Critical Systems Lab - MESCC
Water Pumping Automated System

Ricardo Mendes 1201779

Arthur Gerbelli 1220201

ISEP, January 2024, **Third Delivery**

Contents

1 Introduction

As this is the last report, we will try to summarize all the previous analyses, with the main focus being to achieve a conclusion and not only repeat the specificity of the last reports.

Although the following chapters are tightly defined, the report should be read as a whole.

2 Requirement Specifications

2.1 System Requirements

There was no change in the system requirements. Below, we listed all the identified system requirements and a comment about its implementation on the prototype.

SR-1 .1: While the water level is above the minimum level, WPS shall have a pump working. Implemented

.2: When the water level is below the minimum level, WPS shall have all pumps stopped. Implemented

.3: If the water level is above the maximum level, then the WPS shall trigger an alarm at the Remote Status Station (RSS). Implemented

.4: A second pump shall be turned on only when the water level is above 2/3 the maximum water level. Implemented

.5: When only one pump is available, the maximum water level shall be reduced to 2/3. Implemented

.6: If the readings of the sensor are uneven, the system shall choose the worst case scenario, following the table below: Implemented

sensor #2 \ sensor #1	**0**	**1**	**2**	**3**	**4**
0	-	1	2	3	4
1	1	1	1	1	4
2	2	1	2	2	4
3	3	1	2	3	4
4	4	4	4	4	4

1: below min; **2**: above min; **3**: above med; **4**: above max.

0: no connection to the sensor - if both sensors are unavailable, the alarm shall be triggered.

SR-2 .1: The status of all WPS shall be displayed on all RSS. Partially Implemented: the hardware available was not enough to add a second WPS and RSS.

.2: If the alarm is ON, the button in the RSS shall only disable it. Implemented

.3: The RSS shall have an independent power supply from the WPS. Not Implemented

.4: The alarm on the RSS shall have an independent power supply from the RSS itself and from the WPS. Not Implemented

SR-3 .1: The status of all WPS shall be visible on a web page. Implemented

SR-4 .1: To improve the system´s communication reliability, a cluster of 2(two) MQTT brokers shall be deployed. Not Implemented

2.2 Hazard Analysis

We added one more entry to the hazard analysis, H-9. Below, we added some comments about the implementation of the identified mitigations in the prototype.

H-1: see SR-1.5

H-2:
- **Description:** Both pumps stopped working.
- **Mitigation:** Trigger alarm. Implemented

H-3:
- **Description:** A pump doesn't turn OFF when the water level in bellow minimum.
- **Mitigation:** Trigger alarm. Not Implemented

H-4: see SR-1.6

H-5: see SR-2.3 and SR-2.4

H-6:
- **Description:** RSS are not getting information from WPS.
- **Mitigation:** Implement a cluter of MQTT Brokers or remove this single point of failure by adopting DDS. Trigger alarm. Partially Implemented: the alarm is triggered

H-7:
- **Description:** RSS stops working.
- **Mitigation:** Have redundancy by having multiple RSS and each one displaying all statuses from all WPS. Not Implemented

H-8:
- **Description:** Control Unit stops working.
- **Mitigation:** Implement redundancy by having a cluster of nodes running the Control Unit. If the number of nodes is 3 we can implement a voting system and run the same process with the same input in parallel. This would improve the system's fault tolerance. Not Implemented

H-9:
- **Description:** Rapid wear of the first water pump.
- **Mitigation:** Distribute evenly the work done by the pumps. This can be done, by choosing a random pump when one is need first. Not Implemented

2.3 System Structure and Traceability

This last chapter tries to map the implementation of the prototype to the model developed during the previous two deliveries.

During the implementation of the system, we noticed that the Web Server is a subsystem independent of the RSS.

Although, both are ways to visualize the WPS status, they have very different objectives and levels of criticality. The RSS must alert the maintenance team; the web

version is, in our understanding, only a nice thing to have. And most of all, we can remove one service without affecting the other. Because of this, we made some changes to the System Structure diagram, as it is possible to visualize below.

