Komunikacja sieciowa

Wi-Fi, TCP/IP, JSON i MQTT



Podstawy Systemów Mikroprocesorowych ▪ Wykład 6
dr inż. Dariusz Tefelski

Od czujnika do usługi sieciowej – co dziś chcemy zrozumieć?



Wykłady 4–5 kończyły się na komunikacji z **peryferium**. Dziś dane opuszczają mikrokontroler i trafiają do **innego hosta w sieci**.

Po wykładzie chcemy umieć:

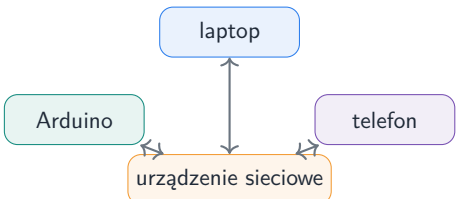
- odróżnić Wi-Fi od Internetu,
- wyjaśnić MAC, IP, port, DNS i DHCP,
- rozróżnić TCP i UDP,
- połączyć Raspberry Pi Pico 2 W z WLAN,
- przygotować dane w formacie JSON,
- opublikować i odebrać dane przez MQTT.

Minimum z sieci komputerowych

Tylko pojęcia potrzebne do zrozumienia kodu na mikrokontrolerze

Co to jest sieć komputerowa?

- sieć to grupa urządzeń, które potrafią **wymieniać dane**,
- urządzeniem może być komputer, telefon, drukarka, Raspberry Pi Pico 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.

Internet

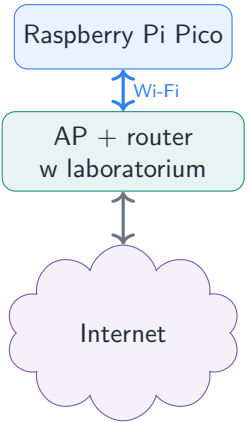
- to ogromna „sieć sieci”,
- łączy wiele niezależnych sieci lokalnych,
- pozwala dotrzeć do usług działających poza naszym laboratorium.

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.

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 Raspberry Pi Pico łączy się radiowo z AP, a router daje mu drogę dalej.



Co naprawdę dzieje się po `wlan.connect()`?

skanowanie AP → uwierzytelnienie → asocjacja z AP → DHCP adres IP → gotowe do IP

Wi-Fi nie jest synonimem Internetu

1. Raspberry Pi Pico 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

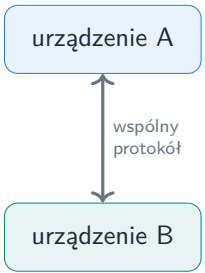
Komunikat „Wi-Fi connected" mówi tylko, że udało się wykonać **pierwszy etap**. Nie dowodzi jeszcze, że działa DNS, połączenie z serwerem ani usługa IoT.

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.



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.

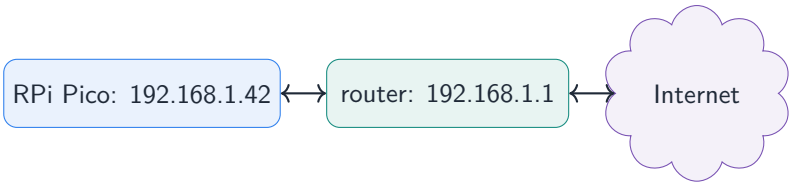
Przykład

`192.168.1.42`

To nie jest „numer Raspberry Pi Pico 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

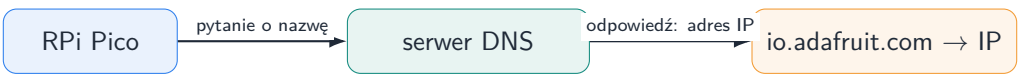
- po dołączeniu do Wi-Fi Raspberry Pi Pico jeszcze nie wie, jaki ma mieć adres IP,
- **DHCP** to mechanizm, dzięki któremu urządzenie może automatycznie dostać konfigurację,
- serwer DHCP zwykle działa w routerze,
- oprócz adresu IP może przekazać m.in. informację o bramie i serwerze DNS.

W praktyce

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

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?”.



DNS: `getaddrinfo()` przygotowuje adres dla socketu

```
import socket

info = socket.getaddrinfo(
    'example.org', # host
    80, # port
    0, # rodzina adresow
    socket.SOCK_STREAM # typ
        socketu
)

addr = info[0][-1]
print(addr)
```

Znaczenie argumentów:

- `host` — nazwa domenowa lub adres hosta,
- `port` — numer usługi, tutaj 80,
- `af=0` — nie narzucamy rodziny adresów; funkcja może zwrócić obsługiwane warianty,
- `SOCK_STREAM` — filtrujemy wyniki do adresów odpowiednich dla połączenia strumieniowego, czyli TCP.

