MQTT

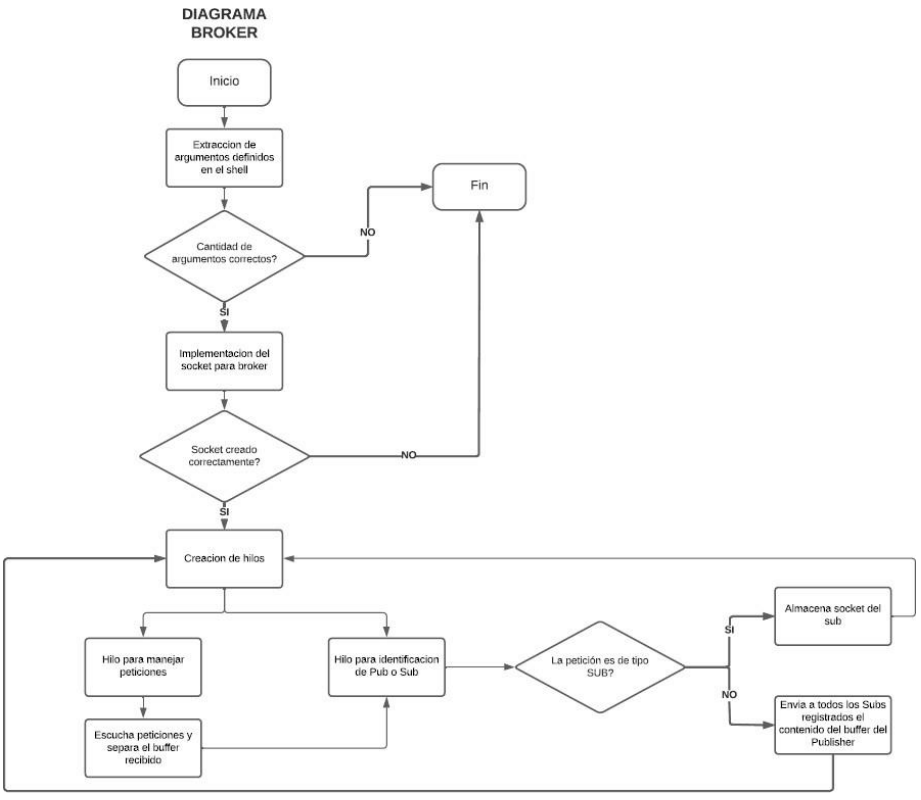
Publisher wokwi:

https://wokwi.com/projects/350106104190992980

Diagrams

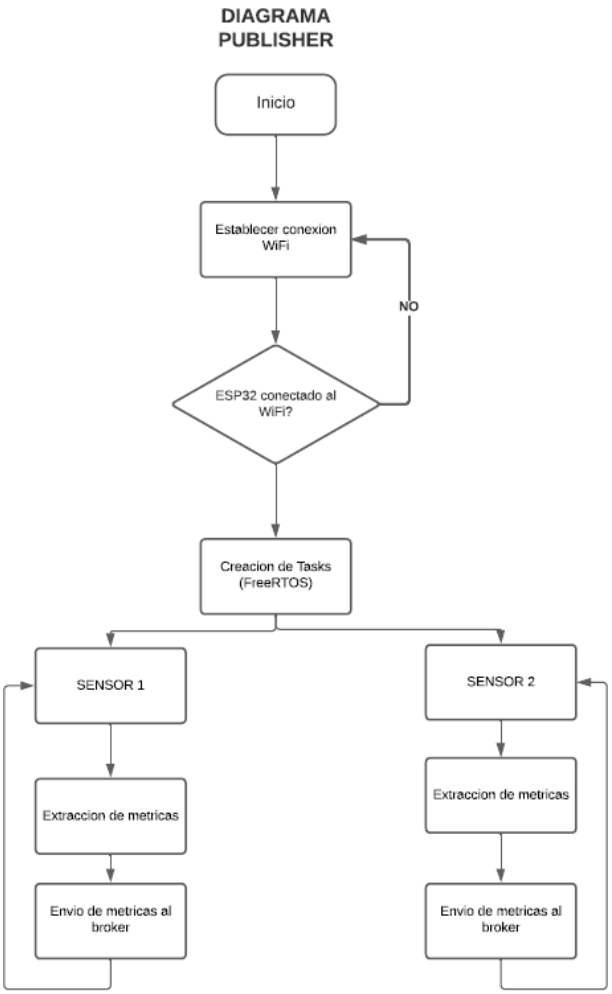
Broker

The broker starts by extracting arguments from the shell, and then creates the socket for the broker. Afterwards, inside an infinite while loop, one thread is created to receive requests, and another to determine whether the request corresponds to a Publisher or Subscriber. Due to the implemented signals, the analysis thread must wait for the request thread to finish execution and extract its buffer. In this way, the analysis thread can identify from the buffer whether it is a Publisher or a Subscriber. If it turns out to be a Subscriber, its socket will be stored in an array; if it is a Publisher, the buffer content will be sent to all Subscribers registered with the broker.