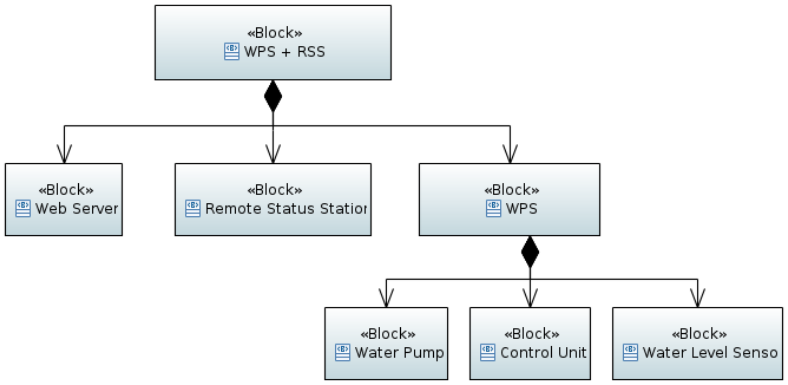

Figure 1: System Structure diagram

Looking at the prototype's deployment diagram 2, we can clearly see that, the output of the modeling, traces nicely to its implementation. The grayed parts of the diagram don't make up the prototype.

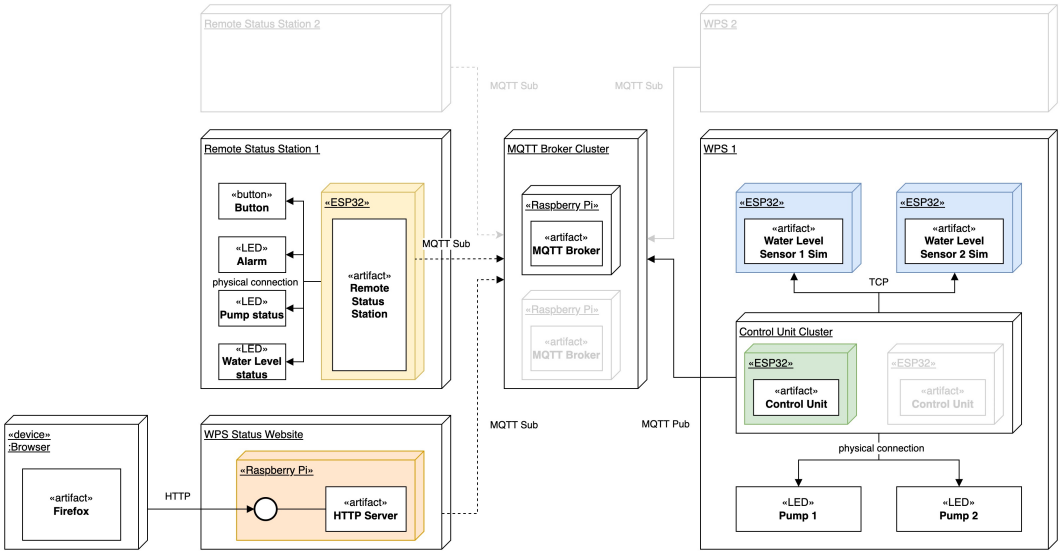

Figure 2: Prototype's deployment diagram

3 Implementation

3.1 CCSYA - Assembly

The following code snippet in implemented on the Sensor Sim, as it was in the previous delivery. The only change was done on one of the comments that had a typo.

```
1  .global getMask
2
3  getMask:    entry a1, 48
4              movi a5, 0          # init increment
5              mov a6, a2          # save inputed water level
6              movi a2, 1          # init return value
7
8  loop:
9              addi a5, a5, 1      # increment by one
10             beq a5, a6, end     # branch to 'end' if equal
11             slli a2, a2, 1      # left shif by 1, i.e. 0001 -> 0010
12             j loop              # jump to loop
13
14 end:
15             retw.n
```

As an input, the code receives the position of the LED that needs to turn ON. So to ensure that only one of the LEDs is turned ON, we start with the value 1 (see line 6). This value in a byte is represented by `'0000 0001'`. From now on, we only shift the leftmost bit, one position at a time, until we reach the selected position.

