Internet Protocols: TCP/IP, HTTP/HTTPS and WebSocket

1.How Protocols Fit Together

A **protocol** is an agreed set of rules for how two computers format, send, receive and acknowledge data. Protocols are organised in **layers**; each layer uses the services of the layer below it.

Layer (TCP/IP model)	Job	Examples
Application	What the app says	HTTP, HTTPS, WebSocket, FTP, SMTP, DNS
Transport	Process-to-process delivery	TCP, UDP
Internet	Addressing and routing between machines	IP (IPv4/IPv6), ICMP
Link / Network Access	Delivery on a local network	Ethernet, Wi-Fi

Encapsulation: when you send data, each layer wraps it with its own header.

```
Application data:  [ HTTP request ]
Transport:         [ TCP header | HTTP request ]            -> segment
Internet:          [ IP header | TCP header | data ]        -> packet
Link:              [ Frame header | IP packet | trailer ]  -> frame
```

The receiver unwraps in reverse order.

OSI vs TCP/IP: OSI has 7 layers (Physical, Data Link, Network, Transport, Session, Presentation, Application). The TCP/IP model merges the top three into Application and the bottom two into Link. Exams ask for both.

---

2. TCP/IP

2.1 IP (Internet Protocol)

- Provides **addressing** and **routing**. Each host has an **IP address**.
- **IPv4**: 32-bit, written like `192.168.1.10` (about 4.3 billion addresses).
- **IPv6**: 128-bit, written like `2001:db8::1` (practically unlimited).
- IP is **connectionless and unreliable (best effort)**: packets can be lost, duplicated, delayed or arrive out of order. IP does not care.
- Data is split into **packets**; each carries source IP, destination IP, **TTL** (hop limit, stops endless loops) and protocol number (6 = TCP, 17 = UDP).
- **Routers** forward packets hop by hop using routing tables.
- **Port numbers** (16-bit, 0 to 65535) identify the application on a host. An IP address plus a port is a **socket address**, e.g. `142.250.80.46:443`.

Common ports: HTTP 80, HTTPS 443, SSH 22, FTP 21, SMTP 25, DNS 53, and 0 to 1023 are well-known (privileged) ports.

2.2 TCP (Transmission Control Protocol)

TCP sits on top of IP and gives applications a **reliable, ordered, byte-stream** connection.

Key features:

1. **Connection-oriented**: a connection is set up before data flows.
2. **Reliable**: lost segments are retransmitted.
3. **Ordered**: sequence numbers let the receiver reassemble bytes in order.
4. **Error checking**: checksum in every segment.
5. **Flow control**: the receiver advertises a *window size* so a fast sender does not overwhelm a slow receiver.
6. **Congestion control**: the sender slows down when the network is congested (slow start, congestion avoidance, fast retransmit).

2.3 Three-Way Handshake (connection setup)

```
Client                          Server
  |  ---- SYN (seq=x) ---------->  |   "I want to connect"
  |  <--- SYN+ACK (seq=y,ack=x+1)  |   "OK, I'm ready too"
  |  ---- ACK (ack=y+1) -------->  |   "Confirmed"
  |        connection ESTABLISHED  |
```

2.4 Four-Way Termination (connection close)

```
Client                          Server
  |  ---- FIN ----------------->  |
  |  <--- ACK ------------------  |
  |  <--- FIN ------------------  |
  |  ---- ACK ----------------->  |   (client waits in TIME_WAIT)
```

2.5 TCP Header Fields (important ones)

Field	Meaning
Source / Destination port	Which apps are talking
Sequence number	Position of the first byte in this segment
Acknowledgement number	Next byte the receiver expects
Flags	SYN, ACK, FIN, RST, PSH, URG
Window size	Flow control (how much the receiver can accept)
Checksum	Error detection

2.6 TCP vs UDP

	TCP	UDP
Connection	Yes (handshake)	No
Reliability	Guaranteed, retransmits	None
Ordering	Ordered	Not ordered
Speed / overhead	Slower, more overhead	Faster, light
Use cases	Web, email, file transfer	Video calls, gaming, DNS, streaming

2.7 Socket Programming (the base for everything else)

A **socket** is the programming endpoint for network communication. Server steps: `socket()`, `bind()`, `listen()`, `accept()`, `recv()/send()`, `close()`. Client steps: `socket()`, `connect()`, `send()/recv()`, `close()`.

