Network Programming

BESE-VI – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

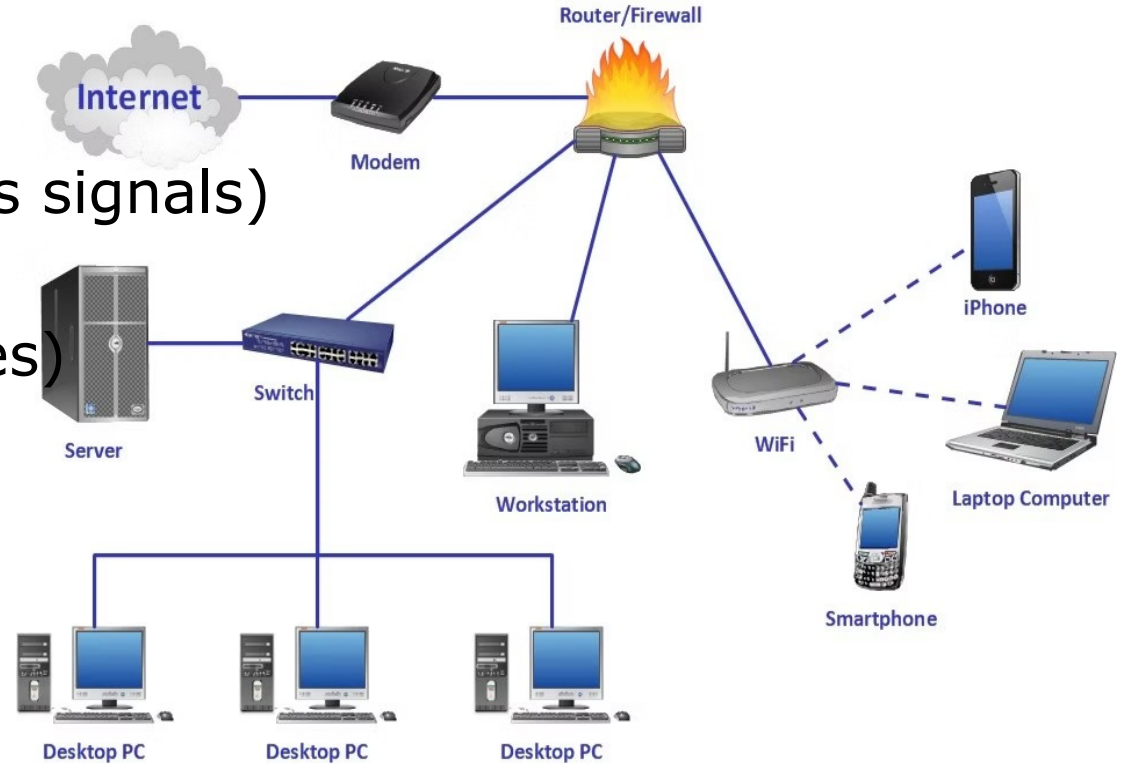
Chapter 1:
Network Programming Fundamentals
(5 hrs)

Outline

1. Introduction and importance of networking and network programming.
2. Fundamentals of Client server and P2P models
3. Basic Network Protocols
   - Characteristics and use cases of TCP, IP, UDP, SCTP
4. TCP state transition diagram.
5. Comparisons of protocols (TCP,UDP,SCTP)
6. Introduction to Sockets
   - Concept of sockets in network communication
   - Types of sockets: Stream (TCP) vs Datagram (UDP)

Introduction of Networking

- ***Networking*** refers to the interconnection of computers and other devices to share resources and information.
- It forms the backbone of modern communication systems, enabling everything from browsing the web to sending emails, video conferencing, and cloud computing.
- Networking involves:
  - Physical connections (cables, wireless signals)
  - Protocols (rules for communication)
  - Addressing systems (like IP addresses)
  - Data transmission methods



Importance of Networking

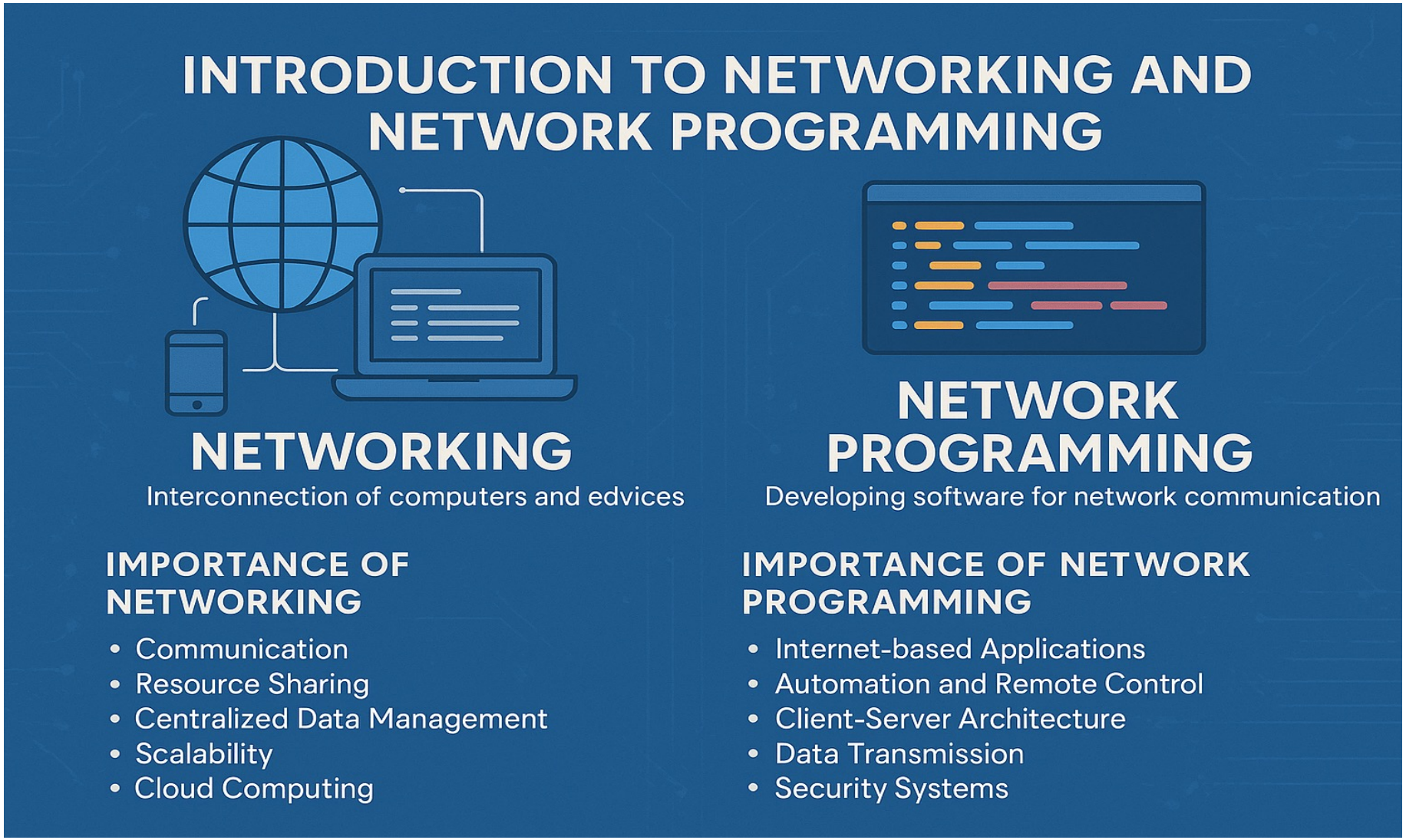
1. *Communication*: Enables instant messaging, emails, voice and video calls globally.
2. *Resource Sharing*: Devices like printers, files, and storage can be shared across networks.
3. *Centralized Data Management*: Networks allow centralized databases and server systems for better data organization and access control.
4. *Scalability*: Networking allows systems to grow by connecting more devices and services.
5. *Cloud Computing*: Networking is essential for cloud-based services, including storage, applications, and platforms.
6. *Distributed Systems*: Powers modern applications that run across multiple machines

