

# Rede Informática

## Introdução Engenharia Informática

---

Mário Antunes

November 10, 2025

Universidade de Aveiro

# Table of Contents i

Sockets: A Base

APIs REST: A Linguagem da Web

WebSockets: O Canal de Tempo Real

MQTT: O Protocolo de IoT

Outros Padrões de Comunicação

**Uma viagem pelos protocolos de comunicação modernos**

Um **socket** é um ponto final (endpoint) para comunicação. É uma abstração (representada como um descritor de ficheiro) na qual o seu programa pode escrever e ler.

- **Analogia:** Um socket é como uma “porta” na sua aplicação. Você atribui-lhe um **número de porta** (o número da porta) no **endereço IP** da sua máquina (o endereço da rua).

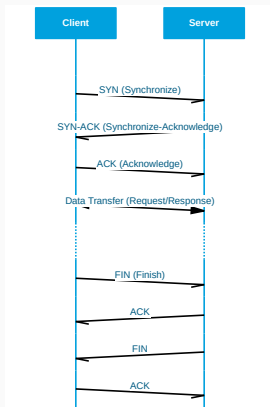
Dois tipos principais para comunicação na internet: **TCP** e **UDP**.

# TCP vs. UDP: Os Dois Pilares i

Característica	TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
<b>Conexão</b>	Orientado à conexão (estabelece uma sessão)	Sem conexão (disparar e esquecer)
<b>Fiabilidade</b>	<b>Fiável:</b> Garante a entrega e a ordem.	<b>Não fiável:</b> Sem garantia de entrega ou ordem.
<b>Overhead</b>	Alto (handshake de 3 vias, ACKs, controlo de fluxo)	Baixo (apenas um pequeno cabeçalho)
<b>Velocidade</b>	Mais lento, devido às verificações de fiabilidade	Mais rápido, sem configuração de conexão ou ACKs
<b>Casos de Uso</b>	Web (HTTP), Email (SMTP), Transferência de Ficheiros (FTP)	Streaming de vídeo, jogos online, DNS, VoIP
<b>Módulo Python</b>	<code>socket . SOCK_STREAM</code>	<code>socket . SOCK_DGRAM</code>

# Padrão de Comunicação TCP (Req/Rep)

O TCP usa um **handshake de 3 vias** para estabelecer uma conexão fiável.



**Figure 1:** Handshake de 3 vias TCP

# Servidor TCP Python (Eco) i

```
# tcp_server.py
import socket

HOST = '127.0.0.1' # Interface loopback padrão
PORT = 65432       # Porta para escutar

# Usar 'with' para gestão automática de recursos
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print(f"Servidor TCP a escutar em {HOST}:{PORT}")
    # conn é um novo objeto socket usável para enviar/receber dados
    # addr é o endereço associado ao cliente
    conn, addr = s.accept()
    with conn:
        print(f"Ligado por {addr}")
        while True:
            data = conn.recv(1024) # buffer de 1KB
            if not data:
                break # Cliente fechou a conexão
            print(f"Recebido: {data.decode()}")
            conn.sendall(data) # Enviar de volta (eco)
```

# Cliente TCP Python i

```
# tcp_client.py
import socket

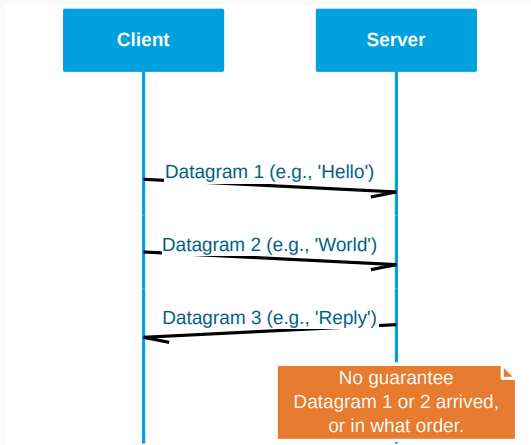
HOST = '127.0.0.1' # O hostname ou IP do servidor
PORT = 65432       # A porta usada pelo servidor

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Ola, mundo') # Enviar como bytes
    data = s.recv(1024)
    print(f"Eco recebido: {data.decode()}")
```



# Padrão de Comunicação UDP (Datagrama)

UDP é “disparar e esquecer”. Nenhuma conexão é estabelecida.



