TCP-Server Example

Routine Introduction

In this section, we will introduce how to use K230 as a TCP server

We open the example code in this section and modify the WIFI connection information part in the code to your home WIFI or mobile hotspot

```
def connect_wifi():
    """
    连接WiFi并返回IP地址
    Connect to WiFi and return IP address
    """
    sta = network.WLAN(0)  # 创建WiFi站点对象 / Create WiFi station object
    sta.connect("WIFI名称", "WIFI密码")  # 连接到指定WiFi / Connect to specified
WiFi
    # 等待直到获取到IP地址 / Wait until IP address is obtained
    while sta.ifconfig()[0] == '0.0.0.0':
        os.exitpoint()
    return sta.ifconfig()[0]  # 返回IP地址 / Return IP address
```

Then click the Run button in the lower left corner to run the routine

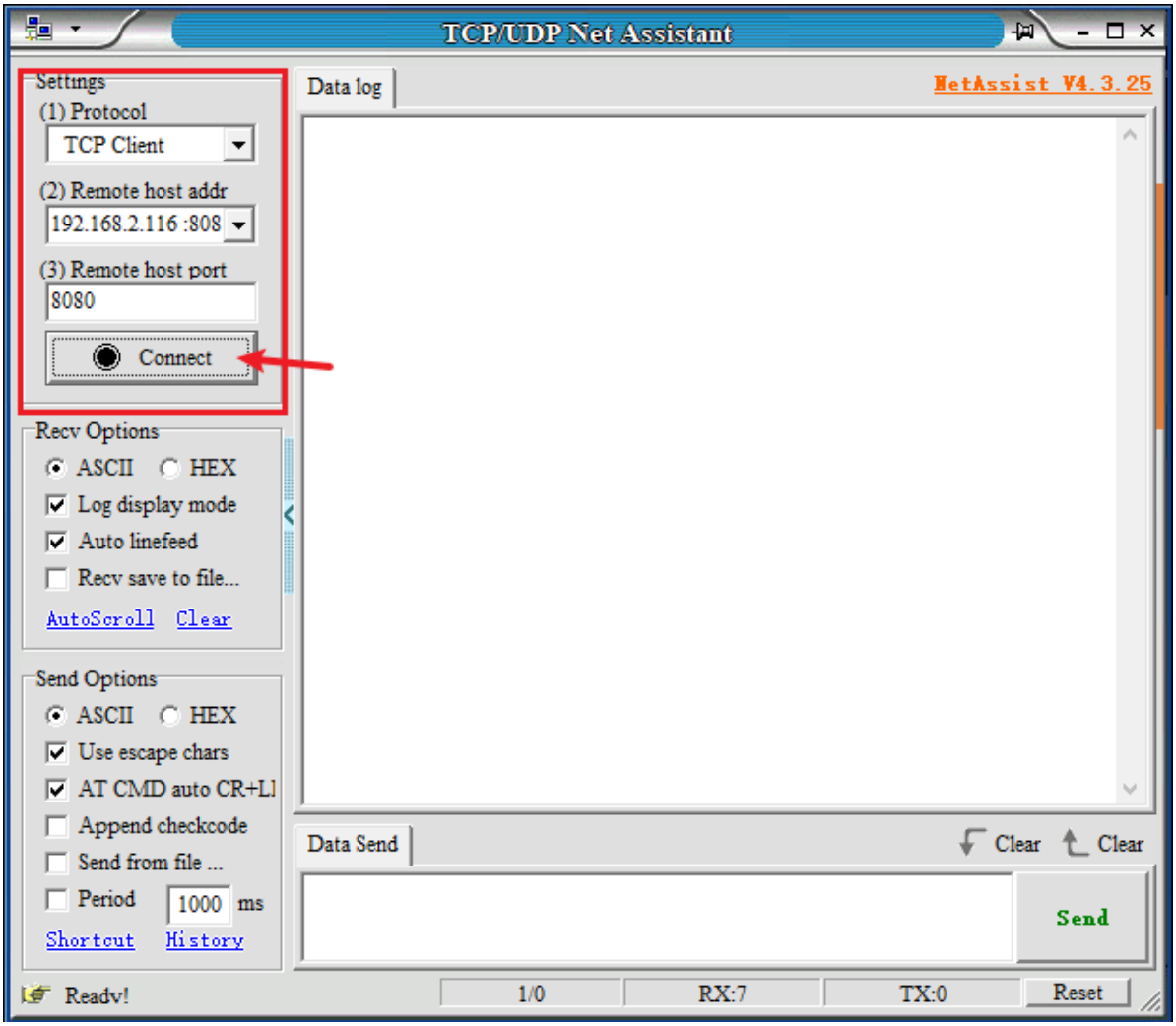
We open the serial terminal and wait for the output message that the server has started successfully