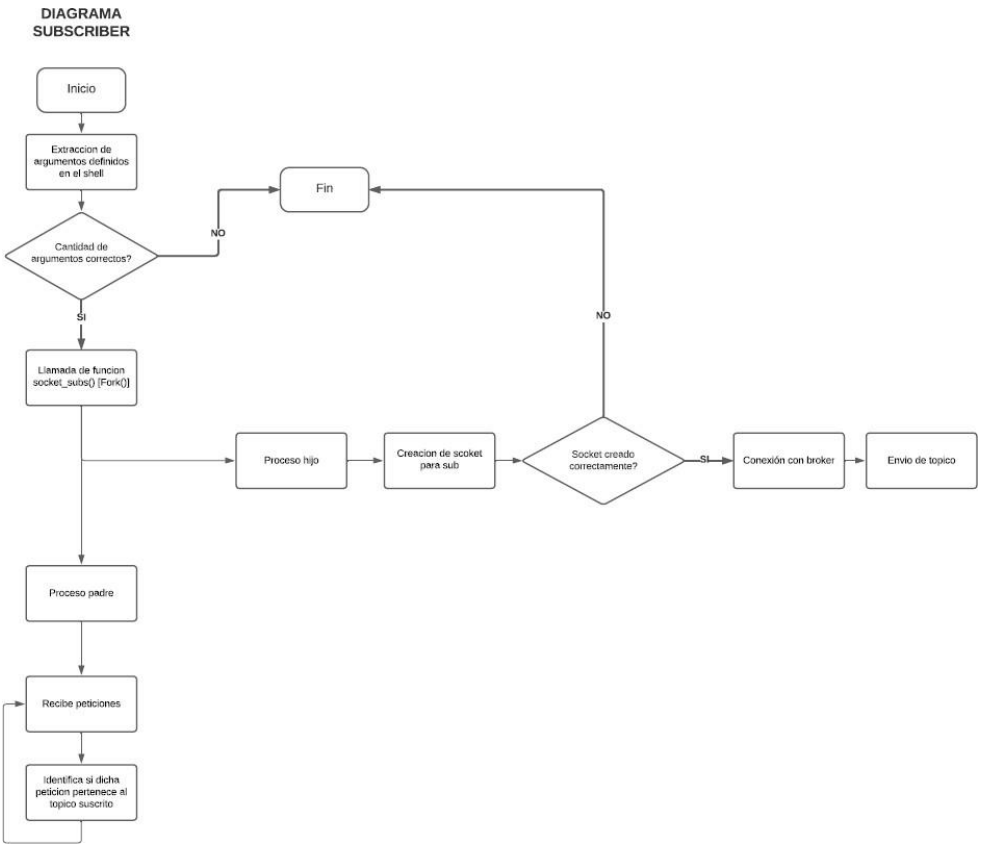
Publisher

The Publisher begins by establishing the WiFi connection of the ESP32 device. If the connection is not successful, it will retry infinitely. After that, two tasks are created (one for each sensor), allowing both sensors to simultaneously extract their metrics (humidity and temperature), which are then sent continuously to the broker.



Subscriber

The Subscriber starts by extracting arguments from the shell. It then executes the socket_subs() function, which handles forking the program so that the child process can create the socket and establish a connection with the broker to send the topic it has subscribed to. Once the child process finishes, the parent process continues and takes over receiving requests from the broker in an infinite loop, analyzing whether the content of the buffer matches the topic to which it subscribed.



Software Requirements

1. **Implementation of Publisher, Broker, and Subscriber**

We begin by implementing the necessary libraries, variables, mutexes, semaphores, and a function called BufferSeparator(), which helps us split the buffer using a delimiter (/). This is because the buffer content follows the format:

- **Publisher:** topic/message
- **Subscriber:** topic

This way, we can determine whether the buffer content belongs to a Subscriber or a Publisher.

Implementation of functions that will be passed as parameters to our threads. Hilo_socket() will be responsible for receiving requests, reading their buffer, and splitting it. Hilo_mqtt() will handle the management between Publishers and Subscribers. These functions will be explained in more detail in the threads and processes section of the document.

4o

```c
//hilo socket
void *hilo_socket(void *args){
        int new_socket = *(int *)args;
        if(new_socket < 0) {
                perror("Error de conexion con el socket del cliente");
                close(new_socket);
                exit(1);
        }
        memset(buffer,0,sizeof(buffer));
        if(read(new_socket, buffer, 1024) < 0) {
                if(strlen(buffer) == 0){
                        close(new_socket);
                }
                perror("Error read");
                close(new_socket);
                exit(1);
        }
        printf("\n");
        strcpy(bufferCopy,buffer);
        bufferSeparator(bufferCopy, bufferSplit, "/");

        sem_wait(&s_mensaje);
        while(bufferSplit[cantidad] != NULL){
                cantidad++;
        }

        sem_post(&s_socket);
        return 0;
}
```

```c
//hilo mqtt
void *hilo_mqtt(void *args) {
        sem_wait(&s_socket);
        int new_socket = *(int *)args;

        if(cantidad == 1) {
                printf("---NUEVO SUSCRIBER---\n");
                pthread_mutex_lock(&mutex);//mutex lock
                socketsSubs[contadorSubs] = new_socket;
                contadorSubs++;
                pthread_mutex_unlock(&mutex);//mutex unlock
                printf("Nuevo suscriber con tópico: %s\n", bufferSplit[0]);
        }else {
                printf("---NUEVO PUBLISHER---\n");
                printf("Publisher envio al tópico: %s el Mensaje: %s\n", bufferSp

                for(int i = 0; i < contadorSubs; i++) {
                        if(write(socketsSubs[i], buffer, strlen(buffer)) < 0) {
                                printf("Error publisher al enviar mensaje\n");
                        }
                }
        }

  return 0;
}
```

A main() function, which will receive the IP address and port through the shell. These will be used to create the socket.

```c
int main(int argc, char **argv) {
        if(argc != 5) {
                perror("Formato: ./broker -i ip -p puerto");
                return -1;
        }

        char opt;
        int puerto;
        char *ip;
        int socketMqtt;
        struct sockaddr_in address;

        while((opt = getopt(argc, argv, "i:p:")) != -1) {
                switch(opt) {
                        case 'i':
                         ip = optarg;
                         break;
                        case 'p':
                         puerto = atoi(optarg);
                         break;
                        default:
                         break;
                }
        }
```

Similarly, the corresponding mutexes and semaphores will be initialized here, which will be discussed later. Additionally, we can see that the socket creation is established.

```c
    }
    //inicializacion mutex -n- semaforos
    pthread_mutex_init(&mutex,NULL);
    sem_init(&s_socket,0,0);
    sem_init(&s_mensaje,0,0);

    //Crear socket
    socketMqtt = socket(AF_INET, SOCK_STREAM, 0);

    if(socketMqtt < 0) {
            perror("Error al crear el socket");
            exit(1);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr(ip);
    address.sin_port = htons(puerto);

    if(bind(socketMqtt, (struct sockaddr *)&address, sizeof(address)) < 0) {
            perror("Error bind");
            close(socketMqtt);
            exit(1);
    }
```

Finally, the threads are created within an infinite while loop.

```c
    if(listen(socketMqtt, 128) < 0) {
            perror("Error listen");
            close(socketMqtt);
            exit(1);
    }
    printf("BROKER ENCENDIDO\n");
    printf("...Escuchando...\n");
    while(1) {
            int new_socket = accept(socketMqtt, NULL, NULL);
            pthread_t hiloSocket;
            pthread_t hiloMqtt;
            pthread_create(&hiloSocket,NULL,hilo_socket, &new_socket);
            pthread_create(&hiloMqtt, NULL, hilo_mqtt, &new_socket);
    }

return 0;
```