Wynik to lista krotek:

```
(family, type, proto, canonname, sockaddr)
```

`info[0][-1]` wybiera pole `sockaddr` z pierwszego wyniku — właśnie ten obiekt przekazujemy później do `connect()`.

Klient i serwer — dwie role w rozmowie

Klient

- rozpoczyna kontakt,
- prosi o dane albo usługę,
- RPi Pico 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.

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**.

TCP i UDP: dwa różne sposoby dostarczania danych

	TCP	**UDP**
połączenie	najpierw zestawiamy połączenie	wysyłamy niezależne datagramy
niezawodność	kolejność, retransmisje, kontrola dostarczenia	brak gwarancji dostarczenia i kolejności
narzut	większy	mały
typowe użycie	HTTP, MQTT, SSH	DNS, streaming, proste telemetrie czasu rzeczywistego
MicroPython	`SOCK_STREAM`	`SOCK_DGRAM`

TCP nie „wie", gdzie kończy się komunikat aplikacji – widzi **strumień bajtów**. Granice komunikatów definiuje dopiero protokół wyższego poziomu, np. HTTP lub MQTT.

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 skupimy się na **MQTT**; HTTP pozostaje użytecznym modelem porównawczym request/response.

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

`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

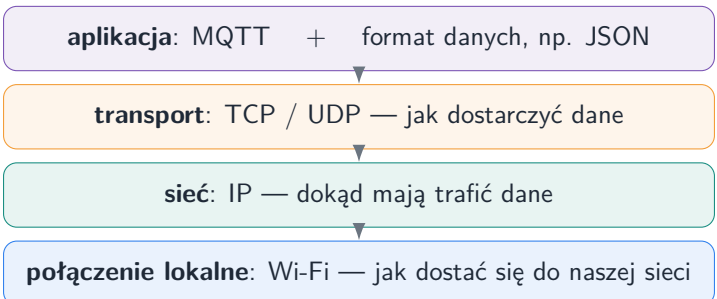
```
{
    "temperature": 21.7,
    "humidity": 48,
    "alarm": false
}
```

- **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.

Socket: programowy koniec komunikacji sieciowej



Socket łączy protokół transportowy z parą **adres IP + port**. Program czyta i zapisuje bajty prawie jak do pliku.

Klient TCP:

1. rozwiąż nazwę DNS,
2. utwórz socket,
3. `connect()`,
4. `write()`/`read()`,
5. `close()`.

Serwer TCP: `bind()` → `listen()` → `accept()`.

Pytanie kontrolne 1

RPi Pico ma poprawny adres IP i potrafi połączyć się z serwerem po jego adresie IP, ale nie działa nazwa `api.example.org`. Co najlepiej sprawdzić najpierw?

A. Czy brama domyślna ma poprawny adres

B. Czy DNS poprawnie rozwiązuje nazwę `api.example.org`

C. Czy RSSI połączenia Wi-Fi jest wystarczająco wysokie

D. Czy wybrano właściwy port usługi na serwerze

Pytanie kontrolne 1

RPi Pico ma poprawny adres IP i potrafi połączyć się z serwerem po jego adresie IP, ale nie działa nazwa `api.example.org`. Co najlepiej sprawdzić najpierw?

A. Czy brama domyślna ma poprawny adres

B. Czy DNS poprawnie rozwiązuje nazwę `api.example.org`

C. Czy RSSI połączenia Wi-Fi jest wystarczająco wysokie

D. Czy wybrano właściwy port usługi na serwerze

Wi-Fi w RPi Pico 2 W

Od radia CYW43439 do poprawnego adresu IP

Co właściwie realizuje Wi-Fi w RPi Pico 2 W?



RPi Pico 2 W używa zewnętrznego układu bezprzewodowego **CYW43439**, połączonego z RP2350 przez wewnętrzny interfejs SPI. Programista nie obsługuje jednak tego SPI bezpośrednio – korzysta z `network.WLAN`.

- Wi-Fi 802.11n,
- pasmo 2.4 GHz,
- wbudowana antena,
- tryb station i access point.

To inny SPI niż interfejsy SPI0/SPI1 używane przez studenta do peryferiów.

