# Network programing

## Introdução Engenharia Informática

Mário Antunes

November 10, 2025

Universidade de Aveiro

# Table of Contents i

**A journey through modern communication protocols**

# Sockets: The Foundation i

A **socket** is an endpoint for communication. It's an abstraction (represented as a file descriptor) that your program can write to and read from.

- **Analogy:** A socket is like a "door" in your application. You give it a **port number** (the door number) on your machine's **IP address** (the street address).

Two main types for internet communication: **TCP** and **UDP**.

# TCP vs. UDP: The Two Pillars i

| Feature | TCP (Transmission Control Protocol) | UDP (User Datagram Protocol) |
|---|---|---|
| **Connection** | Connection-oriented (establishes a session) | Connectionless (fire and forget) |
| **Reliability** | **Reliable:** Guarantees delivery & order. | **Unreliable:** No guarantee of delivery or order. |
| **Overhead** | High (3-way handshake, ACKs, flow control) | Low (just a small header) |
| **Speed** | Slower, due to reliability checks | Faster, no connection setup or ACKs |
| **Use Cases** | Web (HTTP), Email (SMTP), File Transfer (FTP) | Streaming video, online gaming, DNS, VoIP |
| **Python Module** | `socket.SOCK_STREAM` | `socket.SOCK_DGRAM` |

# TCP Communication Pattern (Req/Rep)

TCP uses a **3-way handshake** to establish a reliable connection.

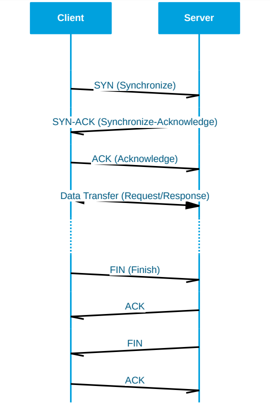



**Figure 1:** TCP 3-way handshake

# Python TCP Server (Echo) i

```python
# tcp_server.py
import socket

HOST = '127.0.0.1'  # Standard loopback interface
PORT = 65432        # Port to listen on

# Use 'with' for automatic resource management
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print(f"TCP server listening on {HOST}:{PORT}")
    # conn is a new socket object usable to send/recv data
    # addr is the address bound to the client
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024) # 1KB buffer
            if not data:
                break # Client closed connection
            print(f"Received: {data.decode()}")
            conn.sendall(data) # Echo back
```

# Python TCP Client i

```python
# tcp_client.py
import socket

HOST = '127.0.0.1'  # The server's hostname or IP
PORT = 65432        # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world') # Send as bytes
    data = s.recv(1024)
    print(f"Received echo: {data.decode()}")
```

# UDP Communication Pattern (Datagram)

UDP is “fire and forget.” No connection is established.

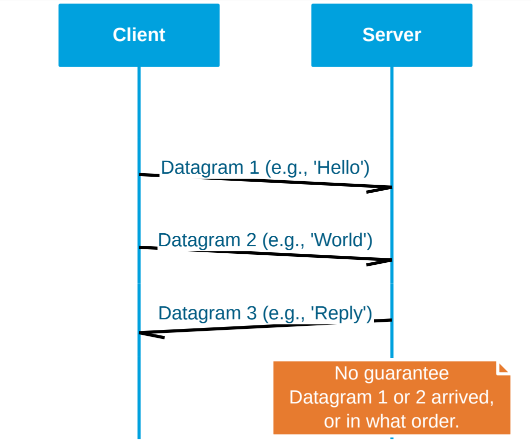



**Figure 2:** UDP - does not use session concepts

# Python UDP Server (Echo) i

```python
# udp_server.py
import socket

HOST = '127.0.0.1'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind((HOST, PORT))
    print(f"UDP server listening on {HOST}:{PORT}")

    while True:
        # recvfrom returns data AND the address of the sender
        data, addr = s.recvfrom(1024)
        print(f"Received {data.decode()} from {addr}")

        if not data:
            break

        s.sendto(data, addr) # Echo back to the sender
```

# Python UDP Client i

```python
# udp_client.py
import socket

HOST = '127.0.0.1'
PORT = 65432
MESSAGE = b'Hello, UDP!'

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.sendto(MESSAGE, (HOST, PORT))

    data, addr = s.recvfrom(1024)
    print(f"Received echo: {data.decode()} from {addr}")
```

## The Problem: Blocking I/O i

Traditional sockets are **blocking**.