Subscriber

We begin by implementing the necessary libraries and declaring variables. Likewise, we implement our bufferSeparator() function, which will separate the content of the buffer sent by the broker.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <errno.h>

//variables
int socketSubs;
struct sockaddr_in address;
int puerto;
char *ip;
char *topico;
char buffer[1024] = {0};
char *bufferSplit[2];
int error = 0;

//bufferSeparator
void bufferSeparator(char *buffer, char *bufferSplit[], char *sep) {
        char *mensaje;
        int i = 0;
        mensaje = strtok(buffer, sep);

        while(mensaje != NULL) {
                bufferSplit[i] = mensaje;
                mensaje = strtok(NULL, sep);
                i++;
        }
}
```

A function called socket_subs() was also implemented, which essentially handles creating the socket for the subscriber and sending the topic with which it registered to the broker. This is done by implementing a child process using fork(), which will be explained later in the threads and processes section.

```c
//proceso socket
void socket_subs(){
        pid_t pid;
        pid = fork();
        if(pid == 0){
                socketSubs = socket(AF_INET, SOCK_STREAM, 0);
                if(socketSubs < 0) {
                        error = 1;
                        perror("Error creando socket para subscriber");
                        exit(1);
                }
                printf("SUBSCRIBER ENCENDIDO\n");

                address.sin_family = AF_INET;
                address.sin_port = htons(puerto);
                address.sin_addr.s_addr = inet_addr(ip);

                if(connect(socketSubs, (struct sockaddr *)&address, sizeof(address)) < 0){
                        error = 1;
                        perror("Error de conexion");
                        close(socketSubs);
                        exit(1);
                }
                printf("Enviando topico al broker...\n");

                if(write(socketSubs, topico, strlen(topico)) < 0) {
                        error = 1;
                        perror("Error al enviar topico al Broker");
                        close(socketSubs);
                        exit(1);
                }
                printf("Subscriber suscrito al BROKER con el tópico: %s\n", topico);

        }else{
                wait(NULL);
                if(error = 1){
                        exit(1);
                }
        }
}
```

At the same time, a main function was defined, whose purpose is to extract the arguments related to the IP address to connect to, the port, and the topic to which the subscriber will be subscribed. After this, the socket_subs() function is called, which performs the corresponding socket implementation along with sending the topic to the broker.

```c
int main(int argc, char **argv) {

        if(argc != 7) {
                perror("Formato: ./subscriber -i direcciónIP -p puerto -t topico");
                return -1;
        }

        char opt;

        while((opt = getopt(argc, argv, "i:p:t:")) != -1) {
                switch(opt) {
                        case 'i':
                         ip = optarg;
                         break;
                        case 'p':
                         puerto = atoi(optarg);
                         break;
                        case 't':
                         topico = optarg;
                         break;
                        default:
                         break;
                }
        }
        //Proceso para creacion de socket y envio al broker
        socket_subs();
```

Finally, once the child process has finished, an infinite while loop will be established to continuously listen for requests sent by the broker. This allows the subscriber to analyze whether the received message corresponds to the topic it subscribed to.

```c
    //recibir del broker
    while(1) {
            if(read(socketSubs, buffer, 1024) < 0) {
                    perror("Error read");
                    close(socketSubs);
                    exit(1);
            }
            bufferSeparator(buffer, bufferSplit, "/");

            if(strcmp(bufferSplit[0], topico) == 0) {
                    printf("Nuevo mensaje recibido al topico suscrito: %s\n", bufferSplit[1]);
            }

            memset(buffer, 0, sizeof(buffer));
    }
```

Publisher

The publisher was implemented using an ESP32 board, where FreeRTOS was used to send metrics from two sensors in parallel through a socket.
We started by implementing the necessary libraries and defining variables, such as the identifier for each sensor, the topics, the HOST and port to which it will connect to send metrics, and arrays to store the data that will later be sent.

Then, functions were implemented to establish a WiFi connection for the ESP32.

```
30  // Variables
31  WiFiClient espClient;
32
33  char *charTemp;
34  char *charHumedad;
35
36  void initWiFi(void);
37  void reconnectWiFi(void);
38
39  /* Inicializa wifi */
40  void initWiFi(void)
41  {
42    delay(10);
43    Serial.println("------Conexión WI-FI------");
44    Serial.print("Conectando con la red: ");
45    Serial.println(SSID);
46    Serial.println("Espere");
47
48    reconnectWiFi();
49  }
50


53  void reconnectWiFi(void)
54  {
55
56    if (WiFi.status() == WL_CONNECTED)
57      return;
58
59    WiFi.begin(SSID, PASSWORD);
60
61    while (WiFi.status() != WL_CONNECTED) {
62      delay(100);
63      Serial.print(".");
64    }
65
66    Serial.println();
67    Serial.print("Conectado con exito a la red");
68    Serial.print(SSID);
69    Serial.println(" IP obtenido: ");
70    Serial.println(WiFi.localIP());
71  }
72
```

Additionally, a function was defined to be passed as a parameter when creating the tasks for each sensor (in this case, 2). This function is essentially responsible for extracting the metrics provided by the sensor (temperature and humidity), and then sending them to the broker.

```
79  void dht22(void *pvParameter) {
80
81    int dhtPin = *((int*)pvParameter);
82    DHTesp dhtSensor;
83    dhtSensor.setup(dhtPin, DHTesp::DHT22);
84
85
86    while (1 ) {
87      TempAndHumidity  data = dhtSensor.getTempAndHumidity();
88
89      Serial.println("Temp: " + String(data.temperature, 2) + "°C");
90      Serial.println("Humidity: " + String(data.humidity, 1) + "%");
91
92      //obtencion de temp y humedad
93      delay(200);
94      sprintf(strTemperature, "%.2fC", data.temperature);
95      sprintf(strHumidity, "%.2f", data.humidity);
96      //formato de envio topico/contenido
97      String temperaturaPublish = String(TOPIC_PUBLISH_TEMPERATURE) + "/" + Stri
98      String humedadPublish = String(TOPIC_PUBLISH_HUMIDITY) + "/" + String(strH
99
100     //conversion char* para enviar al socket
101     charTemp = (char*)temperaturaPublish.c_str();
102     charHumedad = (char*)humedadPublish.c_str();
103     //envio de humedad
104     client.connect(HOST,PORT);
105     client.write(charHumedad);
106     //envio de temperatura
107     client.connect(HOST,PORT);
108     client.write(charTemp);
109
110     vTaskDelay(1000 /portTICK_PERIOD_MS);
111   }
```

Finally, the initWifi() function was called to connect the device to the internet, and the tasks (one for each sensor) were initialized to begin parallelizing the sensors and sending metrics.