Program 1: TCP echo server (`tcp_server.py`)

```
import socket

HOST, PORT = '127.0.0.1', 5000

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind((HOST, PORT))
    server.listen()
    print(f'Server listening on {HOST}:{PORT}')

    conn, addr = server.accept()          # blocks until a client connects
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)        # receive up to 1024 bytes
            if not data:                  # empty = client closed
                break
            print('Received:', data.decode())
            conn.sendall(b'Echo: ' + data)
```

Program 2: TCP client (`tcp_client.py`)

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client:
    client.connect(('127.0.0.1', 5000))   # triggers the 3-way handshake
    client.sendall(b'Hello server')
    reply = client.recv(1024)
    print('Server replied:', reply.decode())
```

Run the server first in one terminal, then the client in another.

Program 3: Multi-client TCP server using threads (`threaded_server.py`)

```
import socket
import threading

def handle_client(conn, addr):
    print('Client connected:', addr)
    with conn:
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data.upper())
    print('Client disconnected:', addr)

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind(('0.0.0.0', 5000))
server.listen()
print('Listening on port 5000...')

while True:
    conn, addr = server.accept()
    threading.Thread(target=handle_client, args=(conn, addr), daemon=True).start()
```

Program 4: UDP server and client (for comparison)

```
# udp_server.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)   # DGRAM = UDP
s.bind(('127.0.0.1', 6000))
print('UDP server ready')
while True:
    data, addr = s.recvfrom(1024)       # no connection, just datagrams
    print('From', addr, ':', data.decode())
    s.sendto(b'Got it', addr)
```

```
# udp_client.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b'Hello UDP', ('127.0.0.1', 6000))
reply, _ = s.recvfrom(1024)
print(reply.decode())
```

Useful network commands: `ping host` (reachability, uses ICMP), `traceroute host` (Linux/macOS) or `tracert host` (Windows), `netstat -an` or `ss -tuln` (open ports and connections), `nslookup domain` (DNS lookup), `ipconfig` / `ifconfig` / `ip addr` (your IP).

---

3. HTTP (HyperText Transfer Protocol)

HTTP is an **application-layer** protocol for transferring web resources (HTML, images, JSON, etc.). It runs over TCP (port 80) and follows the **client-server, request-response** model.

3.1 Key Characteristics

- **Stateless**: the server does not remember previous requests. State is added with **cookies, sessions or tokens**.
- **Text-based** (HTTP/1.x) and human-readable headers.
- **Request/response**: the client always initiates.
- Resources are identified by a **URL**: `scheme://host:port/path?query#fragment`, e.g. `https://example.com:443/products?id=5#reviews`.

3.2 Request Structure

```
GET /index.html HTTP/1.1            <- request line (method, path, version)
Host: example.com                   <- headers
User-Agent: Mozilla/5.0
Accept: text/html
                                    <- blank line
(optional body)                     <- body (for POST/PUT)
```

3.3 Response Structure

```
HTTP/1.1 200 OK                     <- status line
Content-Type: text/html
Content-Length: 1256