- `s.accept()` **blocks** until a client connects.
- `conn.recv(1024)` **blocks** until data arrives.

If you are handling one client, all other clients must wait!

**Traditional Solution: Multi-threading**

- Creates one thread per client.
- **Complex:** Thread safety (locks, race conditions).
- **High Resource Use:** RAM, OS-level context switching.

# The Problem: Blocking I/O ii

- **Python Issue (GIL):** The Global Interpreter Lock (GIL) in CPython prevents true parallel execution of Python code, limiting this approach.

# The Solution: Async IO i

**Asynchronous I/O (`asyncio` in Python)** allows a single thread to manage many connections.

- It works for both **TCP** and **UDP** using an **event loop** to monitor sockets.
- When a socket is "ready" (e.g., has data), the loop runs the corresponding code.
- The `async` and `await` keywords "pause" a function, allowing the loop to work on other things, instead of blocking the whole thread.
  *This is **concurrency**, not parallelism. It's about waiting efficiently.*

# Python `asyncio` TCP Server i

This server can handle thousands of clients concurrently.

```python
# asyncio_server.py
import asyncio

async def handle_client(reader, writer):
    """Callback for each new client connection"""
    addr = writer.get_extra_info('peername')
    print(f"Connected by {addr}")

    try:
        while True:
            data = await reader.read(1024)
            if not data:
                break

            message = data.decode()
            print(f"Received from {addr}: {message}")

            # Echo back
            writer.write(data)
            await writer.drain() # Wait until buffer is flushed
```

# Python `asyncio` TCP Server ii

```
    except asyncio.CancelledError:
        print(f"Connection with {addr} cancelled.")
    finally:
        print(f"Closing connection with {addr}")
        writer.close()
        await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_client, '127.0.0.1', 65432)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

# Python asyncio UDP Server i

AsyncIO also works for UDP, using a slightly different "Protocol" based approach.

```
# asyncio_udp_server.py
import asyncio

class EchoServerProtocol(asyncio.DatagramProtocol):
    def connection_made(self, transport):
        self.transport = transport
        print("UDP Server (asyncio) started")

    def datagram_received(self, data, addr):
        message = data.decode()
        print(f"Received {message} from {addr}")
        self.transport.sendto(data, addr) # Echo back

async def main():
    loop = asyncio.get_running_loop()
    print("Starting UDP server on 127.0.0.1:65432")

    # Create the datagram endpoint
    transport, protocol = await loop.create_datagram_endpoint(
```

# Python asyncio UDP Server ii

```
        lambda: EchoServerProtocol(),
        local_addr=('127.0.0.1', 65432))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour
    finally:
        transport.close()