```
110 void setup() {
111   Serial.begin(115200);
112   initWiFi();
113
114   xTaskCreate(&dht22, "DHT22v1", 1024*4, (void*)&DHT_PIN, 5, NULL);
115   xTaskCreate(&dht22, "DHT22v2", 1024*4, (void*)&DHT_PIN2, 5, NULL);
116
117 }
118
119
120 void loop() {
121
122 }
```

1. Creation of Child Processes using fork / Thread Implementation

For the broker section, a multithreaded solution was chosen, in which two threads are defined: hiloSocket and hiloMqtt.
We can also see the creation of these threads, where each one is assigned a specific function passed as a parameter.

```
        }
        printf("BROKER ENCENDIDO\n");
        printf("...Escuchando...\n");
        while(1) {
                int new_socket = accept(socketMqtt, NULL, NULL);
                pthread_t hiloSocket;
                pthread_t hiloMqtt;
                pthread_create(&hiloSocket,NULL,hilo_socket, &new_socket);
                pthread_create(&hiloMqtt, NULL, hilo_mqtt, &new_socket);
        }

return 0;
```

The hilo_socket(void *args) function is responsible for creating a socket to establish connections with clients and listen for their requests. Through this function, after reading the socket's content, we can store that content in our buffer and then make a copy of it to use the bufferSeparator() function. This allows us to separate the buffer content and store it in our bufferSplit array by topics and message (in the case of a Publisher).

```
//hilo socket
void *hilo_socket(void *args){
        int new_socket = *(int *)args;
        if(new_socket < 0) {
                perror("Error de conexion con el socket del cliente");
                close(new_socket);
                exit(1);
        }
        memset(buffer,0,sizeof(buffer));
        if(read(new_socket, buffer, 1024) < 0) {
                if(strlen(buffer) == 0){
                        close(new_socket);
                }
                perror("Error read");
                close(new_socket);
                exit(1);
        }
        printf("\n");
        strcpy(bufferCopy,buffer);
        bufferSeparator(bufferCopy, bufferSplit, "/");

        sem_wait(&s_mensaje);
        while(bufferSplit[cantidad] != NULL){
                cantidad++;
        }

        sem_post(&s_socket);
        return 0;
}
```

The hilo_mqtt(void *args) function is essentially responsible for determining whether the buffer content corresponds to a Subscriber or a Publisher. It does this by analyzing the number of elements in bufferSplit. If bufferSplit contains only one element, it is a Subscriber, since it will only have the topic. Otherwise, it is a Publisher, as its content will consist of both a topic and a message.

```
//hilo mqtt
void *hilo_mqtt(void *args) {
        sem_wait(&s_socket);
        int new_socket = *(int *)args;

        if(cantidad == 1) {
                printf("---NUEVO SUSCRIBER---\n");
                pthread_mutex_lock(&mutex);//mutex lock
                socketsSubs[contadorSubs] = new_socket;
                contadorSubs++;
                pthread_mutex_unlock(&mutex);//mutex unlock
                printf("Nuevo suscriber con tópico: %s\n", bufferSplit[0]);
        }else {
                printf("---NUEVO PUBLISHER---\n");
                printf("Publisher envio al tópico: %s el Mensaje: %s\n", bufferSplit[0], bufferSpli$

                for(int i = 0; i < contadorSubs; i++) {
                        if(write(socketsSubs[i], buffer, strlen(buffer)) < 0) {
                                printf("Error publisher al enviar mensaje\n");
                        }
                }
        }

  return 0;
}
```

Process Creation for Subscriber:

On the other hand, in the Subscriber section, a child process was implemented using the fork() function. This process is called within the socket_subs() function, which is responsible for defining the Subscriber's socket and sending the specified topic to the broker. Additionally, we can see that the parent process waits for the child to finish before continuing and listening to the various requests sent through the broker.

```
    }
    //Proceso para creacion de socket y envio al broker
    socket_subs();

    //recibir del broker
    while(1) {
            if(read(socketSubs, buffer, 1024) < 0) {
                    perror("Error read");
                    close(socketSubs);
                    exit(1);
```

```
//proceso socket
void socket_subs(){
        pid_t pid;
        pid = fork();
        if(pid == 0){
                socketSubs = socket(AF_INET, SOCK_STREAM, 0);
                if(socketSubs < 0) {
                        error = 1;
                        perror("Error creando socket para subscriber");
                        exit(1);
                }
                printf("SUBSCRIBER ENCENDIDO\n");

                address.sin_family = AF_INET;
                address.sin_port = htons(puerto);
                address.sin_addr.s_addr = inet_addr(ip);

                if(connect(socketSubs, (struct sockaddr *)&address, sizeof(address)) < 0){
                        error = 1;
                        perror("Error de conexion");
                        close(socketSubs);
                        exit(1);
                }
                printf("Enviando topico al broker...\n");

                if(write(socketSubs, topico, strlen(topico)) < 0) {
                        error = 1;
                        perror("Error al enviar topico al Broker");
                        close(socketSubs);
                        exit(1);
                }
                printf("Subscriber suscrito al BROKER con el tópico: %s\n", topico);

