Lab of IoT

Vincenzo Gallicchio MAT: 0522500642

December 2019

1 Introduction

An IoT application is usually made of many battery constrained smart sensing devices which are connected to the internet; these devices should be capable of collecting, smartly computing and sending data to other devices and applications of interest. This project is about the development of a typical and realistic IoT application: it is composed by a sensing micro-controller (such as Arduino) responsible for periodically monitoring the environment, an edge server which collects, filters and reacts to data coming from the sensors within the wireless sensor network and a cloud application accessible from many other ad-hoc applications that are interested in the implemented Wireless Sensor Network meaningful data. This document will be divided into four main sections: the first one will explain the functioning and operations of the entire system, the second will focus on the micro-controller physical circuit and software implementation, the third will talk about the implemented web server and the last one will be about the used Cloud Application.

2 Proposed System

In this system an Arduino WiFi Rev2 is used to collect data about environment temperature, humidity, gas and light. This "Thing" is also equipped with an infrared sensor capable of sensing movements in order to detect the presence of wild animals surrounding the micro-controller. The latter is in fact supposed to be deployed in a rural and possibly hostile environment. The Arduino will be battery powered and for this reason it will have some power consumption functionalities (e.g. a much bigger sending data period) which can be activated directly through an hardware access, in response to a low battery level or through an HTTP API call performed by the server. Given the battery life importance, especially in some situations in which the sensor is not easily accessible, Arduino software logic and circuit have been kept as simple and thin as possible. For this reason, it uses HTTP Rest protocol to communicate. There are though a few moments in which it performs a heavier data computation. In some cases Arduino must be able to perform some checks and react to some critical measurements as fast as possible; in fact, after it classifies an

alert situation it immediately calls the server software module responsible for handling that kind of problem. So, Arduino complexity is pretty plain , it collects and sends data to a server using a WiFi technology in a local LAN. The server components are a little bit more complex, they collects data from the sensor, visualize them in real time through a user friendly interface, save them in a main memory buffer, use a thread to periodically perform some aggregating computations on data contained in the buffer and only then empty the buffer (in order to avoid filling the whole memory), send the aggregated data to the cloud using the MQTT Protocol and manage critical situations such as a strangely high temperature, in this latter case the server will be able to contact the WSN manager directly through a telegram chat. The cloud will be responsible for receiving and indexing meaningful data to other interested applications, in the case of this project a really simple android app has been developed as a possible model of a subscriber.

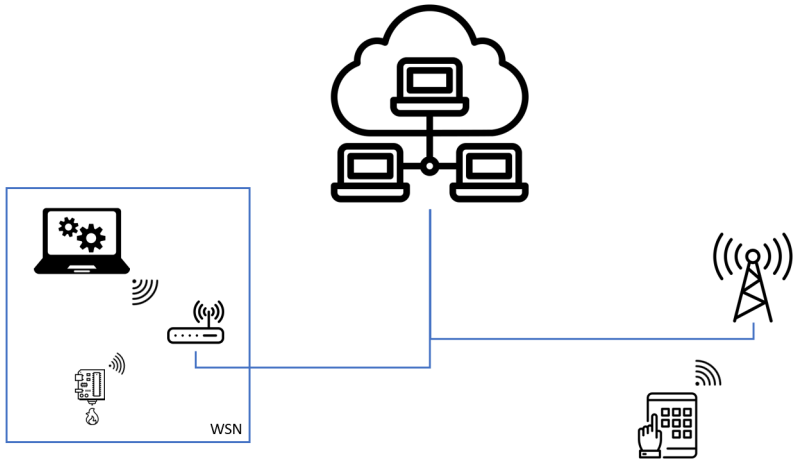

Figure 1: System

3 Arduino

The Arduino micro-controller is connected to a breadboard and all the sensing devices are plugged on there. Here' s a list of the utilized devices and their connections as an attempt to describe the circuit.