asyncio.run(main())
```

# REST APIs: The Language of the Web i

Sockets are powerful but low-level. Most modern web services don't expose sockets directly. They use **APIs** (Application Programming Interfaces).

**REST** (REpresentational State Transfer) is the most common architectural style for web APIs.

- It's **not a protocol**, but a set of rules.
- It builds *on top of* HTTP (which builds on top of TCP).
- It's **stateless**: Every request must contain all info needed to process it.

# REST Communication Pattern (Req/Rep) i

Client-server communication over HTTP.

- **Resource:** An entity (e.g., `/users`, `/products/123`).
- **Verbs:** HTTP methods (GET, POST, PUT, DELETE).
- **Data:** Usually sent/received as **JSON**.

# REST Communication Pattern (Req/Rep) ii

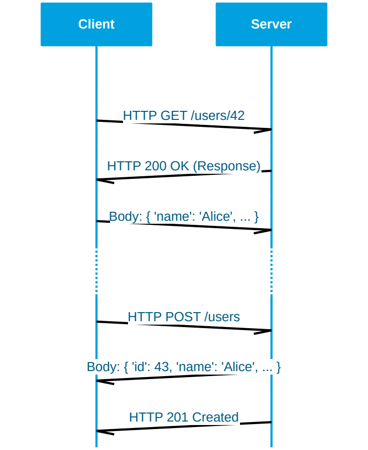



**Figure 3:** HTTP comunication

# Data Format: JSON i

**JSON** (JavaScript Object Notation) is the *de facto* standard for data exchange in REST APIs.

- Lightweight and human-readable.
- Easy for machines to parse and generate.
- Based on JavaScript object syntax.

**Example:**

```json
{
  "id": 123,
  "username": "api_user",
  "isActive": true,
  "roles": ["admin", "editor"],
  "lastLogin": {
    "date": "2025-11-07",
    "ip": "192.0.2.1"
  }
}
```

# Example: FastAPI (Python) i

**FastAPI** is a modern, high-performance Python web framework for building APIs. It's built on `asyncio`.

1. Install: `pip install fastapi "uvicorn[standard]"`
2. Save as `main.py`:

```python
# main.py
from fastapi import FastAPI

app = FastAPI()

# In-memory "database"
items = {
    1: {"name": "Laptop", "price": 1200},
    2: {"name": "Mouse", "price": 50}
}

@app.get("/")
```

# Example: FastAPI (Python) ii

```python
def read_root():
    return {"message": "Hello, API!"}

@app.get("/items/{item_id}")
def read_item(item_id: int):
    if item_id in items:
        return items[item_id]
    return {"error": "Item not found"}
```

3. Run: `uvicorn main:app --reload`
4. Access in browser:
   `http://127.0.0.1:8000/items/1`

# Sockets vs. REST APIs i

| Feature | Raw Sockets | REST APIs |
|---|---|---|
| **Level** | **Low-level** (OS-level) | **High-level** (Application-level) |
| **Protocol** | TCP, UDP (custom protocol) | HTTP/HTTPS (standardized) |
| **State** | Can be stateful | **Stateless** by design |
| **Data Format** | Anything (binary, custom text) | Usually JSON |
| **Use When...** | Custom protocol, high-speed, games, persistent connection. | Web services, mobile apps, public APIs, interoperability. |

# Sockets vs. REST APIs ii

***Key takeaway:*** *You could build a REST API on raw sockets… but you'd just be re-inventing HTTP. REST gives you a massive head start with standardization, security (HTTPS), and tooling.*

# WebSockets: The Real-Time Channel i

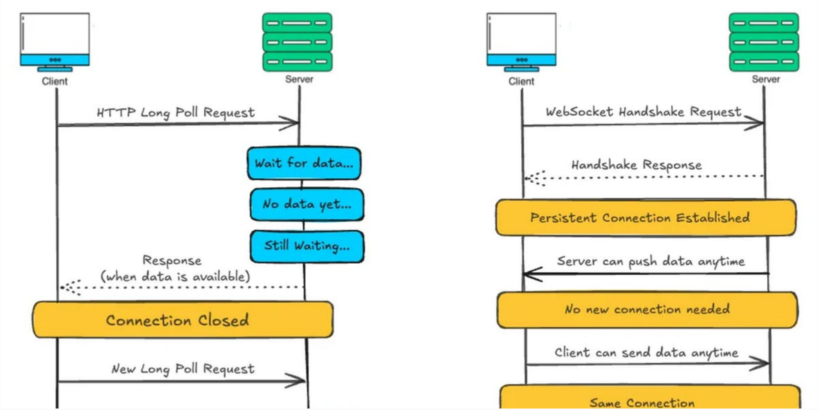
What if REST is too slow? What if the server needs to push data to the client *without* being asked?

- HTTP is a client-pull (Req/Rep) model.
- Inefficient “polling” (asking “any updates?” every 2 seconds) is a common, bad workaround.

**Solution: WebSockets**

- A **persistent, full-duplex** (bidirectional) connection.
- Starts as a standard HTTP “Upgrade” request.
- Once established, it’s a “raw” TCP-like channel for sending messages.

# WebSockets: The Real-Time Channel ii



# WebSocket Communication Pattern

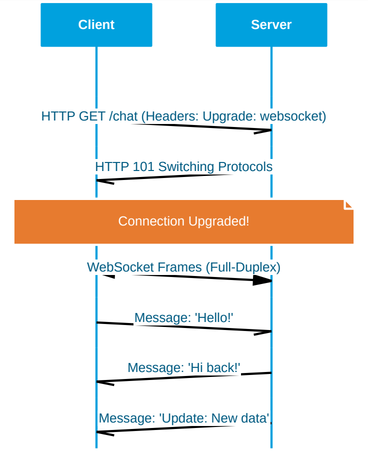



**Figure 4:** WebSocket Communication

# WebSocket Example: JavaScript Client i

The browser is the *native* platform for WebSockets.