The output is an 8-bit integer. If we have as output the decimal number `'16'`, and convert it to binary `´0000 1000´`, we can see that we want to turn ON the LED in the 4th position. The following code implements this logic.

```
// more code
  mask = getMask(level); // ASM code
  implementMask(mask);
// more code

void implementMask(int8_t mask)
{
  int8_t n_bit = 0;
  while (n_bit < MAX_LEVELS) {
    if (mask & 0x01) {
      digitalWrite(STATES[n_bit], HIGH);
    }
    else {
      digitalWrite(STATES[n_bit], LOW);
    }
    n_bit++;
    mask = mask >> 1;
  }
}
```

3.2 RTAES - Concurrency and Real-Time Scheduling

3.2.1 Implementation of the Tasks and Concurrency control

As a reminder, the following tasks are being run in a ESP-WROOM-32 on a single core:

- **Task 1**: Retrieve data from sensors (TCP Client);
- **Task 2**: Process data and give instructions to pumps;
- **Task 3**: Publish WPS status to MQTT broker (MQTT client).

The creation of the tasks was achieved by using the **xTaskCreatePinnedToCore(...)** [1] function from the FreeRTOS API. The function **xTaskCreate(...)** was also available, but we wanted to ensure that the proposed tasks would run on the same core.

```
xTaskCreatePinnedToCore(
                    requestSensorData,   /* Task function. */
                    "Task1",     /* name of task. */
                    10000,       /* Stack size of task */
                    NULL,        /* parameter of the task */
                    1,           /* priority of the task */
                    &Task1,      /* Task handle to keep track of created task */
                    0);          /* pin task to core 0 */
```

The access control to the critical section was done thanks to the use of a single semaphore: `xSemaphoreHandle xMutex;`.

3.2.2 Real Time Scheduling

It must be openly said that this part was the trickiest one.

To check in practical terms, the results of the schedulability calculation done in the previous report, some changes needed to be added to the code.

Although the execution times for Task 2 look to be very close to the ones achieved in the calculations, the same doesn't apply for Task 1 and Task 3. This is partly the fault of communication latency. Both tasks need to request or send information using a wireless connection and the TCP/IP protocol.

Another problem was the method implemented to track the task's execution time. The granularity wasn't enough, especially for Task 2.

As a practical solution, we chose to update the values from the last report in a proportionally way.

Table 1: Theoretical values (left) and implemented values (right) in *ms*:

Task	Ci	Ti
1	60	100
2	25	200
3	35	250

Task	Ci	Ti
1	1200	2000
2	500	4000
3	700	5000

The values on the right side were achieved by running the code in different situations. A closer look, shows that we increased all values **x 20**.

To track the execution time we added the following code to each task:

```
for (;;) {
    unsigned long start_time = millis();
    unsigned long finish_time;
    unsigned long duration;

    // Code -------------------

    finish_time = millis();
    duration = finish_time - start_time;
    Serial.print("TASK N - Execution Time[ms]: ");
    Serial.println(duration);
}
```

And here are some of the outputs:

```
TASK 3 - Duration[ms]: 40
TASK 1 - Duration[ms]: 917
TASK 2 - Duration[ms]: 475
TASK 1 - Duration[ms]: 658
TASK 1 - Duration[ms]: 754
TASK 3 - Duration[ms]: 770  < above WCET
TASK 2 - Duration[ms]: 476
TASK 1 - Duration[ms]: 649
TASK 1 - Duration[ms]: 799
TASK 3 - Duration[ms]: 29
TASK 2 - Duration[ms]: 477
TASK 1 - Duration[ms]: 176
TASK 1 - Duration[ms]: 997
TASK 2 - Duration[ms]: 481
TASK 1 - Duration[ms]: 187
TASK 3 - Duration[ms]: 157
TASK 1 - Duration[ms]: 990
TASK 2 - Duration[ms]: 475
TASK 1 - Duration[ms]: 1036
```

It is important to state, that given the short execution time of Task 2, we needed to add a sleep function to increase it and so bring it closer to the values on the right side table 1.