        }else{
                wait(NULL);
                if(error = 1){
                        exit(1);
                }
        }
}
```

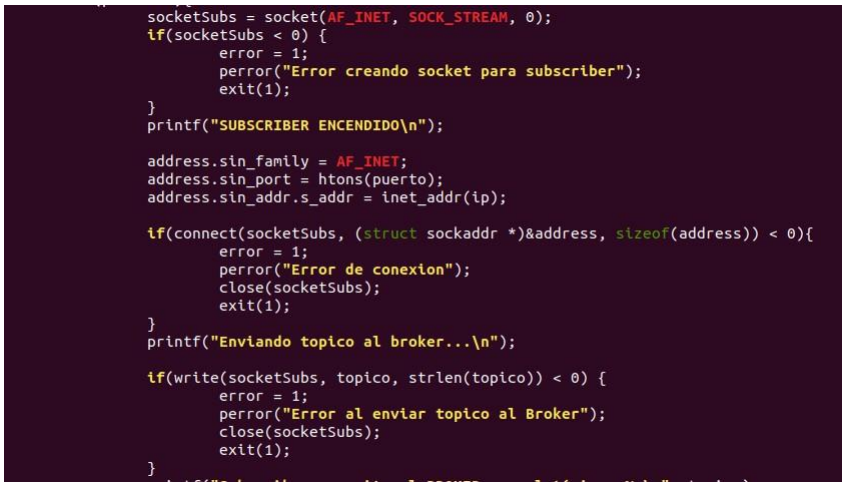
2. IPC Implementation (pipes, sockets, etc.)

For this project, sockets were used to establish the different connections between the broker, the publisher, and the subscriber.

Socket broker:

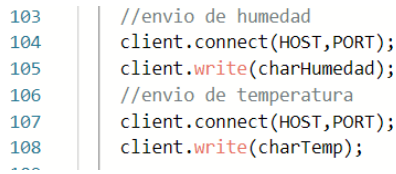
```
socketMqtt = socket(AF_INET, SOCK_STREAM, 0);

if(socketMqtt < 0) {
        perror("Error al crear el socket");
        exit(1);
}

address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr(ip);
address.sin_port = htons(puerto);

if(bind(socketMqtt, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("Error bind");
        close(socketMqtt);
        exit(1);
}

if(listen(socketMqtt, 128) < 0) {
        perror("Error listen");
        close(socketMqtt);
        exit(1);
}
printf("BROKER ENCENDIDO\n");
printf("...Escuchando...\n");
```

Socket subs:



```
        socketSubs = socket(AF_INET, SOCK_STREAM, 0);
        if(socketSubs < 0) {
                error = 1;
                perror("Error creando socket para subscriber");
                exit(1);
        }
        printf("SUBSCRIBER ENCENDIDO\n");

        address.sin_family = AF_INET;
        address.sin_port = htons(puerto);
        address.sin_addr.s_addr = inet_addr(ip);

        if(connect(socketSubs, (struct sockaddr *)&address, sizeof(address)) < 0){
                error = 1;
                perror("Error de conexion");
                close(socketSubs);
                exit(1);
        }
        printf("Enviando topico al broker...\n");

        if(write(socketSubs, topico, strlen(topico)) < 0) {
                error = 1;
                perror("Error al enviar topico al Broker");
                close(socketSubs);
                exit(1);
        }
```

Socket pub:



```
103   //envio de humedad
104   client.connect(HOST,PORT);
105   client.write(charHumedad);
106   //envio de temperatura
107   client.connect(HOST,PORT);
108   client.write(charTemp);
```

3. Implementation of Mutexes and Semaphores

For the broker section, a mutex and two semaphores were defined to handle synchronization and variable protection within our threads.
We can observe the declaration of a mutex named mutex, and two semaphores called s_socket and s_mensaje.

```
//mutex -n- semaforos
pthread_mutex_t mutex;
sem_t s_socket;
sem_t s_mensaje;
```

Below, we can see their initialization within the main function.

```
    }
    //inicializacion mutex -n- semaforos
    pthread_mutex_init(&mutex,NULL);
    sem_init(&s_socket,0,0);
    sem_init(&s_mensaje,0,0);
```

We can see that there is a function called bufferSeparator, which is responsible for receiving the buffer and splitting it using a delimiter—in this case, "/".

Inside this function, we observe a post(s_mensaje) signal, which notifies the socket thread when the buffer has been completely separated.

```
//BufferSeparator
void bufferSeparator(char *buffer, char *bufferSplit[], char *sep) {
        char *mensaje;
        int i= 0;
        mensaje = strtok(buffer, sep);
        memset(bufferSplit,0,2*sizeof(bufferSplit[0]));
        while(mensaje != NULL) {
                bufferSplit[i] = mensaje;
                mensaje = strtok(NULL, sep);
                i++;
        }
        sem_post(&s_mensaje);
}
```

Within the socket thread, we can observe a wait(s_mensaje) signal, which waits for the buffer to be fully separated before analyzing it and determining the number of elements it contains (1 element – subscriber, 2 elements – publisher).
Additionally, after determining the number of elements, the thread sends a post(s_socket) signal to notify the mqtt thread to proceed with analyzing whether the request refers to a publisher or a subscriber.
Finally, we can see that this thread uses a mutex to protect the storage of subscriber sockets in our array.

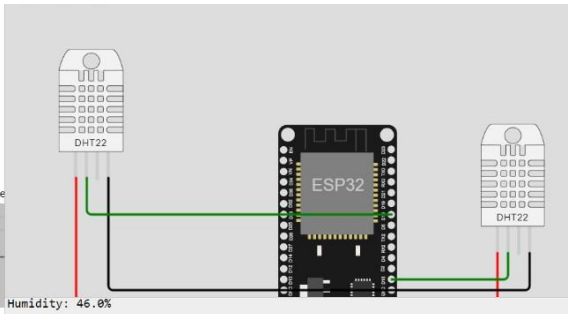
```
//hilo socket
void *hilo_socket(void *args){
        int new_socket = *(int *)args;
        if(new_socket < 0) {
                perror("Error de conexion con el socket del cliente");
                close(new_socket);
                exit(1);
        }
        memset(buffer,0,sizeof(buffer));
        if(read(new_socket, buffer, 1024) < 0) {
                if(strlen(buffer) == 0){
                        close(new_socket);
                }
                perror("Error read");
                close(new_socket);
                exit(1);
        }
        printf("\n");
        strcpy(bufferCopy,buffer);
        bufferSeparator(bufferCopy, bufferSplit, "/");

        sem_wait(&s_mensaje);
        while(bufferSplit[cantidad] != NULL){
                cantidad++;
        }