<html>...</html>                    <- body
```

3.4 Methods

Method	Purpose	Safe	Idempotent
GET	Retrieve a resource	Yes	Yes
POST	Create / submit data	No	No
PUT	Replace a resource	No	Yes
PATCH	Partially update	No	No
DELETE	Remove a resource	No	Yes
HEAD	Like GET, headers only	Yes	Yes
OPTIONS	Which methods are allowed (CORS)	Yes	Yes

Safe means it does not change server state. *Idempotent* means repeating it gives the same result.

3.5 Status Codes

Range	Meaning	Examples
1xx	Informational	101 Switching Protocols
2xx	Success	200 OK, 201 Created, 204 No Content

Range	Meaning	Examples
3xx	Redirection	301 Moved Permanently, 302 Found, 304 Not Modified
4xx	Client error	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 429 Too Many Requests
5xx	Server error	500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable

3.6 Important Headers

Header	Use
Host	Which site (required in HTTP/1.1)
Content-Type	Format of the body (`application/json`, `text/html`)
Content-Length	Size of the body in bytes
Authorization	Credentials (`Bearer <token>`)
Cookie / Set-Cookie	Session state
Cache-Control	Caching rules
User-Agent	Client software
Location	Redirect target

3.6.1 Cookies, Sessions, Caching

- **Cookie**: small data the server sends with `Set-Cookie`; the browser returns it with every later request, so the server can recognise you.
- **Session**: server stores user data and gives the client a session ID (usually in a cookie).
- **Caching**: `Cache-Control`, `ETag` and `Last-Modified` let clients reuse responses; a `304 Not Modified` means the cached copy is still valid.

3.7 HTTP Versions

Version	Highlights
HTTP/1.0	One request per TCP connection
HTTP/1.1	Persistent connections (keep-alive), `Host` header, chunked transfer, but head-of-line blocking
HTTP/2	Binary framing, **multiplexing** many requests on one connection, header compression, server push
HTTP/3	Runs over **QUIC (UDP)** instead of TCP, faster setup, no TCP head-of-line blocking

3.8 Programs

Program 5: Simple HTTP server using Python's standard library (`http_server.py`)

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

class Handler(BaseHTTPRequestHandler):

    def do_GET(self):
        if self.path == '/':
            body = b'<h1>Hello from my HTTP server</h1>'
            self.send_response(200)
            self.send_header('Content-Type', 'text/html')
            self.send_header('Content-Length', str(len(body)))
            self.end_headers()
            self.wfile.write(body)
        elif self.path == '/api/hello':
            body = json.dumps({'message': 'Hello, JSON!'}).encode()
            self.send_response(200)
            self.send_header('Content-Type', 'application/json')
            self.send_header('Content-Length', str(len(body)))
            self.end_headers()
            self.wfile.write(body)
        else:
            self.send_error(404, 'Page not found')

    def do_POST(self):
        length = int(self.headers.get('Content-Length', 0))
        data = json.loads(self.rfile.read(length))
        body = json.dumps({'received': data}).encode()
        self.send_response(201)
        self.send_header('Content-Type', 'application/json')
        self.send_header('Content-Length', str(len(body)))
        self.end_headers()
        self.wfile.write(body)

if __name__ == '__main__':
    server = HTTPServer(('127.0.0.1', 8000), Handler)
    print('Serving on http://127.0.0.1:8000')
    server.serve_forever()
```

Open `http://127.0.0.1:8000` in a browser to test GET.

Program 6: HTTP client using `requests` (`pip install requests`)

```
import requests

# GET with query parameters
r = requests.get('https://httpbin.org/get', params={'name': 'Asha'})
print(r.status_code)        # 200
print(r.headers['Content-Type'])
print(r.json())

# POST with JSON body
r = requests.post('https://httpbin.org/post', json={'user': 'asha', 'age': 20})
```

```
print(r.status_code, r.json()['json'])

# Custom headers
r = requests.get('https://httpbin.org/headers', headers={'X-Demo': 'hello'})
print(r.text)

# Sessions keep cookies between requests
s = requests.Session()
s.get('https://httpbin.org/cookies/set/token/abc123')
print(s.get('https://httpbin.org/cookies').json())
```

Program 7: Raw HTTP request over a TCP socket (shows HTTP is just text over TCP)

```
import socket

host = 'example.com'
request = 'GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n'

with socket.create_connection((host, 80)) as s:
    s.sendall(request.encode())
    response = b''
    while True:
        chunk = s.recv(4096)
        if not chunk:
            break
        response += chunk

print(response.decode(errors='ignore')[:500])
```