Also, to ensure that the tasks didn't extended beyond its deadline, we added the following code:

```
long next_release = 5000 - duration; // 5 seconds deadline for Task 3
if (next_release > 0) {
    vTaskDelay(next_release / portTICK_PERIOD_MS);
```

```
} else {
    Serial.println("TRASK 3 - FAILED Deadline !!!");
}
```

In some cases, Task 1 failed the deadline by a couple of milliseconds. This was because of the defined timeout value of 1 second for each TCP request (one request per sensor). When the two requests reached a timeout, we already had a 2-second execution time. Unfortunately, the library used for the TCP connection has a lower limit of 1 second for the timeout.

We consciously chose not to fix this issue, mostly because a fix would mean a more extended increase of the values, beyond the x20, and when we have at least one sensor responding, the execution time are very close to the ones in the table 1.

```
TASK 1 - Connection failed: 172.20.10.13
TASK 1 - Connection failed: 172.20.10.5
TASK 1 - Duration[ms]: 2009
TASK 1 - FAILED Deadline !!!
```

3.2.3 Concurrency

In terms of concurrency and, as proposed in the last report, we identified as a critical section the read and write of a data structure that encapsulates the WPS status.

```
typedef struct{
  volatile bool alert;
  volatile uint8_t id;
  volatile uint8_t curr_water_level;
  volatile uint8_t curr_pump1_status;
  volatile uint8_t curr_pump2_status;
  volatile uint8_t curr_pump1_state;
  volatile uint8_t curr_pump2_state;
} WPS;
```

The changes to the data structure WPS were made by accessing functions instead of accessing the values directly.

```
void set_curr_pump2_status(uint8_t value) {
  xSemaphoreTake(xMutex, portMAX_DELAY);
  _wps.curr_pump2_status = value;
  xSemaphoreGive(xMutex);
}
```

The status reads, were specially controlled during the publication of the WPS status to the MQTT broker.

```
String getWpsStatus(){
    xSemaphoreTake(xMutex, portMAX_DELAY);
    String message = String(_wps.alert) + "," + String(_wps.id) + "," + String(_wps.curr
    xSemaphoreGive(xMutex);
    return message;
}
```

3.3 COMCS - Communication

3.3.1 TCP pull strategy

We have each sensor acting as a TCP server and the Control Unit connecting as a client to both. The direct TCP request was chosen here due to the simplicity of the network, which has up to two Control Units and two sensors. The MQTT would have an advantage in that it can ensure that the Control Unit gets the message from the sensor, but since we are pulling from the sensors in very short periods, it was not necessary to implement it here.

3.3.2 MQTT communication

We have the MQTT broker running on a host machine for prototype purposes. This machine is connected to the same WLAN, and its address is used by the publishers and subscribers. The Control Unit publishes to this broker the data from the WPS every 5 seconds. As subscribers, we have the RSS and the Web Server. We do not adopt any QoS now, but we may try at least QoS1 [2] depending on the severity of those 5 seconds without the alarm received from the maintenance team.

