Programming for Automation


TEKsystems
Global Services
WORKFORCE
DEVELOPMENT

Python

Develop a passion for learning.

Sockets

A **socket** is a fundamental endpoint for network communication. Sockets provide an **abstraction** that allows programmers to connect devices over a network without needing to manage low-level packet handling.

Sockets operate across:

- **Layer 3 (Network layer)** – IP addressing and routing
- **Layer 4 (Transport layer)** – sending and receiving data

Every time data moves across a network, it does so through sockets.

Client Server Model

Network communication typically follows a **client–server model**.

- A **server** waits for incoming connections
- A **client** initiates a connection to a server
- Roles are defined by behavior, not by the program itself
- A machine can act as both a client and a server
- Clients always start communication; servers respond.

The client–server model defines who initiates communication and who waits.

Listening Sockets

Servers (like web servers) create a **listening socket** so they can wait for clients on a specific port.

- Server chooses an **IP + port** (example: 0.0.0.0:80)
- Server **binds** the socket to that IP + port
- Server enters **listening mode** to wait for clients
- This is what it means for a port to be “open”

A listening socket is how a server advertises, “I’m ready for connections on this port.”

Client Sockets

A **client** is an application that initiates network communication to request data or services from another system.

Examples of clients include web browsers, API consumers, database clients, and command-line tools like curl.

- Clients **create sockets** just like servers
- Client socket is used to initiate a connection
- Client knows the **server address** (IP or hostname)
- Client knows the **destination port**
- Client sends requests and receives responses

A client uses a socket to initiate communication with a server.

UDP

UDP is a connectionless protocol that sends data without establishing a session.

- No handshake or connection setup
- Messages are sent independently
- Delivery and order are not guaranteed
- Lower overhead and faster transmission

UDP prioritizes speed and simplicity over reliability.

When to use UDP

UDP should be used when speed and low overhead are more important than guaranteed delivery.

- Data can be lost without breaking the application
- Messages are small and independent
- Low latency is critical
- No connection setup is required

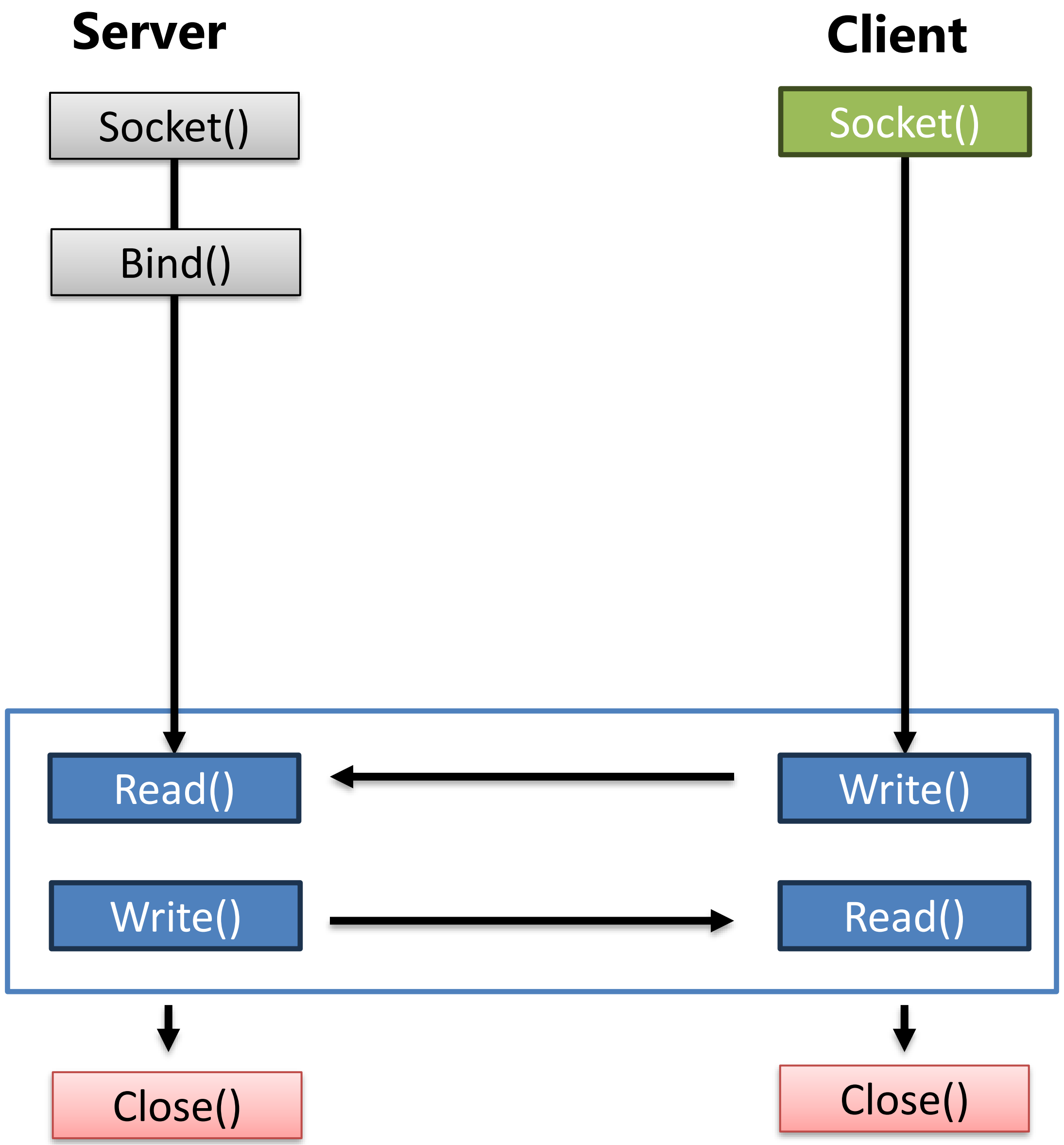
UDP is ideal for applications where occasional loss is acceptable in exchange for speed.