Program 8: Minimal REST API with Flask (`pip install flask`)

```
from flask import Flask, jsonify, request

app = Flask(__name__)
tasks = [{'id': 1, 'title': 'Learn HTTP'}]

@app.get('/tasks')
def list_tasks():
    return jsonify(tasks)

@app.post('/tasks')
def add_task():
    task = {'id': len(tasks) + 1, 'title': request.json['title']}
    tasks.append(task)
    return jsonify(task), 201

@app.delete('/tasks/<int:task_id>')
def delete_task(task_id):
    global tasks
    tasks = [t for t in tasks if t['id'] != task_id]
    return '', 204

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Test with: `curl http://127.0.0.1:5000/tasks` and `curl -X POST -H 'Content-Type: application/json' -d '{"title":"Practice"}' http://127.0.0.1:5000/tasks`.

Program 9: HTTP client in JavaScript (browser or Node 18+)

```
// GET
fetch('https://httpbin.org/get')
  .then(res => res.json())
  .then(data => console.log(data));

// POST with async/await
async function createUser() {
  const res = await fetch('https://httpbin.org/post', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name: 'Asha' })
  });
  console.log(res.status, await res.json());
}
createUser();
```

---

4. HTTPS (HTTP Secure)

HTTPS = HTTP + TLS (Transport Layer Security). Same HTTP messages, but encrypted. Default port **443**. (TLS replaced the older **SSL**.)

4.1 Why HTTP Alone Is Unsafe

HTTP sends everything as plain text, so anyone on the path (Wi-Fi owner, ISP, attacker) can read passwords or modify pages (a **man-in-the-middle attack**).

4.2 What TLS Provides

1. **Confidentiality**: data is encrypted.
2. **Integrity**: tampering is detected (message authentication codes).
3. **Authentication**: the server proves its identity with a **digital certificate**.

4.3 Cryptography Concepts

- **Symmetric encryption** (AES, ChaCha20): same key encrypts and decrypts. Fast, used for the actual data.
- **Asymmetric encryption** (RSA, ECC): public key encrypts or verifies, private key decrypts or signs. Slow, used for key exchange and certificates.
- **Hashing** (SHA-256): one-way fingerprint for integrity.
- **Digital certificate (X.509)**: contains the site's public key, domain name, validity, and the **Certificate Authority (CA)** signature. Browsers trust a set of root CAs; trust flows through a **chain of trust** (root CA, intermediate CA, site certificate).

4.4 TLS Handshake (simplified, TLS 1.3)

```
Client                                  Server
  | -- ClientHello (versions, ciphers,     |
  |      key share) ---------------------> |
  | <-- ServerHello (chosen cipher,        |
```

```
  |      key share), Certificate,          |
  |      Finished ------------------------ |
  |  Client verifies certificate chain     |
  | -- Finished ------------------------> |
  |   Both derive the same session keys    |
  | <======= Encrypted HTTP traffic =====> |
```

Steps in words: (1) agree on TLS version and cipher suite; (2) server sends its certificate; (3) client verifies it against trusted CAs and the domain name; (4) both sides compute a shared **session key** using key exchange (Diffie-Hellman); (5) all later data is encrypted with that symmetric key. TLS 1.3 needs only one round trip; TLS 1.2 needed two.

4.5 HTTP vs HTTPS

	HTTP	HTTPS
Port	80	443
Encryption	None	TLS
Certificate	Not needed	Required
Security	Vulnerable to sniffing and tampering	Protected
SEO / browser trust	Marked "Not secure"	Padlock, preferred by search engines

Related: **HSTS** header forces browsers to always use HTTPS for a site.

4.6 Programs

Program 10: Inspect a server's TLS certificate (`tls_info.py`)

```
import socket
import ssl

hostname = 'www.google.com'
context = ssl.create_default_context()      # verifies certificates by default