3.3.3 Reconnection mechanisms

- **The Control Unit**: Before trying to read from the sensors, check the Wi-Fi connectivity and try to reconnect. If the connection with the sensors fails, it logs to the serial console. Before publishing to the MQTT Broker, it validates the Wi-Fi connection and tries to reconnect if it is not connected. It also tries to connect to the MQTT Broker every time before publishing.
- **The RSS**: It checks for a Wi-Fi connection, and tries to establish the connection in case it drops. It also monitors if it does not receive more messages and raises the alarm on RSS.
- **The WebServer**: We implemented a re-connection mechanism when the MQTT connection drops. This mechanism increases the delay for every re-connection attempt until a maximum time is defined. The number of attempts is also configurable.

3.3.4 Connection lifetime

We do not keep the TCP connection between the Control Unit and sensors, nor between the Control Unit and the MQTT Broker open, to avoid having hanging connections that are stuck. Instead, we connect and disconnect on every task execution span. We have dedicated tasks for publishing to the MQTT Broker and for reading from the sensors via TCP; they are scheduled by FreeRTOS implementation and on each task execution, we connect, execute the job and disconnect.

3.4 Prototype

3.4.1 Overview

An overview of what was implemented in the prototype can be found in diagram 2.

The cluster of MQTT brokers, or Control Units, were not implemented, as were the second WPS and the second RSS. However, the system was developed to accommodate the future implementation of all the missing parts.

With a cluster of MQTT brokers, the producer (Control Units) and the consumers (Web server and RSS) would need to connect to all the available nodes and jump from one to the other if there were some type of failure.

The cluster of Control Units would be implemented by running both systems in parallel but only outputting one of the results. If the computations of both nodes differ, the implementation of a type of voting system needs to be implemented.

A second WPS is easily solved because for each system there must be one MQTT topic. This means one queue per WPS and a consumer subscription to each one of them.

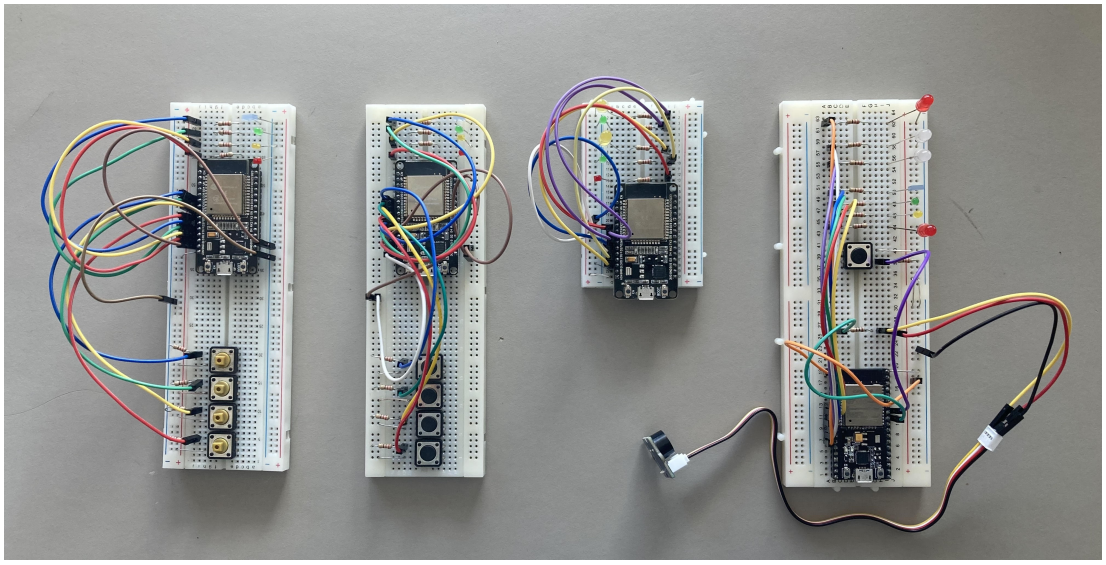

Figure 3: from left to right: Sensor 1, Sensor 2, Control Unit + Pumps, RSS

Below is a representation of the topological layout of the network that will be presented.