UDP Use Cases

- **DNS** – fast name resolution
- **Online gaming** – real-time position and state updates
- **Live video and audio streaming** – timely delivery over perfect delivery
- **Syslog and metrics** – lightweight, fire-and-forget messages

UDP is chosen when late data is worse than lost data.

UDP



Server
Client
Socket()
Bind()
Socket()
Read()
Write()
Write()
Read()
Close()
Close()

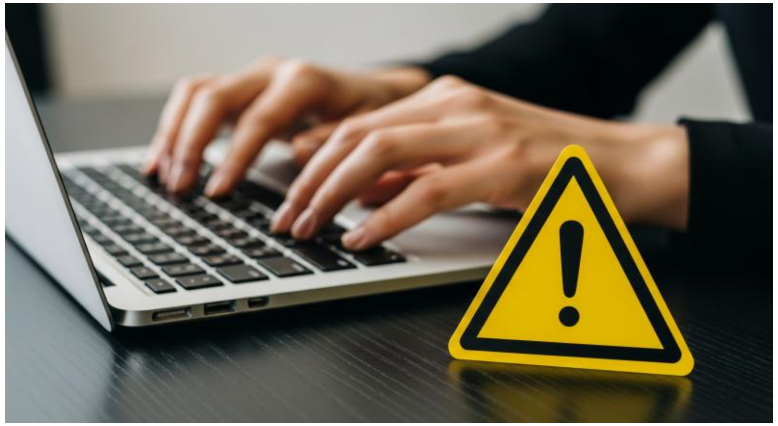
TCP

TCP is a connection-oriented protocol that establishes a reliable session before data is exchanged.

- Connection must be established first
- Uses a handshake to synchronize both sides
- Guarantees data delivery and ordering
- Detects and retransmits lost data

TCP prioritizes reliability over speed.

