);
            viewModel.init(json.getInt("led3"), json.getInt("led4"), json.getInt(WATER));valueLed3.set
            valueLed4.setText(String.valueOf(json.getInt("led4")));
            irrValue.setText(String.valueOf(json.getInt(WATER)));
            json.remove("w");
            json.remove("t");
            json.remove("b");
            json.remove("led_esp");
        } catch (JSONException e) {
            e.printStackTrace();
        }
    });
});
```

Per la ricezione dei dati dal server invece abbiamo adoperato una funzione che passato come parametro l'indirizzo url del server, apriva una richiesta di GET a quel determinato indirizzo per poi mettersi in attesa di ricevere la stringa JSON contenente tutte le informazioni necessarie per l'inizializzazione e l'aggiornamento dell'app.

La comunicazione tra app e controller si basa sull' utilizzo di messaggi JSON che ci ha permesso uno scambio più veloce ed efficiente di informazioni tra le due parti. Per l'invio di un qualsiasi cambiamento infatti andremo a modificare il JSON ricevuto inizialmente dal server aggiornandone le informazioni(per esempio se voglio settare la luminosità del led3 a 4, andremo modificare il valore della chiave "led3" a 4). Una volta fatto ciò spediremo i dati compattati alla seriale apposita dell'arduino, che sarà in grado di riceverli e leggerli.

Controller

Il controller gestirà tramite scheduler due task principali, la prima sarà quella del sistema di irrigazione, mentre l'altra sarà per il sistema della "casa".

Il sistema di irrigazione avrà due stati: OPERATIVO: specificherà che il sistema è operativo e pronto per l'immediato utilizzo. NON OPERATIVO: specificherà che il sistema non sarà operativo. Ciò avverrà in seguito all'attivazione dell'impianto di irrigazione che porterà poi ad una "pausa" dell'impianto per un determinato periodo di tempo;

La "casa" invece avrà gli stati: AUTO: qui l'attivazione dei led e dell'impianto di irrigazione avverrà in maniera automatica a seconda del valori letti dai sensori. MANUAL: qui la "casa" verrà gestita tramite dispositivo mobile ALARM: in questo caso invece

il sistema si metterà in attesa di ricevere dal dispositivo mobile un segnale che gli permettà di disattivare lo stato di allarme e di tornare allo stato precedente.

Per la comunicazione tra App e Arduino utilizzeremo la libreria SoftwareSerial che consente la comunicazione seriale su altri pin digitali di una scheda Arduino. Tramite questa seriale Arduino riceverà dall'app il messaggio sottoforma di JSON con i cambiamente effettuati da app.

```
case MANUAL:
    checkAlarmCondition();
      if(btChannel.available()){
        String msg = btChannel.readString();
       // Serial.println("manual try: " + msg);
        deserializeJson(doc, msg);
        JsonObject root = doc.as<JsonObject>();
        checkChanges(root);
      }
      break;
```

Inoltre Arduino riceverà dal seriale principale i valori letti dall'esp sottoforma di JSON che utilizzerà per effettuare i check necessari per il corretto funzionamento della casa.

```
void RoutineTask::readSerial(){
  String inData = "";
    if(Serial.available() > 0){
      inData = Serial.readString();
      //make inData a JSON obj
      StaticJsonDocument<200> doc1;
      DeserializationError error = deserializeJson(doc1, inData);
      // Test if parsing succeeds.
      if (error) {
        Serial.println("failed serial msg: " + inData);
        Serial.print(F("deserializeJson() failed: "));
        Serial.println(error.f_str());
        return;
      }
      int t = doc1["t"];
      int b = doc1["b"];
      //reads temp and lumionosity
      garden->sensorBoard->temp->setTemp(t);
      garden->sensorBoard->photoresistor->setValue(b);
    }
}
```

Dash-Board

La dashboard è un interfaccia web-based con la quale si può tenere sotto controllo il nostro garden. Per la realizzazione abbiamo utlizzato electron, un modulo di nodeJS con il quale abbiamo creato una pagina web responsive ai cambiamenti del nostro device.

Per poter avere dei valori corretti da mostrare questa pagina ha al suo interno un piccolo server in ascolto sulla porta **http://localhost:3000/garden/dashboard** per poter ricevere le GET effettuate dal service contenti le informazioni necessarie ad mantenere aggiornata la schermata.

Per indicare l'accensione/spegnimento dei led abbiamo usato delle immagini png con lampadine accese o spente, mentre per la

parte di sensori abbiamo non solo mostrato i valori a schermo ma anche utilizzato delle progress bar mappate entro i valori indicatoci nell'assignment.

Sensor-Board

Questa parte del assignment è svolta dall'esp, all'interno della funzione di setup l'esp si collega al wifi, successivamente abbiamo un server per poter gestire alcune richieste HTTP grazie alla libreria ESPAsyncWebServer all'indirizzo IP dell'esp che viene stampato subito prima.

