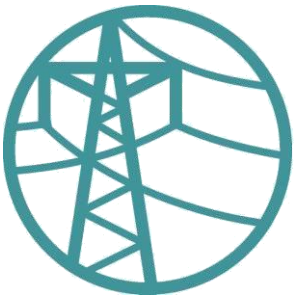


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Komunikacja WiFi pomiędzy ESP32

Projektowanie systemów i sieci komputerowych

Damian Bielecki, 4EFZI P01

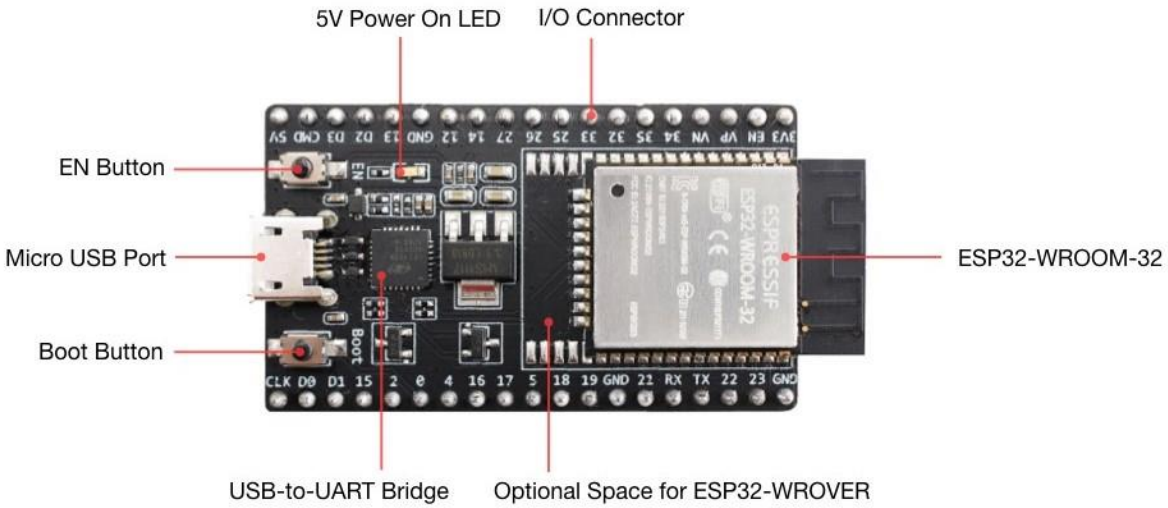
1. Opis projektu

a. Cel projektu

Celem projektu jest przedstawienie możliwości wymiany danych pomiędzy dwoma urządzeniami ESP32 przy pomocy wbudowanego modułu WiFi.

b. Opis platformy

W projekcie wykorzystam płytki deweloperskie z wlutowanym powierzchniowo modułem ESP32-WROOM-32. DevKit został wyposarzony w standarowy raster pinów i złącze USB z układem pozwalającym na programowanie bezpośrednio z komputera. Należy pamiętac że cała płytka pracuje z logiką 3.3V a nie jak typowe układy TTL 5V. Po za WiFi układ oferuje komunikacje Bluetooth, dwu rdzeniowy 32-bitowy procesor o taktowaniu 240Mhz, energooszczędny koprocesor i wiele innych peryferiów.



Rysunek 1. ESP32 devkit

Wykorzystywane WiFi pracuje w zakresie 2,4GHz. Standardowa biblioteka obsługująca łączność bezprzewodową pozwala na:

- Pracę w trybie stacji (STA)
- Tryb programowego punktu dostępowego dla innych urządzeń (AP)
- Równoczesną pracę AP+STA
- różne metody zabezpieczeń połączenia (WEP, WPA2, WPA itp.
- aktywne i pasywne wyszukiwanie dostępnych sieci
- tryb nasłuchiwania pozwalający na odbieranie niezaadresowanych do nas pakietów (standard IEEE802.11.)

