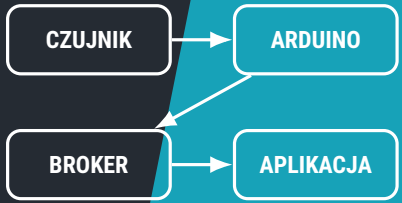
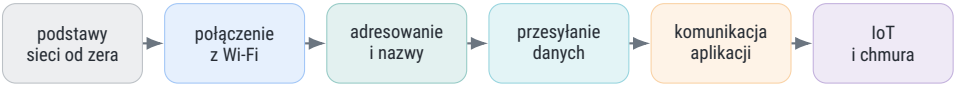
Arduino w Internecie
Od Wi-Fi do MQTT i chmury IoT

Wi-Fi, IP, TCP/UDP, HTTP, JSON, MQTT i bezpieczeństwo



Podstawy Systemów Wbudowanych • Wykład 3
dr inż. Angelika Tefelska

Mapa dzisiejszego wykładu



Najpierw zbudujemy **słownik pojęć**, a dopiero potem będziemy używać ich w kodzie i diagramach.

Od układu lokalnego do Internetu Rzeczy

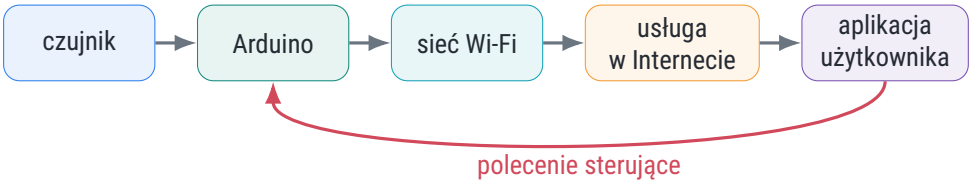
Układ lokalny

- czujnik $\rightarrow$ Arduino,
- decyzja wykonywana na miejscu,
- brak zależności od sieci,
- dane znikają po wyłączeniu zasilania, jeśli ich nie zapisujemy.

Węzeł IoT

- pomiar może trafić do zdalnej usługi,
- dane mogą być archiwizowane i wizualizowane,
- polecenie może przyjść z aplikacji użytkownika,
- pojawiają się opóźnienia, utrata łączności i kwestie bezpieczeństwa.

Typowy przepływ danych w systemie IoT



Dwie niezależne ścieżki

telemetria: urządzenie → użytkownik

sterowanie: użytkownik → urządzenie

Zanim podłączymy Arduino: pięć pytań o komunikację sieciową

1. **Jak urządzenie fizycznie łączy się z siecią?** — u nas bezprzewodowo przez Wi-Fi.
2. **Jak urządzenia są identyfikowane?** — potrzebują adresów.
3. **Jak znaleźć urządzenie lub usługę po nazwie?** — potrzebny jest mechanizm tłumaczenia nazw.
4. **Jak dostarczyć dane?** — możemy wymagać niezawodności albo preferować prostotę i małe opóźnienie.
5. **Skąd odbiorca wie, co znaczą odebrane bajty?** — strony muszą znać ten sam protokół i format danych.

To właśnie uporządkujemy na początku

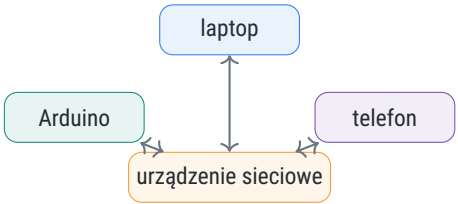
Dopiero po odpowiedzi na te pytania skróty takie jak **IP, DNS, TCP, UDP, HTTP, JSON i MQTT** zaczną mieć konkretne znaczenie.

0. Sieci komputerowe od zera

Minimalny zestaw pojęć potrzebny do zrozumienia dalszej części wykładu.

Co to jest sieć komputerowa?

- sieć to grupa urządzeń, które potrafią **wymieniać dane**,
- urządzeniem może być komputer, telefon, drukarka, Arduino albo serwer,
- urządzenia nie muszą być tego samego typu ani wykonywać tego samego systemu,
- potrzebują jednak wspólnego sposobu przesyłania i interpretowania informacji.



Sieć lokalna i Internet — to nie to samo

Sieć lokalna — LAN

- obejmuje urządzenia w domu, laboratorium lub firmie,
- urządzenia mogą komunikować się nawet bez dostępu do Internetu,
- przykładem jest laptop i Arduino podłączone do tego samego routera.

Internet

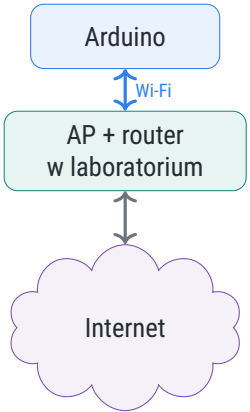
- to ogromna „sieć sieci”,
- łączy wiele niezależnych sieci lokalnych,
- pozwala dotrzeć do usług działających poza naszym laboratorium.

Wi-Fi $\neq$ Internet

Wi-Fi to sposób bezprzewodowego połączenia urządzenia z **siecią lokalną** przez punkt dostępowy. **Internet** to połączenie naszej sieci z innymi sieciami na świecie.
Dlatego telefon może pokazywać ikonę Wi-Fi, a strony internetowe nadal mogą nie działać — np. gdy router utraci łącze z dostawcą Internetu.

Access Point i router — dwie role, często w jednej obudowie

- **Access Point (AP)** zapewnia bezprzewodowe połączenie Wi-Fi z siecią lokalną,
- **router** przekazuje dane pomiędzy różnymi sieciami,
- domowe urządzenie nazywane „routerem Wi-Fi” zwykle pełni obie role jednocześnie,
- w laboratorium Arduino łączy się radiowo z AP, a router daje mu drogę dalej.



Wi-Fi nie jest synonimem Internetu

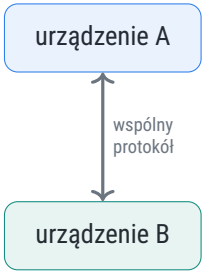
1. Arduino może poprawnie połączyć się z punktem dostępowym.
2. Może dostać adres w sieci lokalnej.
3. Może nawet komunikować się z laptopem w tej samej sieci.
4. A mimo to dostęp do Internetu może nie działać — np. router nie ma połączenia z siecią zewnętrzną.

Diagnostyka

Komunikat „Wi-Fi connected” mówi tylko, że udało się wykonać **pierwszy etap**. Nie dowodzi jeszcze, że działa DNS, połączenie z serwerem ani usługa IoT.

Protokół — wspólne zasady rozmowy

- protokół określa **jak wygląda komunikacja**,
- definiuje kolejność działań, znaczenie pól i reakcję na błędy,
- dwie strony mogą wymieniać dane tylko wtedy, gdy rozumieją te same zasady.



Analogia

Sam fakt, że dwie osoby mają telefony, nie wystarcza do rozmowy. Muszą jeszcze znać wspólny język i zasady prowadzenia rozmowy.

Adres urządzenia — po co jest potrzebny?

- jeśli w sieci jest wiele urządzeń, trzeba wskazać **nadawcę i odbiorcę**,
- podobnie jak przesyłka potrzebuje adresu domu, dane sieciowe potrzebują adresów,
- w praktyce spotkamy kilka rodzajów adresów, używanych na różnych etapach komunikacji.

MAC

Identyfikuje interfejs sieciowy w lokalnej komunikacji. Zwykle zapisujemy go jako sześć bajtów, np. `A4:CF:12:8B:30:9E`.

IP

Adres logiczny używany do komunikacji pomiędzy urządzeniami i sieciami, np. `192.168.1.42`.

IP — adres logiczny urządzenia w sieci

- **IP** = Internet Protocol,
- adres IP mówi, gdzie logicznie znajduje się urządzenie,
- w laboratorium będziemy głównie widzieć IPv4,
- adres IPv4 ma 32 bity i dla wygody zapisujemy go jako cztery liczby 0–255.

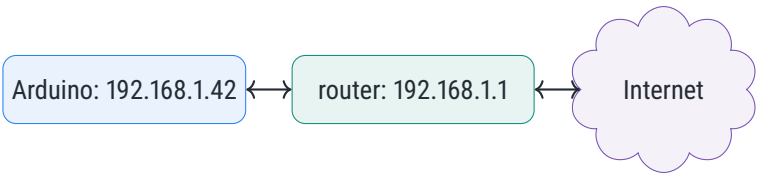
Przykład

192.168.1.42

To nie jest „numer Arduino na zawsze”. Adres może się zmieniać po ponownym podłączeniu do sieci.

Adres prywatny — numer używany wewnątrz sieci lokalnej

- w domu i laboratorium często zobaczymy adresy zaczynające się od `192.168.` albo `10.`,
- takie adresy są używane wewnątrz sieci lokalnych,
- dwa różne domy mogą mieć urządzenie `192.168.1.42` — nie ma konfliktu, bo to różne sieci,
- aby wyjść poza własną sieć, urządzenie korzysta z routera.