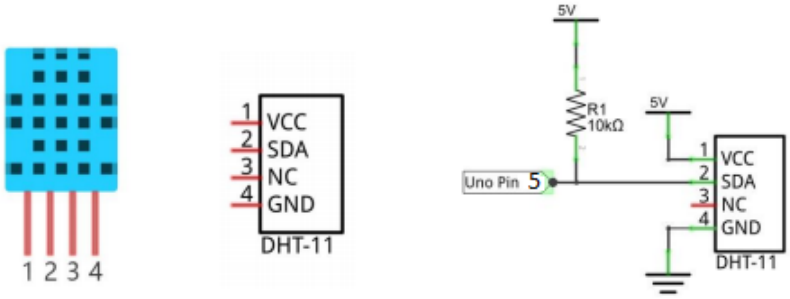

Figure 2: DHT11 for Humidity and Temperature

Figure 3: GAS Sensor

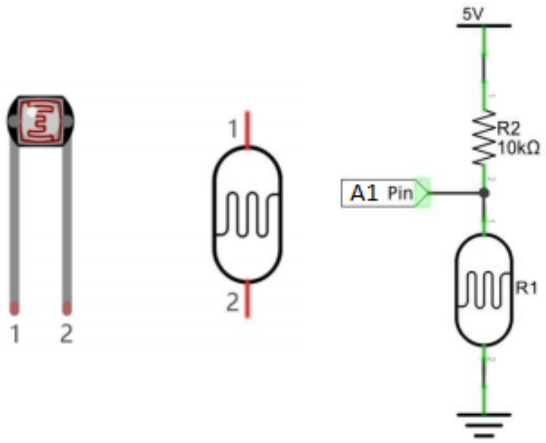

Figure 4: Photoresistor

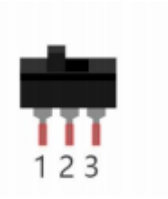

Figure 5: Analog switch used for battery power interruption

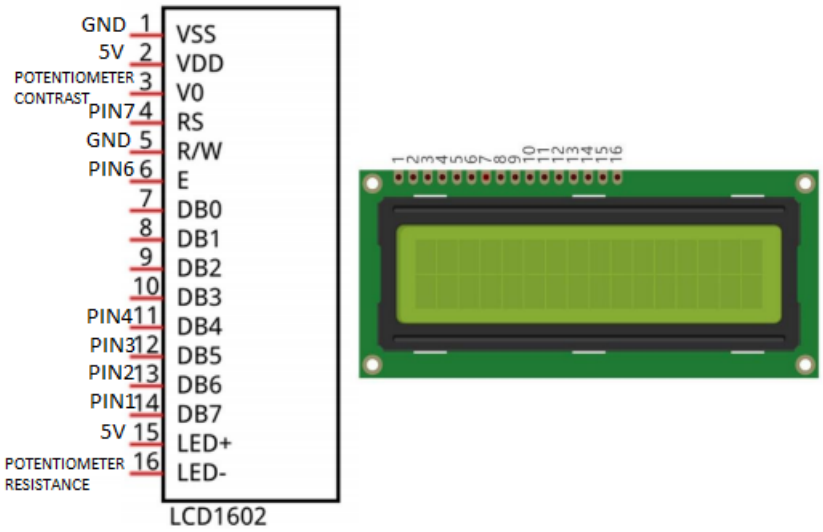

Figure 6: The lcd monitor used to display data

The LCD pin 16 was attached to the potentiometer resistance in order to control the brightness. In this way the screen can be turned on only when needed (e.g. some operators need to check data on the screen).