After the server is successfully started, the serial terminal will output TCP server running on xxxxxx:8080 (if 8080 is occupied, you can change to another port)

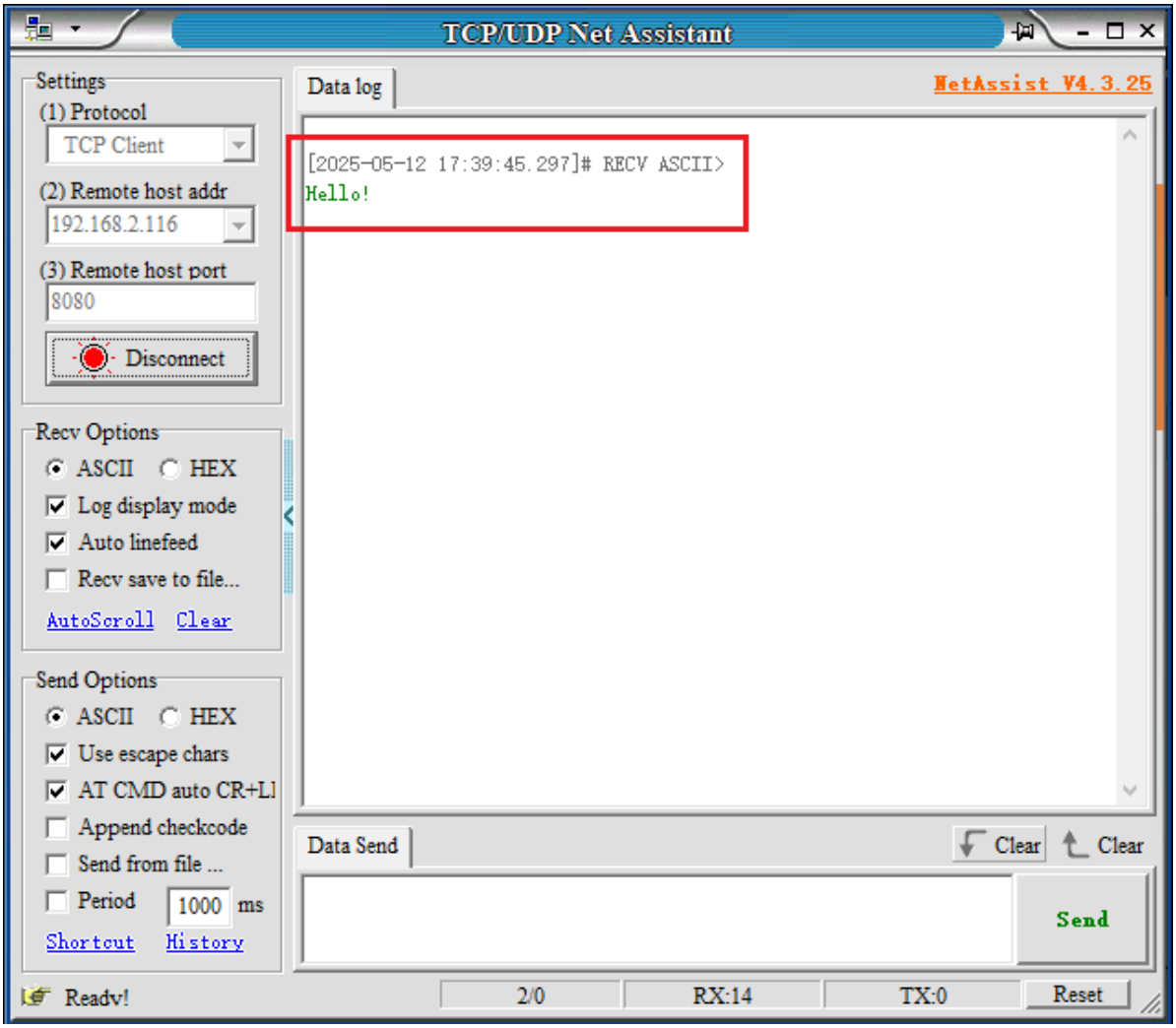
Here is [TCP server running on 192.168.3.47:8080]