DHCP — automatyczne przydzielanie konfiguracji sieciowej

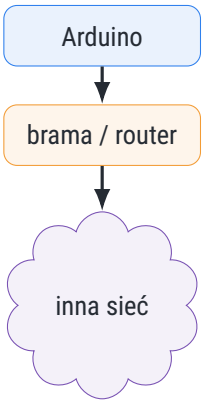
- po dołączeniu do Wi-Fi Arduino jeszcze nie wie, jaki ma mieć adres IP,
- **DHCP** to mechanizm, dzięki któremu urządzenie może automatycznie dostać konfigurację,
- serwer DHCP zwykle działa w routerze,
- oprócz adresu IP może przekazać m.in. informację o bramie i serwerze DNS.

W praktyce

Dlatego zazwyczaj nie wpisujemy adresu IP Arduino ręcznie. Po połączeniu z siecią wyświetlamy adres przydzielony przez DHCP.

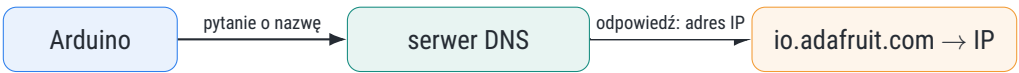
Brama domyślna — gdzie wysłać dane „poza moją sieć”?

- Arduino potrafi bezpośrednio dotrzeć do urządzeń w swojej sieci lokalnej,
- jeśli odbiorca jest gdzie indziej, dane trafiają najpierw do **bramy domyślnej**,
- w typowej sieci bramą jest router,
- router decyduje, gdzie przekazać dane dalej.



DNS — tłumaczenie nazwy na adres IP

- ludziom łatwiej zapamiętać nazwę `io.adafruit.com` niż adres liczbowy,
- sieć potrzebuje jednak adresu IP,
- **DNS** = Domain Name System — system tłumaczący nazwy domenowe na adresy IP,
- program pyta serwer DNS: „jaki adres IP ma ta nazwa?”.



Klient i serwer — dwie role w rozmowie

Klient

- rozpoczyna kontakt,
- prosi o dane albo usługę,
- Arduino często występuje właśnie jako klient.

Serwer

- czeka na żądania klientów,
- udostępnia dane lub funkcję,
- może obsługiwać wielu klientów.

Role dotyczą konkretnej komunikacji — „serwer” nie musi oznaczać ogromnego komputera w centrum danych.

Port — jeden komputer może oferować wiele usług

- adres IP wskazuje **urządzenie**, ale na jednym urządzeniu może działać wiele programów sieciowych,
- **port** to numer pomagający wskazać konkretną usługę,
- analogia: IP = adres budynku, port = numer pokoju,
- port sieciowy nie ma nic wspólnego z pinem GPIO Arduino.

Przykład

`192.168.1.20:80` oraz `192.168.1.20:1883`

Ten sam adres IP, ale dwa różne numery portów mogą oznaczać dwie różne usługi.

TCP — kiedy chcemy, aby dane dotarły poprawnie i w kolejności

- **TCP** = Transmission Control Protocol,
- TCP jest protokołem **połączeniowym**: zanim aplikacja wyśle właściwe dane, klient i serwer wykonują krótką wymianę kontrolną i tworzą logiczne połączenie,
- dopiero w ramach tego połączenia TCP numeruje i kontroluje przesyłane dane,
- jeśli fragment zginie, może zostać wysłany ponownie,
- odbiorca dostaje uporządkowany strumień bajtów.

Koszt

Większa niezawodność oznacza dodatkową komunikację, pamięć i czas. TCP nie jest „zawsze najlepsze” — jest dobre wtedy, gdy jego własności są potrzebne.

UDP — kiedy wysyłamy niezależne datagramy bez zestawiania połączenia

- **UDP** = User Datagram Protocol,
- nadawca wysyła osobne datagramy bez wcześniejszego zestawienia połączenia,
- protokół sam nie gwarantuje, że datagram dotrze,
- nie gwarantuje też kolejności ani automatycznej retransmisji,
- dzięki temu narzut i opóźnienia mogą być mniejsze.

Ważne

„UDP może zgubić dane” nie oznacza, że dane zawsze giną. Oznacza tylko, że **sam UDP nie zapewnia mechanizmu naprawy**.

TCP czy UDP? — prosta analogia

TCP — rozmowa z potwierdzeniami

„Czy słyszysz?” → „Tak.”
„Wysyłam część 1.” → „Mam.”
„Wysyłam część 2.” → „Mam.”

UDP — wrzucanie kolejnych wiadomości

„Temperatura 21.4.”
„Temperatura 21.5.”
„Temperatura 21.6.”

Nadawca nie czeka na potwierdzenie każdej z nich.

Dobór transportu zależy od tego, **co bardziej boli: strata danych czy opóźnienie/narzut**.

Protokół aplikacyjny — co właściwie znaczą przesyłane bajty?

- TCP lub UDP mówi głównie **jak dostarczyć bajty**,
- aplikacja musi jeszcze wiedzieć, **co te bajty oznaczają**,
- dlatego nad transportem działają protokoły aplikacyjne,
- w tym wykładzie poznamy przede wszystkim **HTTP** i **MQTT**.

HTTP

Typowy model: klient wysyła żądanie, serwer odsyła odpowiedź.

MQTT

Typowy model IoT: urządzenia publikują i subskrybują wiadomości przez pośrednika.

Format danych — odbiorca musi wiedzieć, jak odczytać treść

- sieć przesyła ostatecznie **bajty**,
- te same bajty mogą reprezentować tekst, liczbę, obraz, dźwięk albo strukturę danych,
- nadawca i odbiorca muszą uzgodnić format,
- w aplikacjach internetowych bardzo często spotkamy tekstowy format **JSON**.

Przykład tej samej informacji

```
21.7    temperature=21.7    {"temperature":21.7}
```

Znaczenie jest podobne, ale struktura danych jest inna. Program musi wiedzieć, której postaci się spodziewać.

JSON — prosty sposób zapisu danych strukturalnych

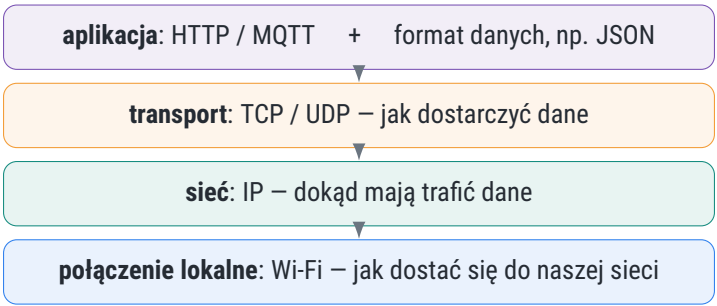
```
{
  "temperature": 21.7,
  "humidity": 48,
  "alarm": false
}
```

- **JSON** = JavaScript Object Notation,
- dane zapisujemy jako pary **klucz : wartość**,
- klucze tekstowe zapisujemy w cudzysłowie,
- wartości mogą być liczbami, tekstem, `true/false`, tablicą, obiektem albo `null`.

Na razie wystarczy

JSON nie jest protokołem sieciowym. To **format danych**, który możemy przesłać np. za pomocą HTTP albo MQTT.

Warstwy — teraz skróty mają już konkretne znaczenie



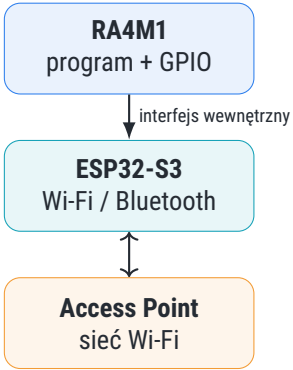
DNS pomaga znaleźć adres IP po nazwie, a **DHCP** automatycznie dostarcza urządzeniu podstawową konfigurację sieciową.

1. Wi-Fi i Arduino UNO R4 WiFi

Najpierw musimy połączyć urządzenie z siecią lokalną — dopiero potem możemy mówić o Internecie.

Arduino UNO R4 WiFi — dwa układy współpracujące ze sobą

- główny mikrokontroler to **Renesas RA4M1**,
- łączność bezprzewodową realizuje moduł z **ESP32-S3**,
- program użytkownika może korzystać z Wi-Fi przez bibliotekę `WiFiS3`,
- z punktu widzenia szkicu ESP32-S3 działa jak moduł komunikacyjny obsługiwany przez bibliotekę.



Źródło: Arduino UNO R4 WiFi — dokumentacja sprzętowa

Co naprawdę daje nam Wi-Fi?

- dostęp do **sieci lokalnej** — niekoniecznie od razu do Internetu,
- możliwość wymiany ramek z punktem dostępowym,
- po konfiguracji IP — możliwość wysyłania pakietów do innych hostów,
- dopiero potem możemy uruchomić protokół aplikacyjny: HTTP, MQTT itd.

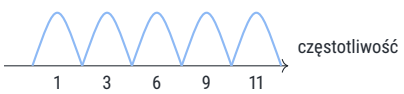
Ważne rozróżnienie

„Połączono z Wi-Fi” nie oznacza jeszcze, że działa DNS, Internet albo broker MQTT.

Wi-Fi w tym kursie: pasmo 2,4 GHz

- UNO R4 WiFi obsługuje Wi-Fi 4 / IEEE 802.11 b/g/n w paśmie 2,4 GHz,
- punkt dostępowy nadaje na określonym **kanale**,
- wiele sieci może być widocznych jednocześnie,
- kanały częściowo nakładają się częstotliwościowo — stąd możliwe zakłócenia.

schematycznie: kanały mogą się nakładać



Źródło: UNO R4 WiFi — datasheet

SSID, BSSID i kanał — trzy różne pojęcia

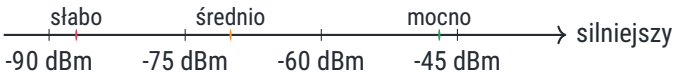
SSID	nazwa sieci widoczna dla użytkownika, np. `PSW-Lab`
BSSID	identyfikator konkretnego punktu dostępowego — zwykle jego adres MAC
kanał	fragment pasma radiowego używany do transmisji

Dlaczego to rozróżniamy?

