AstroCamp／衛星開発ゼミ

電子工作からネットワークへ

TCP/IPの基礎とPico WによるWiFi通信

Raspberry Pi Pico C++ SDK で、PC上のPythonスクリプトと通信する

Week5

AGENDA

本日お話しすること

00 **導入**

シリアル通信とネットワーク通信の違い

01 **TCP/IPの基礎**

階層モデル・WiFiの守備範囲・IPアドレスとポート・TCP／UDP

02 **Pico WでWiFi通信**

ハード構成・ソフトウェアスタック・C++コードの要点

03 **PCと通信する**

構成の選択・Pythonサーバとの接続・動作確認

導入

TCP/IPは「ネットワーク」を前提にした通信規格

シリアル通信：**1本の線を決まった機器間で独占**

ネットワーク：**多数の機器と共有された網**

共有ゆえに起きること

同じ電波を何台もの機器が使う

↓

隣の機器を識別して、電波を分け合う

リンク層（WiFi）

共有ゆえに起きること

相手は他人の網の先にいる

↓

住所で相手を決めて、経路をたどる

インターネット層（IP）

共有ゆえに起きること

落ちる・遅れる・混ざる

↓

宛先のプログラムを分け、確実さを決める

トランスポート層（TCP/UDP）

01

TCP/IPの基礎

通信の仕事を役割ごとに分けて、「どこへ」「どう届けるか」を決める共通ルール。

通信の仕事は4つの層に分業されている

アプリケーション層	「何を話すか」— メッセージの中身と意味	HTTP ／ 自作の形式
トランスポート層	「どう届けるか」— 確実に届ける／軽く送る	TCP ／ UDP
インターネット層	「どこへ届けるか」— 住所を頼りに経路を選ぶ	IP（IPアドレス）
リンク層	「隣まで運ぶ」— 電波や電線で物理的に伝える	WiFi ／ Ethernet

各層は自分の仕事だけを行い、残りは下の層に任せる。だから電波の仕様を知らなくても通信プログラムが書ける。

TCP＝Transmission Control Protocol／UDP＝User Datagram Protocol／IP＝Internet Protocol

WiFiが運ぶのは「電波が直接届く1区間」だけ

- **どの区間にもリンク層がある。**

  ルータより先もIPが直接運ぶのではなく、有線LAN（Local Area Network）や光といった別のリンク層が1区間ずつ運ぶ。
  区間をどうつなぐかを決めているのがIP。

- **「WiFiに接続」＝網の輪に参加する手続き。**

  リンク層の話。あとで出てくるTCPの「接続」とは層も目的も別物。

住所（IPアドレス）と部屋番号（ポート）で相手を指定する

- **IPアドレス ＝ どの機器か。**

  IP（Internet Protocol）。LAN内では 192.168.x.x がルータから自動で配られる
  （DHCP＝Dynamic Host Configuration Protocol）。

- **ポート番号 ＝ どのプログラムか。**

  0〜65535。同じPCの中のプログラムを区別する。

- **2つのペアで宛先が決まる。**

  「192.168.1.10 の 5000番」のように組で指定する。

PC `192.168.1.10` ＝住所

`:5000`	Pythonスクリプト（今日の宛先）
`:80`	Webサーバ
`:22`	SSH

ポート ＝ 部屋番号。同じ住所でも部屋ごとに住人（プログラム）が違う。

シリアルの「当たり前」は、網の中では成り立たない

- **届かないかもしれない。**

  ルータは混雑するとパケットを捨てる。IPは「努力して運ぶ」だけで保証しない（ベストエフォート）。

- **順番が入れ替わるかもしれない。**

  経路は1本ではない。後から送ったパケットが先に着くこともある。

- **機器に届いても「誰宛てか」分からない。**

  IPアドレスが指すのは機器まで。中のどのプログラム宛てかはIPの管轄外。

この3つの穴を埋めるのが**トランスポート層** — ポートで宛先プログラムを指定し、届け方の品質を選ぶ

穴を全部埋めるTCP、埋めずに軽さを取るUDP

TCP

Transmission Control Protocol｜つないでから話す ― 電話

最初に**接続**の手続きをする

届いたか確認し、**欠けたら再送**

送った順に届く（順序保証）

その分、手間と遅延が増える

UDP

User Datagram Protocol｜いきなり送る ― はがき

接続の手続きなしで**即送信**

届いたかどうかは**分からない**

順番が入れ替わることもある

軽くて速い。仕組みが単純

「接続」に注意 ― WiFiの接続は**リンク層**（網への参加。TCP・UDP共通の前提）、TCPの接続は**トランスポート層**（プログラム同士の合意。TCPだけ）

Q. この用途、TCPとUDPどちらを選ぶ?

① 温度センサの値を1秒ごとにPCへ送り続ける　TCP ／ UDP ?