Introduction of Network Programming

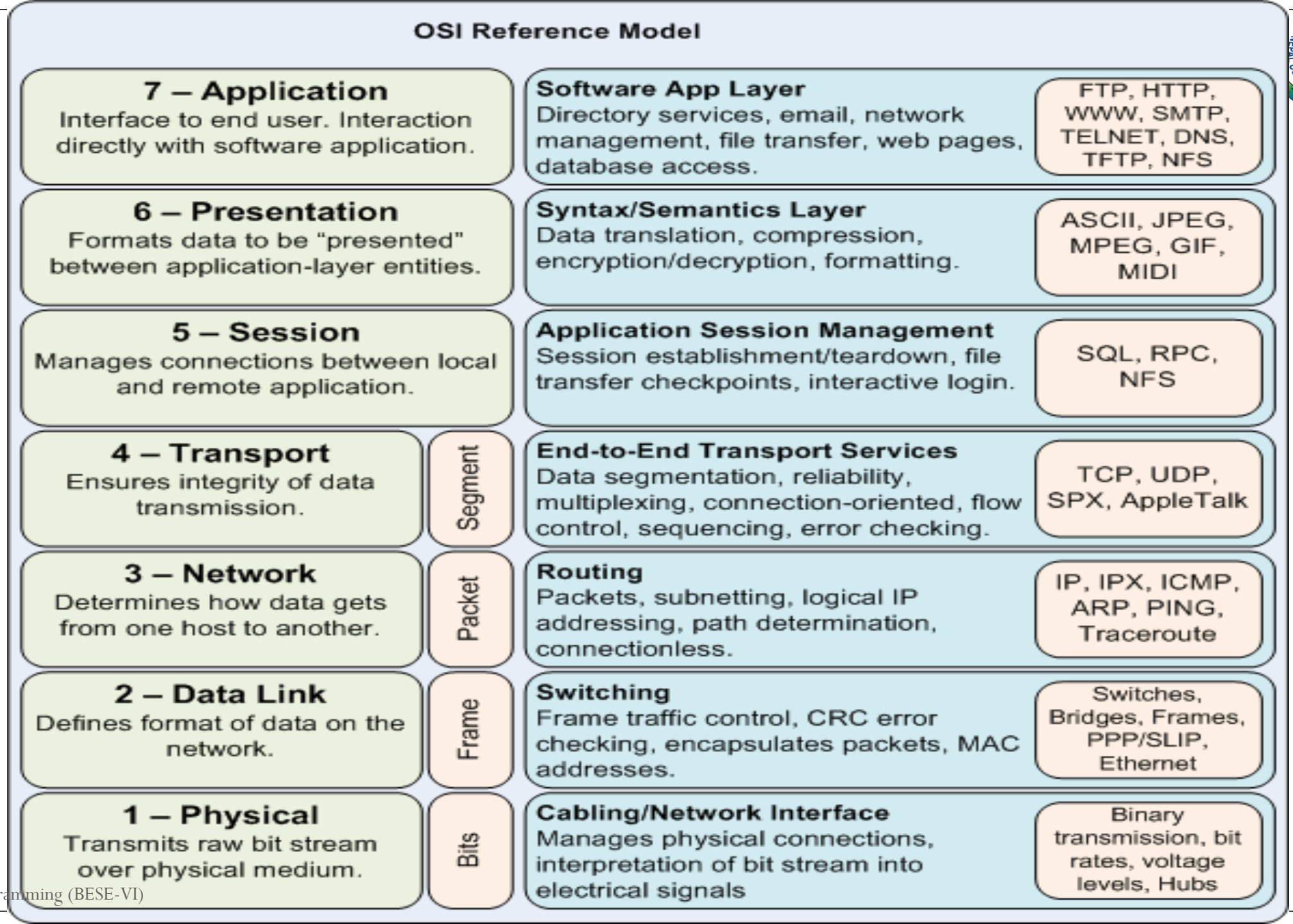
- **Network programming** involves writing software that enables processes (programs) to communicates over computer networks. It focuses on:
  - Creating applications that can send and receive data across networks(using APIs like sockets)
  - Implementing network protocols (like TCP/IP, HTTP, FTP etc)
  - Handling network connections and data transmission
  - Building client-server architectures

Importance of Network Programming

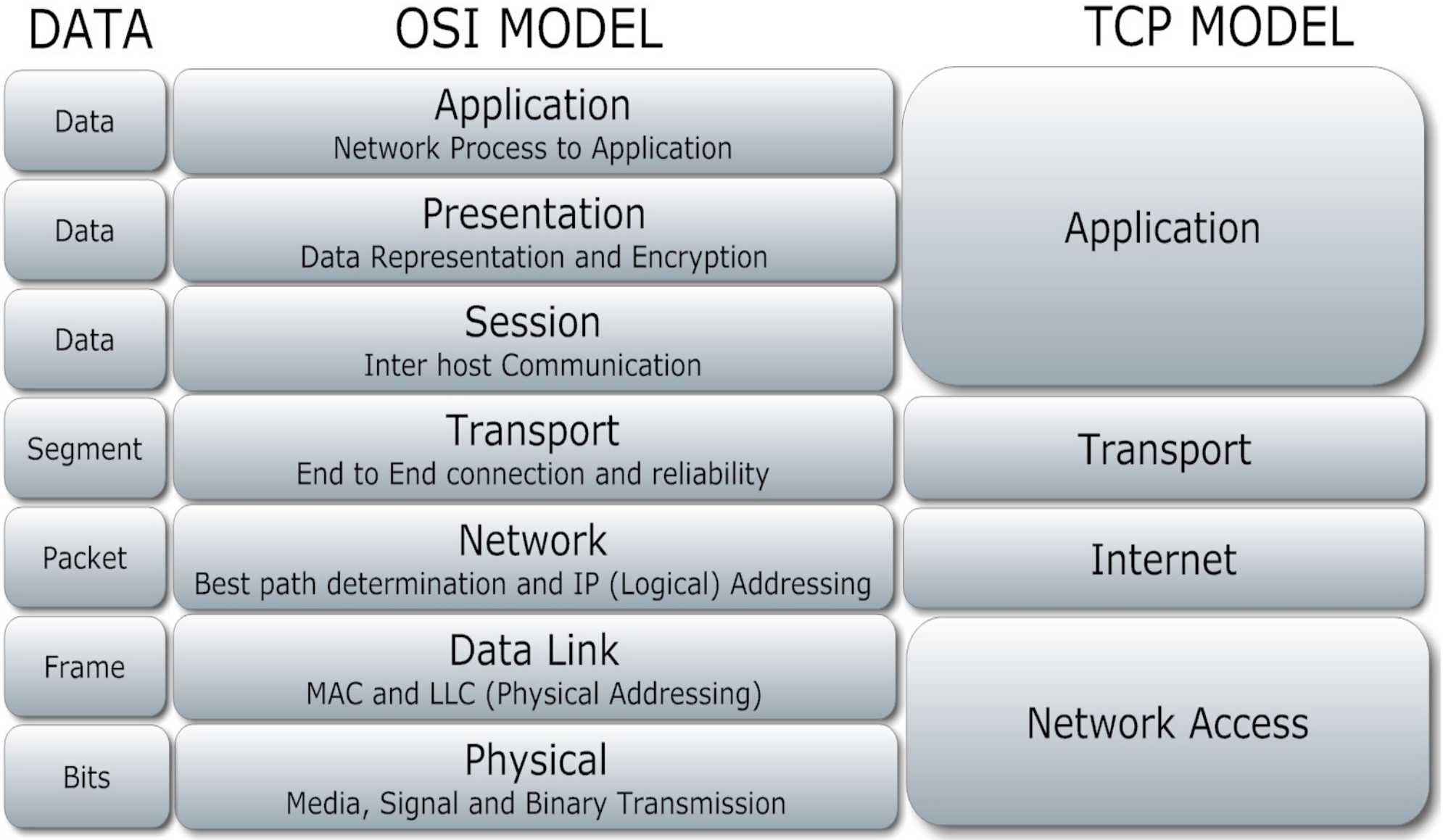
1. Modern Applications: Nearly all significant applications today are networked in some way (Client-Server, P2P etc)
2. Cloud Computing: Understanding networking is crucial for cloud-based development.
3. Internet of Things (IoT): Powers communication between smart devices.
4. Distributed Systems: Needed for systems that run across multiple machines.
5. Data Transmission: Allows real-time data sharing across different systems and locations.
6. Security Systems: NP plays a key role in developing secure communication protocols.