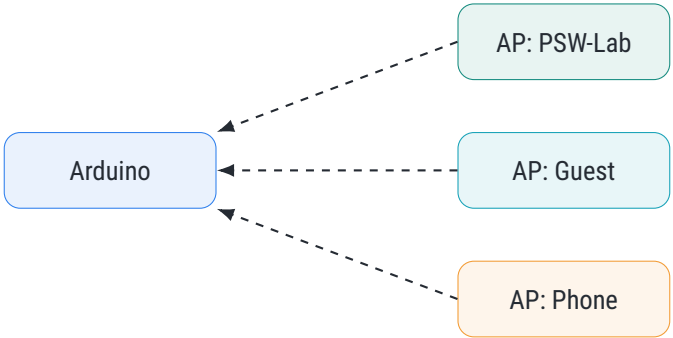
Kilka punktów dostępowych może rozgłaszać **ten sam SSID**, ale mieć inne BSSID i pracować na innych kanałach.

RSSI — jak silny sygnał odbiera urządzenie?

- RSSI = **Received Signal Strength Indicator**,
- w bibliotekach Wi-Fi wartość jest zwykle podawana w dBm,
- jest ujemna: wartość bliższa zeru oznacza silniejszy sygnał,
- RSSI opisuje poziom odebranego sygnału — nie gwarantuje jakości całej komunikacji.



Skanowanie sieci — co robi Arduino?



Skanowanie nie oznacza jeszcze połączenia. Arduino tylko **wykrywa sieci** i odczytuje ich parametry.

WiFiS3: skanowanie dostępnych sieci

```
#include <WiFiS3.h>

void setup() {
  Serial.begin(115200);

  int n = WiFi.scanNetworks();

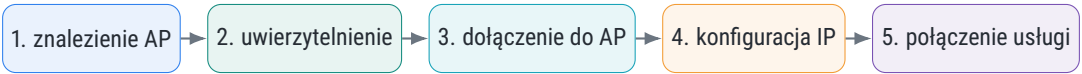
  for (int i = 0; i < n; i++) {
    Serial.print(WiFi.SSID(i));
    Serial.print("  RSSI = ");
    Serial.print(WiFi.RSSI(i));
    Serial.println(" dBm");
  }
}

void loop() { }
```

Przykładowy Serial Monitor

```
PSW-Lab                RSSI = -48 dBm
Phone-Hotspot          RSSI = -71 dBm
Guest                  RSSI = -83 dBm
```

Połączenie z Wi-Fi to kilka etapów



Gdy program „nie łączy się z chmurą”, warto ustalić **na którym etapie** zatrzymał się proces.

Hasło Wi-Fi — do czego służy?

- jest elementem mechanizmu uwierzytelniania do sieci bezprzewodowej,
- nie jest adresem IP ani „hasłem do Internetu”,
- po poprawnym uwierzytelnieniu urządzenie nadal potrzebuje konfiguracji IP,
- poprawne hasło Wi-Fi nie zabezpiecza automatycznie danych na poziomie aplikacji.

Dwie różne warstwy bezpieczeństwa

WPA2/WPA3 chroni łącze Wi-Fi. HTTPS/MQTT over TLS chroni komunikację aplikacyjną z serwerem.

Dane dostępowe poza głównym szkicem

```
// arduino_secrets.h
#define SECRET_SSID "PSW-Lab"
#define SECRET_PASS "tajne_haslo"
```

```
// glowny szkic
#include "arduino_secrets.h"

char ssid[] = SECRET_SSID;
char pass[] = SECRET_PASS;
```

To porządek, nie magia bezpieczeństwa

Oddzielny plik ułatwia pilnowanie sekretów i wykluczenie go z repozytorium, ale sam w sobie nie szyfruje hasła.

WiFi.begin() — rozpoczęcie połączenia

```
#include <WiFiS3.h>
#include "arduino_secrets.h"

char ssid[] = SECRET_SSID;
char pass[] = SECRET_PASS;

void setup() {
  Serial.begin(115200);

  WiFi.begin(ssid, pass);
}
```

- wywołanie inicjuje proces łączenia,
- program powinien potem sprawdzać **stan połączenia**,
- nie zakładamy, że każde wywołanie kończy się sukcesem.

WiFi.status() — połączenie ma stan

Stan	Znaczenie praktyczne
`WL_CONNECTED`	połączenie z siecią zostało zestawione
`WL_IDLE_STATUS`	biblioteka jest w trakcie operacji / oczekiwania
`WL_NO_SSID_AVAIL`	wskazany SSID nie jest dostępny
`WL_CONNECT_FAILED`	próba połączenia nie powiodła się
`WL_CONNECTION_LOST`	połączenie zostało utracone
`WL_DISCONNECTED`	brak aktywnego połączenia

Najprostsze oczekiwanie na połączenie

```
int status = WL_IDLE_STATUS;

while (status != WL_CONNECTED) {
  Serial.print("Laczenie z: ");
  Serial.println(ssid);

  status = WiFi.begin(ssid, pass);
  delay(3000);
}

Serial.println("Polaczono z Wi-Fi");
```

Na start — tak. W docelowym systemie — ostrożnie.

Taka pętla blokuje resztę programu. Później zastąpimy ją logiką stanów i ponownych prób bez zatrzymywania całego urządzenia.

Po połączeniu warto od razu wypisać diagnostykę

```
Serial.print("SSID: ");
Serial.println(WiFi.SSID());

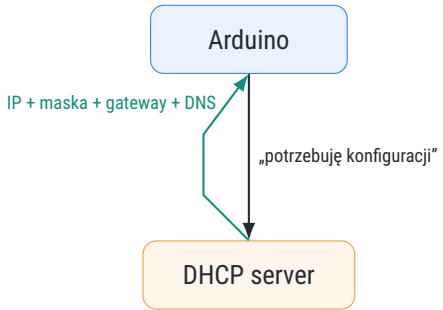
IPAddress ip = WiFi.localIP();
Serial.print("IP: ");
Serial.println(ip);

long rssi = WiFi.RSSI();
Serial.print("RSSI: ");
Serial.print(rssi);
Serial.println(" dBm");
```

To trzy pierwsze informacje, które sprawdzamy, gdy „sieć nie działa”.

DHCP — skąd Arduino dostaje adres IP?

- w typowej sieci nie wpisujemy ręcznie adresu IP,
- serwer DHCP przekazuje urządzeniu parametry sieci,
- zwykle są to: adres IP, maska, brama i DNS,
- adres jest przydzielany na określony czas — **lease**.



Arduino połączyło się z punktem dostępowym. Co jeszcze musi działać, aby połączyć się z usługą pod nazwą `io.adafruit.com`?

1. wystarczy samo poprawne hasło Wi-Fi
2. wystarczy znać adres MAC punktu dostępowego
3. DNS nie jest potrzebny, gdy w programie podajemy nazwę domenową
4. potrzebna jest poprawna konfiguracja IP i droga przez bramę, a przy nazwie domenowej także działający DNS

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Arduino połączyło się z punktem dostępowym. Co jeszcze musi działać, aby połączyć się z usługą pod nazwą `io.adafruit.com`?

1. wystarczy samo poprawne hasło Wi-Fi
2. wystarczy znać adres MAC punktu dostępowego
3. DNS nie jest potrzebny, gdy w programie podajemy nazwę domenową
4. **potrzebna jest poprawna konfiguracja IP i droga przez bramę, a przy nazwie domenowej także działający DNS**

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

2. IP, DHCP, DNS, porty i transport

Żeby zrozumieć HTTP i MQTT, trzeba wiedzieć, jak urządzenie znajduje serwer i jak dociera do konkretnej usługi.

IPv4 w praktyce: adres prywatny i NAT

- IPv4 ma 32 bity, np. `192.168.1.42`,
- w laboratorium najczęściej używamy adresów prywatnych: `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`,
- taki adres identyfikuje urządzenie **wewnątrz** lokalnej sieci.

Po co NAT?

Router może tłumaczyć wiele prywatnych adresów urządzeń na jeden publiczny adres używany po stronie Internetu. Dzięki temu laptop, telefon i Arduino mogą współdzielić jedno łącze.

Maska podsieci: czy odbiorca jest „u mnie”, czy za routerem?

Przykład /24

`192.168.1.42/24` maska `255.255.255.0`

Host B: 192.168.1.80/24

Pierwsze 24 bity części sieciowej są takie same. Urządzenia traktujemy jako należące do tej samej podsieci.

Host C: 192.168.2.80/24

Część sieciowa jest inna. Pakiet trzeba przekazać do routera — bramy domyślnej.

Maska nie jest dodatkowym adresem — mówi, **które bity IP opisują sieć**.

Brama i DNS — dwa różne zadania

Brama domyślna

Odpowiada na pytanie: **„gdzie wysłać pakiet, gdy odbiorca nie jest w mojej podsieci?”**

Najczęściej jest nią router, np. `192.168.1.1`.