        sem_post(&s_socket);
        return 0;
```

```
//hilo mqtt
void *hilo_mqtt(void *args) {
        sem_wait(&s_socket);
        int new_socket = *(int *)args;

        if(cantidad == 1) {
                printf("---NUEVO SUSCRIBER---\n");
                pthread_mutex_lock(&mutex);//mutex lock
                socketsSubs[contadorSubs] = new_socket;
                contadorSubs++;
                pthread_mutex_unlock(&mutex);//mutex unlock
                printf("Nuevo suscriber con tópico: %s\n", bufferSplit[0]);
        }else {
                printf("---NUEVO PUBLISHER---\n");
                printf("Publisher envio al tópico: %s el Mensaje: %s\n", bufferSp

                for(int i = 0; i < contadorSubs; i++) {
                        if(write(socketsSubs[i], buffer, strlen(buffer)) < 0) {
                                printf("Error publisher al enviar mensaje\n");
                        }
                }
        }

  return 0;
}
```

4. Implementation of Sending and Receiving Metrics from Multiple Sensors

Sending metrics from multiple sensors through the Publisher: We can observe how the Publisher is responsible for sending multiple metrics (temperature and humidity) through an infinite while loop, constantly sending these values to the broker.

```
while (1 ) {
  TempAndHumidity  data = dhtSensor.getTempAndHumidity();

  Serial.println("Temp: " + String(data.temperature, 2) + "°C");
  Serial.println("Humidity: " + String(data.humidity, 1) + "%");

  //obtencion de temp y humedad
  delay(200);
  sprintf(strTemperature, "%.2fC", data.temperature);
  sprintf(strHumidity, "%.2f", data.humidity);
  //formato de envio topico/contenido
  String temperaturaPublish = String(TOPIC_PUBLISH_TEMPERATURE) + "/" + String(strTempe
  String humedadPublish = String(TOPIC_PUBLISH_HUMIDITY) + "/" + String(strHumidity);

  //conversion char* para enviar al socket
  charTemp = (char*)temperaturaPublish.c_str();
  charHumedad = (char*)humedadPublish.c_str();
  //envio de humedad
  client.connect(HOST,PORT);
  client.write(charHumedad);
  //envio de temperatura
  client.connect(HOST,PORT);
  client.write(charTemp);

  vTaskDelay(1000 /portTICK_PERIOD_MS);
}
```




```
Humidity: 46.0%
Temp: 10.60°C
Humidity: 50.5%
Temp: 24.00°C
Humidity: 46.0%
Temp: 10.60°C
Humidity: 50.5%
```

Receiving multiple metrics through the Subscriber: We can see an infinite while loop where the Subscriber is responsible for receiving multiple metrics sent by the Publisher through the broker. Once the buffer is received from the broker, the Subscriber checks whether the message corresponds to the topic it subscribed to.

```
//recibir del broker
while(1) {
        if(read(socketSubs, buffer, 1024) < 0) {
                perror("Error read");
                close(socketSubs);
                exit(1);
        }
        bufferSeparator(buffer, bufferSplit, "/");

        if(strcmp(bufferSplit[0], topico) == 0) {
                printf("Nuevo mensaje recibido al topico suscrito: %s\n", bufferSplit[1]);
        }

        memset(buffer, 0, sizeof(buffer));
}
```

5. Management of Multiple Publishers and Subscribers

The management of multiple Publishers and Subscribers is handled by the broker. In this case, we define a variable: an array of sockets related to the Subscribers, which essentially stores the multiple sockets that connect through the broker.

```
//variables
int socketsSubs[21];
int contadorSubs;
char *bufferSplit[2] = {};
char buffer[1024] = {0};
char bufferCopy[1024] = {0};
int cantidad = 0;
```

We can observe that if the buffer contains only one element (a topic), it is identified as a Subscriber. In that case, the Subscriber's socket is stored in the socketsSubs[] array, allowing us to manage multiple Subscribers.
On the other hand, if the message content contains more than one element (topic and message), it is identified as a Publisher. In this case, the broker immediately sends the buffer to all registered Subscribers, so each Subscriber can check whether the content matches the topic they subscribed to—thus enabling the management of multiple Publishers.

```
//hilo mqtt
void *hilo_mqtt(void *args) {
        sem_wait(&s_socket);
        int new_socket = *(int *)args;

        if(cantidad == 1) {
                printf("---NUEVO SUSCRIBER---\n");
                pthread_mutex_lock(&mutex);//mutex lock
                socketsSubs[contadorSubs] = new_socket;
                contadorSubs++;
                pthread_mutex_unlock(&mutex);//mutex unlock
                printf("Nuevo suscriber con tópico: %s\n", bufferSplit[0]);
        }else {
                printf("---NUEVO PUBLISHER---\n");
                printf("Publisher envio al tópico: %s el Mensaje: %s\n", bufferSplit[0], bufferSpli$

                for(int i = 0; i < contadorSubs; i++) {
                        if(write(socketsSubs[i], buffer, strlen(buffer)) < 0) {
                                printf("Error publisher al enviar mensaje\n");
                        }
                }
        }

  return 0;
}
```

6. FreeRTOS

An ESP32 was used for the Publisher, where FreeRTOS was utilized to implement multiple sensors (in this case, two). Two tasks were created (one for each sensor), with each task responsible for obtaining the metrics (temperature and humidity) from its assigned sensor and sending those metrics to the broker in parallel.

```
5   #include "DHTesp.h"
6   #include <WiFi.h>
7   #include <WiFiClient.h>
8   #include "freertos/FreeRTOS.h"
9   #include "freertos/task.h"
```

We can observe how two tasks are created for each sensor (2), where a function named dht22 is passed as a parameter, along with the sensor's identifier and a priority level, which is the same for both tasks.

```
116  void setup() {
117    Serial.begin(115200);
118    initWiFi();
119
120    xTaskCreate(&dht22, "DHT22v1", 1024*4, (void*)&DHT_PIN, 5, NULL);
121    xTaskCreate(&dht22, "DHT22v2", 1024*4, (void*)&DHT_PIN2, 5, NULL);
122
123  }
124
```

This function extracts the sensor's identifier and retrieves its corresponding metrics (humidity and temperature), then sends them through the broker. Thanks to the creation of these tasks, we can perform the transmission of metrics in parallel.