Source: Image Generated by chatgpt.


OSI Reference Model
7 – Application
Interface to end user. Interaction directly with software application.
Software App Layer
Directory services, email, network management, file transfer, web pages, database access.
FTP, HTTP, WWW, SMTP, TELNET, DNS, TFTP, NFS
6 – Presentation
Formats data to be "presented" between application-layer entities.
Syntax/Semantics Layer
Data translation, compression, encryption/decryption, formatting.
ASCII, JPEG, MPEG, GIF, MIDI
5 – Session
Manages connections between local and remote application.
Application Session Management
Session establishment/teardown, file transfer checkpoints, interactive login.
SQL, RPC, NFS
4 – Transport
Ensures integrity of data transmission.
Segment
End-to-End Transport Services
Data segmentation, reliability, multiplexing, connection-oriented, flow control, sequencing, error checking.
TCP, UDP, SPX, AppleTalk
3 – Network
Determines how data gets from one host to another.
Packet
Routing
Packets, subnetting, logical IP addressing, path determination, connectionless.
IP, IPX, ICMP, ARP, PING, Traceroute
2 – Data Link
Defines format of data on the network.
Frame
Switching
Frame traffic control, CRC error checking, encapsulates packets, MAC addresses.
Switches, Bridges, Frames, PPP/SLIP, Ethernet
1 – Physical
Transmits raw bit stream over physical medium.
Bits
Cabling/Network Interface
Manages physical connections, interpretation of bit stream into electrical signals
Binary transmission, bit rates, voltage levels, Hubs

OSI VS TCP/IP


DATA
OSI MODEL
TCP MODEL
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Inter host Communication
Application
Segment
Transport
End to End connection and reliability
Transport
Packet
Network
Best path determination and IP (Logical) Addressing
Internet
Frame
Data Link
MAC and LLC (Physical Addressing)
Bits
Physical
Media, Signal and Binary Transmission
Network Access

Similarities b/w TCP/IP and OSI

1. Both the reference models are based upon layered architecture.
2. The physical layer and the data link layer of the OSI model correspond to the link layer of the TCP/IP model. The network layers and the transport layers are the same in both the models. The session layer, the presentation layer and the application layer of the OSI model together form the application layer of the TCP/IP model.
3. In both the models, protocols are defined in a layer-wise manner.
4. In both models, data is divided into packets and each packet may take the individual route from the source to the destination.

Differences b/w TCP/IP and OSI

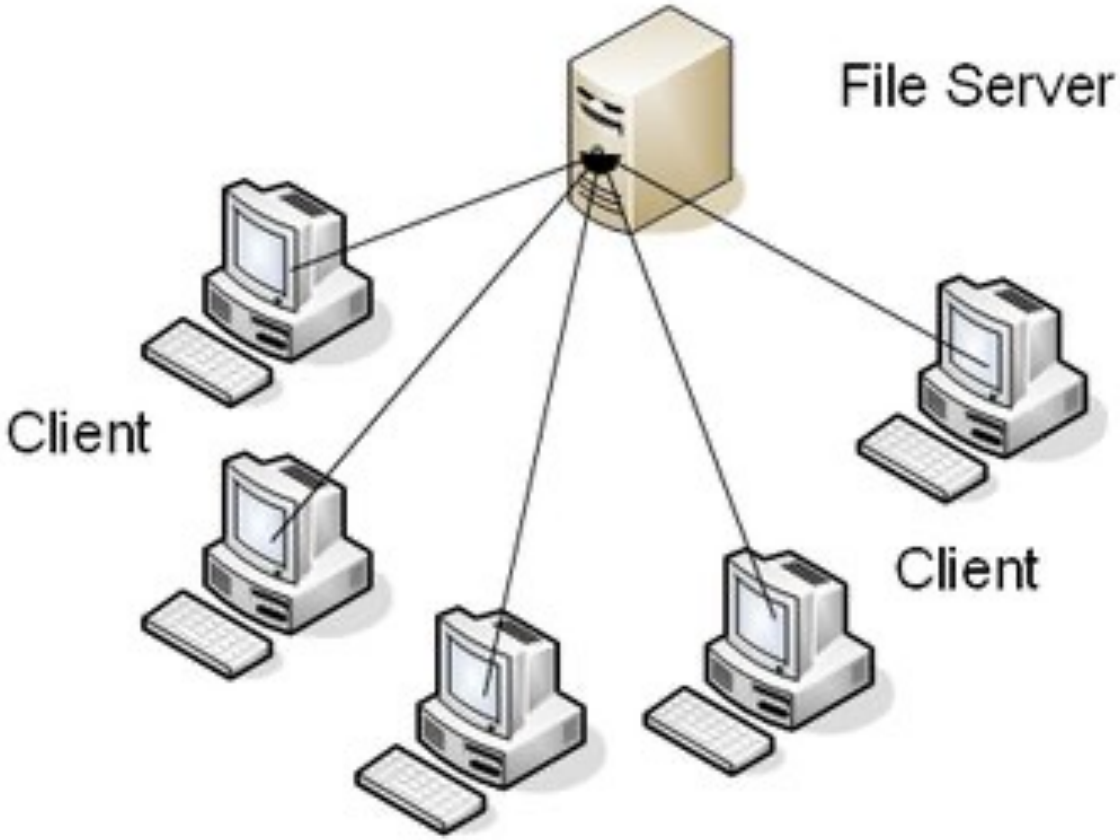
OSI

- OSI model is just a framework but not the implementation model.
- It consists of 7-layers.
- It clearly distinguishes between services, interfaces & protocols.
- The protocols are first defined & then the layers are set or fixed. Thus, the protocols in the OSI model can be easily replaced by the technology change.
- The network layer is connectionless or connection oriented.
- The transport layer is connection oriented.