```html
<!DOCTYPE html>
<html>
<head><title>WebSocket Chat</title></head>
<body>
    <ul id="messages"></ul>
    <input id="messageBox" type="text" />
    <button id="sendButton">Send</button>

    <script>
        const ws = new WebSocket("ws://localhost:8765"); // Connect
        const messages = document.getElementById("messages");
        const messageBox = document.getElementById("messageBox");
        const sendButton = document.getElementById("sendButton");

        // Listen for messages from server
        ws.onmessage = (event) => {
            const li = document.createElement("li");
            li.textContent = `Server: ${event.data}`;
            messages.appendChild(li);
        };
```

# WebSocket Example: JavaScript Client ii

```
        // Send message to server
        sendButton.onclick = () => {
            const message = messageBox.value;
            ws.send(message);
            const li = document.createElement("li");
            li.textContent = `Client: ${message}`;
            messages.appendChild(li);
            messageBox.value = "";
        };

        ws.onopen = () => console.log("Connected to server");
        ws.onclose = () => console.log("Disconnected");
    </script>
</body>
</html>
```

# WebSocket Example: Python Server i

Using the `websockets` library: `pip install websockets`

```python
# ws_server.py
import asyncio
import websockets

connected_clients = set()

async def chat_handler(websocket, path):
    # Register new client
    connected_clients.add(websocket)
    print(f"Client connected: {websocket.remote_address}")

    try:
        # Iterate over messages
        async for message in websocket:
            print(f"Received from {websocket.remote_address}: {message}")

            # Broadcast message to all other clients
            for client in connected_clients:
                if client != websocket:
                    await client.send(f"[{websocket.remote_address[1]}]: {message}")
```

# WebSocket Example: Python Server ii

```python
    except websockets.ConnectionClosed:
        print(f"Client disconnected: {websocket.remote_address}")
    finally:
        # Unregister client
        connected_clients.remove(websocket)

async def main():
    print("Starting WebSocket server on ws://localhost:8765")
    async with websockets.serve(chat_handler, "localhost", 8765):
        await asyncio.Future()  # Run forever