Then we open the network debugging assistant NetAssist.exe in the supporting tools

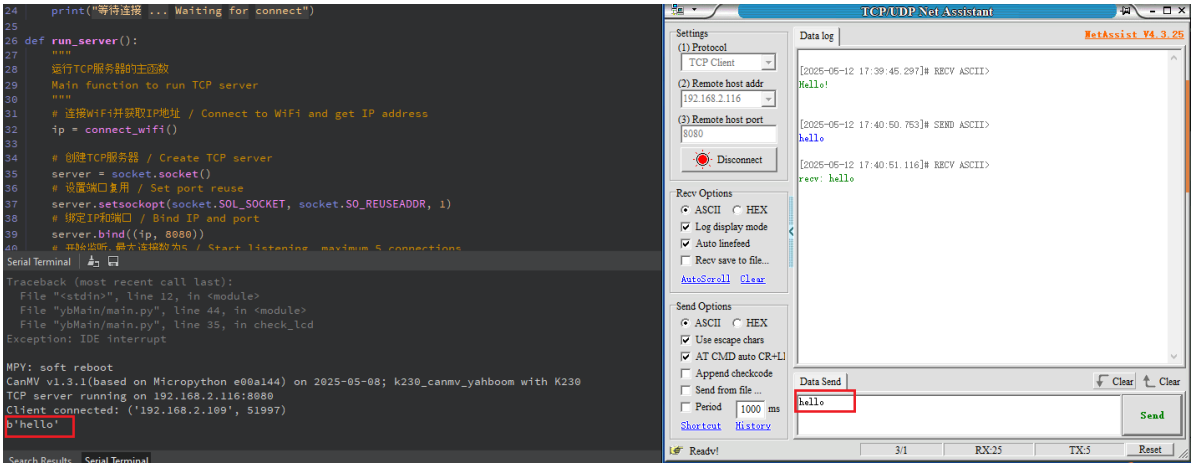
Enter the information in the area in the upper left corner and click Connect



You can see that we received a message from the server running on K230



We send a message in the input box, and the server can also receive the information we sent



> Due to encoding problems, the IDE output looks like garbled code. In fact, we only need to call the decoding function and change print(data) to print(data.decode()) to display the correct content. Here we leave it to you to try it yourself

Code Analysis

For the complete code and bilingual comments, please refer to [Source Code/11.Network/03.tcp/tcp_server.py]

1. Import module part:

```
import socket      # 导入socket模块，用于网络通信 / Import socket module for network communication
import network     # 导入network模块，用于WiFi连接 / Import network module for WiFi connection
import os          # 导入os模块，用于系统操作 / Import os module for system operations
import time        # 导入time模块，用于时间相关操作 / Import time module for time-related operations
```

This part imports the basic modules required by the program, including modules related to network communication, WiFi connection, system operation and time operation.