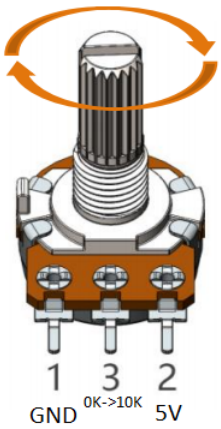

Figure 7: Potentiometer that manage resistance through rotation

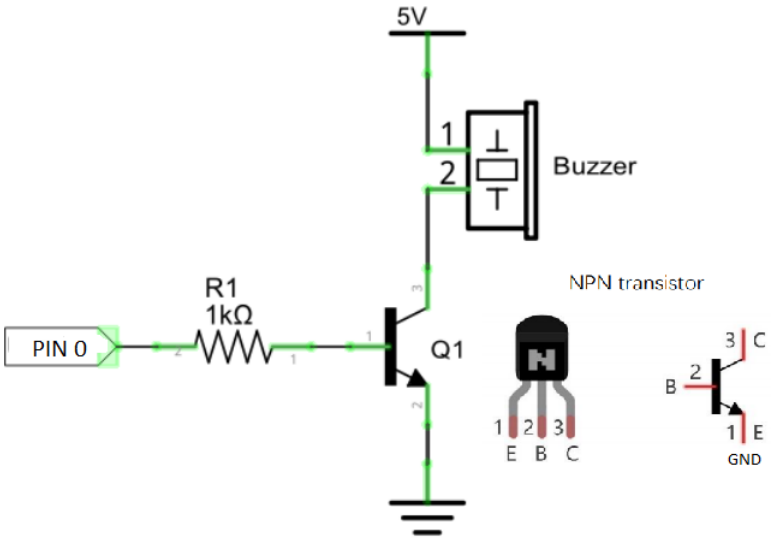

Figure 8: A utility buzzer to make routine switch on/off sound

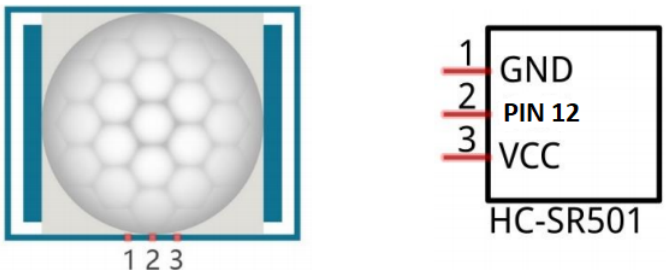

Figure 9: Infrared motion sensor

There's a portion of the circuit strictly linked to the sketch. This portion of the breadboard involves a button connected to the Arduino digital pin 11 and responsible for a processor interruption. When an interruption occurs, the processor will jump to the interrupt function to handle interruption events and then return to the main program.

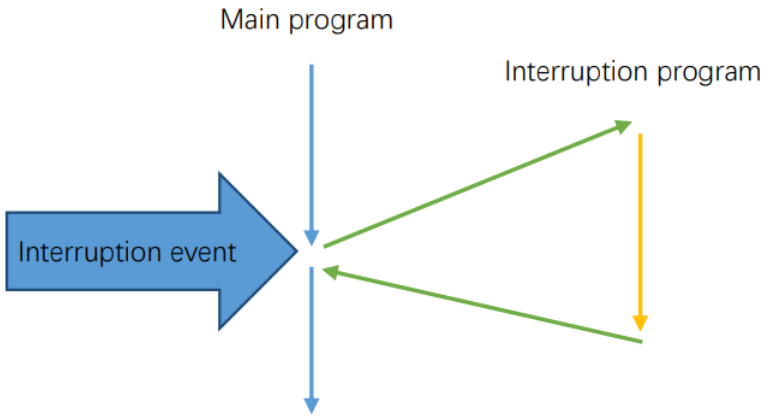

Figure 10: Interruption diagram

The interrupt function has been developed to manage power consumption. In fact, after the button is pressed, the micro-controller will execute a function that will modify the delay time in the loop function increasing it if the objective is to minimize battery consumption and decreasing it otherwise. When the interrupt function is executed for the first time delay time will be increased, this will reflect a much lighter usage of wi-fi that is extremely costly. Much more power saving functions exist, unfortunately they are not yet compatible with the Arduino used in this project. These functions should let use Arduino timed rest and wake up modes in order to preserve much more energy. The Arduino is alimented by a 9V battery which is monitored directly in the software through the analog pin A5, this makes the software capable of measuring the voltage and consequently detect and approximate the battery level. But there is a problem, while Vin pin support a maximum voltage of 12, arduino analog pins support a maximum voltage of 5 over which they go in overflow making the voltage estimation very difficult. That's why two equal resistors have been used in order to divide by two the physical value. (http://www.learningaboutelectronics.com/Articles/How-to-reduce-voltage-with-resistors.php) Then it becomes easy to reconstruct the real voltage, just multiply by 2 the digital one. Anyway, life battery estimation is possible thanks to Arduino current usage information which is approximately 50 mA (http://www.home-automation-community.com/arduino-low-power-how-to-run-atmega328p-for-a-year-on-coin-cell-battery) and thanks to the battery sheet which correlate the decreasing voltage over time with a life period. In alternative we still could estimate battery life in a proportional and mostly empirical manner knowing that Arduino won't work with a voltage lower than 3.5 V and testing a battery lifetime under usage conditions.