When to use TCP

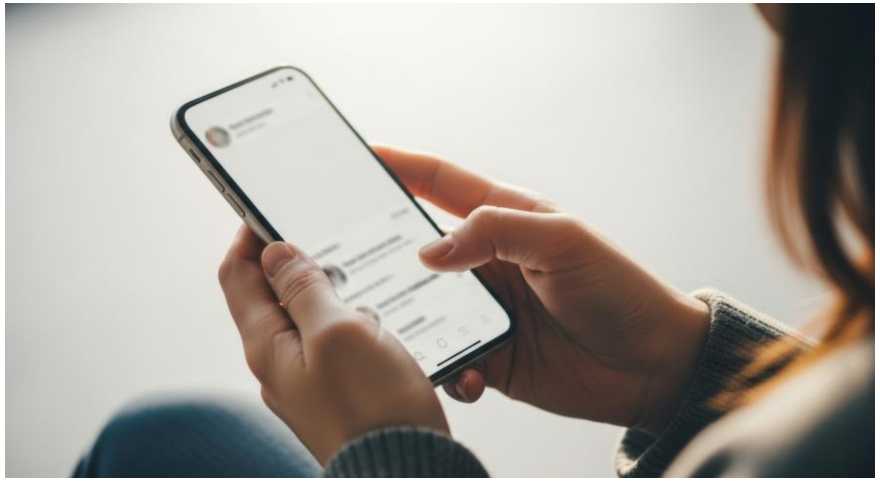


TCP should be used when reliability and correctness are more important than raw speed.

- Data must arrive **complete and in order**
- Lost data must be retransmitted
- Connection state is required
- Suitable for long-lived communication

TCP is ideal for applications where correctness matters more than latency.

TCP Use Cases



- **HTTP / HTTPS** – web traffic and APIs
- **Messaging applications** – chat and notifications
- **File transfers** – complete and correct delivery
- **Databases** – queries and transactions

TCP is chosen when correctness and completeness matter more than speed.