Station i Access Point – RPi Pico może pełnić dwie role

STA – station



RPi Pico **dołącza do istniejącej sieci**. To typowy tryb dla IoT.

AP – access point



RPi Pico **tworzy własną sieć Wi-Fi**. Klient może połączyć się bez routera.

Tryb AP sam w sobie **nie oznacza dostępu do Internetu**; RPi Pico musiałoby dodatkowo routować ruch do innego interfejsu.

Zanim się połączymy, możemy przeskanować otoczenie

```
import network

wlan = network.WLAN()
wlan.active(True)

for ap in wlan.scan():
    ssid, bssid, channel, rssi, \
        security, hidden = ap
    print(ssid, channel, rssi)
```

`scan()` zwraca m.in.:

- **SSID** – nazwę sieci,
- BSSID – MAC punktu dostępowego,
- kanał,
- **RSSI** – poziom odebranego sygnału,
- tryb zabezpieczeń,
- informację, czy SSID jest ukryte.

Skanowanie nie daje jeszcze adresu IP i nie oznacza połączenia z siecią.

Łączenie z Wi-Fi: dodajemy timeout i kontrolę błędów

```
import network
from time import ticks_ms, ticks_diff, sleep_ms

SSID = 'LabWiFi'
PASSWORD = 'haslo'

wlan = network.WLAN()
wlan.active(True)
wlan.connect(SSID, PASSWORD)

start = ticks_ms()
while not wlan.isconnected():
    status = wlan.status()
    if status in (network.STAT_WRONG_PASSWORD,
                  network.STAT_NO_AP_FOUND,
                  network.STAT_CONNECT_FAIL):
        raise OSError('Wi-Fi error: {}'.format(status))
    if ticks_diff(ticks_ms(), start) > 10_000:
        wlan.disconnect()
        raise OSError('Wi-Fi timeout')
    sleep_ms(100)

print(wlan.ipconfig('addr4'))
```

W RP2 po `connect()` ponawianie może trwać bez końca, dlatego aplikacja powinna mieć **własny timeout**.

Po połączeniu sprawdzamy, co naprawdę otrzymaliśmy

```
print(wlan.isconnected())
print(wlan.ipconfig('addr4'))
print(wlan.ifconfig())
print(wlan.config('mac'))
print(wlan.status('rssi'))
```

Warto umieć odczytać:

- własny adres IPv4,
- maskę/prefix sieci,
- bramę,
- DNS,
- MAC interfejsu,
- RSSI połączenia.

`isconnected()` oznacza w trybie STA nie tylko asocjację z AP, ale też posiadanie poprawnego adresu IP.

RPi Pico 2 W jako Access Point

```
import network

ap = network.WLAN(network.WLAN.IF_AP)
ap.config(ssid='RPi-Pico-Lab')
ap.config(max_clients=4)
ap.active(True)

print(ap.ipconfig('addr4'))
```

- telefon/laptop może połączyć się bez istniejącego routera,
- przydatne do lokalnej konfiguracji urządzenia,
- na RPi Pico można wtedy uruchomić np. prosty serwer HTTP,
- dostęp do Internetu **nie pojawia się automatycznie**.

Diagnostyka Wi-Fi: sprawdzamy warstwy po kolei

Objaw	Co sprawdzić?	Narzędzie/API
nie widać sieci	SSID, pasmo 2.4 GHz, zasięg	`wlan.scan()`
nie łączy z AP	hasło, zabezpieczenia, status	`wlan.status()`
połączone, brak IP	DHCP	`ipconfig()/ifconfig()`
IP działa, nazwa nie	DNS	`getaddrinfo()`
DNS działa, usługa nie	port, serwer, firewall	`socket.connect()`
połączenie zrywa się	RSSI, zasilanie, timeouty	`status('rssi')`

Nie zaczynamy od przepisywania całego programu. Najpierw ustalamy, **na którym etapie komunikacja przestaje działać**.

Minimalny klient TCP: DNS → socket → połączenie

```
import socket

addr = socket.getaddrinfo('example.org', 80, 0,
    socket.SOCK_STREAM)[0][-1]

s = socket.socket()
s.settimeout(5)
s.connect(addr)

s.write(b'hello')
data = s.read(20)
s.close()
```

- `SOCK_STREAM` wybiera TCP,
- `settimeout()` zapobiega nieskończonemu blokowaniu,
- socket przesyła **bajty**, nie Pythonowe obiekty,
- protokół aplikacji musi określić, co te bajty znaczą.