2. WiFi connection function:

```
def connect_wifi():
    """
    连接WiFi并返回IP地址
    Connect to WiFi and return IP address
    """
    sta = network.WLAN(0)  # 创建WiFi站点对象 / Create WiFi station object
    sta.connect("WIFI名称", "WIFI密码")  # 连接到指定WiFi / Connect to specified WiFi
    # 等待直到获取到IP地址 / Wait until IP address is obtained
    while sta.ifconfig()[0] == '0.0.0.0':
        os.exitpoint()
    return sta.ifconfig()[0]  # 返回IP地址 / Return IP address
```

This function is responsible for:

- Create a WiFi site object
- Connect to a specified WiFi network
- Wait until a valid IP address is obtained
- Returns the obtained IP address

3. Client shutdown function:

```
def close_client(client):
    """
    关闭客户端连接
    Close client connection
    """
    client.close()  # 关闭连接 / Close connection
    print("等待连接 ... Waiting for connect")
```

This is a simple helper function that closes the client connection and prints a prompt message.

4. Main server function:

```
def run_server():
    """
    运行TCP服务器的主函数
    Main function to run TCP server
    """
    # 连接WiFi并获取IP地址 / Connect to WiFi and get IP address
    ip = connect_wifi()
```

```
    # 创建TCP服务器 / Create TCP server
    server = socket.socket()
    # 设置端口复用 / Set port reuse
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # 绑定IP和端口 / Bind IP and port
    server.bind((ip, 8080))
    # 开始监听，最大连接数为5 / Start listening, maximum 5 connections
    server.listen(5)
    print(f"TCP server running on {ip}:8080")
```

This part of the code completes the initialization of the server:

- Connect to WiFi and obtain an IP address
- Creating a TCP socket
- Set up port reuse (allows the server to immediately use the same port when it restarts)
- Bind IP and port (8080)
- Start listening for connections, the maximum number of waiting connections is 5

5. Main loop part:

```
    while True:
        # 等待客户端连接 / Wait for client connection
        client, addr = server.accept()
        print(f"Client connected: {addr}")
        # 设置非阻塞模式 / Set non-blocking mode
        client.setblocking(False)

        # 发送欢迎消息 / Send welcome message
        client.write(f"Hello!\n".encode())

        while True:
            # 处理接收数据 / Handle received data
            try:
                data = client.read()
                if data:
                    print(data)
                    # 发送确认消息 / Send acknowledgment
                    client.write(b"recv: " + data)
                # 如果收到结束命令，关闭连接 / If end command received, close
connection
                if b"end" in data:
                    close_client(client)
                    break
            except:
                pass
```

This is the core processing logic of the server:

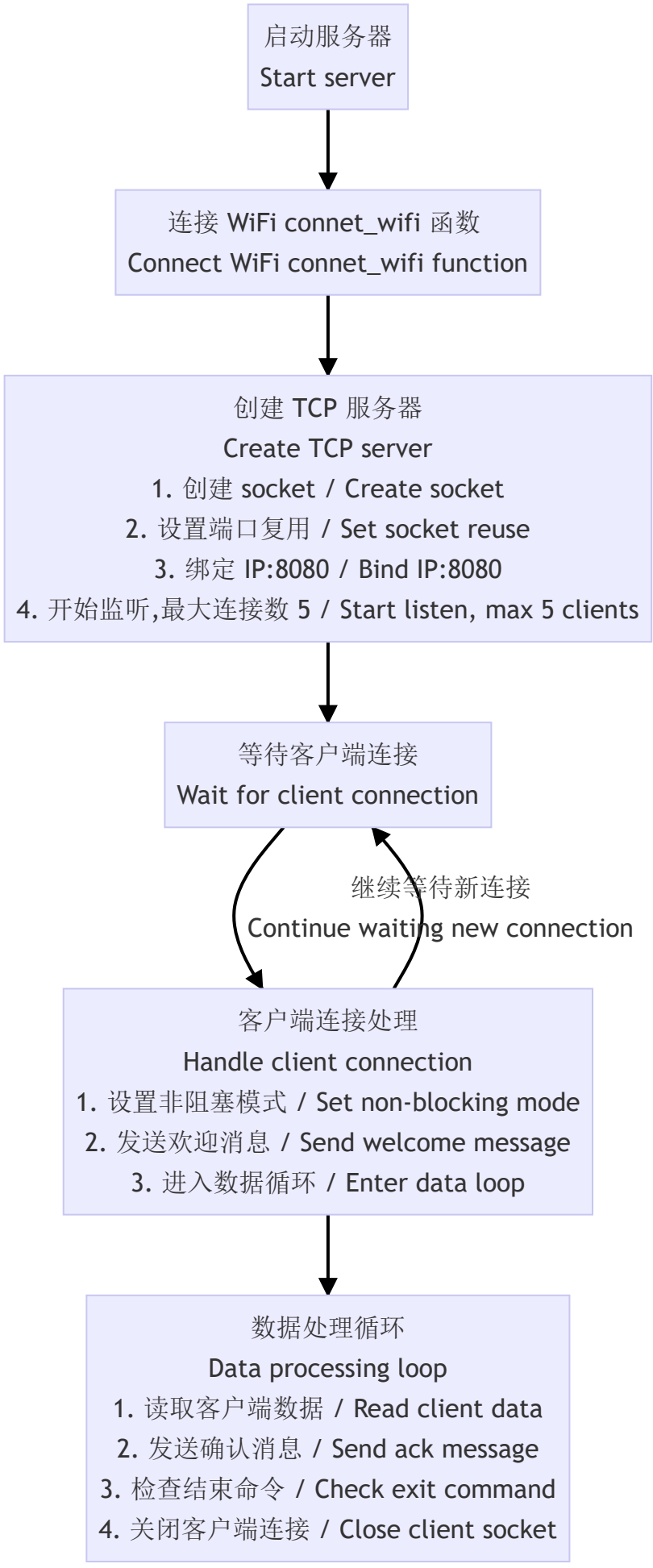
- The outer loop continues to accept new client connections
- When there is a new connection, set it to non-blocking mode
- Send a welcome message
- The inner loop handles communication with the client:
  - Read data sent by the client
  - If data is received, print and send a confirmation message
  - If an "end" command is received, close the connection and exit the inner loop

- Use exception handling to handle read operations in non-blocking mode

6. Server startup:

```
# 启动服务器 / Start server
run_server()
```

The program flow chart is as follows:


启动服务器
Start server
连接 WiFi connet_wifi 函数
Connect WiFi connet_wifi function
创建 TCP 服务器
Create TCP server
1. 创建 socket / Create socket
2. 设置端口复用 / Set socket reuse
3. 绑定 IP:8080 / Bind IP:8080
4. 开始监听,最大连接数 5 / Start listen, max 5 clients
等待客户端连接
Wait for client connection
继续等待新连接
Continue waiting new connection
客户端连接处理
Handle client connection
1. 设置非阻塞模式 / Set non-blocking mode
2. 发送欢迎消息 / Send welcome message
3. 进入数据循环 / Enter data loop
数据处理循环
Data processing loop
1. 读取客户端数据 / Read client data
2. 发送确认消息 / Send ack message
3. 检查结束命令 / Check exit command
4. 关闭客户端连接 / Close client socket

> In this example, the client can normally disconnect the communication with the K230 server by sending "end". However, if the connection is disconnected directly, the server will freeze. To solve this problem, please refer to the other attached source code tcp_server_heartbeat.py in this section.