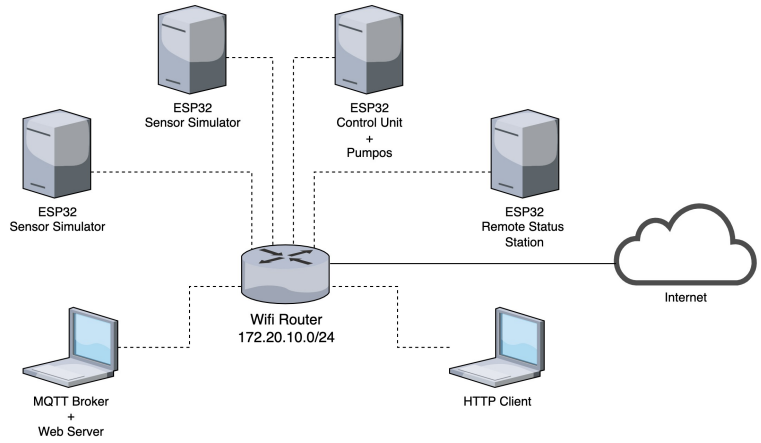

Figure 4: Network diagram

3.4.2 Control Unit

The WPS will publish a message with its status, every 5 seconds.

Looking at the right tabel 1, we can also notice that we are able to do 2 requests to the sensor (Task 1) for each computation of the systems behavior (Task 2). This gives us the possibility to recover from a communication failure when doing one of the two requests.

The main behavior of the system is summurized by all the requirements from SR-1.

Used hardware:

- ESP-32
- 1x alert LED
- 2x pump status LEDs
- 2x pump state LEDs (able or not to do some work)

3.4.3 RSS

The MQTT client in the RSS will, every second, query the broker for a new message. If no new message is available in the last 15 seconds, an alarm will be triggered. This means that the Control Unit, which publishes to the broker every 5 seconds, has three chances to recover from some communication failure.

Although we are simulating only one WPS, subscribing to a second topic with the status of the second WPS is easily done.

Used hardware:

- ESP-32
- 1x buzzer module

- 1x button to stop buzzer
- 1x alert LED
- 2x pump status LEDs
- 4x water level LEDs

3.4.4 Sensor

Used hardware:

- ESP-32
- 4x water level buttons
- 4x water level LEDs

3.4.5 Web Server

The Web Server follows the same logic as the RSS. It consumes the same topics made available by the broker.

We ponder another approach, where the Web server would query the RSS for the WPS status. This solution would give more responsibilities to a critical system (RSS) and make all the implementation more complex and unnecessary.

It is running also in a host machine connected to the WLAN for this prototype. The implementation was made in Python using the module http.server, it is very easy and simple to use and we could have a running server set in a minute. We created a separate index.html file to hold the web page format, and the webserver will read and write to this file.

For the MQTT, we import the module paho.mqtt, which has straightforward methods to set and subscribe to the MQTT messages. We also implemented a re-connection mechanism, described in item x.xx. We have an unsolved bug here which is after a re-connection occurs, the webserver does not receive more messages. We left this indicated on the code.

As an additional tool, we used a module called BeautifulSoup for parsing the HTML page, and for searching for the tag ids that we want to update the values.

4 Team Work

Because of thoughtful analysis during the first delivery, the solution analysis gave us a list of very decoupled subsystems. This enabled us to split the work in a more organized way, as soon as the communication method was defined.

The drafting of the reports was done in cooperation, giving us the opportunity to share some particular knowledge of each subsystem.

In a more concrete way, we create a common repository on Bitbucket, where each main change in the code base is done by creating a Pull Request. This made both of us aware of the work as a whole.

In conclusion, the teamwork ran smoothly, mainly because of an initial good analysis of the system requirements, and also because of the focus on the academic objectives of this work.

References

[1] Espressif FreeRTOS API: https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/api-reference/system/freertos.html

[2] MQTT QoS: https://cedalo.com/blog/understanding-mqtt-qos/