if __name__ == "__main__":
    asyncio.run(main())
```

# Advantages & When to Use i

**Why WebSockets?**

- **Low Latency:** No HTTP overhead for each message.
- **Real-Time:** Server can push data *instantly*.
- **Efficient:** Replaces constant polling, saving bandwidth and server load.

**When to Use?**

- Real-time chat applications
- Live sports tickers or stock feeds
- Multiplayer online games
- Collaborative editing (like Google Docs)

# Advantages & When to Use ii

## Advantage of Browser as Platform

- **Rich UI:** HTML & CSS provide a powerful rendering engine.
- **Powerful Logic:** JavaScript is a mature, high-performance language.
- **Ubiquity:** Runs on every desktop, laptop, and phone.
- **Integrated APIs:** Access to graphics (WebGL), audio, storage, and more.

# MQTT: The IoT Protocol i

What if you have thousands of tiny, battery-powered devices on an unreliable network?

- TCP is too heavy.
- HTTP is *way* too heavy.
- They might not even have a stable IP address.

**MQTT (Message Queuing Telemetry Transport)**

- A lightweight, **publish/subscribe** protocol.
- Designed for constrained devices (IoT) and low-bandwidth networks.
- Minimal overhead (can be a 2-byte header).

# Communication Pattern: Publish/Subscribe i

This is a fundamental shift from Request/Response.

- **Publisher:** Sends messages on a **Topic** (e.g., `home/livingroom/temp`). It doesn't know *who* is listening.
- **Subscriber:** Listens to one or more **Topics**. It doesn't know *who* published the message.
- **Broker:** The central server that receives *all* messages and routes them to the correct subscribers.

**This fully decouples clients from each other.**

# MQTT (Pub/Sub)

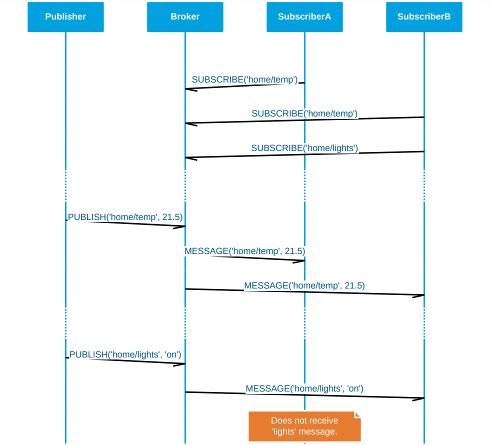



**Figure 5:** MQTT Pub/Sub pattern

# MQTT Protocol Details i

- **Topics:** Hierarchical strings (e.g., `building/floor1/room102/light`).
  - Subscribers can use wildcards:
    - +: Single-level (e.g., `building/+/room102/light`)
    - #: Multi-level (e.g., `building/floor1/#`)
- **Quality of Service (QoS):**
  - **QoS 0:** At most once. (Fire and forget, like UDP)
  - **QoS 1:** At least once. (Guarantees delivery, may have duplicates)
  - **QoS 2:** Exactly once. (Guarantees delivery, no duplicates. Slowest)

# MQTT Protocol Details ii

- **Last Will & Testament (LWT):** A message the broker sends *on behalf of* a client if it disconnects ungracefully. (e.g., `device/123/status` -> “offline”)

# Python Publisher Example i

Using the `paho-mqtt` library: `pip install paho-mqtt`

```
# publisher.py
import paho.mqtt.client as mqtt
import time
import random

def on_connect(client, userdata, flags, rc):
    print(f"Connected with result code {rc}")

client = mqtt.Client()
client.on_connect = on_connect

# Connect to a public broker (test.mosquitto.org)
client.connect("test.mosquitto.org", 1883, 60)
client.loop_start() # Start a background thread for handling network

try:
    while True:
        temperature = round(random.uniform(20.0, 25.0), 2)
        print(f"Publishing: {temperature}")

        # Publish message
```

# Python Publisher Example ii

```
        client.publish("myhome/livingroom/temperature", payload=temperature, qos=0)

        time.sleep(5)
except KeyboardInterrupt:
    print("Publishing stopped")
    client.loop_stop()
```

# Python Subscriber Example i

```python
# subscriber.py
import paho.mqtt.client as mqtt

# Callback when connecting
def on_connect(client, userdata, flags, rc):
    print(f"Connected with result code {rc}")
    # Subscribe to the topic once connected
    client.subscribe("myhome/livingroom/temperature")
    print("Subscribed to 'myhome/livingroom/temperature'")