```
    server.on("/get", HTTP_GET, [](AsyncWebServerRequest *request){
        if (request->getParam("led")->value() == "1"){
        digitalWrite(RED, HIGH);
        } else {
          digitalWrite(RED, LOW);
          }
      // request->send(200, "text/plain", "Led received state");
    });
```

Questa funzione asincrona permette di poter attivare il led in caso di ALARM direttamente dal service tramite una semplice GET con attributo led=1.

Il sensor board quindi si occupa di segnalare lo stato di ALARM che viene triggerato da alcuni parametri, uno dei quali viene rilevato dall'esp, come la temperatura ed anche di rilevare la luce tramite l'apposito sensore.

Dopo avere eseguito una rilevazione, questa viene compattata all'interno di un file JSON e inviata al server di garden service, per fare ciò abbiamo utilizzato ngrok in modo da poter eseguire una POST ad un indirizzo IP pubblico sul quale in realtà è in ascolto il nostro server.

Service

Per la creazione del server di ascolto abbiamo deciso di utilizzare NodeJS, dato che in parte sapevamo come usarlo in parte volevamo imparare qualcosa di nuovo.

La cosa più complicata è stata la gestione della seriale attraverso nodeJS ma alla fine ci siamo riusciti, si deve prima importare due moduli

```
const SerialPort = require('serialport').SerialPort;
const Readline = require('@serialport/parser-readline');
```

Dichiare due oggetti che serviranno a parsare i messaggi con uno specifico delimitatore e una porta seriale con i classici parametri di arduino.

```
const port = new SerialPort({path: 'COM3', baudRate: 9600 , parser:Readline});
const parser = port.pipe(new Readline.ReadlineParser({ delimiter: '\r\n' }))
```

Per poi definire la funzione che sarà in ascolto di tutti i messaggi sulla porta seriale, abbiamo fatto la scelta di usare il formato JSON per la comunicazione e questa funzione tenta di parsare ogni cosa che viene parsata sulla seriale come un JSON e si blocca quando vede ad esempio del normale testo. Qui leggiamo anche il parametro *led_esp* è settato ad uno dall'arduino ossia ci troviamo in uno stato di ALARM e inviamo una GET al server che sta runnando sull'esp.

```
 //serial data
port.on("open", async function () {
  parser.on('data', async function(data) {
   console.log(data);
    var g = data;
    var j = JSON.parse(g);
    gardenObject = jsonConcat(gardenObject, j);
    if(gardenObject["led_esp"] > 0){
      requestify.get('http://192.168.1.22/get?led=1')
      .then(function(response) {
          // Get the response body (JSON parsed or jQuery object for XMLs)
          // console.log(response.getBody());
      });
    } else {
      requestify.get('http://192.168.1.22/get?led=0')
      .then(function(response) {
          // Get the response body (JSON parsed or jQuery object for XMLs)
          // console.log(response.getBody());

});
}});
});
```

Utiizziamo una funzione scritta da noi (jsonConcat) per concatenare il JSON ricevuto da arduino a quello gia presente sul server, questo per risolvere uno dei problemi piu grandi che abbiamo incontrato durante lo sviluppo di questo assignment, ossia il troncamento di messaggi. A quanto pare arduino, dopo un periodo variabile di tempo inviava messaggi troncati che bloccavano l'esecuzione del server e di tutto il processo, apparentemente il buffer della seriale ha una specie di limite di bytes e periodicamente deve svuotare questo buffer, all'inizio avevamo incluso tutti i parametri all'interno di uno stesso JSON direttamente dall'arduino ma questa cosa oltre a non essere precisa dava questo problema sporadicamente.

Perciò dopo la divisione dei JSON dell'esp e dell'arduino è stato più semplice lavorare e dopo la ricezione del JSON dal sensor-board(esp) abbiamo inserito la funzione per poter scrivere sulla seriale i dati appena ricevuti.

```
 //Get per andare a ritirare i dati necessari
app.get('/garden/app/getData', async function(req,res){
  while(gardenObject == null){
      return;
  }
  res.send(gardenObject);
})

function sendOnSerial() {
  var esp_obj = {
    "t" : gardenObject["t"],
    "b" : gardenObject["b"],
  }
  port.write(JSON.stringify(esp_obj), function(err) {
    if (err) {
      return console.log('Error on write: ', err.message)
    }
    console.log('message written on serial')
    console.log(JSON.stringify(esp_obj))
  })
}
```

Link video assignement: https://liveunibo-my.sharepoint.com/:v:/g/personal/michele_nardini2_studio_unibo_it/ESpQAYytkLlLpZoA7LcEQ3YBj_Ib3aXv_u_Frwlo3UYDlQ?e=fu7p5R