DNS

Odpowiada na pytanie: **„jaki adres IP odpowiada nazwie `io.adafruit.com`?”**

DNS nie przesyła właściwych danych aplikacji — pomaga znaleźć adres serwera.

Typowa kolejność: **nazwa** → **DNS** → **IP** → **brama/router** → **serwer**.

Port: na tym samym serwerze może działać wiele usług

- adres IP wskazuje host,
- numer portu wskazuje konkretną usługę / punkt końcowy,
- para `adres:port` pozwala rozróżnić różne usługi na jednym serwerze,
- port sieciowy nie ma nic wspólnego z pinem Arduino.

Usługa	typowy port
HTTP	80
HTTPS	443
MQTT	1883
MQTT/TLS	8883
DNS	53

Klient, serwer i socket — trzy pojęcia, które spotkamy w kodzie

Klient

Rozpoczyna kontakt z usługą. Arduino publikujące dane do chmury jest klientem.

Serwer

Nasłuchuje na określonym porcie i obsługuje przychodzące połączenia.

Socket

Interfejs programistyczny reprezentujący punkt komunikacji. Biblioteka `WiFiClient` ukrywa większość jego szczegółów.

Połączenie TCP rozróżnia m.in. **IP i port obu stron**.

TCP — co dostajemy w zamian za większy narzut?

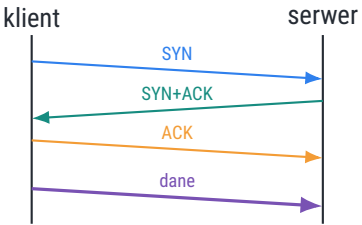
- uporządkowany **strumień bajtów**,
- wykrywanie utraty segmentów i retransmisję,
- kontrolę przepływu — nadawca nie powinien zalewać wolniejszego odbiorcy,
- kontrolę przeciążenia sieci,
- połączenie, które ma swój stan po obu stronach.

W naszym kursie

HTTP/1.1 i klasyczny MQTT najczęściej korzystają z TCP właśnie dlatego, że chcemy otrzymywać dane poprawnie i we właściwej kolejności.

Źródło: RFC 9293 — Transmission Control Protocol

TCP: najpierw handshake, potem strumień danych



- trzy pierwsze komunikaty tworzą logiczne połączenie TCP,
- dopiero potem aplikacja wysyła HTTP lub MQTT,
- TCP **nie zachowuje granic wiadomości**: jedno `write()` nie musi odpowiadać jednemu `read()`,
- granice danych definiuje dopiero protokół aplikacyjny.

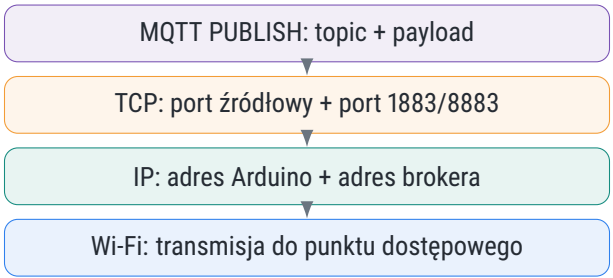
UDP i TCP — dobieramy transport do problemu

	TCP	UDP
Przed wysłaniem danych	handshake / połączenie	brak połączenia
Dostarczenie	retransmisja i kolejność	brak gwarancji protokołu
Granice danych	strumień bajtów	zachowane datagramy
Narzut	większy	mniejszy
Typowe przykłady	HTTP, MQTT	DNS, streaming, prosta telemetria czasu rzeczywistego

Nie ma „lepszego” transportu

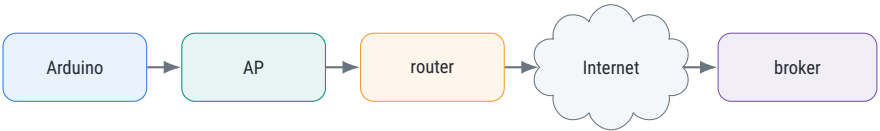
Jeżeli ważniejsza jest kompletność danych — często wybieramy TCP. Jeżeli ważniejszy jest mały narzut i świeżość kolejnych próbek — UDP może być lepszym wyborem.

Ten sam pomiar przechodzi przez kilka protokołów



Jedna wartość `23.6` jest opakowywana kolejnymi nagłówkami na różnych warstwach.

Jak pakiet trafia do brokera w chmurze?



- Arduino nie musi znać wszystkich routerów po drodze,
- zna swoją bramę i adres docelowy,
- kolejne routery przekazują pakiet dalej zgodnie z tablicami routingu.

Co jest potrzebne, aby Arduino połączyło się z brokerem pod nazwą io.adafruit.com?

1. tylko SSID sieci Wi-Fi
2. Wi-Fi + konfiguracja IP + DNS + osiągalność serwera i portu
3. tylko adres MAC brokera
4. koniecznie statyczny adres IP Arduino

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Co jest potrzebne, aby Arduino połączyło się z brokerem pod nazwą io.adafruit.com?

1. tylko SSID sieci Wi-Fi
2. **Wi-Fi + konfiguracja IP + DNS + osiągalność serwera i portu**
3. tylko adres MAC brokera
4. koniecznie statyczny adres IP Arduino

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

3. HTTP i JSON

Zanim przejdziemy do MQTT, poznajmy najpopularniejszy model komunikacji aplikacyjnej w Internecie.

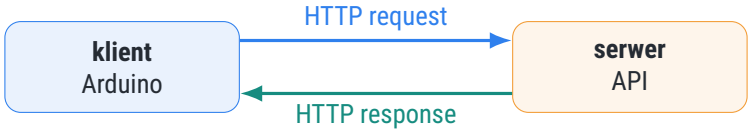
URL — co oznaczają poszczególne fragmenty?

Przykład

```
https://api.example.com:443/sensors/7?limit=10
```

`https`	schemat / protokół aplikacyjny + TLS
`api.example.com`	nazwa hosta
`443`	port
`/sensors/7`	ścieżka zasobu
`?limit=10`	parametr zapytania

HTTP — model request / response



Klient zadaje pytanie / wysyła dane. Serwer odsyła kod statusu, nagłówki i ewentualne body.

Anatomia żądania HTTP GET

Surowe żądanie

```
GET /api/temperature HTTP/1.1
Host: example.com
Accept: application/json
Connection: close
```

- pierwsza linia: metoda + ścieżka + wersja HTTP,
- kolejne linie: nagłówki,
- pusta linia kończy sekcję nagłówków.

Anatomia odpowiedzi HTTP

Przykładowa odpowiedź

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 20

{"temperature":23.6}
```

- kod 200 oznacza sukces żądania,
- `Content-Type` mówi, jak interpretować body,
- body może zawierać JSON, HTML, tekst, obraz itd.

Kody statusu HTTP — czy żądanie się udało?

Klasa	Znaczenie	Przykład
2xx	sukces	200 OK, 201 Created
3xx	przekierowanie	301 Moved Permanently
4xx	problem po stronie klienta	400 Bad Request, 401 Unauthorized, 404 Not Found
5xx	problem po stronie serwera	500 Internal Server Error, 503 Service Unavailable

Źródło: RFC 9110 — HTTP Semantics

Nagłówki HTTP — metadane o wiadomości

- `Host` — do jakiej nazwy serwera kierujemy żądanie,
- `Content-Type` — format body,
- `Content-Length` — długość body,
- `Authorization` / klucze API — dane uwierzytelniające,
- `Connection` — sposób obsługi połączenia.

Klucz API w nagłówku nadal jest sekretem

To, że nie znajduje się w URL, nie oznacza automatycznie, że jest zaszyfrowany — potrzebujemy TLS/HTTPS.

Metody HTTP i REST — jak opisujemy operację na zasobie?

GET	pobierz dane / reprezentację zasobu
POST	wyślij nowe dane do przetworzenia
PUT	ustaw lub zastąp wskazany zasób
DELETE	usuń wskazany zasób

Idea REST

Ścieżka wskazuje **zasób**, a metoda opisuje **co chcemy z nim zrobić**. Np. `/devices/7/temperature` wskazuje temperaturę urządzenia 7.

GET — odczyt

„Podaj ostatnią temperaturę”

```
GET /feeds/temp/last
```

POST — wysłanie danych

„Zapisz nowy pomiar”

```
POST /feeds/temp/data
```

Ważne

To konkretne API definiuje, jakie ścieżki istnieją, jakie body jest wymagane i co dokładnie oznacza dana operacja.

JSON — tekstowy format wymiany danych

Przykład

```
{"temperature": 23.6, "led": true}
```

- łatwy do odczytania przez człowieka,
- reprezentuje obiekty, tablice i wartości proste,
- często używany w API HTTP,
- może też być payloadem wiadomości MQTT.

Źródło: RFC 8259 — JSON

JSON — podstawowe typy wartości

Typ	Przykład
string	`"room-225"`
number	`23.6`
boolean	`true`
null	`null`
array	`[21.3, 21.5, 21.6]`
object	`{"temp":23.6, "unit":"C"}`

JSON — obiekt może zawierać zagnieżdżone dane

Przykład telemetrii

```
{
  "device": "uno-r4-01",
  "sensor": {"type": "LM35", "pin": "A0"},
  "values": [23.4, 23.5, 23.6],
  "online": true
}
```

Jeden komunikat może przenosić więcej informacji niż pojedyncza liczba.

Najczęstsze błędy w JSON

Błędne