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

# Servidor UDP Python (Eco) i

```
# 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"Servidor UDP a escutar em {HOST}:{PORT}")

    while True:
        # recvfrom retorna dados E o endereço do remetente
        data, addr = s.recvfrom(1024)
        print(f"Recebido {data.decode()} de {addr}")

        if not data:
            break

    s.sendto(data, addr) # Enviar de volta (eco) para o remetente
```

# Cliente UDP Python i

```
# udp_client.py
import socket

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

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

    data, addr = s.recvfrom(1024)
    print(f"Eco recebido: {data.decode()} de {addr}")
```

## O Problema: I/O Blocante (Blocking I/O) i

Sockets tradicionais são **blocantes**.

- `s.accept()` **bloqueia** até um cliente se ligar.
- `conn.recv(1024)` **bloqueia** até chegarem dados.

Se está a tratar de um cliente, todos os outros clientes têm de esperar!

### Solução Tradicional: Multi-threading

- Cria uma thread por cliente.
- **Complexo:** Segurança entre threads (locks, race conditions).

## O Problema: I/O Blocante (Blocking I/O) ii

- **Elevado Uso de Recursos:** RAM, mudança de contexto (context switching) ao nível do SO.
- **Problema do Python (GIL):** O Global Interpreter Lock (GIL) no CPython impede a verdadeira execução paralela de código Python, limitando esta abordagem.

**I/O Assíncrono** (`asyncio` em **Python**) permite que uma única thread gira muitas conexões.

- Funciona tanto para **TCP** como para **UDP** usando um **event loop** (ciclo de eventos) para monitorizar os sockets.
- Quando um socket está “pronto” (ex: tem dados), o loop executa o código correspondente.
- As palavras-chave `async` e `await` “pausam” uma função, permitindo que o loop trabalhe noutras coisas, em vez de bloquear a thread inteira.

## A Solução: I/O Assíncrono (Async IO) ii

*Isto é **concorrência**, não paralelismo. Trata-se de esperar eficientemente.*

# Servidor TCP asyncio Python i

Este servidor pode lidar com milhares de clientes concorrentemente.

```
# asyncio_server.py
import asyncio

async def handle_client(reader, writer):
    """Callback para cada nova conexão de cliente"""
    addr = writer.get_extra_info('peername')
    print(f"Ligado por {addr}")

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

            message = data.decode()
            print(f"Recebido de {addr}: {message}")

            # Enviar de volta (eco)
            writer.write(data)
```



# Servidor TCP asyncio Python ii

```
        await writer.drain() # Esperar até o buffer ser descarregado (flushed)

except asyncio.CancelledError:
    print(f"Conexão com {addr} cancelada.")
finally:
    print(f"A fechar conexão com {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'A servir em {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

# Servidor UDP asyncio Python i

O AsyncIO também funciona para UDP, usando uma abordagem ligeiramente diferente baseada em “Protocolo”.

```
# asyncio_udp_server.py
import asyncio

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

    def datagram_received(self, data, addr):
        message = data.decode()
        print(f"Recebido {message} de {addr}")
        self.transport.sendto(data, addr) # Enviar de volta (eco)

async def main():
    loop = asyncio.get_running_loop()
    print("A iniciar servidor UDP em 127.0.0.1:65432")

    # Criar o endpoint do datagrama
    transport, protocol = await loop.create_datagram_endpoint(
```

# Servidor UDP asyncio Python ii

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

try:
    await asyncio.sleep(3600) # Servir por 1 hora
finally:
    transport.close()

asyncio.run(main())
```

Sockets são poderosos, mas de baixo nível. A maioria dos serviços web modernos não expõe sockets diretamente. Eles usam **APIs** (Application Programming Interfaces).