② 設定ファイルをPicoへ転送する（1バイトも欠けてはいけない）　TCP ／ UDP ?

③ 動作ログをPCへ送る　TCP ／ UDP ?

ヒント：欠けたら困るか? 遅れたら困るか?

A. 欠けて困るならTCP、鮮度が命ならUDP

- **① センサ値 → UDPが定番。**

  1秒後に新しい値が来る。1個の欠けより、再送で古い値が遅れて届く方が困る。

- **② ファイル転送 → TCP一択。**

  1バイト欠けても壊れる。全部を順番どおり届ける保証が必要。

- **③ ログ → 要件次第。**

  確実な記録ならTCP、多少欠けてよい速報ならUDP。「欠けたら困るか」で決める。

下の層への入口「ソケット」— ファイルのように読み書きする

層の中での位置づけ

アプリケーション層
あなたのプログラム

↕

ソケット ＝ 下の層を使うための窓口
作るときにTCPかUDPを選ぶ。プログラムとの境目はここ1か所だけ

↕

トランスポート層	TCP ／ UDP
インターネット層	IP（IPアドレス）
リンク層	WiFi

ここから下はOS・ライブラリの仕事。プログラムからは見えない

使い方は4ステップ

① 作る	TCPかUDPを選んで作る	`socket()`
② 接続する	IPアドレスとポートを指定（TCPのみ）	`connect()`
③ 送る・受け取る	バイト列を読み書き	`send()` ／ `recv()`
④ 閉じる	後片付け	`close()`

ファイル操作（open → read/write → close）と同じ発想。C++でもPythonでも共通の流れ

02

Pico WでWiFi通信

ハードとソフトの全体像をつかんでから、C++ SDK（Software Development Kit）のコードを読む。

無線は専用チップCYW43439の仕事 — RP2040は依頼するだけ

対応は**2.4GHz帯のみ**。5GHz専用のSSID（Service Set Identifier）には接続できない。

PCではOSがやる仕事を、Picoでは lwIP が肩代わりする

あなたのC++コード	送りたいデータを組み立てる	アプリケーション層
pico-sdk（cyw43_arch）	初期化・WiFi接続を数行の関数に	つなぎ役
lwIP	lightweight IPの略。TCP/IPを実装した軽量ライブラリ。今日の主役	トランスポート層＋インターネット層
CYW43439 ドライバ＋チップ	電波の送受信	リンク層

CMakeで `pico_cyw43_arch_lwip_threadsafe_background` をリンクすると、この一式がまとめて使える。

WiFi接続は実質3関数 — 初期化・STAモード・接続

```
#include "pico/cyw43_arch.h"

int main() {
    // ① 無線チップを初期化
    if (cyw43_arch_init()) return -1;

    // ② 子機（ステーション）モードに
    cyw43_arch_enable_sta_mode();

    // ③ 接続を待つ（最大30秒）
    if (cyw43_arch_wifi_connect_timeout_ms(
            "MyHomeWiFi", "password",
            CYW43_AUTH_WPA2_AES_PSK, 30000)) {
        return -1;  // 失敗
    }
    // ここに来ればIPアドレス取得済み
}
```