Program został napisany w C++ przy wykorzystaniu ESP-IDF (Espressif IoT Development Framework)

2. Metody połączenia dwóch ESP32

Przedstawione niżej przykłady będą korzystały z klasy WIFiConnection do łączenia się z siecią wifi przy pomocy nazwy sieci i hasła dostępowego. Definicja klasy:

```
#define WIFI_CONNECTED_BIT BIT0
#define WIFI_FAIL_BIT      BIT1

//Singleton
class WiFiConnection
{
    public:
    static WiFiConnection& instance();
    void connect(const char*, const char*);
    bool isConnected();


    const int WIFI_CONNECTED_BIT2 = BIT0;
    private:
    WiFiConnection();

    static void event_handler(void* arg, esp_event_base_t event_base,
                int32_t event_id, void* event_data);

    static void wifi_disconect_event(void* arg, esp_event_base_t event_base,
                int32_t event_id, void* event_data);

    /* FreeRTOS event group to signal when we are connected*/
    EventGroupHandle_t s_wifi_event_group;
    const char *TAG;
    bool _isConnected;

    esp_event_handler_instance_t instance_wifi_disconnected;
};
```

Nawiązanie połączenia w funkcji głównej:

```
void app_main(void)
{
   System::getInstance().initFileSystem();
   WiFiConnection::instance().connect("900/2100/2", "ppp123abcAyA");

   if(WiFiConnection::instance().isConnected())
      ESP_LOGE("SYSTEM", "Wifi error - ");
```

Powyższa klasa implementuje wzorzec singleton, i korzysta ze standardowej biblioteki esp i freeRTOS. Biblioteka po wywołaniu metody connect, inicjalizuje sterownik wifi, podpina funkcje zwrotne dla zdarzeń i rozpoczyna proces łączenia z punktem dostępowym.

Wynik wywołanych funkcji:

```
I (1913) wifi:AP's beacon interval = 102400 us, DTIM period = 2
I (2613) esp_netif_handlers: sta ip: 192.168.1.162, mask: 255.255.255.0, gw: 192.168.1.1
I (2613) WIFI_CONNECTION: Successfully connected with ip 192.168.1.162
```

a. ESP32-serwer tcp – ESP32

Jednym ze sposobów na połączenie dwóch esp32 ze sobą jest wykorzystanie serwera tcp pozwalającego na wysyłanie dowolnych danych. W moim przypadku serwerem tcp jest program napisany w pythonie odbierający i wyświetlający dane. W realnym rozwiązaniu serwer odebrane dane od jednego klienta przekazywał by do drugiego. Dużą zaletą jest możliwość uruchomienia serwera poza lokalną siecią co pozwala na połączenie mikrokontrolerów z różnych sieici. Łączność wykonywana jest przy pomocy biblioteki ASIO. Tworzony jest obiekt wskazujący na adres i port serwera, który podawany jest jako argument konstruktora klasy tcpClient.

```
 asio::io_context io_context;
   asio::ip::tcp::endpoint
serverAddress(asio::ip::address_v4::from_string("192.168.1.239"), 3333);
   tcpClient client(io_context, serverAddress);
```

Kolejnym krokiem jest uruchomienie silnika ASIO w oddzielnym wątku tak aby nie blokował wykonania głównej części programu. Jego zadaniem jest przetwarzanie wszystkich asynchronicznych zadań wejścia wyjścia biblioteki asio. Aby funkcja run() nie zakończyła się a więc przestała przetwarzać nasze zapytania uruchamiamy jej sztuczne zadanie.

```
// idle task
   asio::executor_work_guard<decltype(io_context.get_executor())>
work{io_context.get_executor()};
   std::cout << "ASIO engine is up and running" << std::endl;
   xTaskCreate([](void* ptr) -> void {
      ((asio::io_context*)ptr)->run();
   }, "asio", 4096, &io_context, 10, NULL);
```

Obiekt klienta tcp otwiera soket na wskazanym adresie. W pętli głównej programu uruchamiana jest odpowiednia procedura, wysyłające dane do połączonego serwera tcp.

```
tcpClient::tcpClient(asio::io_context& context, asio::ip::tcp::endpoint&
endpoint)
    :_context(context), _socket(context){
    this->_socket.connect(endpoint);
}
```

Pętla główna:

```
while(true)
{
    char buff[100];
    itoa(rand()%20 + 10, buff, 10);
    client.write(std::string(buff));
    vTaskDelay(1000);
}
```

Przykładowe wykorzystanie:

Esp co określony czas wysyła wylosowane dane imitujące odczytaną temperaturę.

```
I (1910) wifi:AP's beacon interval = 102400 us, DTIM period = 2
I (2560) esp_netif_handlers: sta ip: 192.168.1.162, mask: 255.255.255.0, gw: 192.168.1.1
I (2560) WIFI_CONNECTION: Successfully connected with ip 192.168.1.162
W (2560) esp32_asio_pthread: pthread_condattr_setclock: not yet supported!
W (2630) wifi:<ba-add>idx:0 (ifx:0, 90:f6:52:91:49:38), tid:0, ssn:3, winSize:64
W (7770) wifi:<ba-del>idx
W (8330) wifi:<ba-add>idx:0 (ifx:0, 90:f6:52:91:49:38), tid:0, ssn:4, winSize:64
ASIO engine is up and running
Wyslano dane => 11
W (13680) wifi:<ba-del>idx
Wyslano dane => 21
W (18330) wifi:<ba-add>idx:0 (ifx:0, 90:f6:52:91:49:38), tid:0, ssn:7, winSize:64
W (23670) wifi:<ba-del>idx
Wyslano dane => 17
W (29930) wifi:<ba-add>idx:0 (ifx:0, 90:f6:52:91:49:38), tid:0, ssn:9, winSize:64
```

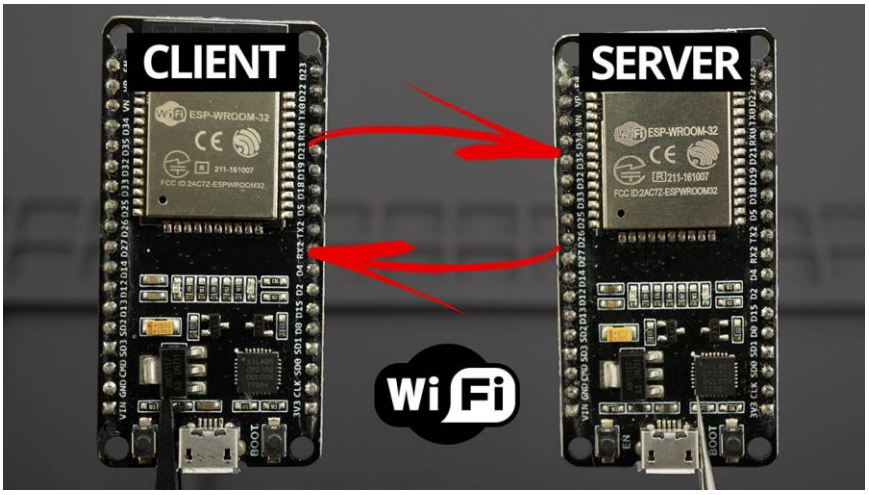
Rysunek 2. Nawiązane połączenie z WiFi i wysłane dane

Serwer odebrane dane w zależności od potrzeby może zapisać lub wysłać dane do pozostałych węzłów sieci czy innego serwera agregującego dane.

```
PS C:\Users\Damian\Desktop\esp32-tcps> python .\server.py
Connection from: ('192.168.1.162', 64506)
from connected user: 11
from connected user: 21
from connected user: 17
from connected user: 25
```

Rysunek 3. Serwer odbierający dane od klienta

b. ESP32-ESP32



Rysunek 4.Bezpośrednie połączenie [2]

Esp32 pozwala na stworzenie programowego punktu dostępowego co umożliwia nam połączenie dwóch sterowników bez zewnętrznego serwera. Na sterowniku tworzącym sieć wifi uruchamiamy serwer tcp pozwalający nam na dowolną wymianę danych.

Sieć tworzona jest przy pomocy biblioteki WiFiConnection i metody createNetwork

```
  System::getInstance().initFileSystem();
   WiFiConnection::instance().createNetwork("esp32Test", "ppp123abcAyA");
```

Metoda createNetwork korzysta ze standardowego api jak w przypadku pracy w trybie klienta. Podobnie jak w poprzednim przypadku tworzony jest wątek dla kontekstu biblioteki asio. Oprogramowanie klienta jest takie samo jak w przypadku esp-serwer-esp a więc dalej przedstawię tylko implementacje serwera tcp na esp.

```
   asio::io_context io_context;
   server s(io_context, 3333);
   std::cout << "ASIO engine is up and running" << std::endl;
   xTaskCreate([](void* ptr) -> void {
      ((asio::io_context*)ptr)->run();
   }, "tcp_server", 4096, &io_context, 10, NULL);

   std::cout<<"Serwer jest uruchomiony"<<std::endl;
```

Największą różnica względem poprzedniego programu jest stworzenie obiektu klasy server, przyjmującej kontekst odpowiadający za przychodzące połączenia.