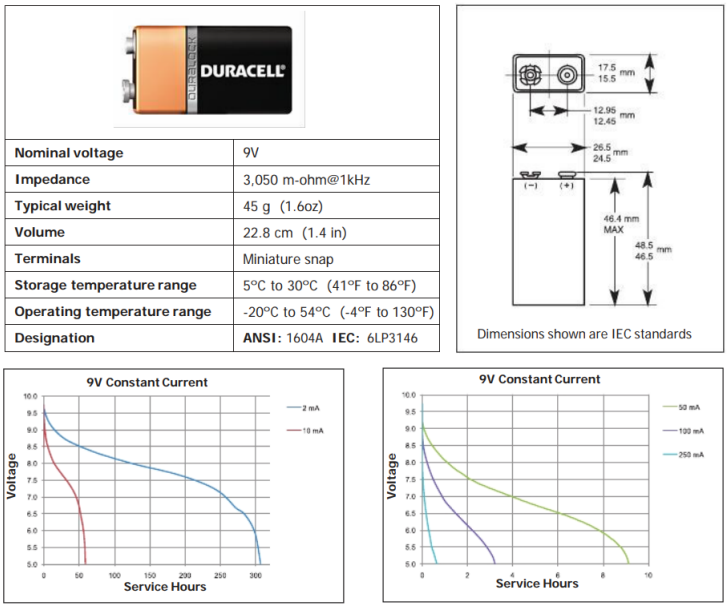

Figure 11: Battery document

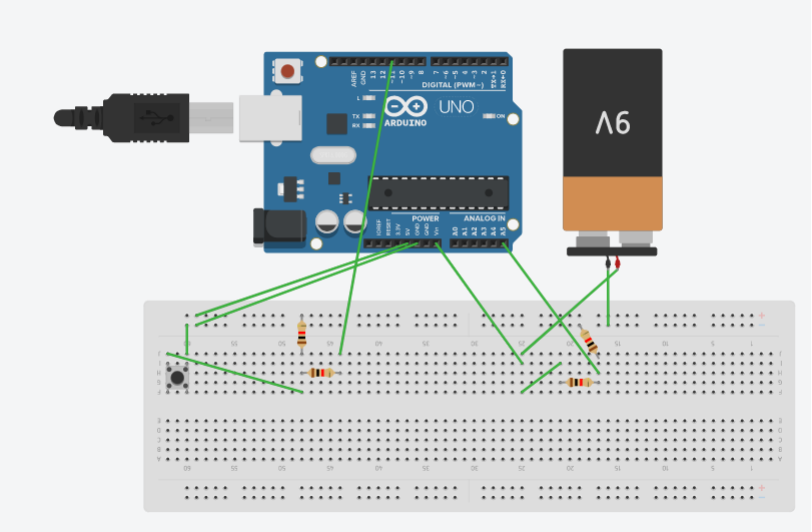

Figure 12: Arduino interrupt and battery circuit

Arduino has an initial "delay time" of two seconds at the end of which it re-executes the loop function body.

```
void setup(){
  Serial.begin(9600);
  connectToWiFi();
  //attaching interrupt to the button
  attachInterrupt(11, savePower, CHANGE);
  initializeDevices();

  //timed events schedulation
  timer.every(6000, getBatteryVoltage);
  timer.every(60000, updateTime);
  startWebServer();
}
```

Figure 13: Start up

```
void loop() {
  timer.tick(); // tick the timer
  switchGo = true; //to handle button double push

  humid = getHumidity();
  temp = getTemperature();
  gas = readGas();
  light = detectLight();
  animalDetected = (digitalRead(sensorPin) > 0);
  displayAll(); //display on the lcd meaningful data

  sendDataToEdgeServer(); //REST API HTTP GET
  if(!criticalPowerMode)
    checkForHTTPRequest();

  delay(timeToSense);
}
```

Figure 14: Loop function

In the loop function there's a boolean variable switchGo used to handle button double push wrong detection in the interrupt function. There are some functions for environment sensing like getHumidity(), getTemperature() and readGas(), the latter two have a particular structure, they in fact have to be able to detect warning values and react to them.

```
float readGas(){
  float gas = analogRead(A0);
  if(gas >80.0 && alarmPermission)
    makeGetCall(gas, "gas");
  return gas;
}
```

Figure 15: readGas function

If the Arduino senses an unusually high value it contacts an edge server component for management. The latter will contact the WSN manager on telegram.

```
void makeGetCall(float warningValue, String type){
  String string = (String)warningValue;
  Serial.println("\nStarting connection to server...");
  // if you get a connection, report back via serial:
  if (client.connect(serverIP, 8080)) {
    Serial.println("connected to server");
    // Make a HTTP request:
    client.println("GET /IoTEdgeNode/AlertModule?value=" + string + "&type=" + type + " HTTP/1.1");
    client.println(serverIPHost);
    client.println();
  }else
    Serial.println("no connection");

  numberOfMinutes = 0;
  alarmPermission = false;
}
```

Figure 16: Alert management function

Back to the loop method there are also two utilities for light and infrared detection stored in two variables to be sent to the edge server. The method displayAll() simply visualize some of the gathered data on the lcd screen plus an approximation of the battery life in hours and a dynamic battery icon for the power level, the latter is visualized thanks to other two methods: getBatteryVoltage() and batteryLevel() (for the battery picture).