```
{temperature: 23.6}
{'temperature': 23.6}
{"temperature": 23.6,}
```

Poprawne

```
{"temperature": 23.6}
```

- nazwy pól są stringami w podwójnych cudzysłowach,
- przecinki oddzielają kolejne elementy,
- nie zostawiamy przecinka po ostatnim elemencie.

Budowanie prostego JSON-a ręcznie w Arduino

```
float temp = 23.6;
int ledState = 1;

String json = "{\"temperature\":";
json += String(temp, 1);
json += ",\"led\":";
json += String(ledState);
json += "}";

Serial.println(json);
```

Wynik w Serial Monitor

```
{"temperature":23.6,"led":1}
```

Dla prostego przykładu — OK

Przy większych strukturach bezpieczniej użyć biblioteki do JSON, zamiast ręcznie pilnować cudzysłowów, przecinków i typów.

HTTP bez dodatkowej biblioteki — WiFiClient

```
WiFiClient client;

if (client.connect("example.com", 80)) {
  client.println("GET /api/value HTTP/1.1");
  client.println("Host: example.com");
  client.println("Connection: close");
  client.println();
}
```

- sami składamy tekst żądania HTTP,
- sami musimy potem odebrać i zinterpretować odpowiedź,
- dobrze pokazuje, że HTTP to po prostu protokół tekstowy przesyłany przez połączenie.

ArduinoHttpClient — wyższa warstwa abstrakcji

```
#include <ArduinoHttpClient.h>

WiFiClient wifi;
HttpClient http(wifi, "example.com", 80);

http.get("/api/value");

int statusCode = http.responseStatusCode();
String body = http.responseBody();
```

Biblioteka nadal używa klienta sieciowego, ale ukrywa ręczne składanie nagłówków HTTP.

HTTP POST z JSON — co trzeba wysłać?

Idea

```
POST /api/data HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: ...

{"temperature":23.6}
```

- metoda mówi „wysyłam dane”,
- `Content-Type` określa format,
- body zawiera właściwy JSON.

Sieć może blokować program tak samo jak delay()

- oczekiwanie na połączenie może trwać długo,
- serwer może nie odpowiedzieć,
- DNS może się nie udać,
- odczyt odpowiedzi bez timeoutu może zatrzymać logikę aplikacji.

W systemie wbudowanym

Komunikacja sieciowa jest tylko jednym z zadań. Urządzenie powinno nadal obsługiwać czujniki, przyciski i bezpieczeństwo lokalne.

HTTP a MQTT — dwa różne modele

	HTTP	MQTT
Model	request / response	publish / subscribe
Kto inicjuje?	klient wysyła żądanie	publisher publikuje, broker rozsyła
Stan połączenia	może być krótki	zwykle długotrwałe połączenie
Sterowanie w dół	klient musi odpytać / utrzymać kanał	naturalna subskrypcja
IoT	API, konfiguracja, REST	telemetria i zdarzenia

4. MQTT — publish / subscribe

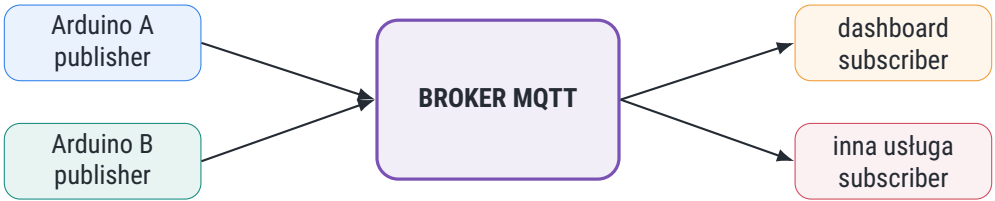
Lekki protokół komunikacyjny zaprojektowany tak, aby urządzenia i aplikacje nie musiały znać się bezpośrednio.

Po co MQTT w IoT?

- wiele urządzeń wysyła małe porcje danych,
- odbiorców może być wielu i mogą zmieniać się niezależnie,
- chcemy łatwo przesyłać zdarzenia w obu kierunkach,
- urządzenie nie powinno znać adresów wszystkich aplikacji, które korzystają z danych.

Rozwiązanie: wszystkie klienty łączą się z **brokerem**.

Broker rozdziela nadawców od odbiorców



Publisher i subscriber to role, nie typ urządzenia

- ten sam klient może jednocześnie **publikować** i **subskrybować**,
- Arduino może publikować temperaturę,
- to samo Arduino może subskrybować polecenie dla LED,
- dashboard może publikować polecenia i jednocześnie subskrybować telemetrię.

Lab 5

Arduino jest równocześnie publisherem feedu `temperature` i subscriberem feedu `led`.

Topic i payload — adres logiczny oraz treść

Przykład

topic: `room225/uno01/temperature`

payload: `23.6`

- topic mówi **czego dotyczy** wiadomość,
- payload to właściwa treść,
- broker używa topicu do decyzji, którym subskrybentom przekazać wiadomość.

Hierarchia topiców — ukośnik ma znaczenie organizacyjne

Przykładowe tematy

```
buildingA/floor2/room225/temperature
buildingA/floor2/room225/humidity
buildingA/floor2/room226/temperature
```

- dobrze zaprojektowane tematy tworzą czytelną hierarchię,
- łatwiej potem filtrować dane i dodawać kolejne urządzenia,
- unikamy topiców typu `data1`, `test2`, `x`.

Wildcard + — dokładnie jeden poziom

Subskrypcja