```
79  void dht22(void *pvParameter) {
80
81    int dhtPin = *((int*)pvParameter);
82    DHTesp dhtSensor;
83    dhtSensor.setup(dhtPin, DHTesp::DHT22);
84
85
86    while (1 ) {
87      TempAndHumidity  data = dhtSensor.getTempAndHumidity();
88
89      Serial.println("Temp: " + String(data.temperature, 2) + "°C");
90      Serial.println("Humidity: " + String(data.humidity, 1) + "%");
91
92      //obtencion de temp y humedad
93      delay(200);
94      sprintf(strTemperature, "%.2fC", data.temperature);
95      sprintf(strHumidity, "%.2f", data.humidity);
96      //formato de envio topico/contenido
97      String temperaturaPublish = String(TOPIC_PUBLISH_TEMPERATURE) + "/" + Stri
98      String humedadPublish = String(TOPIC_PUBLISH_HUMIDITY) + "/" + String(strH
99
100     //conversion char* para enviar al socket
101     charTemp = (char*)temperaturaPublish.c_str();
102     charHumedad = (char*)humedadPublish.c_str();
103     //envio de humedad
104     client.connect(HOST,PORT);
105     client.write(charHumedad);
106     //envio de temperatura
107     client.connect(HOST,PORT);
108     client.write(charTemp);
109
110     vTaskDelay(1000 /portTICK_PERIOD_MS);
111   }
```

(12) Implement a log that records the connection established between the broker and the clients, as well as the messages exchanged.

Broker

To implement this log, we will create a function called makeLog(char *registro). This function receives a character string corresponding to the messages printed to the console, such as the broker's startup, the connections established between the broker and clients, along with their respective messages, etc.
Inside makeLog(), the first step is to call the fork() function to create a child process, ensuring that the parent process is not affected when execvp() is executed—since execvp() immediately replaces the currently running process (in this case, the broker). Using execvp(), we can run the logger command, which allows us to write the logs into the system log (syslog), and then apply the -s flag so that the output is printed via STDERR, which is then redirected to the file log.txt.

```
char registro[255]; // LOG

//mutex -n- semaforos
pthread_mutex_t mutex;
sem_t s_socket;
sem_t s_mensaje;

//log
void makeLog(char *registro) {
        pid_t pid;
        pid = fork();
        if(pid == 0) {
                char comando[1024];
                sprintf(comando, "logger --no-act -t Broker -s %s >> log.txt 2>&1", registro);
                char *argumentos[] = {"sh","-c", comando, NULL};

                execvp("/bin/sh", argumentos);
        }
        else {
                wait(NULL);
        }
}
```

We can observe that within the hilo_mqtt function—which is responsible for handling Publishers and Subscribers—before printing their established connections and messages to the console, these are stored in the registro variable using the sprintf() function. Afterwards, the makeLog() function is called, passing this content as a parameter to record it in the log.txt file.

```
//hilo mqtt
void *hilo_mqtt(void *args) {
        sem_wait(&s_socket);
        int new_socket = *(int *)args;

        if(cantidad == 1) {
                sprintf(registro,"***NUEVO SUSCRIBER***");
                printf("%s\n", registro);
                makeLog(registro);// log
                pthread_mutex_lock(&mutex);//mutex lock
                socketsSubs[contadorSubs] = new_socket;
                contadorSubs++;
                pthread_mutex_unlock(&mutex);//mutex unlock
                sprintf(registro,"Nuevo suscriber con tópico: %s", bufferSplit[0]);
                printf("%s\n", registro);
                makeLog(registro); //log
        }else {
                sprintf(registro,"***NUEVO PUBLISHER***");
                printf("%s\n", registro);
                makeLog(registro);//log
                sprintf(registro,"Publisher envio al tópico: %s el Mensaje: %s", bufferSplit[0], bufferSplit[1])
                printf("%s\n", registro);
                makeLog(registro);//log

                for(int i = 0; i < contadorSubs; i++) {
                        if(write(socketsSubs[i], buffer, strlen(buffer)) < 0) {
                                sprintf(registro, "Error publisher al enviar mensaje");
                                printf("%s\n", registro);
                                makeLog(registro); //log
```

Another example can be seen within the program's main function, where at the start of its execution, the startup messages are also stored in the log record and added to the log using the makeLog() function.

```
    sprintf(registro,"BROKER ENCENDIDO");
    printf("%s\n", registro);
    makeLog(registro);
    sprintf(registro,"...Escuchando...");
    printf("%s\n", registro);
    makeLog(registro);
    while(1) {
            int new_socket = accept(socketMqtt, NULL, NULL);
            pthread_t hiloSocket;
            pthread_t hiloMqtt;
            pthread_create(&hiloSocket,NULL,hilo_socket, &new_socket);
            pthread_create(&hiloMqtt, NULL, hilo_mqtt, &new_socket);
    }

urn 0;
```

```
 GNU nano 2.9.3                                log.txt

<13>Jan 14 03:24:04 Broker: BROKER ENCENDIDO
<13>Jan 14 03:24:04 Broker: ...Escuchando...
<13>Jan 14 03:24:14 Broker: ***NUEVO SUSCRIBER***
<13>Jan 14 03:24:14 Broker: Nuevo suscriber con tópico: nodos/+/temperatura
<13>Jan 14 03:25:24 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:24 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:24 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:24 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:25 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:25 Broker: Publisher envio al tópico: nodos/nodo 18/temperatura el Mensaje: 19.60C
<13>Jan 14 03:25:25 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:25 Broker: Publisher envio al tópico: nodos/nodo 18/temperatura el Mensaje: 19.60C
<13>Jan 14 03:25:29 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:29 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:29 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:29 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:30 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:30 Broker: Publisher envio al tópico: nodos/nodo 18/temperatura el Mensaje: 19.60C
<13>Jan 14 03:25:31 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:31 Broker: Publisher envio al tópico: nodos/nodo 18/temperatura el Mensaje: 19.60C
<13>Jan 14 03:25:38 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:38 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:38 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:38 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:39 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:39 Broker: Publisher envio al tópico: nodos/nodo 18/temperatura el Mensaje: 19.60C
<13>Jan 14 03:25:40 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:40 Broker: Publisher envio al tópico: nodos/nodo 18/temperatura el Mensaje: 19.60C
<13>Jan 14 03:25:47 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:47 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:48 Broker: ***NUEVO PUBLISHER***
<13>Jan 14 03:25:48 Broker: Publisher envio al tópico: nodos/nodo 18/humedad el Mensaje: 50.50
<13>Jan 14 03:25:48 Broker: ***NUEVO PUBLISHER***
```

(13) Implement N-level Topics

Subscriber

For the implementation of N-level topics, two pointer arrays were first established: pubSplit and subSplit. These structures will contain the sub-topics—also known as N-level topics—associated with both the Subscriber and the Publishers, which are sent through the broker.