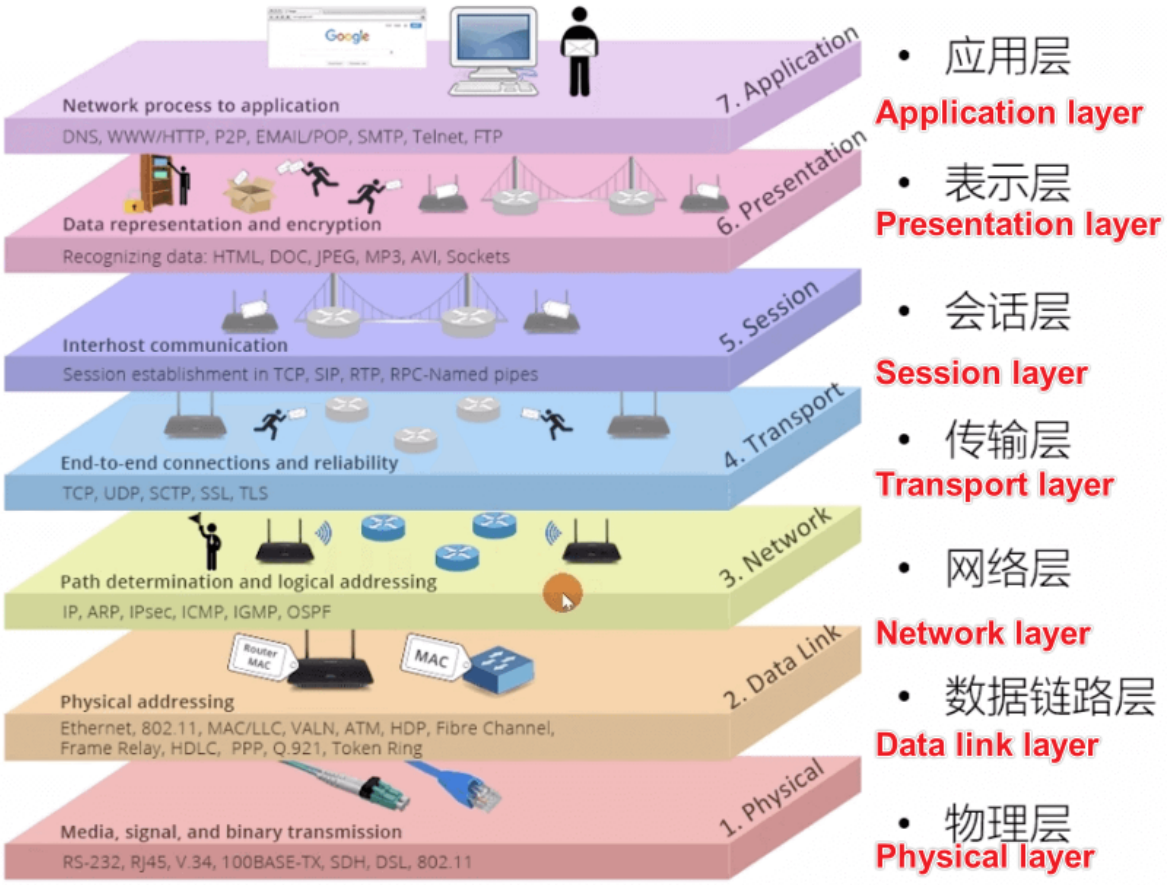
TCP

We briefly describe the TCP protocol here

Basic knowledge of TCP

1.1 What is TCP?

- **Positioning** : Transport layer protocol (OSI layer 4)
- **Features** : connection-oriented, reliable transmission, flow control, congestion control
- **Analogy** : Like a courier company's "insured package service" (ensuring complete delivery)