- **接続完了まで待つ関数。**

  戻り値0で成功。IPアドレスはDHCPで自動取得される。

- **まだLANに参加しただけ。**

  データの送受信はこの後のTCP／UDPで行う。

抜粋。CMakeLists.txtで
pico_cyw43_arch_lwip_threadsafe_background のリンクが必要。

lwIPは「呼んで待つ」ではなく「起きたら呼ばれる」— コールバック方式

自分から呼ぶ関数

`tcp_connect(...)` 接続を依頼

`tcp_write(...)` 送信データを渡す

lwIPから呼ばれる関数（事前に登録）

`on_connected(...)` 接続が完了したとき

`on_recv(...)` データが届いたとき

GPIO割り込みハンドラと同じ発想 —「イベントが起きたら、登録した関数が呼ばれる」

接続を依頼し、完了したら送る — TCPクライアントの骨格

```
// ① PCB（≒ソケット）を作り、コールバックを登録
struct tcp_pcb *pcb = tcp_new();
tcp_recv(pcb, on_recv);         // 受信時に呼ばれる

// ② サーバ（PC）へ接続を依頼
tcp_connect(pcb, &server_ip, 5000, on_connected);

// ③ 接続完了で lwIP から呼ばれる
err_t on_connected(void *arg,
        struct tcp_pcb *pcb, err_t err) {
    tcp_write(pcb, "hello from pico\n", 16,
              TCP_WRITE_FLAG_COPY);
    tcp_output(pcb);            // 送信を確定
    return ERR_OK;
}
```

- **PCBがソケットの役。**

  PCB＝Protocol Control Block（接続を管理する台帳）。
  作る→接続→送受信→閉じる、の流れは第1章と同じ。

- **writeとoutputはセット。**

  writeはバッファに積むだけ。outputで送信を促す。

抜粋。エラー処理・切断処理（tcp_close）は省略。

2.6 コード③ UDP送信

UDPは「詰めて、投函する」だけ — 接続もコールバックも不要

```
// ① PCBを作り、宛先（PC）を用意
struct udp_pcb *pcb = udp_new();
ip_addr_t dest;
ipaddr_aton("192.168.1.10", &dest);

// ② データをpbuf（lwIPのバッファ）に詰める
const char *msg = "temp=25.4\n";
struct pbuf *p = pbuf_alloc(PBUF_TRANSPORT,
                    strlen(msg), PBUF_RAM);
memcpy(p->payload, msg, strlen(msg));

// ③ 投函 – 届いたかの確認はしない
udp_sendto(pcb, p, &dest, 5000);
pbuf_free(p);  // 忘れるとメモリ枯渇
```

- **TCPよりずっと短い。**
  接続手続きがないぶんコードも単純。定期送信のセンサ向き。
- **pbuf_freeを忘れない。**
  マイコンはメモリが少ない。解放漏れは送信失敗に直結。

抜粋。1秒ごとに②③を繰り返せば計測ノードになる。

03

PCと通信する

受け手はPythonの標準ライブラリで10行。組み合わせて通信を成立させる。

「先に待つ側」がサーバ、「話しかける側」がクライアント

- **通信は「待つ側」がいないと始まらない。**

  網の相手は、こちらの都合では聞いていない。だから先に「待つ係」を決めておく。線が常設のシリアルには無かった概念。

- **機械の種類ではなく、プログラムの役割。**

  ソケットを使うプログラム同士の分担で、アプリケーション側の概念。IPやWiFiなど下の層は両者を対等に扱い、この区別を知らない。

- **役割はいつでも入れ替えられる。**

  同じPCがサーバにもクライアントにもなれる。どちらを待つ側にするかは次のスライドで選ぶ。

どちらを「待つ側」にするか — 正解は用途で決まる

前提：PicoとPCは同じルータの下（同じLAN）にいる。待つ側＝サーバ、話しかける側＝クライアント。

案A：PCが待つ

Pico＝クライアント ／ PC＝サーバ

Picoは**起きたら送るだけ**。データを集める用途に自然

待つ側のIPが必要 → PCのIPは**ipconfigですぐ分かる**

接続を始められるのは**Pico側だけ**。
（接続後は双方向に送り合える）

案B：Picoが待つ

Pico＝サーバ ／ PC＝クライアント

PCから**好きなタイミングで問い合わせ**・操作できる

複数の端末から同じPicoへアクセスする用途に自然

PicoのIPはDHCPで変わりうる → **固定IP等の工夫が必要**

以降のコード例は案Aで進める。関数の使い方は構成を入れ替えても同じ。

3.3 コード④ PC側（TCP）

受け手のTCPサーバはPython標準ライブラリで10行

```
import socket

srv = socket.socket(socket.AF_INET,
                    socket.SOCK_STREAM)  # TCP
srv.bind(("0.0.0.0", 5000))  # 5000番で受付
srv.listen(1)

conn, addr = srv.accept()    # Picoの接続を待つ
print("connected:", addr)

while True:
    data = conn.recv(1024)  # 受信を待つ
    if not data:
        break               # 切断された
    print(data.decode())
conn.close()
```