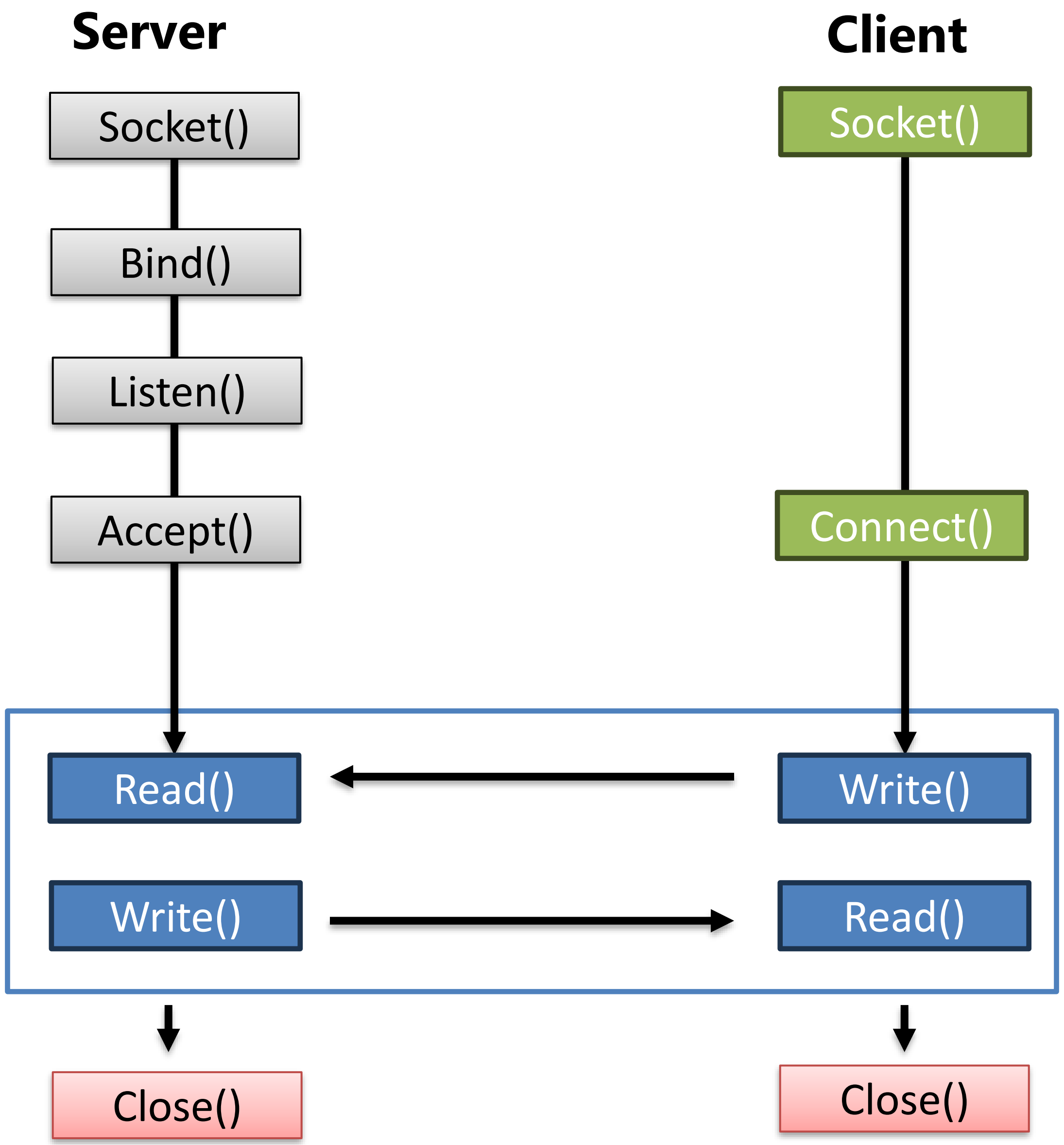
TCP Accepting Connections

When a client connects to a listening port, the server uses accept() to take the next connection from the waiting queue.

- Client initiates the connection (connect())
- Server receives the connection request (queued by the OS)
- Server calls accept() to get a **new socket** for that client
- The original listening socket stays open to accept more clients

Closing sentence: accept() creates a dedicated connection socket for one client while the server keeps listening for others.

TCP


Server
Client
Socket()
Bind()
Listen()
Accept()
Socket()
Connect()
Read()
Write()
Write()
Read()
Close()
Close()

TCP Connection Handshake

For TCP communication, a connection must be established before data can be exchanged between a client and a server.

- Client initiates the connection request
- Server acknowledges the request
- Client confirms the acknowledgement
- Connection is established

The TCP handshake ensures both sides are ready to communicate.

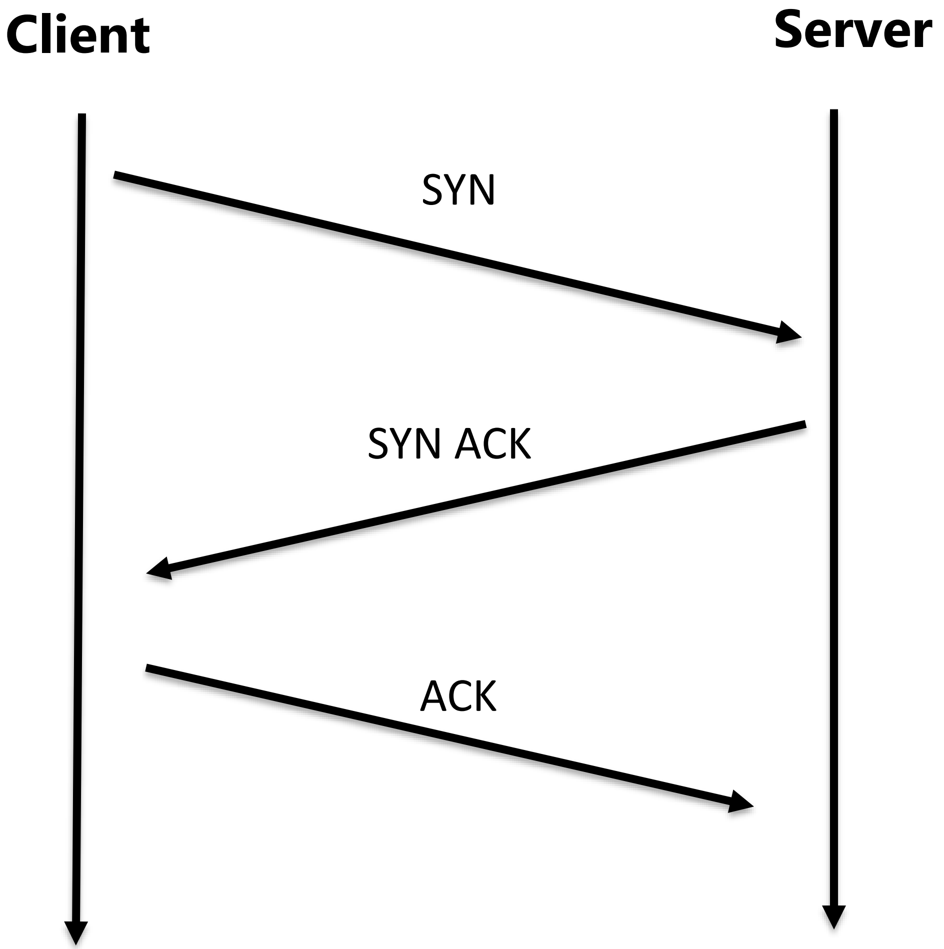
TCP Three Way Handshake (SYN/SYN-ACK/ACK)

TCP uses a three-step handshake to establish a reliable connection and synchronize both sides before data transfer begins.