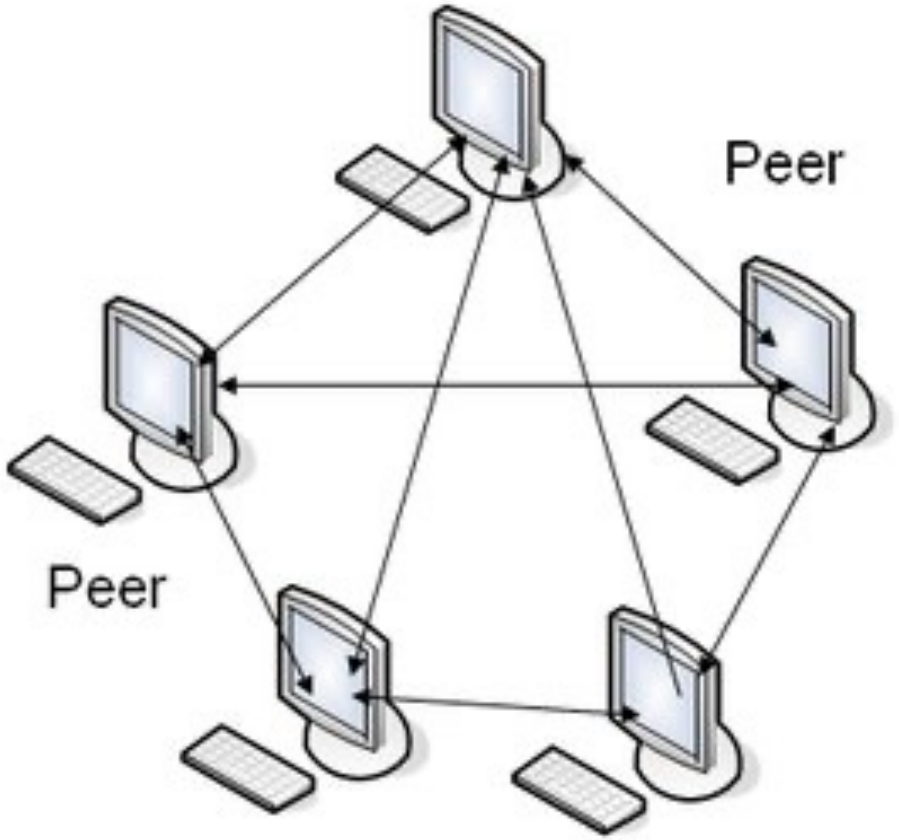
TCP/IP

- It is an implementation model.
- It has just 4-layers.
- TCP/IP model fails to distinguish between services, interfaces & protocols.
- The layers are first set & then the protocols are defined. Thus, there may difficulties to replace it by the technology change.
- The network layer is connectionless.
- The transport layer is connectionless or connection oriented.

Fundamentals of Client Server and P2P models



Client server mode



Peer to Peer mode

Client-Server Model

- A networking model where ***clients*** (e.g., web browsers, mobile apps) request services or resources (e.g., files, websites) and ***servers*** (e.g., web servers, databases) provide them.
- **Examples:** Web Browsing (Browser = client, Web Server = server), Email Services, Online Games, Banking Systems etc.

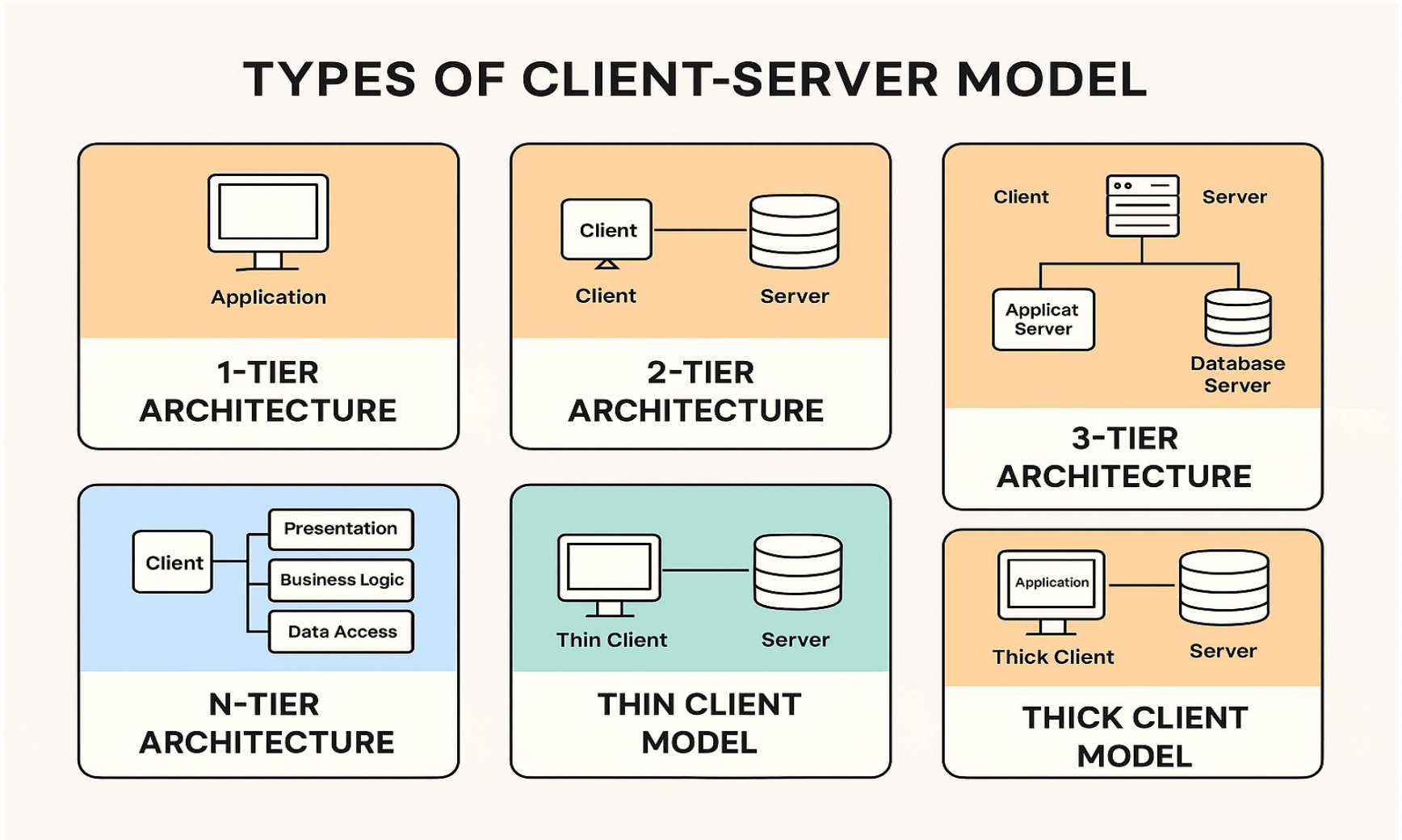
Key Characteristics:

- Efficient for large networks with many users
- Central management of data and security-Servers manage data and authentication
- Easier to back up and maintain
- Scalable infrastructure- Servers can handle multiple clients

Disadvantages:

- Single point of failure-Server failure can disrupt the whole system
- Can be expensive to set up and maintain servers

Types of Client-Server Model



Comparison of types of CS Model

Model	Layers /Tiers	Description	Processing Location	Use Case Examples	Pros	Cons
1-Tier	1	Everything in a single system (UI, logic, data)	Client device	Local apps like Notepad, MS Excel	Simple, easy to build	No data sharing or remote access
2-Tier	2	Client interacts directly with database server	Split between client/server	Small DB apps, LAN banking systems	Fast and simple communication	Limited scalability, tight coupling
3-Tier	3	Middle tier (application server) added between client and database	Shared between all tiers	Web apps (e.g., shopping sites)	Better performance & maintainability	More complex than 2-tier
N-Tier	3+	Additional layers (e.g., presentation, logic, data access, etc.)	Distributed across tiers	Enterprise & cloud-based apps	High flexibility, modular, secure	Complex design & management
Thin Client	2 (lightweight client)	Client depends heavily on server for processing	Server	Cloud desktops, Citrix, Google Docs	Low cost on client side, easy updates	Requires strong server & network
Thick Client	2 (heavy client)	Client handles major processing; server for storage	Client + server	Local software syncing with DB	Good offline capability, fast local use	Difficult updates & higher client load

Peer-to-Peer (P2P) Model

- A decentralized architecture where ***peers (nodes)*** act as both clients and servers, sharing resources directly.

Key Characteristics:

- *Decentralized*: No central server; peers communicate directly.
- *Resilience*: No single point of failure.
- *Efficiency*: Bandwidth/resources are distributed.
- *Examples*: BitTorrent (file sharing), Blockchain (Bitcoin), VoIP (Earlier version of Skype).

Working Mechanisms:

1. Peers discover each other (via trackers or Distributed Hash Tables (DHT)).
2. Data is split into chunks and shared directly.
3. Each peer contributes upload bandwidth.