Samo wysłanie `b'hello'` do portu 80 nie jest poprawnym HTTP.

Pytanie kontrolne 2

Co oznacza `wlan.isconnected() == True` w trybie station?

A. RPi Pico widzi co najmniej jedną sieć Wi-Fi

B. RPi Pico automatycznie ma dostęp do każdego serwera w Internecie

C. RPi Pico jest połączone z AP i ma poprawną konfigurację IP

D. DNS na pewno działa

Pytanie kontrolne 2

Co oznacza `wlan.isconnected() == True` w trybie station?

A. RPi Pico widzi co najmniej jedną sieć Wi-Fi

B. RPi Pico automatycznie ma dostęp do każdego serwera w Internecie

C. RPi Pico jest połączone z AP i ma poprawną konfigurację IP

D. DNS na pewno działa

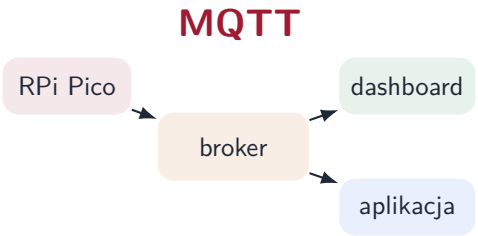
MQTT

Publish/subscribe zamiast bezpośredniego pytania serwera

Dlaczego IoT często używa MQTT zamiast samych requestów HTTP?



klient musi znać serwer i bezpośrednio wysłać do niego żądanie.



nadawca publikuje temat do brokera; odbiorcy subskrybują interesujące ich tematy.

Nadawca i odbiorca nie muszą znać swoich adresów – obie strony znają **broker**.

Topic organizuje dane – nie jest adresem IP

Przykładowa hierarchia:

```
lab/room1/temperature
lab/room1/humidity
lab/room1/led/set
lab/room2/temperature
```

Broker dopasowuje opublikowany **topic** do subskrypcji.

Wildcards w subskrypcji:

- `+` – dokładnie jeden poziom,
- `#` – wszystkie dalsze poziomy.

Przykład:

```
lab/+/temperature
```

odbierze temperatury ze wszystkich pomieszczeń na tym poziomie.

QoS, retained i keepalive – trzy praktyczne mechanizmy MQTT

QoS 0	"at most once" – mały narzut, brak potwierdzenia na poziomie MQTT
QoS 1	"at least once" – potwierdzenie; możliwy duplikat
QoS 2	"exactly once" – bardziej złożona wymiana
retained	broker zachowuje ostatnią wiadomość tematu i może wysłać ją nowemu subskrybentowi
keepalive	klient okresowo utrzymuje/monitoruje połączenie z brokerem

Biblioteka `umqtt.simple` z micropython-lib obsługuje **QoS 0 i 1**; QoS 2 celowo pominięto, aby ograniczyć złożoność i rozmiar kodu.

Pierwsza publikacja przez `umqtt.simple`

```
from umqtt.simple import MQTTClient

client = MQTTClient(
    'pico-01',
    '192.168.1.20',
    port=1883,
    keepalive=60
)

client.connect()
client.publish(
    b'lab/room1/temperature',
    b'23.4',
    qos=0
)
client.disconnect()
```

Pakiet można doinstalować jednorazowo przez `mip.install('umqtt.simple')`.

- `pico-01` – identyfikator klienta,
- adres – broker MQTT,
- topic i payload w `umqtt.simple` są bajtami,
- 1883 to typowy port MQTT bez TLS.

Publikacja nie wymaga, aby dashboard był aktualnie uruchomiony. RPi Pico komunikuje się z brokerem.

Subskrypcja: wiadomość trafia do callbacku

```
from umqtt.simple import MQTTClient
from machine import Pin
from time import sleep_ms

led = Pin('LED', Pin.OUT)

def on_message(topic, msg):
    if msg == b'ON':
        led.on()
    elif msg == b'OFF':
        led.off()

client = MQTTClient('pico-01', '192.168.1.20')
client.set_callback(on_message)
client.connect()
client.subscribe(b'lab/room1/led/set')

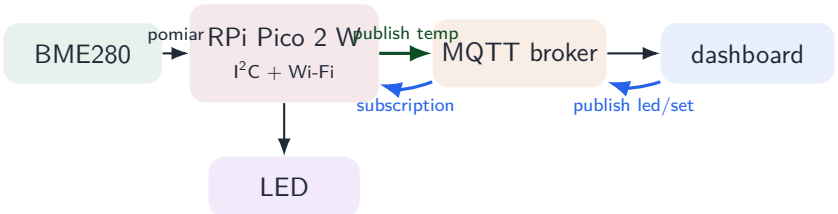
while True:
    client.check_msg()
    sleep_ms(50)
```

`check_msg()`:

- sprawdza, czy broker przesłał wiadomość,
- jeśli tak – wywołuje callback,
- jeśli nie – wraca od razu.