```
//variables
int socketSubs;
struct sockaddr_in address;
int puerto;
char *ip;
char *topico;
char buffer[1024] = {0};
char *bufferSplit[2];
int error = 0;
//pub
char topicoPub[1024];
char *pubSplit[20];
//sub
char *subSplit[20];
```

We begin with the N-level topics for the Subscriber itself. To do this, we first set the entire subSplit array to NULL, and then use the bufferSeparator function. Using the topic received as a parameter (e.g., general_topic/subtopic1/subtopic2), the function separates the content using the "/" delimiter and stores each sub-topic in the array according to the extracted sections.

```
                    case 't':
                     topico = optarg;
                     break;
                    default:
                     break;
            }
    }
    //Proceso para creacion de socket y envio al broker
    socket_subs();

    //sub multinivel
    memset(subSplit, 0, sizeof(subSplit));
    bufferSeparator(topico, subSplit, "/");
```

The bufferSeparator() function was previously implemented and explained during the development of the partial project. This function essentially takes a string and splits it using a delimiter, then stores each subsection into an array.

```
//bufferSeparator
void bufferSeparator(char *buffer, char *bufferSplit[], char *sep) {
        char *mensaje;
        int i = 0;
        mensaje = strtok(buffer, sep);

        while(mensaje != NULL) {
                bufferSplit[i] = mensaje;
                mensaje = strtok(NULL, sep);
                i++;
        }
}
```

On the other hand, for the N-level topics from the Publishers obtained through the buffer, we start by splitting the buffer using the bufferSeparator function with the delimiter $, since the buffer content follows the format:

general_topic/subtopic1/subtopic2$MESSAGE

In this way, both the topic section and the message are separated and stored in our array called bufferSplit. This array is then passed as a parameter to the analysis() function.

```
        //recibir del broker
        while(1) {
                if(read(socketSubs, buffer, 1024) < 0) {
                        perror("Error read");
                        close(socketSubs);
                        exit(1);
                }
                bufferSeparator(buffer, bufferSplit, "$");
                strcpy(topicoPub, bufferSplit[0]);
                analisis(bufferSplit); //wildcards

                memset(buffer, 0, sizeof(buffer));
        }
```

Inside the analysis() function, we first set the pubSplit array to NULL, and then use the bufferSeparator function with the “/” delimiter to separate the sub-topics and store them within the array.

```
void analisis(char *bufferSplit[]) {

  //Setea NULL todo el contenido
  memset(pubSplit, 0, sizeof(pubSplit));
  bufferSeparator(bufferSplit[0], pubSplit, "/");
```

(14) wildcard (+)

Subscriber

Once the sub-topics registered by the Subscriber and those from the Publishers obtained through the buffer have been separated, we proceed to analyze them. Within the analysis function, we use a while(pubSplit[i] != NULL && subSplit[i] != NULL) loop, meaning that as long as both arrays have sub-topics, they will be analyzed.
We begin by checking whether sub-topic *i* of the Subscriber is a wildcard "+". If it is, we increment the counter *i* by 1 and use continue to move to the next iteration—thus allowing any sub-topic from the Publisher to pass.

```
void analisis(char *bufferSplit[]) {

  //Setea NULL todo el contenido
  memset(pubSplit, 0, sizeof(pubSplit));
  bufferSeparator(bufferSplit[0], pubSplit, "/");
  int i = 0;
  while(pubSplit[i] != NULL && subSplit[i] != NULL) {
    if(strcmp(subSplit[i], "+") == 0){
      i++;
      continue;
    }
    else if(strcmp(subSplit[i], pubSplit[i]) == 0 && subSplit[i+1] == NULL && pubSplit[i+1] == NULL) {
      printf("Nuevo mensaje recibido con TOPICO %s: %s\n", topicoPub, bufferSplit[1]);
      break;
    }
    else if(strcmp(subSplit[i], pubSplit[i]) != 0) {
      break;
    }
   i++;
  }
}
```

Another condition is that if sub-topic *i* is the same for both the Subscriber and the Publisher, and they are also the last sub-topics for each, then we proceed to receive the message and print it to the console. It's important to highlight the check for the last sub-topics, as it confirms that all previous sub-topics in the loop were also equal.

Finally, if sub-topic *i* of the Subscriber and Publisher are not equal, we simply exit the loop and continue listening for requests.

References

[1] This page describes the RTOS xtaskcreate() freertos API function which is part of the RTOS Task Control API. FreeRTOS is a professional grade, small footprint, open source rtos for microcontrollers. FreeRTOS. (2022, January 26). Retrieved December 7, 2022, from https://www.freertos.org/a00125.html

[2] Writing rtos tasks in freertos - implementing tasks as forever loops. FreeRTOS. (2022, January 26). Retrieved December 7, 2022, from https://www.freertos.org/implementing-a-FreeRTOS-task.html

[3] Mutex vs semaphore. GeeksforGeeks. (2021, April 1). Retrieved December 7, 2022, from https://www.geeksforgeeks.org/mutex-vs-semaphore/

[4] Yung, Z. (2022, November 21). MQTT publish/subscribe with Mosquitto Pub/Sub Examples. Cedalo. Retrieved December 7, 2022, from https://cedalo.com/blog/mqtt- subscribe-publish-mosquitto-pub-sub-example/

[5] 262588213843476. (n.d.). Simple socket example in C. Gist.

Retrieved December 7, 2022, from https://gist.github.com/browny/5211329

[6] Mqtt. (n.d.). Servers · MQTT/mqtt.org wiki. GitHub. Retrieved December 7, 2022, from https://github.com/mqtt/mqtt.org/wiki/servers