**REST** (REpresentational State Transfer) é o estilo arquitetural mais comum para APIs web.

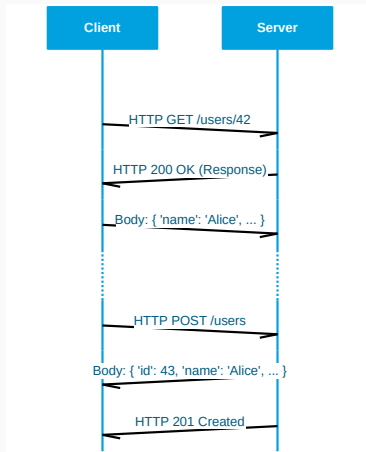
- **Não é um protocolo**, mas sim um conjunto de regras.
- Constrói *sobre* o HTTP (que constrói sobre o TCP).
- É **stateless** (sem estado): Cada pedido deve conter toda a informação necessária para o processar.

# Padrão de Comunicação REST (Req/Rep) i

Comunicação cliente-servidor sobre HTTP.

- **Recurso:** Uma entidade (ex: /users, /products/123).
- **Verbos:** Métodos HTTP (GET, POST, PUT, DELETE).
- **Dados:** Geralmente enviados/recebidos como **JSON**.

# Padrão de Comunicação REST (Req/Rep) ii



**Figure 3:** HTTP communication

# Formato de Dados: JSON i

**JSON** (JavaScript Object Notation) é o padrão *de facto* para troca de dados em APIs REST.

- Leve e legível por humanos.
- Fácil de processar (parse) e gerar por máquinas.
- Baseado na sintaxe de objetos JavaScript.

## Exemplo:

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

## Exemplo: FastAPI (Python) i

**FastAPI** é uma framework web Python moderna e de alta performance para construir APIs. É construída sobre `asyncio`.

1. Instalar: `pip install fastapi`  
`"uvicorn[standard]"`
2. Guardar como `main.py`:

```
# main.py
from fastapi import FastAPI

app = FastAPI()

# "Base de dados" em memória
items = {
    1: {"name": "Laptop", "price": 1200},
    2: {"name": "Mouse", "price": 50}
}
```



## Exemplo: FastAPI (Python) ii

```
@app.get("/")
def read_root():
    return {"message": "Ola, 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. Executar: `uvicorn main:app --reload`

4. Aceder no browser:

`http://127.0.0.1:8000/items/1`

# Sockets vs. APIs REST i

Característica	Sockets Puros	APIs REST
<b>Nível Protocolo</b>	<b>Baixo nível</b> (Nível de SO) TCP, UDP (protocolo personalizado)	<b>Alto nível</b> (Nível de Aplicação) HTTP/HTTPS (padronizado)
<b>Estado</b>	Pode ser stateful (com estado)	<b>Stateless</b> (sem estado) por design
<b>Formato Dados</b>	Qualquer coisa (binário, texto personalizado)	Normalmente JSON
<b>Usar Quando...</b>	Protocolo personalizado, alta velocidade, jogos, conexão persistente.	Serviços Web, apps móveis, APIs públicas, interoperabilidade.

**Conclusão:** *Você poderia construir uma API REST sobre sockets puros... mas estaria apenas a reinventar o HTTP. O REST dá-lhe uma enorme vantagem com padronização, segurança (HTTPS) e ferramentas.*

# WebSockets: O Canal de Tempo Real i

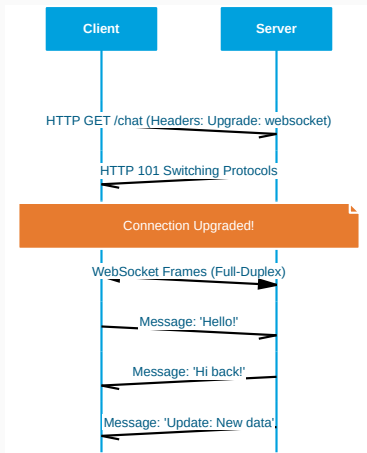
E se o REST for muito lento? E se o servidor precisar de enviar (push) dados para o cliente *sem* que lhe seja pedido?

- O HTTP é um modelo de *client-pull* (Req/Rep).
- O “polling” ineficiente (perguntar “há atualizações?” a cada 2 segundos) é uma solução comum, mas má.

## Solução: WebSockets

- Uma conexão **persistente e full-duplex** (bidirecional).
- Começa como um pedido HTTP “Upgrade” padrão.
- Uma vez estabelecido, é um canal “puro” semelhante ao TCP para enviar mensagens.

# Padrão de Comunicação WebSocket



**Figure 4:** WebSocket Communication

# Exemplo de WebSocket: Cliente JavaScript i

O browser é a plataforma *nativa* para WebSockets.

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

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

    // Escutar por mensagens do servidor
    ws.onmessage = (event) => {
      const li = document.createElement("li");
      li.textContent = `Servidor: ${event.data}`;
      messages.appendChild(li);
    };
  </script>
</body>
</html>
```