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as tls:
        print('TLS version :', tls.version())
        print('Cipher      :', tls.cipher())
        cert = tls.getpeercert()
        print('Issuer      :', dict(x[0] for x in cert['issuer']))
        print('Subject     :', dict(x[0] for x in cert['subject']))
        print('Valid from  :', cert['notBefore'])
        print('Valid until :', cert['notAfter'])
```

Program 11: HTTPS request (certificates verified automatically)

```
import requests

r = requests.get('https://www.python.org')
print(r.status_code, r.url)
```

```
# requests verifies the certificate; a bad one raises requests.exceptions.SSLError
# Never use verify=False in real code.
```

Program 12: Generate a self-signed certificate (terminal, needs OpenSSL)

```
openssl req -x509 -newkey rsa:2048 -nodes -keyout key.pem -out cert.pem -days 365 -
subj '/CN=localhost'
```

Program 13: HTTPS server using that certificate (`https_server.py`)

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
import ssl

server = HTTPServer(('127.0.0.1', 8443), SimpleHTTPRequestHandler)

context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile='cert.pem', keyfile='key.pem')
server.socket = context.wrap_socket(server.socket, server_side=True)

print('HTTPS server on https://127.0.0.1:8443')
server.serve_forever()
```

The browser will warn because the certificate is self-signed (no trusted CA). That is expected for local testing. In production, use a free CA such as **Let's Encrypt**.

---

5. WebSocket

5.1 The Problem WebSocket Solves

HTTP is request-response: the server cannot push data to the client on its own. Workarounds were inefficient:

- **Polling**: client asks every few seconds (wasteful).
- **Long polling**: server holds the request open until data exists (complex, still one-way per request).

WebSocket (RFC 6455) gives a **persistent, full-duplex (two-way), low-latency** connection over a single TCP connection. Either side can send messages at any time.

5.2 How It Works

1. Client sends a normal HTTP request asking to **upgrade**:

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

2. Server agrees:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
```

```
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

3. The connection stays open and now carries **WebSocket frames** instead of HTTP messages. Either side can send text or binary frames, plus **ping/pong** (keep-alive) and **close** frames.

```
Client                          Server
  | --- HTTP Upgrade request --> |
  | <-- 101 Switching Protocols  |
  | <====== open channel =====>  |
  | --- message --------------> |
  | <-- message --------------- |   (server pushes anytime)
  | --- close ----------------> |
```

- URL schemes: `ws://` (plain, port 80) and `wss://` (over TLS, port 443). Always use `wss://` in production.
- Frames have very small headers (2 to 14 bytes) compared with HTTP headers on every request.

5.3 HTTP vs WebSocket

	HTTP	WebSocket
Direction	Client asks, server answers	Both sides send anytime
Connection	Per request (or keep-alive)	One long-lived connection
Overhead	Headers on every request	Small frame headers
Best for	Pages, REST APIs	Chat, live scores, games, trading, collaborative editing, notifications

5.4 Related Technologies

- **Server-Sent Events (SSE)**: server to client only, simpler, auto-reconnect.
- **Socket.IO**: library built on WebSocket (with fallbacks, rooms, reconnection).
- **WebRTC**: peer-to-peer audio/video.

5.5 Programs

Install the library first: `pip install websockets`

Program 14: Echo WebSocket server (`ws_server.py`)

```
import asyncio
import websockets

async def echo(websocket):
    async for message in websocket:          # runs for each incoming message
        print('Received:', message)
        await websocket.send(f'Echo: {message}')

async def main():
    async with websockets.serve(echo, 'localhost', 8765):
        print('WebSocket server on ws://localhost:8765')
        await asyncio.Future()               # run forever
```

```
asyncio.run(main())
```

Program 15: WebSocket client (`ws_client.py`)

```
import asyncio
import websockets

async def main():
    async with websockets.connect('ws://localhost:8765') as ws:
        await ws.send('Hello WebSocket')
        print(await ws.recv())
        await ws.send('Second message')
        print(await ws.recv())

asyncio.run(main())
```