SYN: Client sends a synchronization request asking to start a connection
SYN-ACK: Server acknowledges the request and agrees to synchronize
ACK: Client confirms the acknowledgement and completes the connection

This exchange synchronizes connection state and sequence numbers on both sides.

TCP Three Way Handshake



Blocking IO

In blocking I/O, a program pauses execution when a network operation cannot complete immediately.

- accept() waits for a client to connect
- recv() waits for data to arrive
- send() may wait if buffers are full
- Execution resumes only after the operation completes

Blocking I/O pauses execution at network calls until the OS completes the operation.

Non-Blocking IO

In non-blocking I/O, network operations return immediately even if they cannot complete.

- Operations do not pause execution
- Calls may return partial data or no data
- Program must check readiness and retry later
- Often combined with polling or event notification

Non-blocking I/O avoids pausing execution by deferring work until data is ready.

When to use Blocking vs Non-Blocking

Blocking and non-blocking I/O are chosen based on simplicity, scale, and concurrency needs.

Blocking is simpler and easier to reason about
Blocking works well with threads or processes

Non-blocking scales better with many connections
Non-blocking is common in high-performance servers

Blocking favors simplicity, while non-blocking favors scalability.

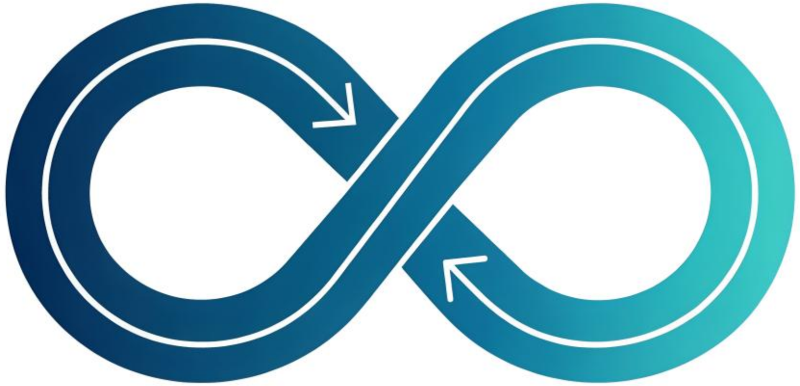
Thread-Per-Client Model

The thread-per-client model uses **blocking** I/O, but handles multiple clients by assigning each one its own thread.

- Each client connection runs in a separate thread
- Blocking calls only pause the current thread
- Other clients continue running in parallel
- Simpler than non-blocking or async designs

Blocking is isolated per thread, allowing concurrency without changing blocking behavior.

Async Event Loop Model



The async event loop model **avoids blocking** by handling many clients within a single thread.

- Uses non-blocking I/O
- One event loop manages many connections
- Operations resume only when data is ready
- No thread is blocked waiting on network calls

Async replaces blocking with event-driven execution to scale efficiently.

WebSockets

WebSockets provide a persistent, bidirectional communication channel between a client and server.

- Built on top of **TCP**
- Connection begins as an **HTTP request**
- Upgraded to a long-lived socket connection
- Allows server and client to send data at any time
- Focuses more on Layer 7 of OSI model

WebSockets enable real-time communication over a single persistent connection.

WebSockets Use Cases

WebSockets are used when applications need real-time, bidirectional communication over a persistent connection.

- Live chat and messaging applications
- Real-time dashboards and monitoring
- Multiplayer games and collaborative tools
- Notifications and live updates

WebSockets are ideal when both client and server need to push data instantly without repeated requests

Lab: Python Sockets

Congratulations

In this class, you learned that **sockets are the foundation of network communication**, enabling applications to send and receive data over a network.

You explored **TCP and UDP**, understanding when reliability and ordering matter versus when speed and simplicity are more important.

You also learned how **TCP uses the three-way handshake** to establish a reliable connection before data is transferred.

With these concepts applied in **Python**, you now have the foundation to understand, build, and troubleshoot networked applications.