```
lab/225/+/temperature
```

Pasuje do:

- `lab/225/uno01/temperature`,
- `lab/225/uno02/temperature`.

Nie pasuje do:

- `lab/225/uno01/sensor/temperature` — dodatkowy poziom.

Wildcard # — wszystkie dalsze poziomy

Subskrypcja

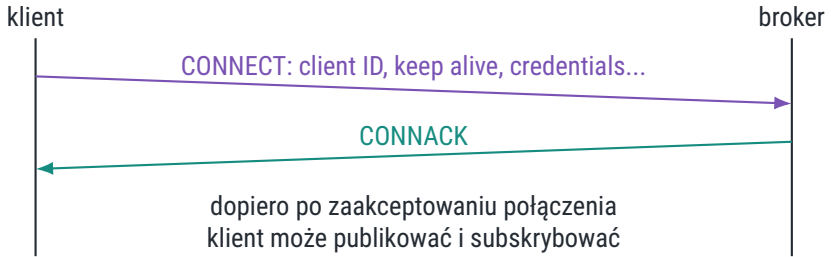
```
lab/225/#
```

Pasuje do wszystkich tematów zaczynających się od `lab/225/`, np. temperatury, LED, statusu i przyszłych sensorów.

Uwaga

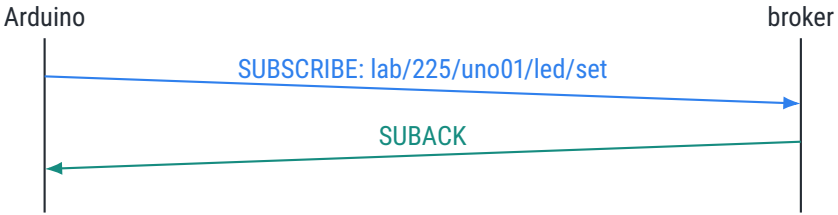
Wildcardy występują w **subskrypcjach**. Publisher publikuje do konkretnego topicu.

MQTT CONNECT — najpierw klient przedstawia się brokerowi

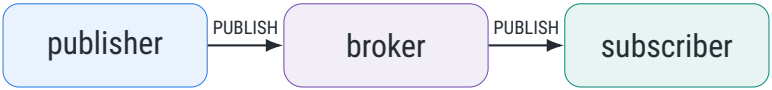


Źródło: OASIS MQTT 5.0

SUBSCRIBE — klient mówi, co chce odbierać



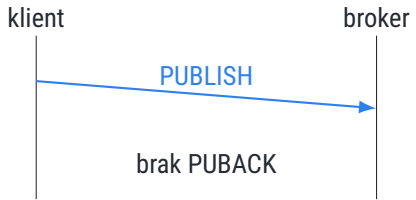
Broker zapisuje subskrypcję i od tej chwili przekazuje pasujące publikacje.



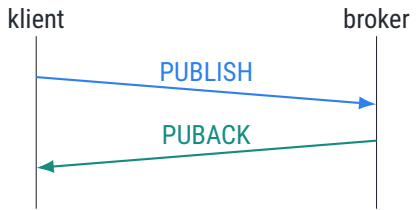
Broker sprawdza topic i przekazuje wiadomość wszystkim pasującym subskrybentom.

QoS 0 & 1— at most once

- publisher wysyła PUBLISH,
- brak potwierdzenia na poziomie MQTT,
- wiadomość może zginąć,
- najmniejszy narzut.



- PUBLISH jest potwierdzany przez PUBACK,
- brak potwierdzenia może spowodować retransmisję,
- wiadomość powinna dotrzeć co najmniej raz,
- aplikacja musi brać pod uwagę możliwość duplikatu.



QoS 2 — exactly once

- najbardziej rozbudowany mechanizm dostarczenia MQTT,
- wymaga dodatkowej wymiany pakietów i stanu po obu stronach,
- redukuje ryzyko wielokrotnego przetworzenia tej samej wiadomości,
- ma większy narzut — nie wybieramy go „bo jest najlepszy”, tylko gdy jest potrzebny.



Który QoS wybrać?

QoS	Własność	Przykład
0	szybko, bez potwierdzenia	częsta telemetria, gdzie pojedyncza utrata nie jest krytyczna
1	co najmniej raz	zdarzenia, które powinny dotrzeć; aplikacja toleruje duplikaty
2	dokładnie raz w sensie protokołu	rzadkie przypadki, gdy dodatkowy narzut jest uzasadniony

Dobieramy QoS do **konsekwencji utraty i duplikacji**, a nie do „ważności” danych w ogóle.

QoS 1 i duplikaty — dlaczego aplikacja powinna być odporna?

- klient wysyła wiadomość,
- broker ją odbiera, ale PUBACK ginie,
- klient nie wie, że broker ją dostał i wysyła ponownie,
- dlatego odbiorca może zobaczyć tę samą informację więcej niż raz.

Wniosek projektowy

Polecenie „ustaw LED=1” jest odporne na powtórzenie. Polecenie „zwiększ licznik o 1” już niekoniecznie.

Last Will and Testament — wiadomość na wypadek nagłego zniknięcia klienta

- klient ustala LWT podczas CONNECT,
- broker przechowuje topic i payload „ostatniej woli”,
- jeśli połączenie zrywa się niepoprawnie, broker może opublikować LWT,
- typowe zastosowanie: `device/status = offline`.

Po co?

Inne elementy systemu mogą wykryć, że urządzenie zniknęło bez wysłania poprawnego komunikatu „disconnect”.

Keep Alive — skąd broker wie, że klient jeszcze żyje?

- klient deklaruje interwał keep alive,
- jeśli przez pewien czas nie wysyła innych pakietów, może wysyłać PINGREQ,
- broker odpowiada PINGRESP,
- brak aktywności pozwala brokerowi wykryć martwe połączenie.

Dlatego biblioteka musi „pracować” w loop()

Jeśli zablokujemy program na długo, klient może nie obsłużyć keep alive ani przychodzących wiadomości.

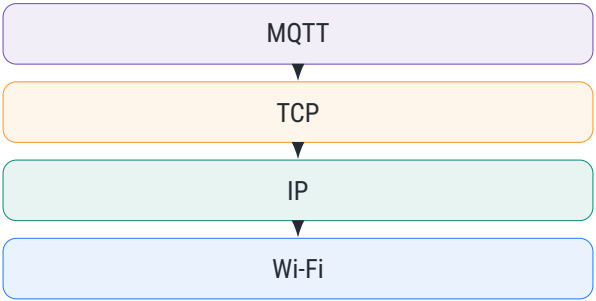
Client ID — broker musi rozróżniać klientów

- klient MQTT przedstawia się identyfikatorem podczas CONNECT,
- dwa aktywne klienty z tym samym ID mogą się wzajemnie rozłączać — zależnie od brokera,
- identyfikator powinien być stabilny i unikalny w obrębie brokera,
- nie wkładamy do niego sekretów — to identyfikator, nie hasło.

Najważniejsze pakiety MQTT

Pakiet	Rola
CONNECT / CONNACK	rozpoczęcie sesji
SUBSCRIBE / SUBACK	ustawienie subskrypcji
PUBLISH	wiadomość aplikacyjna
PUBACK	potwierdzenie QoS 1
PINGREQ / PINGRESP	utrzymanie / kontrola połączenia
DISCONNECT	poprawne zakończenie

MQTT zwykle działa na TCP



MQTT definiuje komunikację aplikacyjną; TCP dostarcza strumień bajtów między klientem i brokerem.

HTTP vs MQTT — wracamy do porównania z większą wiedzą

	HTTP	MQTT
Topologia	klient ↔ serwer	klient ↔ broker ↔ klient
Adresowanie danych	URL / zasób	topic
Odpowiedź	bezpośrednia, response	niezależne publikacje
Sterowanie asynchroniczne	mniej naturalne	naturalne przez subscribe
Stan połączenia	często krótszy	zwykle utrzymywany

Arduino ma odbierać polecenie sterujące możliwie szybko po zmianie wartości w chmurze. Który mechanizm najlepiej do tego pasuje?

1. cykliczny restart połączenia Wi-Fi
2. zmiana adresu IP po każdym poleceniu
3. MQTT z subskrypcją odpowiedniego topicu
4. zapisywanie poleceń wyłącznie na kartę SD

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Arduino ma odbierać polecenie sterujące możliwie szybko po zmianie wartości w chmurze. Który mechanizm najlepiej do tego pasuje?

1. cykliczny restart połączenia Wi-Fi
2. zmiana adresu IP po każdym poleceniu
3. **MQTT z subskrypcją odpowiedniego topicu**
4. zapisywanie poleceń wyłącznie na kartę SD

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Który mechanizm MQTT najlepiej pasuje do informacji „urządzenie nagle przestało być osiągalne”?

1. Last Will and Testament
2. retained temperature
3. zmiana SSID
4. HTTP 404

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Który mechanizm MQTT najlepiej pasuje do informacji „urządzenie nagle przestało być osiągalne”?

1. **Last Will and Testament**
2. retained temperature
3. zmiana SSID
4. HTTP 404

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

5. Adafruit IO — MQTT w praktyce

Laboratorium korzysta z wygodnej warstwy abstrakcji: feedów, dashboardu i biblioteki Adafruit IO Arduino.

Co w naszym systemie robi Adafruit IO?

- udostępnia usługę MQTT,
- przechowuje dane w **feedach**,
- pozwala wizualizować je na dashboardzie,
- dashboard może również generować wartości sterujące,
- biblioteka Arduino ukrywa część szczegółów topiców i MQTT.

Feed w Adafruit IO

Logiczny strumień danych przechowywany przez platformę, np. `temperature`.

Topic MQTT

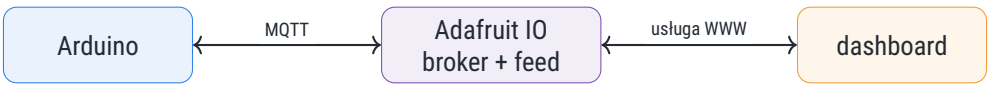
Nazwa używana przez broker MQTT do routowania wiadomości.

Publikacja do feedu przez MQTT oznacza wysłanie wiadomości do **powiązanego topicu**.

Dla feedu o kluczu temperature

```
username/feeds/temperature
```

- nazwa użytkownika identyfikuje właściciela,
- `feeds` wskazuje przestrzeń feedów,
- ostatni segment to klucz konkretnego feedu.



Przycisk na dashboardzie zapisuje wartość do feedu; broker dostarcza ją subskrybującemu Arduino.

AIO Key — traktujemy jak hasło

- klucz służy do uwierzytelniania do konta / usług Adafruit IO,
- nie umieszczamy go w sprawozdaniu ani publicznym repozytorium,
- najlepiej trzymać go w pliku z sekretami,
- wyciek klucza oznacza konieczność jego unieważnienia / zmiany.

Biblioteka Adafruit IO Arduino — co ukrywa?

- zestawienie połączenia z usługą,
- szczegóły topiców feedów,
- publikowanie wartości do feedu,
- odbieranie wiadomości i wywoływanie callbacków,
- utrzymanie połączenia i obsługę MQTT w tle biblioteki.

Dzięki temu kod laboratoryjny jest krótszy, ale warto wiedzieć, **co dzieje się pod spodem**.

Obiekt feedu — uchwyt do konkretnego strumienia danych

```
AdafruitIO_Feed *temperature = io.feed("temperature");
AdafruitIO_Feed *led = io.feed("led");
```

- `temperature` i `led` są wskaźnikami do obiektów feedów,
- biblioteka na ich podstawie buduje odpowiednie topiki MQTT,
- potem używamy metod takich jak `save()`, `onMessage()` i `get()`.

io.connect() — uruchomienie połączenia z usługą

```
Serial.print("Connecting to Adafruit IO");
io.connect();

while (io.status() < AIO_CONNECTED) {
  Serial.print(".");
  delay(500);
}

Serial.println();
Serial.println(io.statusText());
```

To przykład startowy

Blokująca pętla jest łatwa dydaktycznie. W systemie działającym ciągle lepiej zaprojektować ponowne łączenie tak, aby nie zatrzymywać całej aplikacji.

io.run() — jedna z najważniejszych linii w programie

```
void loop() {
  io.run();

  // pozostala logika programu
}
```

- biblioteka przetwarza przychodzące dane,
- utrzymuje komunikację z usługą,
- obsługuje mechanizmy keep alive / ping i stan połączenia,
- dlatego `loop()` nie powinno być długo blokowane.

Źródło: Adafruit IO Arduino — biblioteka i przykłady

Publikacja temperatury — pojedyncza linia na poziomie aplikacji

```
float temperatureC = 23.6;

temperature->save(temperatureC);
```

Pod spodem

Biblioteka zamienia wartość na dane zgodne z API Adafruit IO, publikuje je do właściwego topicu MQTT, a broker zapisuje punkt w feedzie i udostępnia go dashboardowi.

millis() — publikacja co 5 s bez blokowania odbioru wiadomości

- `loop()` może wykonywać się tysiące razy na sekundę,
- czujnik temperatury nie potrzebuje takiej częstotliwości,
- usługi chmurowe mają limity i zasoby.