The main function sendDataToEdgeServer() connects to the edge webserver module through HTTP 1.1 and periodically sends some data through a get call while checkForHTTPRequest keeps listening to the internal web server in order to detect any calls coming from WSN components.

```
void sendDataToEdgeServer(){
  Serial.println("\nStarting connection to server...");
  if (client.connect(serverIP, 8080)) {
    Serial.println("Data sent to server");
    // Make a HTTP request:
    client.println("GET /IoTEdgeNode/SensorDataProcessor?temp=" + temp + "&gas="+gas+
    "&humid=" + humid + "&light=" + light +"&infrared="+animalDetected+" HTTP/1.1");
    client.println(serverIPHost);
    client.println();
  }else
    Serial.println("no connection");
}
```

Figure 17: Main function used to send data

4 Edge server

The edge server can be any computer connected to the internet that hosts a web server and that is reachable through an ip address, normally in the context of IoT the edge device should be as close as possible to the sensor network for performance reasons. The selected system web server is apache tomcat, it hosts a java web application developed with java servlet technology. The latter's main components are called Servlets, they are http reachable processing units. For the main requirements of this project the web application uses several components : java classes, servlet classes and html pages. The first ones are needed to model and process data coming from a "thing" they are: SensorData and ProcessingUnit. One models the various types of sensed data into a class and the other possesses a buffer of SensorData instances and offers means to process and aggregate sets of data through some IoT common functions like mean(), max(), min().

```java
public class SensorData {
    private String SenderMacID;
    private String location;
    private float temperature;
    private float gas;
    private String light;
    private float humidity;
    private String infrared;

    public SensorData(float temperature, float gas, String light, float humidity,
            String infrared) {
        super();
        this.temperature = temperature;
        this.gas = gas;
        this.light = light;
        this.humidity = humidity;
        this.infrared = infrared;
    }

    public SensorData(float temperature, float humidity, float gas) {
        super();
        this.temperature = temperature;
        this.gas = gas;
        this.humidity = humidity;
    }

    public SensorData() {
        // TODO Auto-generated constructor stub
    }
```

Figure 18: SensorData class

```java
public class ProcessingUnit {
    private ArrayList<Object[]> measurementListToMine;

    public ProcessingUnit() {
        measurementListToMine = new ArrayList<Object[]>();
    }

    public void endMiningPhase() {
        measurementListToMine = new ArrayList<Object[]>();
    }

    public void addMeasurement(SensorData measurement) {
        Object[] timeInstance = new Object[2];
        timeInstance[0] = measurement;
        timeInstance[1] = new Date();
        this.measurementListToMine.add(timeInstance);
    }

    public SensorData getDataMean() {
        SensorData toProcess;
        SensorData minedData;
        float totalTemp = 0.0f;
        float totalUmid = 0.0f;
        float totalGas = 0.0f;
        int NoI = measurementListToMine.size();
        for(Object[] measurement : measurementListToMine) {
            //get data only
            toProcess = (SensorData)measurement[0];
            totalTemp += toProcess.getTemperature();
            totalUmid += toProcess.getHumidity();
```