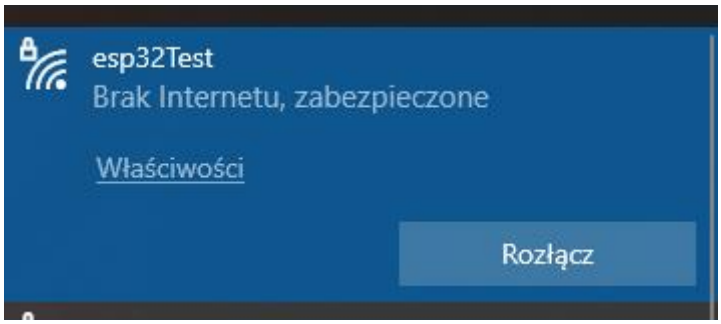
```
server::server(asio::io_context& io_context, short port)
    :acceptor_(io_context, tcp::endpoint(tcp::v4(), port))
{
    do_accept();
}
void server::do_accept()
```

```
{
    acceptor_.async_accept(
        [this](std::error_code ec, tcp::socket socket)
        {
            if (!ec)
            {
                std::make_shared<session>(std::move(socket))->start();
            }
            do_accept();
        });
}
```

Konstruktor serwera informuje bibliotekę że obsługuje przychodzące połączenia tcp dla ipv4 i w ciele funkcji uruchamia asynchroniczną metodę czekającą na połączenie. Jeżeli klient ze chce nawiązać połączenie z naszym serwerem to zostanie wywołana funkcja lambda, która stworzy sesje użytkownika i przekaże jej otwarte gniazdo.

```
void session::start()
{
    // this->sendData("Hello! user!");
    this->do_read();
}
void session::do_read()
{
    auto self(shared_from_this());
    socket_.async_read_some(asio::buffer(data_, max_length),
        [this, self](std::error_code ec, std::size_t length)
        {
            if (!ec)
            {
                data_[length] = 0;
                do_write(length);
                std::cout<<" DATA => "<< data_<<std::endl;
            }
        });
}
```

Na sesji wykonywana jest metoda start uruchamiająca asynchroniczną funkcje odbierającą nadsyłane dane. W funkcji start możemy wysłać podstawowe dane pozwalające na konfiguracje wymienianych danych pomiędzy sterownikami. W tym przypadku po odczytaniu danych biblioteka wywołuje funkcje zwrotną, ta wyświetla dane z bufora i odsyła je do klienta. W rzeczywistym sterowniku w tym miejscu odczytany by był nagłówek na podstawie, którego podjęto by dalsze działania na danych.



Rysunek 5. Utworzona sieć wifi

Poniżej widać logi z esp i serwera dhcp, który przydzielił adres dla nowo połączonego laptopa. Esp ma adres 192.168.4.1 a laptop kolejny wolny 192.168.4.2

```
I (882) WIFI_CONNECTION: wifi_init_softap finished. SSID:esp32Test password:ppp123abcAyA
W (892) esp32_asio_pthread: pthread_condattr_setclock: not yet supported!
ASIO engine is up and running
Serwer jest uruchomiony
I (1508792) wifi:new:<1,1>, old:<1,1>, ap:<1,1>, sta:<255,255>, prof:1
I (1508792) wifi:station: 10:6f:d9:25:a6:93 join, AID=1, bgn, 40U
I (1508912) WIFI_CONNECTION: station 10:6f:d9:25:a6:93 join, AID=1
I (1508962) esp_netif_lwip: DHCP server assigned IP to a station, IP is: 192.168.4.2
W (1509012) wifi:<ba-add>idx:2 (ifx:1, 10:6f:d9:25:a6:93), tid:0, ssn:3, winSize:64
```