```
const unsigned long interval = 5000;
unsigned long previousPublish = 0;

void loop() {
  io.run();

  unsigned long now = millis();
  if (now - previousPublish >= interval) {
    previousPublish = now;

    float temperatureC = readTemperature();
    temperature->save(temperatureC);
  }
}
```

`readTemperature()` to nasza funkcja odczytująca LM35 — jej definicja jest tutaj pominięta dla czytelności.

Subskrypcja — funkcja callback reaguje na wiadomość

```
void handleMessage(AdafruitIO_Data *data) {
  int state = data->toInt();

  if (state == 1) {
    digitalWrite(LED_PIN, HIGH);
  }
  else {
    digitalWrite(LED_PIN, LOW);
  }
}
```

Callback jest wywoływany, gdy biblioteka przetworzy nową wiadomość dla subskrybowanego feedu.

Jak „podłączyć” callback do feedu?

```
AdafruitIO_Feed *led = io.feed("led");

void setup() {
  pinMode(LED_PIN, OUTPUT);

  led->onMessage(handleMessage);
  io.connect();
}
```

- `onMessage()` rejestruje funkcję obsługi,
- nie wywołujemy `handleMessage()` sami,
- biblioteka wywoła ją po odebraniu nowej wartości.

led->get() — odtworzenie ostatniej wartości po starcie

```
while (io.status() < AIO_CONNECTED) {
  delay(500);
}

led->get();
```

- urządzenie po restarcie nie zna poprzedniego stanu LED,
- `get()` prosi usługę o aktualną / ostatnią wartość feedu,
- callback może dzięki temu ustawić wyjście zgodnie z ostatnim stanem zapisanym w chmurze.

Pełny węzeł IoT — część 1: definicje i setup()

```
#include "AdafruitIO_WiFi.h"
#include "arduino_secrets.h"

const int LED_PIN = 8;
const int TEMP_PIN = A0;

AdafruitIO_WiFi io(SECRET_IO_USERNAME, SECRET_IO_KEY,
                   SECRET_SSID, SECRET_PASS);

AdafruitIO_Feed *temperature = io.feed("temperature");
AdafruitIO_Feed *led = io.feed("led");

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);

  led->onMessage(handleMessage);
  io.connect();

  while (io.status() < AIO_CONNECTED) delay(500);
  led->get();
}
```

Pełny węzeł IoT — część 2: loop() i callback

```
const unsigned long interval = 5000;
unsigned long previousPublish = 0;

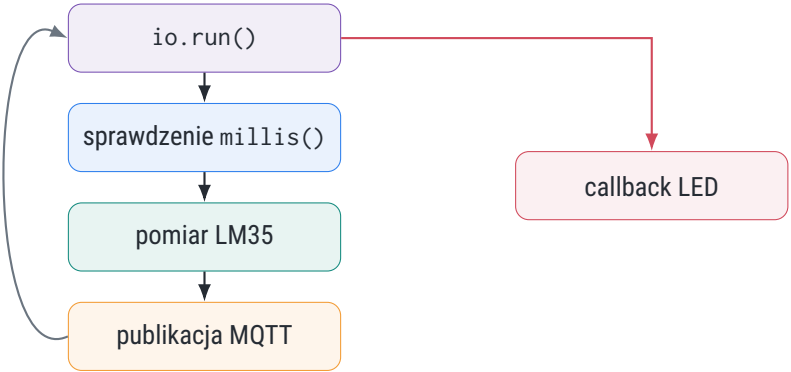
void loop() {
  io.run();

  unsigned long now = millis();
  if (now - previousPublish >= interval) {
    previousPublish = now;

    int raw = analogRead(TEMP_PIN);
    float voltage = raw * 5.0 / 1023.0;
    float temperatureC = voltage * 100.0;
    temperature->save(temperatureC);
  }
}

void handleMessage(AdafruitIO_Data *data) {
  int state = data->toInt();
  if (state == 1) digitalWrite(LED_PIN, HIGH);
  else digitalWrite(LED_PIN, LOW);
}
```

Ten program wykonuje kilka zadań „naraz”



Brak długiego `delay()` pozwala regularnie obsługiwać zarówno telemetrię, jak i polecenia.

Dlaczego w loop() powinno być regularnie wykonywane io.run()?

1. bo zwiększa rozdzielczość ADC
2. bo obsługuje komunikację, przychodzące wiadomości i utrzymanie połączenia
3. bo zmienia SSID sieci
4. bo generuje PWM dla diody

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Dlaczego w loop() powinno być regularnie wykonywane io.run()?

1. bo zwiększa rozdzielczość ADC
2. **bo obsługuje komunikację, przychodzące wiadomości i utrzymanie połączenia**
3. bo zmienia SSID sieci
4. bo generuje PWM dla diody

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

6. Niezawodność i bezpieczeństwo

Prawdziwy system IoT musi działać również wtedy, gdy Wi-Fi zniknie, serwer nie odpowie albo przyjdą błędne dane.

Sieć zawodzi — i to jest normalne

- użytkownik wyłącza router,
- urządzenie wychodzi poza zasięg,
- DHCP / DNS chwilowo nie odpowiada,
- połączenie TCP zostaje zerwane,
- broker wykonuje restart lub odrzuca logowanie,
- Internet działa wolno albo z dużymi opóźnieniami.

Projektujemy na awarie

Pytanie nie brzmi „czy połączenie kiedyś zniknie?”, tylko **„co zrobi urządzenie, kiedy zniknie?”**.

Utrata Wi-Fi — najpierw ją wykryj

```
if (WiFi.status() != WL_CONNECTED) {
  Serial.println("Brak Wi-Fi");
  // rozpocznij logike ponownego laczenia
}
```

- nie zakładamy, że połączenie z setup() będzie trwało wiecznie,
- stan połączenia można monitorować również w trakcie pracy.

Prosty reconnect — próby w odstępach czasu

```
const unsigned long retryInterval = 5000;
unsigned long lastRetry = 0;

void loop() {
  unsigned long now = millis();

  if (WiFi.status() != WL_CONNECTED && now - lastRetry >= retryInterval) {
    lastRetry = now;
    WiFi.begin(ssid, pass);
  }

  // pozostala logika nadal moze dzialac
}
```

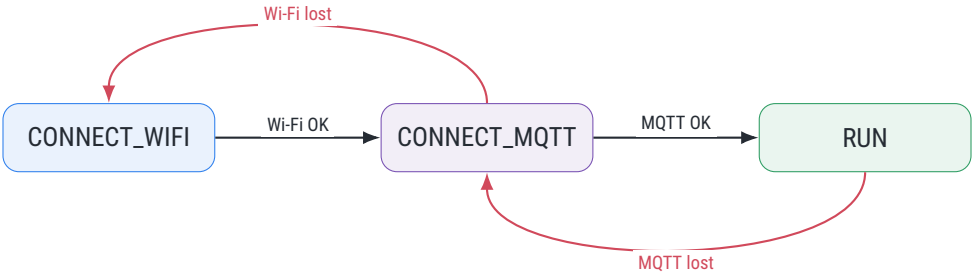
Backoff — po kolejnych błędach nie próbuj coraz szybciej

- pierwsza ponowna próba może być szybka,
- po kolejnych niepowodzeniach wydłużamy odstęp,
- zmniejsza to obciążenie urządzenia, AP i serwera,
- po sukcesie wracamy do krótkiego interwału startowego.

Przykład

1 s $\rightarrow$ 2 s $\rightarrow$ 4 s $\rightarrow$ 8 s $\rightarrow$ maks. 30 s

Najczytelniej: maszyna stanów połączenia



Stan mówi, **co próbujemy osiągnąć teraz**, zamiast ukrywać wszystko w blokujących pętlach.

Timeout — każda operacja sieciowa powinna mieć granicę czasu

- połączenie może nigdy nie zostać zestawione,
- odpowiedź może nie przyjść,
- czekanie bez końca jest szczególnie niebezpieczne w systemie sterującym,
- po timeout przechodzimy do obsługi błędu i późniejszej próby.

Zasada

„Nie dostałem odpowiedzi” jest poprawnym stanem systemu i musi mieć przewidzianą obsługę.

Co zrobić z pomiarem, gdy chmura jest niedostępna?

- pominąć — jeśli pojedyncza próbka nie jest ważna,
- zapisać ostatnią wartość i wysłać później,
- buforować kilka / kilkaset próbek w pamięci,
- zapisywać lokalnie na nośniku,
- decyzja zależy od wymagań systemu i dostępnej pamięci.

Nie ma jednej dobrej odpowiedzi

Inaczej projektujemy termometr pokojowy, inaczej rejestrator pomiarów naukowych.

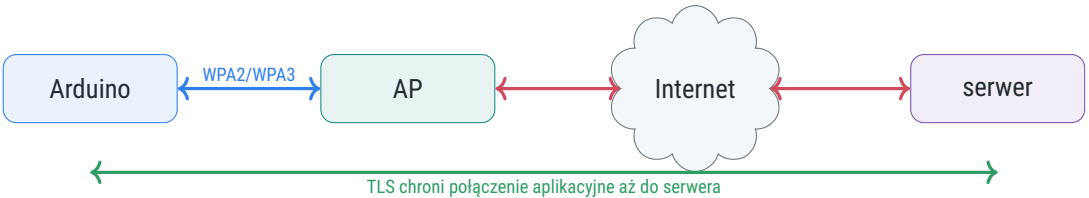
Dane przychodzące z sieci są wejściem — trzeba je walidować