`wait_msg()` jest wersją blokującą. Gdy program ma też mierzyć czujniki, zwykle wygodniejszy jest `check_msg()`.

Pełny przykład IoT: dane w jedną stronę, komenda w drugą



Jedno urządzenie może jednocześnie być **publisherem** wyników i **subscriberem** komend. Broker rozdziela oba strumienie według tematów.

Sieć się rozłączy – dobry program musi to traktować jako normalny przypadek

Co może zniknąć?

- Wi-Fi / AP,
- DHCP lub DNS,
- połączenie TCP,
- broker MQTT,
- dostęp do Internetu.

Wzorzec obsługi:

1. wykryj błąd/timeout,
2. zamknij uszkodzony socket,
3. sprawdź/reconnect Wi-Fi,
4. odtwórz połączenie aplikacyjne,
5. ponów operację tylko wtedy, gdy ma to sens.

`umqtt.robust` istnieje jako rozszerzenie `umqtt.simple` i dodaje automatyczne ponawianie połączenia przy błędach sieciowych.

Bezpieczeństwo: sieć daje wygodę, ale też nową powierzchnię ataku

Minimum dla projektu:

- nie publikuj haseł w repozytorium,
- nie wystawiaj bez potrzeby portów do Internetu,
- uwierzytelniaj dostęp do brokera/usługi,
- stosuj TLS, gdy przesyłasz dane przez niezaufaną sieć,
- aktualizuj firmware/biblioteki.

Ważne rozróżnienie:

`Wi-Fi z hasłem` chroni przede wszystkim **dostęp do lokalnej sieci bezprzewodowej**.

Nie zastępuje szyfrowania end-to-end pomiędzy RPi Pico a serwerem.

W systemie embedded zawsze sprawdzamy, czy

konkretna biblioteka i build poprawnie weryfikują certyfikaty TLS.

Pytanie kontrolne 3

RPi Pico publikuje temperaturę, a wiele dashboardów ma ją odbierać bez znajomości adresu IP urządzenia i bez bezpośredniego odpytywania RPi Pico. Który model najlepiej pasuje?

A. MQTT: publisher – broker – subscriber

B. HTTP: dashboard cyklicznie odpytuje serwer uruchomiony na RPi Pico

C. Raw TCP: RPi Pico utrzymuje bezpośrednie połączenie TCP z dashboardem

D. UDP: RPi Pico wysyła datagramy bezpośrednio na adres IP i port dashboardu

Pytanie kontrolne 3

RPi Pico publikuje temperaturę, a wiele dashboardów ma ją odbierać bez znajomości adresu IP urządzenia i bez bezpośredniego odpytywania RPi Pico. Który model najlepiej pasuje?

A. MQTT: publisher – broker – subscriber

B. HTTP: dashboard cyklicznie odpytuje serwer uruchomiony na RPi Pico

C. Raw TCP: RPi Pico utrzymuje bezpośrednie połączenie TCP z dashboardem

D. UDP: RPi Pico wysyła datagramy bezpośrednio na adres IP i port dashboardu

Najważniejsze pojęcia + źródła techniczne

- Wi-Fi daje lokalne połączenie radiowe; Internet wymaga routingu IP.
- MAC identyfikuje interfejs lokalnie, IP host, a port usługę.
- DHCP przydziela konfigurację; DNS tłumaczy nazwy na adresy.
- TCP daje niezawodny strumień bajtów; UDP wysyła datagramy bez takiej gwarancji.
- HTTP jest przykładem request/response; w IoT skupiamy się na MQTT: publish/subscribe przez broker.
- W programie sieciowym timeout i reconnect są elementem normalnej logiki, nie „awarią wyjątkową”.

Źródła techniczne:

Raspberry Pi – dokumentacja serii RPi Pico / RPi Pico 2 W:
`https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html`

MicroPython v1.29.0 – RP2, `network.WLAN`, `socket`:
`https://docs.micropython.org/en/v1.29.0/`

micropython-lib – `requests`, `umqtt.simple`:
`https://github.com/micropython/micropython-lib`