Rysunek 6. Wykryte połączenie wifi

Klient na laptopie łączy się z serwerem tcp i oczekuje na dane do wysłania, imituje on drugie esp łączące się do sieci i wysyłane dane.

```
PS C:\Users\Damian\Desktop\esp-esp> python.exe .\client.py
Połączenie z 192.168.4.1:3333
test
Odebrane => test
tu moze byc json
Odebrane => tu moze byc json
lub dane binnarne
Odebrane => lub dane binnarne
```

Rysunek 7. Klient w pythonie łączący się z serwerem

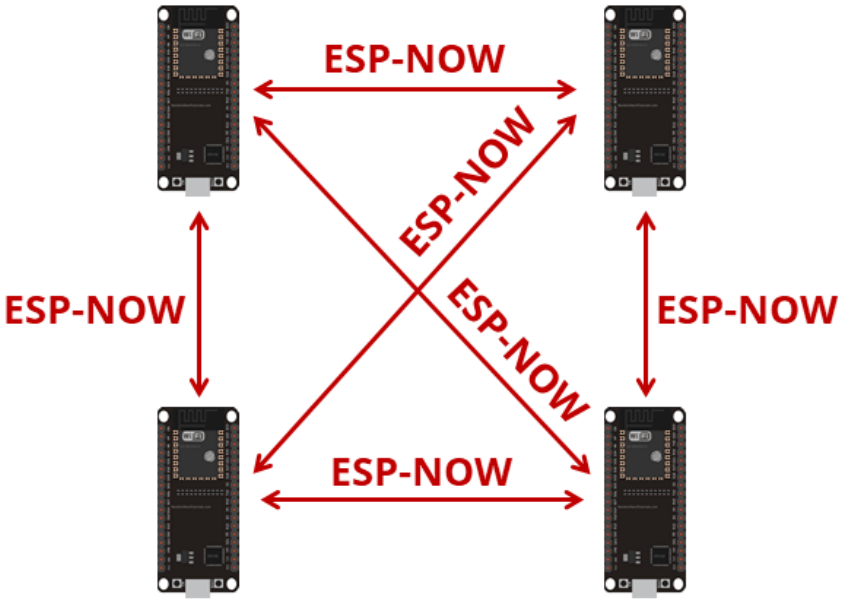
Serwer tcp odbiera i wyświetla dane w konsoli

```
I (8952) esp_netif_lwip: DHCP server assigned IP to a station, IP is: 192.168.4.2
W (8982) wifi:<ba-add>idx:2 (ifx:1, 10:6f:d9:25:a6:93), tid:0, ssn:3, winSize:64
 DATA => test
 DATA => tu moze byc json
 DATA => lub dane binnarne
```