# Exemplo de WebSocket: Cliente JavaScript ii

```
// Enviar mensagem para o servidor
sendButton.onclick = () => {
  const message = messageBox.value;
  ws.send(message);
  const li = document.createElement("li");
  li.textContent = `Cliente: ${message}`;
  messages.appendChild(li);
  messageBox.value = "";
};

ws.onopen = () => console.log("Ligado ao servidor");
ws.onclose = () => console.log("Desligado");
</script>
</body>
</html>
```

# Exemplo de WebSocket: Servidor Python i

## Usando a biblioteca websockets: `pip install websockets`

```
# ws_server.py
import asyncio
import websockets

connected_clients = set()

async def chat_handler(websocket, path):
    # Registrar novo cliente
    connected_clients.add(websocket)
    print(f"Cliente ligado: {websocket.remote_address}")

    try:
        # Iterar sobre as mensagens
        async for message in websocket:
            print(f"Recebido de {websocket.remote_address}: {message}")

            # Transmitir (broadcast) mensagem para todos os outros clientes
            for client in connected_clients:
                if client != websocket:
```



# Exemplo de WebSocket: Servidor Python ii

```
        await client.send(f"[{websocket.remote_address[1]}]: {message}")

except websockets.ConnectionClosed:
    print(f"Cliente desligado: {websocket.remote_address}")
finally:
    # Cancelar registro do cliente
    connected_clients.remove(websocket)

async def main():
    print("A iniciar servidor WebSocket em ws://localhost:8765")
    async with websockets.serve(chat_handler, "localhost", 8765):
        await asyncio.Future() # Executar para sempre

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

## Porquê WebSockets?

- **Baixa Latência:** Sem o overhead do HTTP para cada mensagem.
- **Tempo Real:** O servidor pode enviar (push) dados *instantaneamente*.
- **Eficiente:** Substitui o polling constante, poupando largura de banda e carga no servidor.

## Quando Usar?

- Aplicações de chat em tempo real
- Resultados desportivos ou cotações da bolsa ao vivo
- Jogos online multijogador
- Edição colaborativa (como o Google Docs)

### Vantagem do Browser como Plataforma

- **UI Rica:** HTML & CSS fornecem um motor de renderização poderoso.
- **Lógica Poderosa:** JavaScript é uma linguagem madura e de alta performance.
- **Ubiquidade:** Corre em todos os desktops, portáteis e telemóveis.
- **APIs Integradas:** Acesso a gráficos (WebGL), áudio, armazenamento e mais.

# MQTT: O Protocolo de IoT i

E se tiver milhares de pequenos dispositivos a bateria numa rede não fiável?

- TCP é muito pesado.
- HTTP é *muito* mais pesado.
- Podem nem sequer ter um endereço IP estável.