Figure 19: ProcessingUnit class

In reality, there is another java class called Subscriber, it is just a MQTT utility class that exposes methods to interact with a Cloud MQTT Broker towards which the processed data will be sent. But arduino logically and directly interact with the servlet classes that are: AlertModule, SensorDataProcessor. The first one is called every time the microcontroller detects unexpected temperature and gas values. These values are passed to the AlertModule get function which will contact an API offered from the IFTTT platform which will contact the sensor manager directly on telegram.

```java
@WebServlet("/AlertModule")
public class AlertModule extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final Logger LOGGER = Logger.getLogger(AlertModule.class.getName());

    private String value;
    private String type;
    private String eventType;
    /**
     * @see HttpServlet#HttpServlet()
     */
    public AlertModule() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        if((type = request.getParameter("type"))!=null) {
            switch(type) {
                case "temperature":
                    this.eventType = "temperature";
                    this.parseTemperatureParameter(request);
                case "gas":
                    this.eventType = "gas";
                    this.parseGasParameter(request);
            }

            URL url = new URL("https://maker.ifttt.com/trigger/alert/with/key/icISbsI1EL3PgyhFElAKW-qOy3TQWjj1W66Ht6YhhVN?value1=" +
                    this.eventType+"&value2="+this.value);
            HttpURLConnection.setFollowRedirects(false);
            HttpURLConnection con = (HttpURLConnection) url.openConnection();

            LOGGER.info(con.getResponseCode()+"");
```

Figure 20: AlertModule servlet

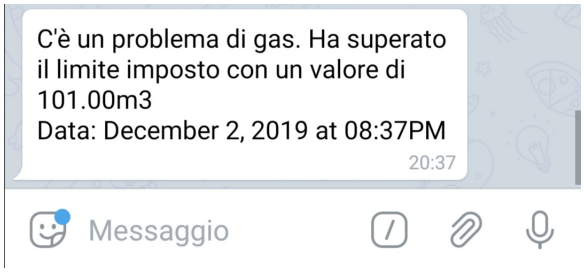

Figure 21: AlertMessage example

SensorDataProcessor is the core component of the server, it receives sensor data at a rate decided by the arduino, it uses received information to create a thread shared SensorData instance, it logs the information on a console and finally create a new global variable containing the sensor data, the latter is useful for web application modules that shares information.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    boolean proceed = this.parseParameters(request);
    if(proceed) {
        this.sD = new SensorData(temperature, gas, isLightOn, humid, infrared);
        synchronized(this) {
            this.pU.addMeasurement(sD);
        }
        LOGGER.info(this.sD.toString() + " - " + new Date());
        getServletContext().setAttribute( "realTimeMeasurement", new SensorData(temperature, gas, isLightOn, humid, infrared));
    }
}

private boolean parseParameters(HttpServletRequest request) {
    boolean proceed = true;
    if(request.getParameter("temp") != null){
        temperature = Float.parseFloat(request.getParameter("temp"));
    }else
        proceed = false;

    if(request.getParameter("humid") != null){
        humid = Float.parseFloat(request.getParameter("humid"));
    }else
        proceed = false;

    if(request.getParameter("gas") != null){
        gas = Float.parseFloat(request.getParameter("gas"));
    }else
        proceed = false;
```

Figure 22: SensorDataProcessor get

When this component is created for the first time it initializes a timed thread which will be responsible to use a ProcessingUnit class to process data and then send the latter to the cloud using an MQTT Publisher.

```
public void init(ServletConfig config) throws ServletException{
    super.init(config );
    try {
        this.subscriber = new Subscriber();
    } catch (MqttException e) {
        e.printStackTrace();
    }

    //Initialization
    SensorData sharedData =  new SensorData(0.0f, 0.0f, "Nessun dato", 0.0f, "Nessun Dato");
    this.sD =  new SensorData(0.0f, 0.0f, "Nessun dato", 0.0f, "Nessun Dato");

    //add a SensorData instance as a variable available through the entire web app
    getServletContext().setAttribute( "realTimeMeasurement", sharedData);

    pU = new ProcessingUnit();
    TimedUpload timerTask = new TimedUpload();
    //running timer task as daemon thread
    Timer timer = new Timer(true);
    //Every minute sends relevant data to Cloud
    timer.scheduleAtFixedRate(timerTask, 0, 1*60000);
}
```

Figure 23: SensorDataProcessor initialization

```
    private class TimedUpload extends TimerTask {
        @Override
        public void run() {
            try {
                completeUpload();
            } catch (MqttException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        private void completeUpload() throws MqttException {
            synchronized(this) {
                //Send data to cloud for other applications' needs
                subscriber.sendMessage("sensor/measurements", "MAX: " + pU.getDataMax().toString() + " - " +
                        "MEAN: " + pU.getDataMean().toString() + " - " + "MIN: " + pU.getDataMin().toString());