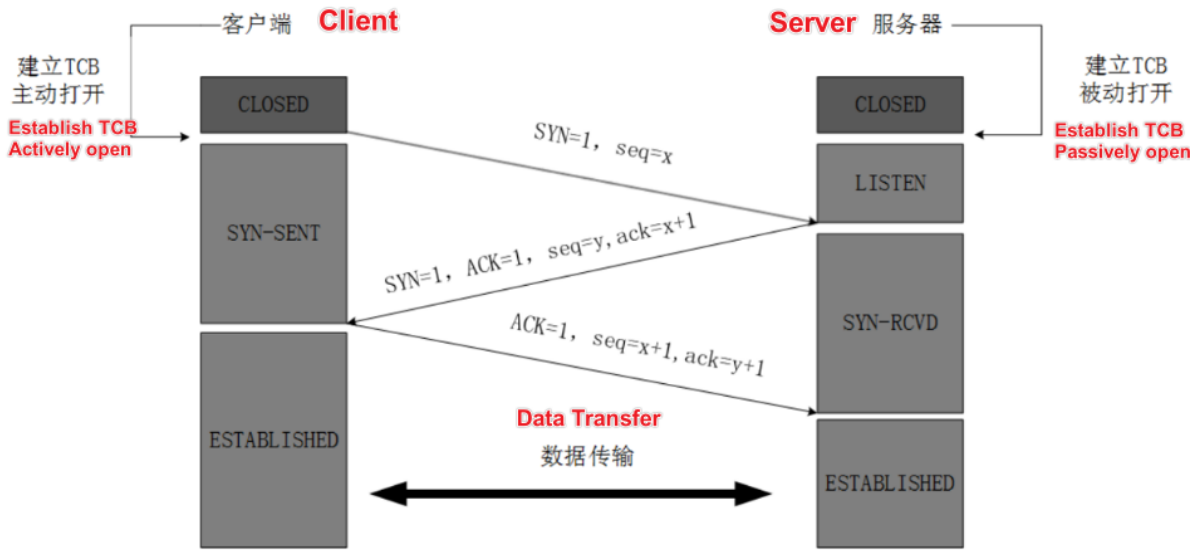
(Diagram description: In the OSI model, TCP is located above the IP layer (network layer) and below the application layer)

1.2 TCP vs UDP

characteristic	TCP	UDP
Connection	Connection-Oriented	No connection
reliability	Reliable transmission	Do your best
speed	Slow	Very fast
Head size	20-60 bytes	8 bytes
Typical Applications	Web/Email	Video streaming/gaming

TCP connection management (three stages illustrated)