Disadvantages:

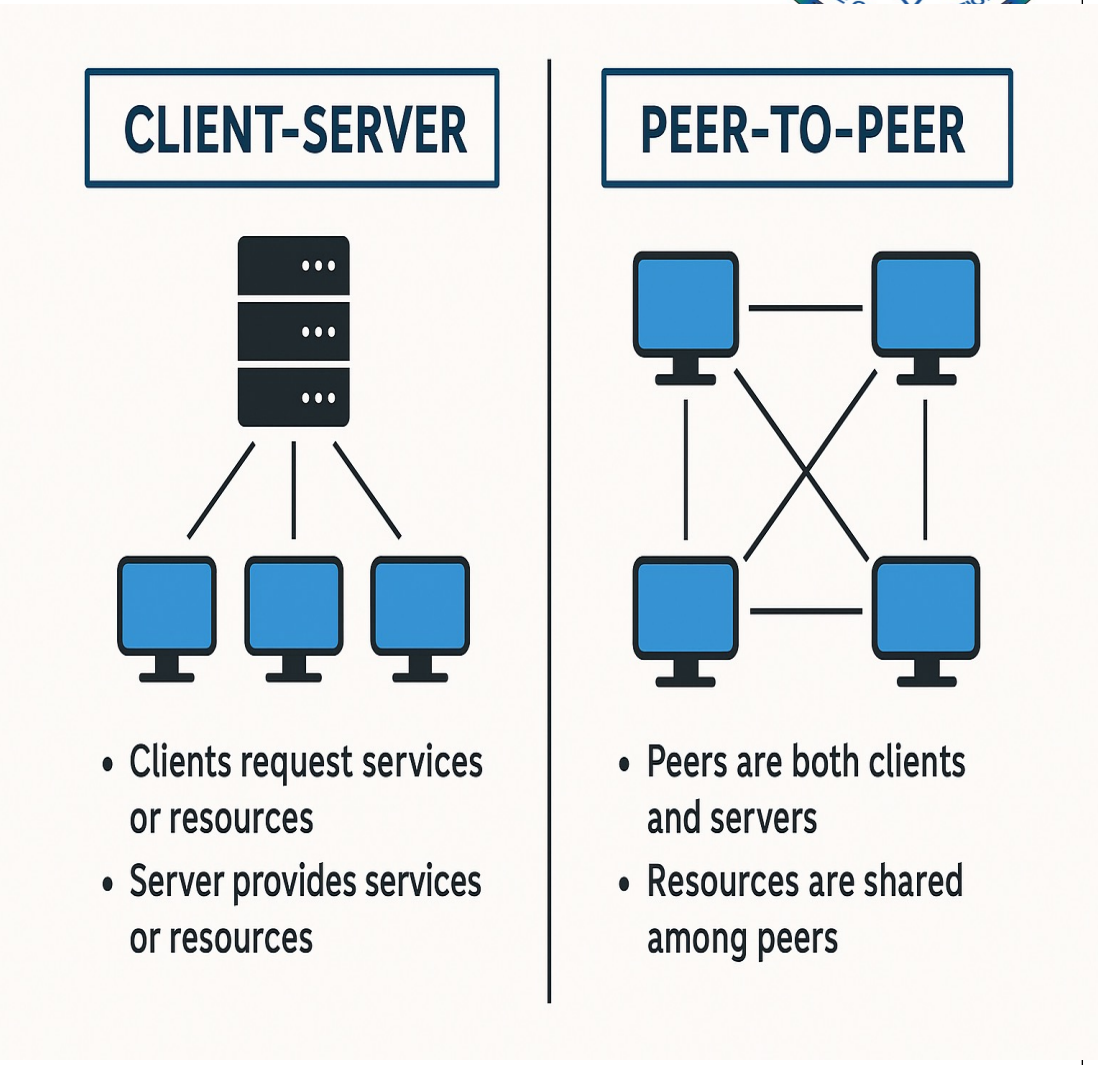
- Harder to manage and secure
- Data may not be reliably available
- Performance depends on peer reliability

Types of P2P Models

Type	Description	Examples	Pros	Cons
1. Pure P2P	Every peer has equal capabilities and responsibilities. No central server.	BitTorrent (initial design), Gnutella	Full decentralization, high fault tolerance	Poor search efficiency, redundant traffic
2. Hybrid P2P	A central server assists with functions like indexing or peer discovery.	Napster, Skype (earlier versions)	Faster searching, better resource management	Central server can be a single point of failure
3. Structured P2P	Uses a fixed protocol (e.g., DHT) to place and retrieve data efficiently.	Chord, Kademlia, CAN	Efficient routing and searching (logarithmic)	More complex protocols, rigid structure
4. Unstructured P2P	Peers connect randomly. Searching is done via flooding or random walk.	Gnutella, Freenet	Easy to join, flexible topology	High bandwidth consumption for searching
5. Super-Peer Model	Some powerful peers (super-peers) act like mini-servers for weaker peers.	Modern Skype, Kazaa	Efficient resource usage, improved scalability	Super-peer failure can affect many nodes
6. Blockchain-based P2P	Uses distributed ledger technology for secure and immutable data sharing.	Bitcoin, Ethereum	High security, transparency, no central control	High energy use (for some), scalability issues

CS Model vs P2P Model

Feature	Client-Server	Peer-to-Peer
Control	Centralized	Decentralized
Scalability	Scales with server resources	Scales with number of peers
Examples	Web services, email, banking systems	File sharing, blockchain, Skype
Reliability	Depends on server	Depends on peers
Cost	High (infrastructure/server)	Low (just peers)
Use Cases	Web, Email, Cloud	File sharing, Blockchain



Basic Network Protocols

- Characteristics and use cases of TCP, IP, UDP, SCTP

Few Protocols and Description

Protocol	Description
IPv4	Internet Protocol version 4. IPv4 uses 32-bit addresses and provides packet delivery service for TCP, UDP, SCTP, ICMP, and IGMP.
IPv6	Internet Protocol version 6. IPv6 uses 128-bit addresses.
TCP	Transmission Control Protocol. TCP is a **connection-oriented** protocol that provides a **reliable**, **full-duplex** byte stream to its users
UDP	User Datagram Protocol. UDP is a **connectionless** protocol, and UDP sockets are an example of datagram sockets.
SCTP	Stream Control Transmission Protocol. SCTP is a **connection-oriented** protocol that provides a reliable full-duplex association
ICMP	Internet Control Message Protocol. ICMP handles error and control information between routers and hosts.
IGMP	Internet Group Management Protocol. IGMP is used with multicasting.
ARP	Address Resolution Protocol. ARP maps an IPv4 address into a hardware address (such as an Ethernet address). ARP is normally used on broadcast networks such as Ethernet, token ring, and FDDI, and is not needed on point-to-point networks.
RARP	Reverse Address Resolution Protocol. RARP maps a hardware address into an IPv4 address.