                System.out.println("MAX: " + pU.getDataMax().toString() + " - " +
                        "MEAN: " + pU.getDataMean().toString() + " - " + "MIN: " + pU.getDataMin().toString());
                pU.endMiningPhase();
            }
        }
    }
}
```

Figure 24: Timed thread inner class

Thus, SensorDataProcessor servlet updates two variables, one is shared between the timed thread and the main program, the other one is uploaded in a web application global space for a web page to access it and show data in real time through a user friendly interface.

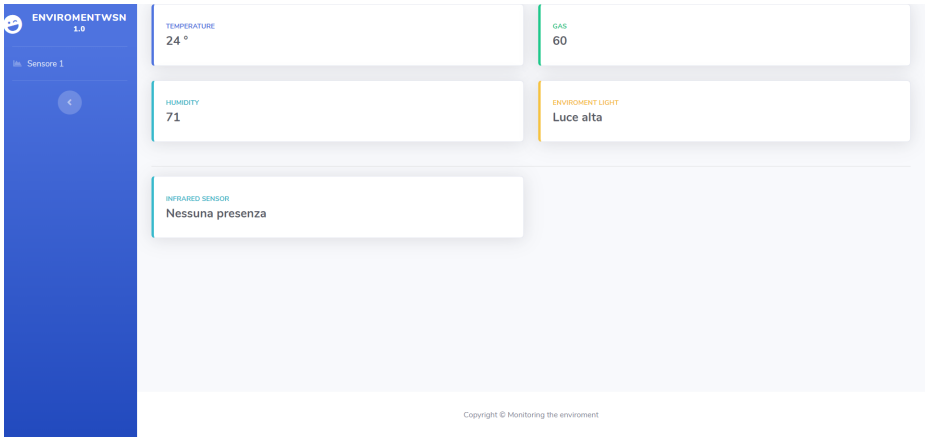

Figure 25: Real time data visualization

This web page uses a java script timed function to query the global variable every two second for information. So, the function uses ajax to asynchronly call a servlet called DataUpdater which will consult the global variable and return its content as a json structure. The interface also offerd a toggle button to use an Arduino API through HTTP get call. (It contacts the arduino web server).

```
$(document).ready(function() {
  // Thread for servlet called every two seconds
  setInterval(function(){

      $.ajax({
                  type : "POST",
                  url : "data_updater",
                  dataType : "html",
                  async: true,
                  success : function(msg) {
                      var msg = eval(msg);
                      var temperature = msg[0];
                      var humidity = msg[1];
                      var gas = msg[2];
                      var light = msg[3];
                      var infrared = msg[4];
                      $('#temperature').text(temperature);
                      $('#gas').text(gas);
                      $('#humidity').text(humidity);
                      $('#light').text(light);
                      $('#infrared').text(infrared);
                  },
                  error : function(err) {
                      console.log("error: "+err)
                  }
              });
  },2000);

});
```

Figure 26: Javascript function in update.js file

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    ServletContext servletContext = getServletContext();
    SensorData sD = (SensorData) servletContext.getAttribute( "realTimeMeasurement" );
    if(sD == null)
        sD = new SensorData(0.0f, 0.0f, "Nessun dato", 0.0f, "Nessun Dato");
    sendResponse(sD, response);
}

/**
 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // TODO Auto-generated method stub
    doGet(request, response);
}

private void sendResponse(SensorData sD, HttpServletResponse response) throws IOException {
    JsonArray jsonArray = new JsonArray();

    jsonArray.add(sD.getTemperature());
    jsonArray.add(sD.getHumidity());
    jsonArray.add(sD.getGas());
    jsonArray.add('"'+sD.getLight()+'"');
    jsonArray.add('"'+sD.getInfrared()+'"');
    PrintWriter out = response.getWriter();
    out.print(jsonArray);
    out.flush();
    out.close();
}
```

Figure 27: DataUpdater servlet

5 CLOUD

The used cloud web application is CloudMQTT, a hosted message broker for the Internet of Things. It receives filtered data from the edge server and saves them on the selected topic, then it notifies all the applications interested and subscribed to that topic. For a demonstration purpose a simple android app connected to the internet has been developed. It updates data on the screen each time the MQTT receives a new message. The android app only needs to

subscribe to a Cloud service through internet, it has nothing to do with the wireless sensor network.

6 Conclusions and Future works

This project wants to be a model of a typical IoT product but it isn't complete yet. In fact all the security requirements are missing and the system isn't capable of dynamically adding new sensors to the network. (If we wanted to build a much broader measurement network all the device should be added manually to the server) Future development phases will comprehend dynamic sensor self-configuration in the sensor network, security protocols implementation, authentication and validation of sensors.