Rysunek 8. Odbierane dane

c. ESPNOW

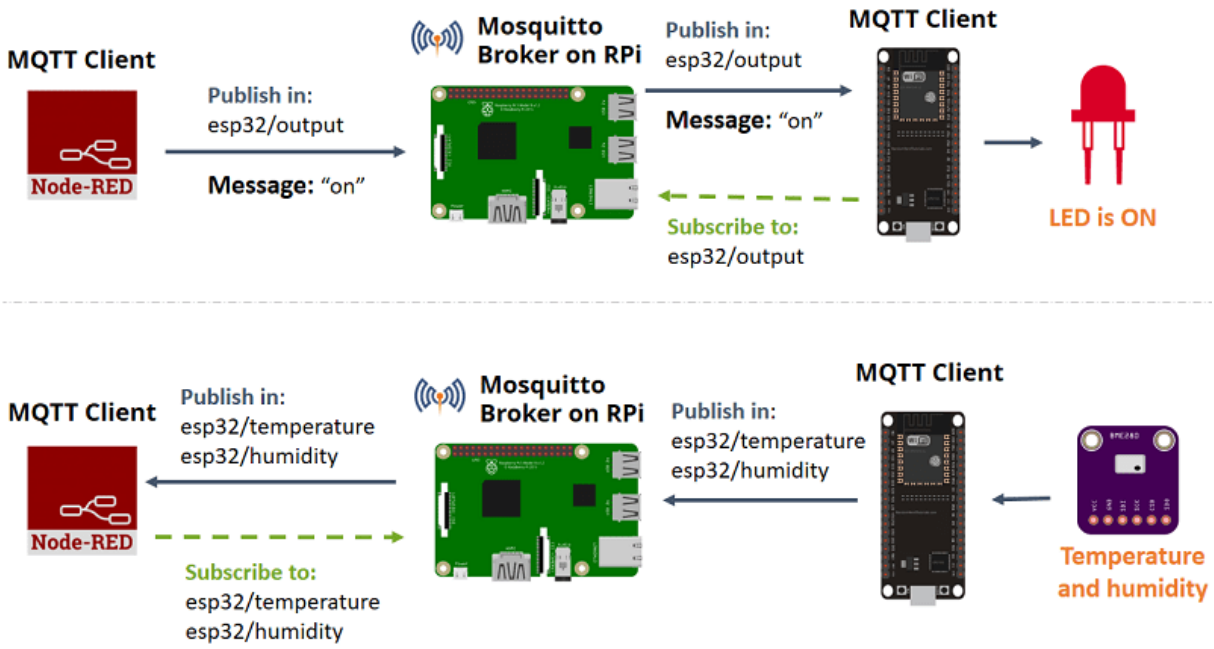
EspNow to bezpołączeniowy protokół komunikacyjny pracujący na tej samej częstotliwości co WiFi. Przed nawiązaniem komunikacji konieczne jest sparowanie urządzeń. Wysyłane dane są enkapsulowane i wysyłane w trybie peer-to-peer. Maksymalny rozmiar danych to 250 bajtów. Łączność może być nawiązana pomiędzy 20 urządzeniami bez szyfrowania i 10 z szyfrowaniem. Protokół ten ze względu na prostotę połączenia może być szeroko stosowany w IOT do budowy prostej sieci czujników i sterowników. Espnow pozwala na połączenie wielu węzłów do sieci i wymianę danych w trybie rozgłoszeniowym lub pomiędzy dwoma urządzeniami.



Rysunek 9. Sieć ESPNow [3]

d. MQTT

MQTT – to oparty o wzorzec publikacja-subskrypcja bardzo lekki protokół transmisji danych pozwalający na wymianę informacji w sieciach IOT. Protokół dzięki zmniejszeniu prędkości wymiany danych gwarantuje zwiększoną niezawodność. Wymiana danych odbywa się poprzez serwer pośredniczący nazywanym brokerem mqtt. Każdy z klientów łączy się z brokerem a następnie subskrybuje dany temat, w którym może również publikować informacji. Każdy klient subskrybujący dany temat otrzyma aktualizacje gdy ten się zmieni.



Rysunek 10. Schemat logiczny połączenia pomiędzy systemami [1]

Uruchomienie serwera

Serwer zostanie uruchomiony w wirtualnym środowisku dockera i obrazem emqx, który możemy pobrać przy pomocy polecenia:

```
$ docker pull emqx
```

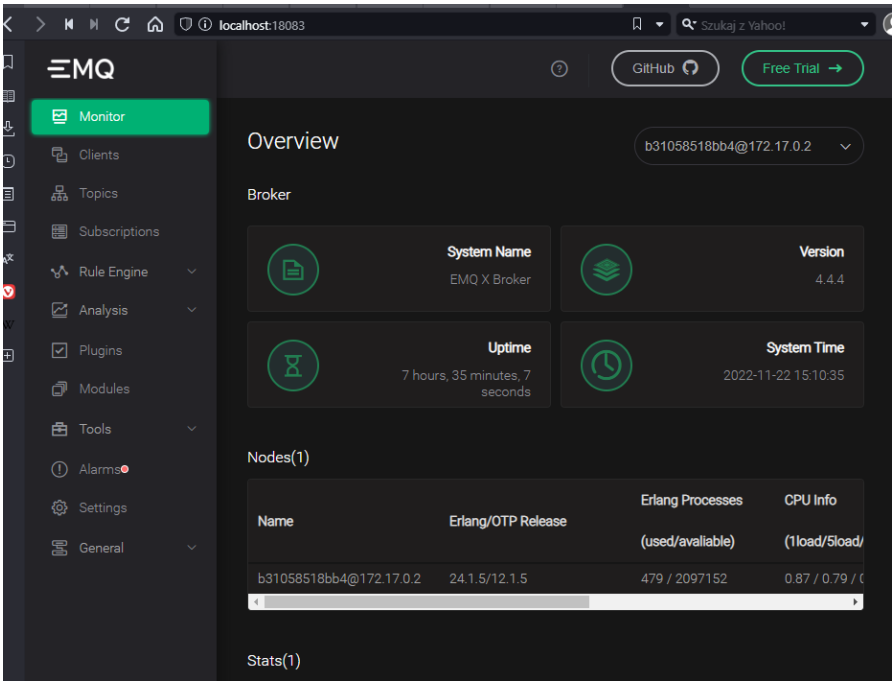
Kontener uruchamiany przy pomocy polecenia:

```
$ docker run -d --name emqx -p 18083:18083 -p 1883:1883 emqx:latest
```