# Callback when a message is received
def on_message(client, userdata, msg):
    print(f"Topic: {msg.topic} | Message: {msg.payload.decode()}")

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("test.mosquitto.org", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks,
# and handles reconnecting.
client.loop_forever()
```

# Other Communication Patterns i

Sockets, REST, WebSockets, and MQTT cover most cases, but other powerful patterns exist.

We'll look at two major examples:

- **Message Brokers (e.g., RabbitMQ):**
  - **Smart server, dumb clients.**
  - Manages complex routing, persistence, and delivery guarantees.
- **Brokerless Sockets (e.g., ZeroMQ):**
  - **Smart clients, no server.**
  - A library that provides high-level patterns (Pub/Sub, Push/Pull) over raw sockets.

# Pattern: Message Broker (RabbitMQ) i

Uses the **AMQP** protocol (or others). It's a server that acts as a post office.

- **Producer:** Sends a message to an `Exchange`.
- **Exchange:** Routes the message to one or more `Queues` based on rules ("routing key").
- **Queue:** A durable buffer that holds messages.
- **Consumer:** Pulls messages from a `Queue`.

**Advantages:**

- **Reliability:** Queues can persist messages to disk.
- **Decoupling:** Producer and Consumer don't know about each other.

# Pattern: Message Broker (RabbitMQ) ii

- **Complex Routing:** Fanout (broadcast), topic, and direct routing.
- **Load Balancing:** Multiple consumers can read from one queue.

**Use Case:** Microservice backends, task queues (e.g., Celery), financial transactions.

# Pattern: Brokerless (ZeroMQ / ØMQ) i

ZeroMQ is **not** a broker. It's a **socket library on steroids.** It gives you patterns, not just a raw data stream.

- **How it works:** You import `zmq` and create sockets with *patterns*.
- **Common Patterns:**
  - `REQ/REP`: Like REST, but faster and bi-directional.
  - `PUB/SUB`: Like MQTT, but *without a central broker*. (Subscribers connect directly to Publisher).
  - `PUSH/PULL`: Distributes work to a pool of "worker" nodes.

**Advantages:**

# Pattern: Brokerless (ZeroMQ / ØMQ) ii

- **Blazing Fast:** Can use in-process, IPC, or TCP communication.
- **Lightweight:** No single point of failure (broker).
- **Simple:** Easy to embed in any application.

**Use Case:** High-speed data (HPC), financial trading, inter-process communication.

# Summary: Choosing the Right Tool i

- **Raw Sockets (TCP/UDP):**
  - **Use:** Custom protocols, high-performance needs, games.
  - **Pattern:** Request/Response (or custom).
- **REST API (HTTP):**
  - **Use:** Standard web services, public APIs, mobile app backends.
  - **Pattern:** Request/Response.
- **WebSockets:**
  - **Use:** Real-time web (chat, live feeds, collaborative editing).
  - **Pattern:** Full-Duplex / Bidirectional.
- **MQTT:**
  - **Use:** IoT, constrained devices, unreliable networks.

# Summary: Choosing the Right Tool ii

  - **Pattern:** Publish/Subscribe (via Broker).
- **RabbitMQ (Broker):**
  - **Use:** Reliable microservice communication, task queues.
  - **Pattern:** Queues & Exchanges.
- **ZeroMQ (Brokerless):**
  - **Use:** High-speed, low-latency messaging.
  - **Pattern:** Various (Pub/Sub, Push/Pull, etc.).

# 📚 Further Resources i

**Python Docs**

- `socket` — Low-level networking interface
- `asyncio` — Asynchronous I/O

**Frameworks & Libraries**

- FastAPI
- `websockets` library
- Paho-MQTT Client
- RabbitMQ (and Python tutorial)
- ZeroMQ (and Python guide)

# 📚 Further Resources ii

**Protocols & Concepts**

- MQTT Standard
- WebSocket (MDN)
- The Global Interpreter Lock (GIL)