```
int state = data->toInt();

if (state == 0) {
  digitalWrite(LED_PIN, LOW);
}
else if (state == 1) {
  digitalWrite(LED_PIN, HIGH);
}
else {
  Serial.println("Niepoprawne polecenie");
}
```

Nie zakładamy, że każda wiadomość z brokera ma sensowną wartość.

WPA2/WPA3 i TLS — dwie różne warstwy ochrony



WPA2/WPA3

Chroni **lokalne łącze radiowe** między urządzeniem a punktem dostępowym. Nie szyfruje automatycznie dalszej drogi przez Internet.

TLS

Szyfruje dane aplikacyjne w drodze do serwera i pozwala klientowi zweryfikować jego tożsamość na podstawie certyfikatu. HTTPS = HTTP + TLS; MQTT także może działać przez TLS.

Certyfikat — po co klientowi lista zaufania?

- serwer przedstawia certyfikat,
- klient sprawdza, czy certyfikat jest wiarygodny i pasuje do nazwy hosta,
- bez weryfikacji certyfikatu szyfrowanie może nie chronić przed atakiem typu man-in-the-middle,
- urządzenia wbudowane muszą mieć sposób przechowywania odpowiednich certyfikatów / CA.

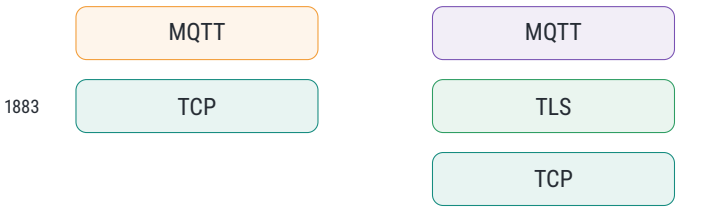
MQTT na porcie 1883 i MQTT/TLS na 8883 — co się zmienia?

1883 — typowo MQTT bez TLS

- najpierw tworzymy połączenie TCP,
- po nim przesyłamy pakiety MQTT,
- sam MQTT **nie szyfruje** topicu ani payloadu.

8883 — typowo MQTT przez TLS

- TCP nadal znajduje się pod spodem,
- między TCP i MQTT dochodzi warstwa TLS,
- TLS szyfruje transmisję i weryfikuje serwer.



Numery 1883 i 8883 są **konwencją często stosowaną przez brokerów**; konkretna usługa może używać innych portów.

Minimalne uprawnienia — klucz nie powinien otwierać wszystkiego

- urządzenie powinno mieć tylko te uprawnienia, których potrzebuje,
- osobny token dla urządzenia ogranicza skutki wycieku,
- nie używamy konta administratora jako prostego „hasła do sensora”,
- gdy usługa wspiera ACL, ograniczamy dostęp do konkretnych topiców / feedów.

ACL = Access Control List

To zestaw reguł mówiących, **kto może wykonać jaką operację na jakim zasobie**, np. urządzenie może publikować do `device/01/temperature`, ale nie może czytać ani zmieniać topiców innych urządzeń.

Zasada least privilege

Każdy element systemu dostaje **najmniejszy zestaw uprawnień** potrzebny do wykonania swojej roli.

Aktualizacje firmware — bezpieczeństwo nie kończy się w dniu uruchomienia

- biblioteki i firmware modułu komunikacyjnego mogą zawierać poprawki błędów,
- urządzenie pracujące latami wymaga planu aktualizacji,
- aktualizacja nie może przypadkowo ujawniać sekretów,
- zdalne OTA jest wygodne, ale samo musi być zabezpieczone.

Źródło: Arduino OTA — przykład mechanizmu aktualizacji

Które zdanie jest prawdziwe?

1. WPA2 szyfruje automatycznie MQTT przez cały Internet
2. TLS chroni komunikację aplikacyjną niezależnie od lokalnego zabezpieczenia Wi-Fi
3. plik `arduino_secrets.h` szyfruje hasło w pamięci programu
4. QoS 2 zastępuje uwierzytelnianie użytkownika

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Które zdanie jest prawdziwe?

1. WPA2 szyfruje automatycznie MQTT przez cały Internet
2. **TLS chroni komunikację aplikacyjną niezależnie od lokalnego zabezpieczenia Wi-Fi**
3. plik `arduino_secrets.h` szyfruje hasło w pamięci programu
4. QoS 2 zastępuje uwierzytelnianie użytkownika

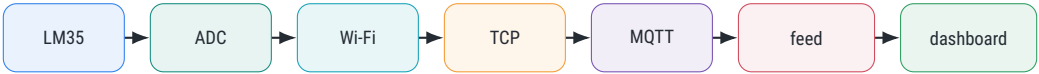
?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

7. Projektujemy kompletny węzeł IoT

Na koniec składamy wszystkie warstwy w jeden system i uczymy się diagnozować go krok po kroku.

Kompletny system — od napięcia LM35 do wykresu w przeglądarce



Każdy blok można testować osobno. To najlepsza strategia diagnostyczna.

Dwie drogi w systemie IoT: telemetria i sterowanie

Telemetria: urządzenie → chmura

1. ADC odczytuje napięcie LM35.
2. Program przelicza je na temperaturę.
3. Biblioteka buduje publikację MQTT.
4. Dane idą po TCP/IP przez Wi-Fi.
5. Broker zapisuje wartość w feedzie.
6. Dashboard aktualizuje wykres.

Sterowanie: chmura → urządzenie

1. Użytkownik zmienia Toggle na dashboardzie.
2. Adafruit IO zapisuje nową wartość feedu `led`.
3. Broker wysyła publikację do subskrybentów.
4. Arduino odbiera ją w aktywnym połączeniu MQTT.
5. `io.run()` uruchamia callback.
6. Callback ustawia stan pinu D8.

Gdzie szukać błędu? — diagnostyka od dołu

1. Czy moduł widzi SSID?
2. Czy `WiFi.status()` daje `WL_CONNECTED`?
3. Czy Arduino dostało sensowny adres IP?
4. Czy działa DNS / czy osiągalna jest usługa?
5. Czy uwierzytelnienie do MQTT się udaje?
6. Czy topic / feed jest poprawny?
7. Czy callback jest zarejestrowany?
8. Czy `io.run()` wykonuje się regularnie?

Nie zaczynaj od zmiany wszystkiego naraz

Testuj kolejne warstwy i zapisuj, co już na pewno działa.

Co jeśli Internet zniknie, ale lokalne sterowanie jest krytyczne?

- lokalne czujniki nadal działają,
- lokalne reguły bezpieczeństwa powinny nadal działać,
- telemetrię można oznaczyć jako offline / buforować,
- po powrocie sieci próbujemy wznowić połączenie.

Zła architektura

„Jeśli chmura nie działa, nie potrafię wyłączyć grzałki lokalnie.”

Przykład projektowy: inteligentna szklarnia

- czujniki: temperatura + wilgotność gleby,
- aktuatory: wentylator + pompa,
- telemetria: publikacja co 30 s,
- sterowanie zdalne: tryb AUTO / MANUAL,
- lokalne zabezpieczenie: maksymalny czas pracy pompy niezależny od chmury,
- stan online/offline publikowany do osobnego topicu.

Szklarnia — przykładowe topiki

Telemetria

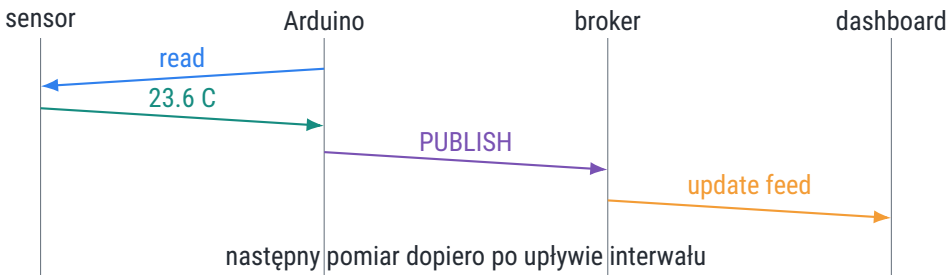
```
greenhouse/01/temperature
greenhouse/01/soil
greenhouse/01/status
```

Sterowanie

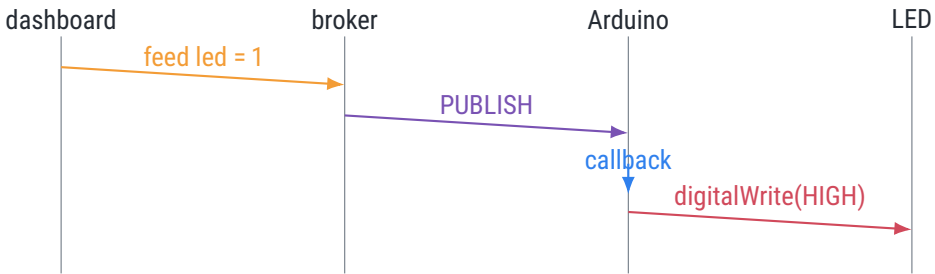
```
greenhouse/01/mode/set
greenhouse/01/pump/set
greenhouse/01/fan/set
```

Rozdzielenie `/set` i `/status` pomaga odróżnić żądanie od rzeczywistego stanu urządzenia.

Sekwencja pomiaru w dobrze działającym węźle



Sekwencja polecenia sterującego



Dziękuję za uwagę!