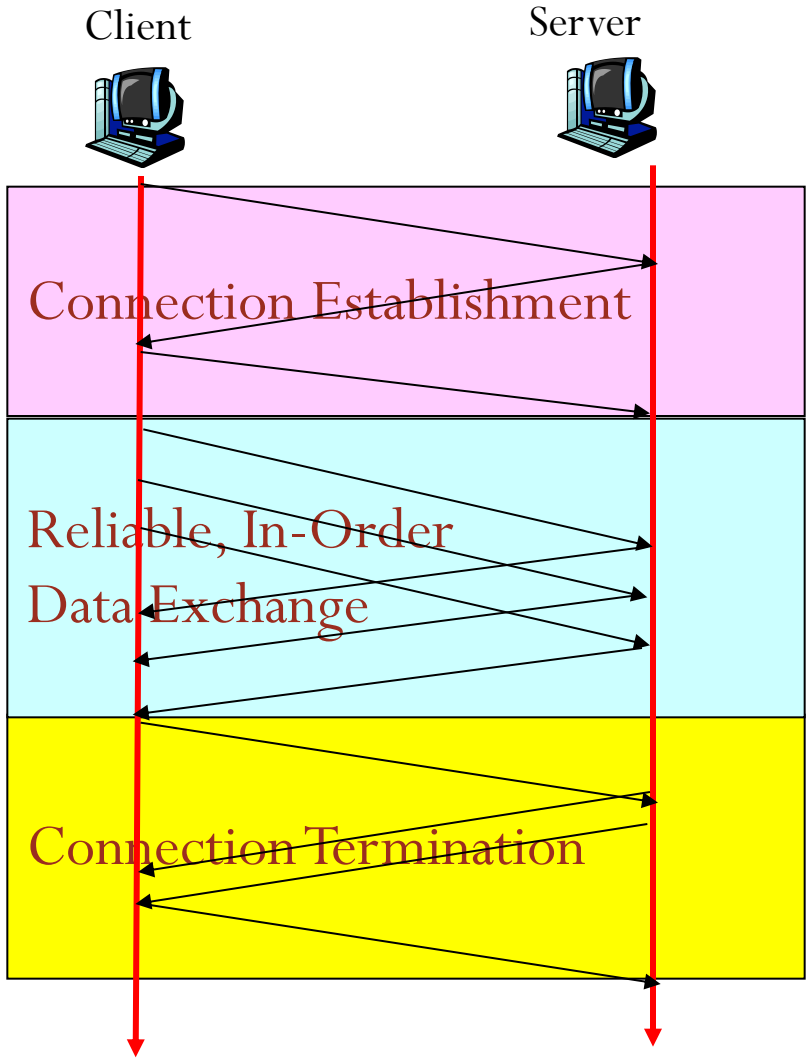
Transmission Control Protocol (TCP)

1. ***Connection***: TCP provides connections between clients and servers. A TCP client establishes a connection with a server, exchanges data across the connection, and then terminates the connection.
2. ***Reliability***: TCP requires acknowledgment when sending data. If an acknowledgment is not received, TCP automatically retransmits the data and waits a longer amount of time.
3. ***Round-trip time*** (RTT): TCP estimates RTT between a client and server dynamically so that it knows how long to wait for an acknowledgment.
4. ***Sequencing***: TCP associates a sequence number with every byte (**segment**, unit of data that TCP passes to IP.) it sends. TCP reorders out-of-order segments and discards duplicate segments.
5. ***Flow control*****:** is a set of procedures to restrict the amount of data that sender can send. Stop and Wait Protocol is a **flow control** protocol where sender sends one data packet to the receiver and then stops and waits for its acknowledgement from the receiver.
6. ***Full-duplex***: an application can send and receive data in both directions on a given connection at any time.

TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point (unicast):
  - one sender, one receiver
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
  - State resides only at the END systems – Not a virtual circuit!
- full duplex data:
  - bi-directional data flow in same connection (A->B & B->A in the same connection)
  - MSS: maximum segment size
- reliable, in-order *byte steam:*
  - no "message boundaries"
- send & receive buffers
  - buffer incoming & outgoing data
- flow controlled:
  - sender will not overwhelm receiver
- congestion controlled:
  - sender will not overwhelm network

Typical TCP Transaction



- A TCP Transaction consists of 3 Phases
  1. **Connection Establishment**
     - Handshaking between client and server
  2. **Reliable, In-Order Data Exchange**
     - Recover any lost data through retransmissions and ACKs
  3. **Connection Termination**
     - Closing the connection

TCP Connection Establishment

The following scenario occurs when a TCP connection is established.

1. The server must be prepared to accept an incoming connection. This is normally done by calling **socket**, **bind**, and **listen** and is called a passive open.
2. The client issues an active open by calling **connect**. This causes the client TCP to send a "**synchronize**" (**SYN**) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the **SYN**; it just contains an IP header, a TCP header, and possible TCP options.
3. The server must acknowledge (**ACK**) the client's **SYN** and the server must also send its own **SYN** containing the initial sequence number for the data that the server will send on the connection. The server sends its **SYN** and the **ACK** of the client's **SYN** is a single segment.
4. The client must acknowledge the server's **SYN**.

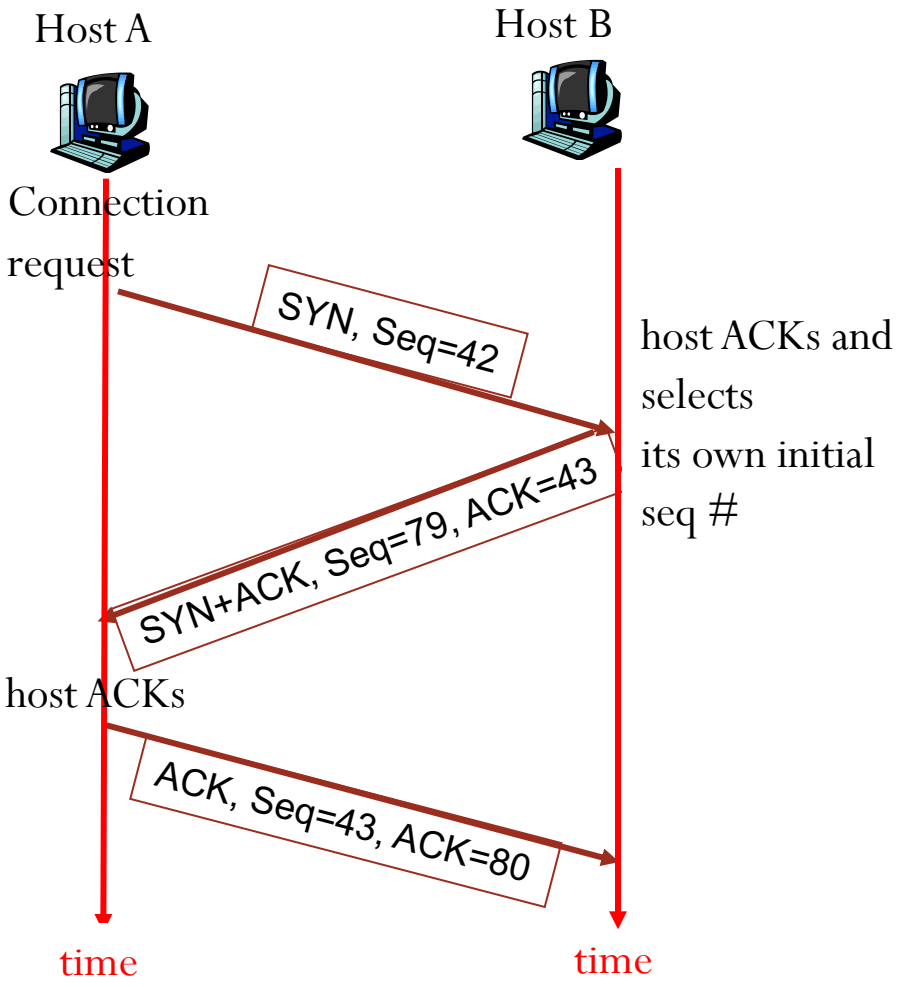
Connection Establishment (cont)

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies a random initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data