Uruchomiony kontener:

Rysunek 11. uruchomiony kontener z emqx

Widok panelu zarządzania:



Rysunek 12. Panel sterowania serwerem

Implementacja

Po połączeniu się z siecią wifi, inicjalizujemy strukturę konfiguracyjną klienta mqtt:

```
esp_mqtt_client_config_t mqtt_cfg;
memset(&mqtt_cfg, 0, sizeof(mqtt_cfg));
mqtt_cfg.username = "admin";
mqtt_cfg.password = "public";
mqtt_cfg.port     = 1883;
mqtt_cfg.host     = "192.168.1.239";
```

Zainstalowany broker mqtt ma stworzonego domyślnego użytkownika admin z hasłem „public”. Podany adres jest adresem naszego komputera w sieci lokalnej. Na jej podstawie tworzymy instancje klienta i łączymy się z brokerem.

```
 esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
 esp_mqtt_client_start(client);
 esp_mqtt_client_register_event(client, esp_mqtt_event_id_t::MQTT_EVENT_ANY,
mqtt_event_handler, NULL);
```

Ostatnie funkcja odpowiada za przechwytywanie eventów. Po połączeniu esp z wifi i serwerem mqtt subskrybowane są dwa tematy:

```
 int msg_id;
 msg_id = esp_mqtt_client_subscribe(client, "/czujniki/smog", 0);
 ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);

 msg_id = esp_mqtt_client_subscribe(client, "/czujniki/temperatura1", 1);
 ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);
```

Uruchomienie

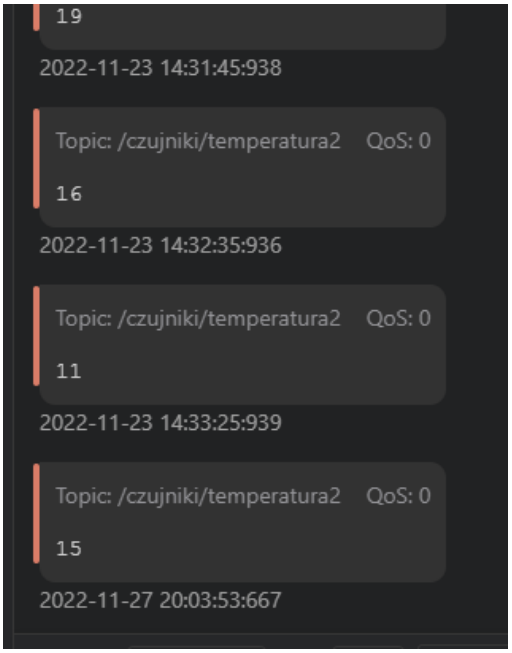
```
I (2122) wifi:AP's beacon interval = 102400 us, DTIM period = 2
I (2692) esp_netif_handlers: sta ip: 192.168.1.162, mask: 255.255.255.0, gw: 192.168.1.1
I (2692) WIFI_CONNECTION: Successfully connected with ip 192.168.1.162
I (2702) MQTT_EXAMPLE: Other event id:7
W (2742) wifi:<ba-add>idx:0 (ifx:0, 90:f6:52:91:49:38), tid:0, ssn:3, winSize:64
I (2762) MQTT_EXAMPLE: MQTT_EVENT_CONNECTED
W (7872) wifi:<ba-del>idx
I (12692) MQTT_EXAMPLE: sent subscribe successful, msg_id=20090
I (12692) MQTT_EXAMPLE: sent subscribe successful, msg_id=55352
I (12692) MQTT_EXAMPLE: sent publish successful, msg_id=48174
W (12712) wifi:<ba-add>idx:0 (ifx:0, 90:f6:52:91:49:38), tid:0, ssn:7, winSize:64
I (12982) MQTT_EXAMPLE: Other event id:3
I (13012) MQTT_EXAMPLE: Other event id:3
I (13022) MQTT_EXAMPLE: Other event id:5
W (18122) wifi:<ba-del>idx
```