## MQTT (Message Queuing Telemetry Transport)

- Um protocolo leve de **publish/subscribe** (publicar/subscrever).
- Desenhado para dispositivos com limitações (IoT) e redes de baixa largura de banda.
- Overhead mínimo (pode ter um cabeçalho de 2 bytes).

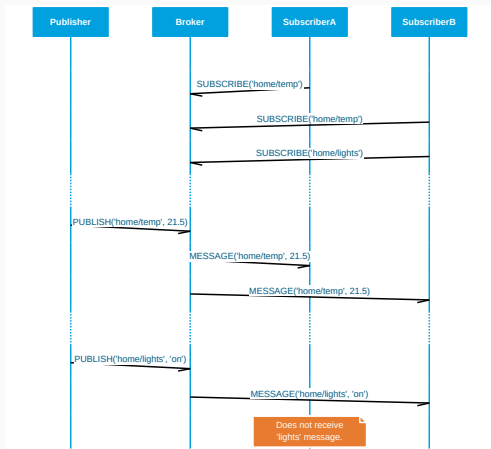
## Padrão de Comunicação: Publish/Subscribe i

Esta é uma mudança fundamental em relação ao Request/Response.

- **Publisher:** Envia mensagens num **Tópico** (ex: home/livingroom/temp). Não sabe *quem* está a ouvir.
- **Subscriber:** Escuta um ou mais **Tópicos**. Não sabe *quem* publicou a mensagem.
- **Broker:** O servidor central que recebe *todas* as mensagens e as encaminha para os subscritores corretos.

**Isto desacopla totalmente os clientes uns dos outros.**

# MQTT (Pub/Sub)



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

# Detalhes do Protocolo MQTT i

- **Tópicos:** Strings hierárquicas (ex: `building/floor1/room102/light`).
  - Subscritores podem usar wildcards:
    - `+`: Nível único (ex: `building/+ /room102/light`)
    - `#`: Multi-nível (ex: `building/floor1/#`)
- **Qualidade de Serviço (QoS):**
  - **QoS 0:** No máximo uma vez. (Disparar e esquecer, como o UDP)
  - **QoS 1:** Pelo menos uma vez. (Garante entrega, pode ter duplicados)
  - **QoS 2:** Exatamente uma vez. (Garante entrega, sem duplicados. O mais lento)



- **Last Will & Testament (LWT):** Uma mensagem que o broker envia *em nome de* um cliente se este se desligar abruptamente. (ex: `device/123/status -> "offline"`)

# Exemplo de Publisher Python i

Usando a biblioteca paho-mqtt: `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"Ligado com o código de resultado {rc}")

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

# Ligar a um broker público (test.mosquitto.org)
client.connect("test.mosquitto.org", 1883, 60)
client.loop_start() # Iniciar uma thread de fundo para tratar da rede

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

# Exemplo de Publisher Python ii

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

time.sleep(5)
except KeyboardInterrupt:
    print("Publicação parada")
    client.loop_stop()
```

# Exemplo de Subscriber Python i

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

# Callback ao ligar
def on_connect(client, userdata, flags, rc):
    print(f"Ligado com o código de resultado {rc}")
    # Subscrever o tópico assim que estiver ligado
    client.subscribe("myhome/livingroom/temperature")
    print("Subscrito a 'myhome/livingroom/temperature'")

# Callback quando uma mensagem é recebida
def on_message(client, userdata, msg):
    print(f"Tópico: {msg.topic} | Mensagem: {msg.payload.decode()}")