Three-way handshake

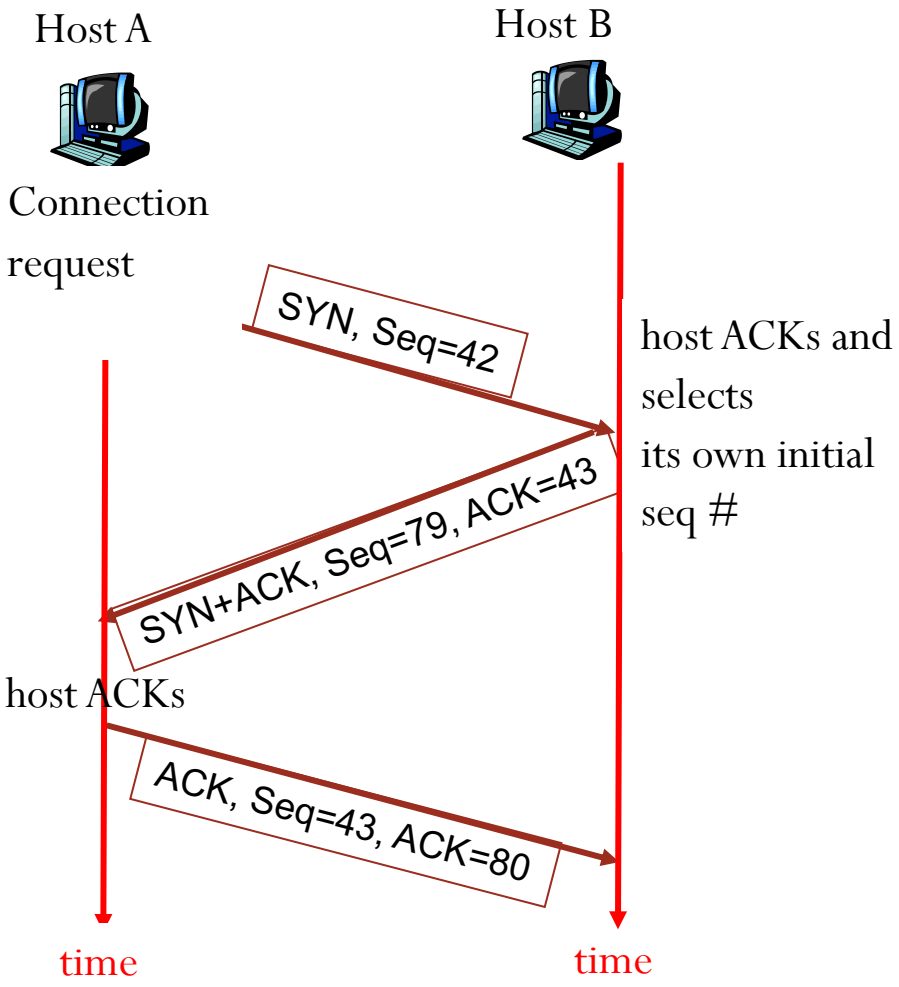
Connection Establishment (cont)

Seq. #'s:

- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK



Three-way handshake

TCP Starting Sequence Number Selection

- Why a random starting sequence #? Why not simply choose 0?
  - To protect against two incarnations of the same connection reusing the same sequence numbers too soon
  - That is, while there is still a chance that a segment from an earlier incarnation of a connection will interfere with a later incarnation of the connection
- How?
  - Client machine seq #0, initiates connection to server with seq #0.
  - Client sends one byte and client machine crashes
  - Client reboots and initiates connection again
  - Server thinks new incarnation is the same as old connection

TCP Connection Termination

1. One application calls **close** first, and we say that this end performs the **active close**. This end's TCP sends a **FIN** segment, which means it is **finished sending data**.
2. The other end that receives the **FIN** performs the **passive close**. The received FIN is acknowledged by **TCP**. The receipt of the **FIN** is also passed to the application as an end-of-file, since the receipt of the **FIN** means the application will not receive any additional data on the connection.
3. Sometime later, the application that received the end-of-file will close its socket. This causes its **TCP** to send a **FIN**.
4. The TCP on the system that receives this final **FIN** (the end that did the active close) acknowledges the **FIN**.

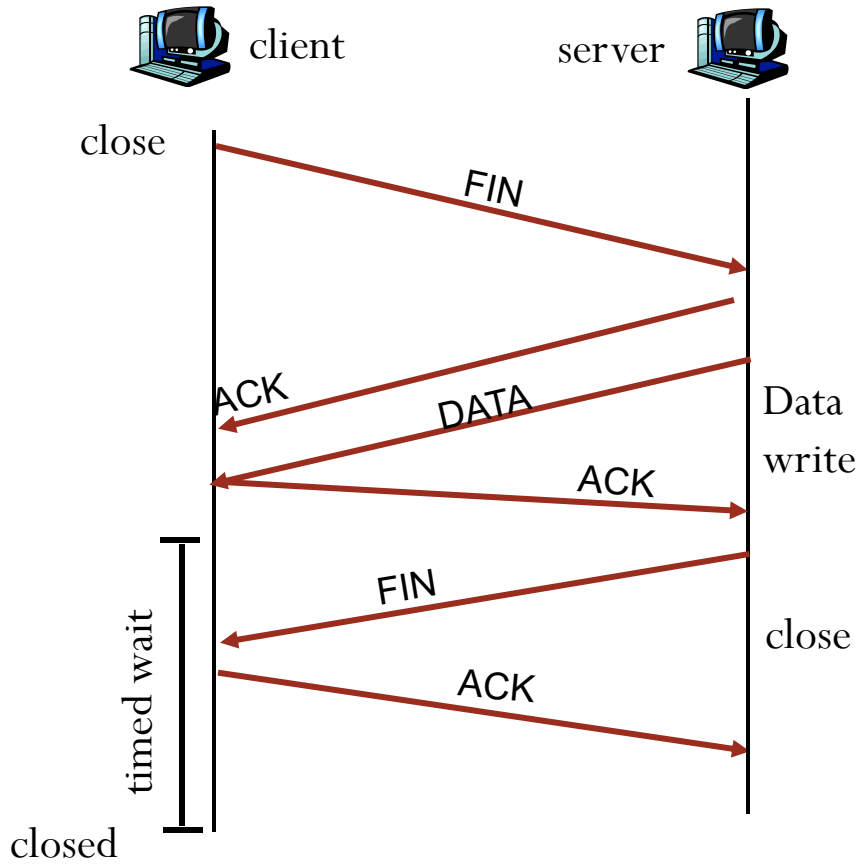
TCP Connection Termination

Closing a connection:

client closes socket:
clientSocket.close();

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Server might send some buffered but not sent data before closing the connection. Server then sends FIN and moves to Closing state.



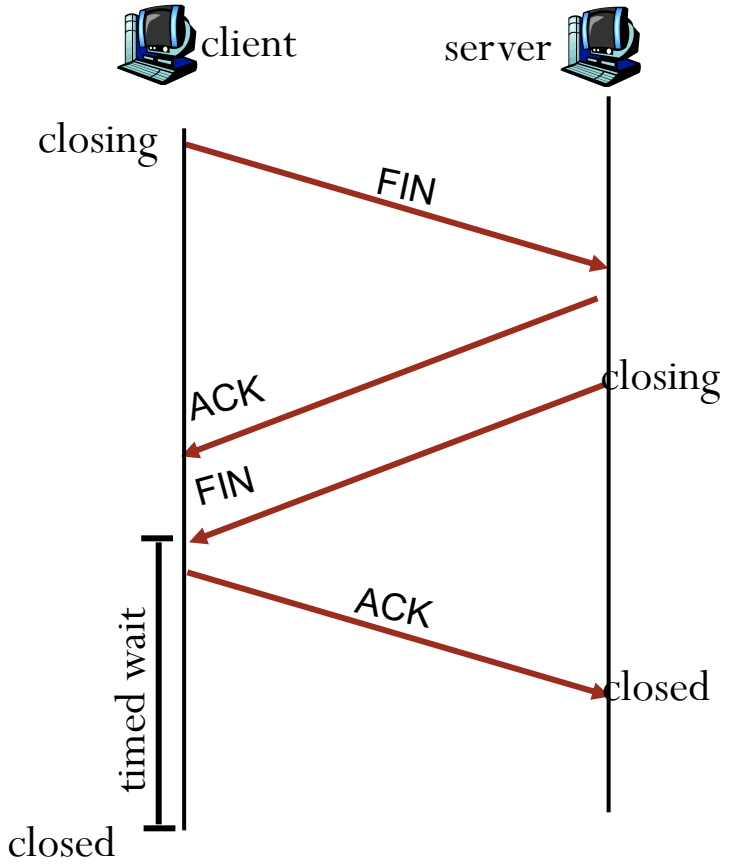
TCP Connection Termination

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

- Why wait before closing the connection?
  - If the connection were allowed to move to CLOSED state, then another pair of application processes might come along and open the same connection (use the same port #s) and a delayed FIN from an earlier incarnation would terminate the connection.