Rysunek 13. Odbierane eventy mqtt

Client ID	Username	IP Address	Keepaliv
mqttx_abd77ca7	admin	172.17.0.1:39220	60
ESP32_4cC2FC	admin	172.17.0.1:39236	120

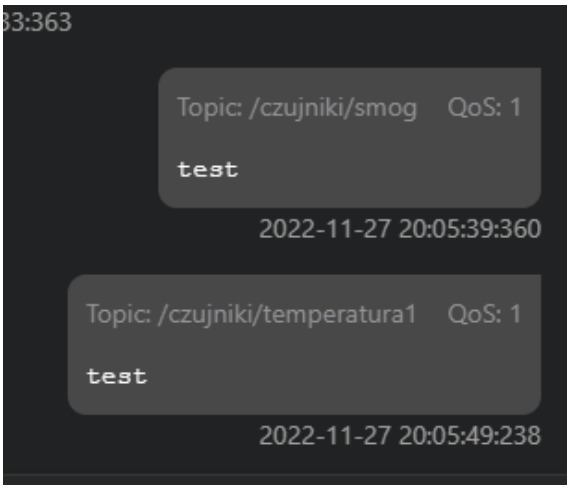
Rysunek 14. Podłączeni klienci mqtt

Esp w pętli głównej publikowało losowe dane w temacie /czujniki/temperatura2.

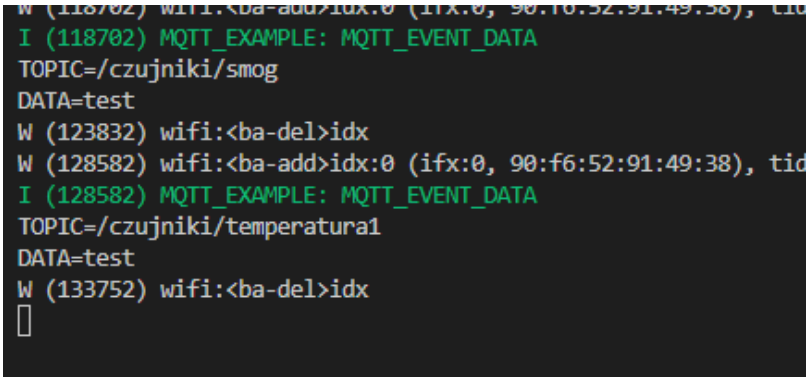


Rysunek 15.Odbierane dane na kliencie wysyłane przez esp

Korzystając z klienta na PC MQTTX połączyłem się z serwerem i wysłałem wiadomości testowe, które zostały odebrane i wyświetlone przez esp.



Rysunek 16. Wysłane dane z klienta na pc



Rysunek 17. Odebrane dane przez esp

Podsumowanie

W projekcie zostały przedstawione różne sposoby połączenia dwóch mikrokontrolerów ESP32 korzystając z sieci wifi. Przedstawione przykłady wysyłały proste komunikaty tekstowe jednak można je łatwo przerobić tak aby w zależności od zastosowania wysyłały i odbierały dane. Rozwiązania wykorzystujące pośredni serwer mają większe wymagania jednak pozwalają na wykonanie połączenia z różnych lokalnych sieci. Esp to układy szeroko stosowane w IOT i w wielu przypadkach będą poza zasięgiem sieci bezprzewodowej. W takich przypadkach możemy wykorzystać sieć LoRAWAN, cechującą się niskim zużyciem energii.

Bibliografia

[1] https://randomnerdtutorials.com/esp32-mqtt-publish-subscribe-arduino-ide/

[2] https://randomnerdtutorials.com/esp32-client-server-wi-fi/

[3] https://randomnerdtutorials.com/esp-now-esp32-arduino-ide/