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

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

# Chamada bloqueante que processa o tráfego de rede, despacha callbacks
# e trata da reconexão.
client.loop_forever()
```

## Outros Padrões de Comunicação i

Sockets, REST, WebSockets e MQTT cobrem a maioria dos casos, mas existem outros padrões poderosos.

Vamos ver dois exemplos principais:

- **Message Brokers (ex: RabbitMQ):**
  - **Servidor inteligente, clientes “burros”.**
  - Gere roteamento complexo, persistência e garantias de entrega.
- **Sockets sem Broker (ex: ZeroMQ):**
  - **Clientes inteligentes, sem servidor.**
  - Uma biblioteca que fornece padrões de alto nível (Pub/Sub, Push/Pull) sobre sockets puros.

## Padrão: Message Broker (RabbitMQ) i

Usa o protocolo **AMQP** (ou outros). É um servidor que atua como uma estação de correios.

- **Producer:** Envia uma mensagem para um Exchange (Troca).
- **Exchange:** Encaminha a mensagem para uma ou mais Queues (Filas) com base em regras ("routing key").
- **Queue:** Um buffer durável que retém mensagens.
- **Consumer:** Retira (pull) mensagens de uma Queue.

### Vantagens:

- **Fiabilidade:** As filas podem persistir mensagens em disco.

## Padrão: Message Broker (RabbitMQ) ii

- **Desacoplamento:** O Producer e o Consumer não sabem um do outro.
- **Roteamento Complexo:** Fanout (broadcast), tópico e roteamento direto.
- **Balanceamento de Carga:** Múltiplos consumidores podem ler de uma fila.

**Caso de Uso:** Backends de microserviços, filas de tarefas (ex: Celery), transações financeiras.

## Padrão: Sem Broker (ZeroMQ / ØMQ) i

ZeroMQ **não** é um broker. É uma **biblioteca de sockets “com esteroides”**. Dá-lhe padrões, não apenas um fluxo de dados puro.

- **Como funciona:** Você importa zmq e cria sockets com *padrões*.
- **Padrões Comuns:**
  - REQ/REP: Como REST, mas mais rápido e bidirecional.
  - PUB/SUB: Como MQTT, mas *sem um broker central*. (Subscritores ligam-se diretamente ao Publisher).
  - PUSH/PULL: Distribui trabalho para um conjunto (pool) de “trabalhadores” (workers).



### Vantagens:

- **Extremamente Rápido:** Pode usar comunicação in-process, IPC ou TCP.
- **Leve:** Sem ponto único de falha (broker).
- **Simples:** Fácil de incorporar em qualquer aplicação.

**Caso de Uso:** Dados de alta velocidade (HPC), negociação financeira, comunicação entre processos.

# Resumo: Escolher a Ferramenta Certa i

- **Sockets Puros (TCP/UDP):**
  - **Uso:** Protocolos personalizados, necessidades de alta performance, jogos.
  - **Padrão:** Request/Response (ou personalizado).
- **API REST (HTTP):**
  - **Uso:** Serviços web padrão, APIs públicas, backends de apps móveis.
  - **Padrão:** Request/Response.
- **WebSockets:**
  - **Uso:** Web em tempo real (chat, feeds ao vivo, edição colaborativa).
  - **Padrão:** Full-Duplex / Bidirecional.
- **MQTT:**

## Resumo: Escolher a Ferramenta Certa ii

- **Uso:** IoT, dispositivos com limitações, redes não fiáveis.
- **Padrão:** Publish/Subscribe (via Broker).
- **RabbitMQ (Broker):**
  - **Uso:** Comunicação fiável entre microsserviços, filas de tarefas.
  - **Padrão:** Filas (Queues) & Trocas (Exchanges).
- **ZeroMQ (Sem Broker):**
  - **Uso:** Mensagens de alta velocidade e baixa latência.
  - **Padrão:** Vários (Pub/Sub, Push/Pull, etc.).



## Documentação Python

- `socket` — Interface de rede de baixo nível
- `asyncio` — I/O Assíncrono

## Frameworks & Bibliotecas

- `FastAPI`
- Biblioteca `websockets`
- Cliente `Paho-MQTT`
- `RabbitMQ` (e tutorial Python)
- `ZeroMQ` (e guia Python)



### Protocolos & Conceitos

- [Padrão MQTT](#)
- [WebSocket \(MDN\)](#)
- [O Global Interpreter Lock \(GIL\)](#)