- **流れは第1章のソケットのまま。**

  作る → 待つ（bind/listen/accept）→ 読み書き → 閉じる。

- **"0.0.0.0" ＝ どの口でも受付。**

  このPCが持つすべてのネットワーク口で5000番を開く。

UDP受信は接続なし — bindしてrecvfromするだけ

```
import socket

sock = socket.socket(socket.AF_INET,
                     socket.SOCK_DGRAM)  # UDP
sock.bind(("0.0.0.0", 5000))

while True:
    # 届いたデータと送り主が返る
    data, addr = sock.recvfrom(1024)
    print(addr, data.decode())
```

- **listenもacceptもない。**
  接続という概念がないため、開いたらすぐ受信できる。
- **送り主のアドレスも分かる。**
  recvfromが (データ, 送信元) を返す。複数のPicoの受け口を1つにできる。

「待つ側が先」— サーバを起動してからPicoを動かす

① PCのIPを調べる
ipconfig ／ ip addr。Picoのコードの宛先に書く

→

② サーバを起動
python server.py — 待つ側を先に動かす

→

③ ビルド＆書き込み
cmake＋make → BOOTSELを押しながら接続、UF2をコピー

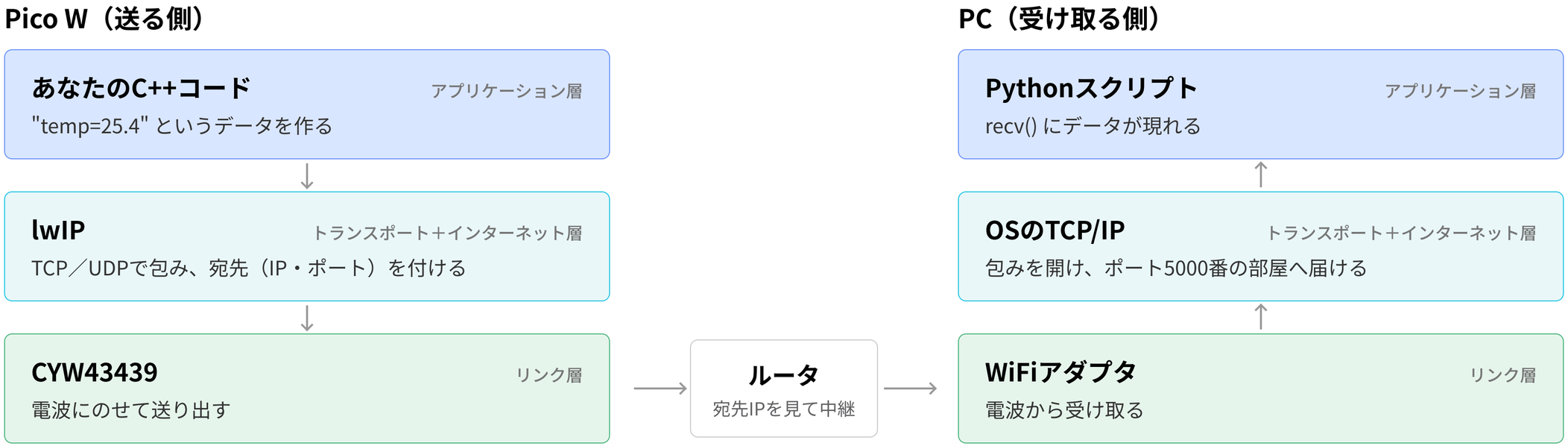
→

④ 受信を確認
PCのターミナルに hello from pico が表示されれば成功

```
$ python server.py
connected: ('192.168.1.23', 54321)
hello from pico
```

まとめ

全体像：データは層を下り、網を渡り、層を上る



送る側は層を下り、受け取る側は層を上る。今日の内容はすべてこの図の中にある。