2.1 Three-way handshake to establish a connection

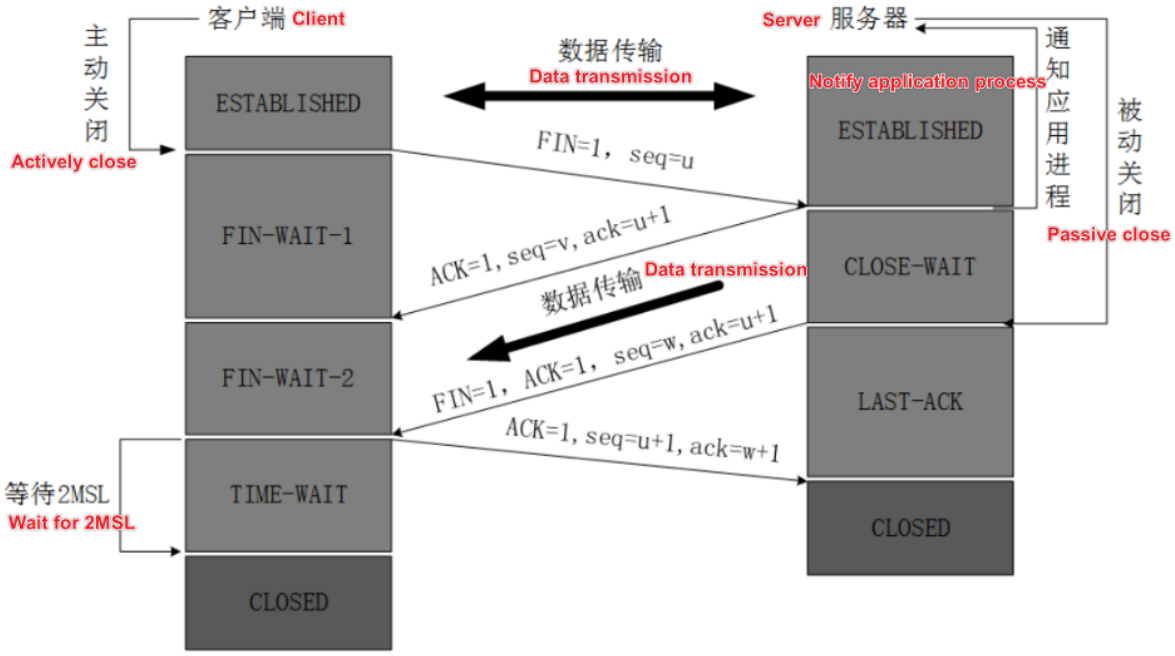


(Diagram description: The client and server interact three times through SYN, SYN-ACK, and ACK)

1. **SYN** (client→server) "I want to establish a connection, my initial sequence number is X"
2. **SYN-ACK** (Server → Client) "Received! My initial sequence number is Y, looking forward to your next X+1"
3. **ACK** (Client → Server) "Confirmed receipt, let's start transmitting data!"

> Why not two handshakes? To prevent invalid connection requests from suddenly arriving and causing resource waste (network delay problem)

2.2 Four waves to terminate the connection



(Diagram describing the four interactive steps of FIN and ACK)

1. **FIN** (A→B)
   "I am closing the connection"
2. **ACK** (B→A)
   "Close request received"
3. **FIN** (B→A)
   "I'm ready to close too"
4. **ACK** (A→B)
   "Confirmation of closure, connection termination"

> **TIME_WAIT state** : The active closing party waits for 2MSL time to ensure that the last ACK is received

TCP reliable transmission core mechanism

3.1 Data numbering and confirmation

- **Sequence Number** : Each byte has a unique number
- **Ack Number** : "I have received all data before X"
- **Example** :

```
Sender: Send a data packet with [sequence number = 1, length = 100]
Receiver: Reply [ACK=101] (expecting the next data packet to start from 101)
```

3.2 Timeout Retransmission

- **RTO (Retransmission Timeout)** : dynamically calculated timeout
- **Fast retransmit** : Retransmit immediately after receiving 3 duplicate ACKs

3.3 Sliding Window Protocol

- **Window size** : The available buffer size advertised by the receiver
- **Dynamic adjustment** : real-time changes based on network conditions

---

Advanced control mechanisms

4.1 Flow Control

- **Principle** : Control the sending rate through the receiving window (rwnd)
- **Zero window detection** : Send a detection packet when the receive window is 0

4.2 Congestion Control

1. **Slow start** : exponential growth until threshold
2. **Congestion avoidance** : linear growth
3. **Fast recovery** : halving the window when packet loss occurs

4.3 TCP request message header structure

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options (if Data Offset > 5)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             Data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Practical application scenarios

1. **HTTP communication** : Each web page load goes through a TCP connection
2. **File transfer** : FTP and other protocols rely on TCP reliability
3. **Database connection** : MySQL and other databases use TCP by default

---

FAQs

Q1: How to deal with TCP packet sticking problem?
Through application layer protocol design (such as adding length headers or delimiters)

Q2: Why do some live videos use TCP?

When image quality is prioritized, use TCP; when real-time is prioritized, use UDP+QUIC

Q3: How to maintain a long connection?

Use the Keep-Alive mechanism to send detection packets regularly

> The attached source code tcp_server_heartbeat.py mentioned in this article refers to the Keep-Alive mechanism