User Datagram Protocol (UDP)

- **UDP** is a simple **transport-layer protocol**. (RFC 768)
- The application writes a message to a **UDP socket**, which is then encapsulated in a **UDP datagram**, which is then further encapsulated as an IP datagram, which is then sent to its destination.
- There is no guarantee that a **UDP datagram** will ever reach its final destination, that order will be preserved across the network, or that datagrams arrive only once.
- With network programming using UDP is **its lack of reliability**. If a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not delivered to the UDP socket and is **not automatically retransmitted**.
- Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data.
- UDP provides a **connectionless service**, as there need not be any long-term relationship between a UDP client and server.

Stream Control Transmission Protocol (SCTP)

1. Like **TCP**, SCTP provides **reliability**, **sequencing**, **flow control**, and **full-duplex data transfer**.
2. Unlike TCP, SCTP provides:
   - I. **Association** instead of "connection": An association refers to a communication between two systems, which may involve more than two addresses due to **multihoming**.
   - II. **Message-oriented**: provides sequenced delivery of individual records. Like UDP, the length of a record written by the sender is passed to the receiving application.
   - III. **Multihoming**: allows a single SCTP endpoint to support multiple IP addresses. This feature can provide increased robustness against network failure.

Comparison Table: TCP, IP, UDP, SCTP

Feature / Protocol	TCP	IP	UDP	SCTP
Full Form	Transmission Control Protocol	Internet Protocol	User Datagram Protocol	Stream Control Transmission Protocol
Layer	Transport Layer	Network Layer	Transport Layer	Transport Layer
Connection Type	Connection-oriented	Connectionless	Connectionless	Connection-oriented
Reliability	Reliable – with acknowledgments & retransmissions	Unreliable – no guarantees	Unreliable – no guarantees	Reliable – with multi-streaming and retransmissions
Ordering	Maintains order of packets	No ordering	No ordering	Maintains order per stream
Flow Control	Yes – using windowing	No	No	Yes – built-in flow & congestion control
Error Checking	Yes – checksum	Basic checksum only	Optional checksum	Strong – built-in error detection
Speed	Slower – due to overhead for reliability	Depends on routing	Fast – minimal overhead	Moderate – balance between reliability and performance
Use Cases	Web, email, file transfer, remote login	Packet delivery & routing across networks	Streaming, VoIP, DNS, online games	Telecom signaling, VoIP, high-availability systems

Protocol Comparison

Services/Features	SCTP	TCP	UDP
Connection-oriented	yes	yes	no
Full duplex	yes	yes	yes
Reliable data transfer	yes	yes	no
Partial-reliable data transfer	optional	no	no
Ordered data delivery	yes	yes	no
Unordered data delivery	yes	no	yes
Flow control	yes	yes	no
Congestion control	yes	yes	no
ECN capable	yes	yes	no
Selective ACKs	yes	optional	no
Preservation of message boundaries	yes	no	yes
Path MTU discovery	yes	yes	no
Application PDU fragmentation	yes	yes	no
Application PDU bundling	yes	yes	no
Multistreaming	yes	no	no
Multihoming	yes	no	no
Protection against SYN flooding attacks	yes	no	n/a
Allows half-closed connections	no	yes	n/a
Reachability check	yes	yes	no
Pseudo-header for checksum	no (uses vtags)	yes	yes
Time wait state	for vtags	for 4-tuple	n/a

Usage Scenarios & Applications of TCP, UDP, IP, and SCTP

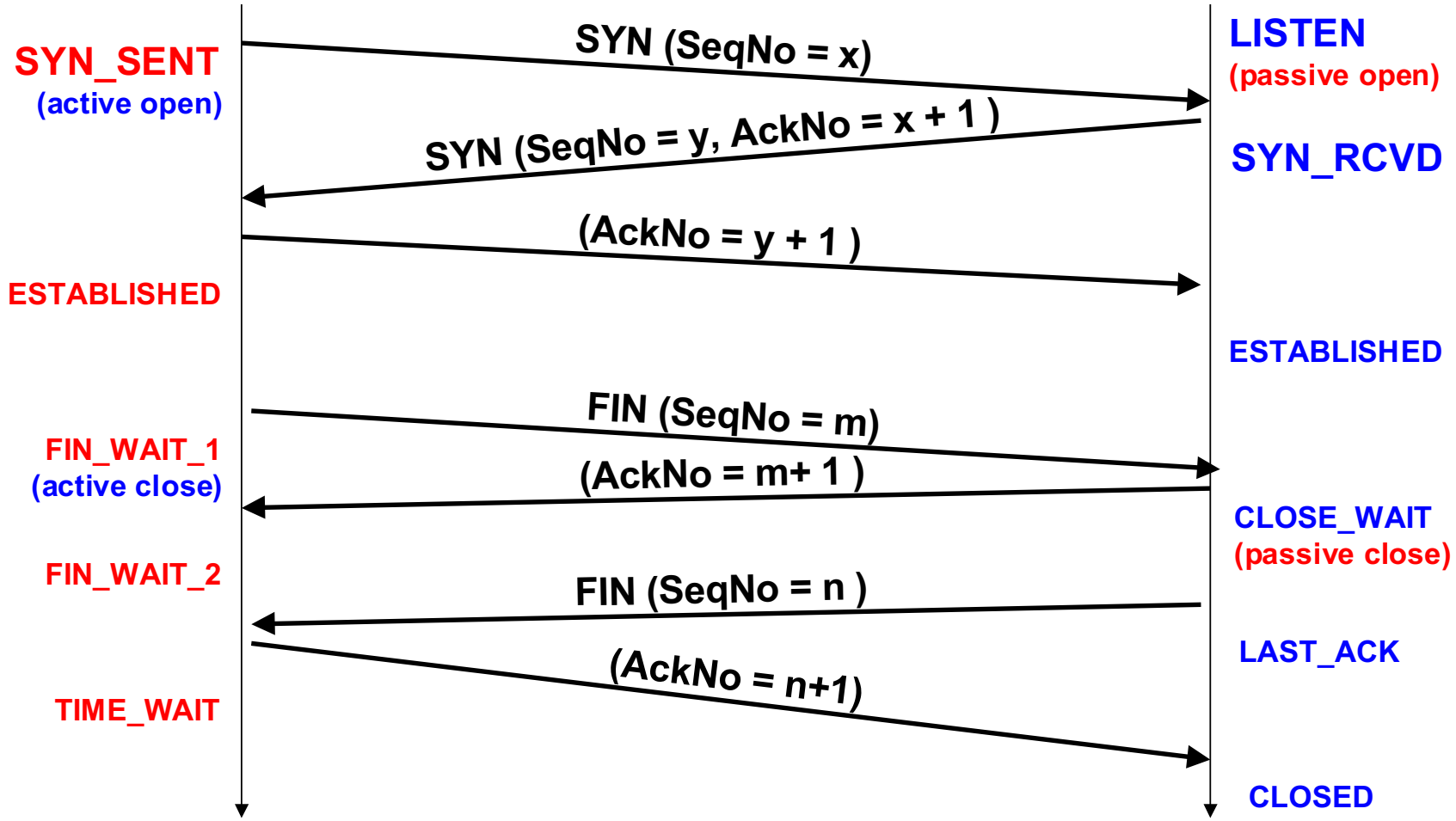
Protocol	Type	Key Characteristics	Common Usage Scenarios	Real-world Applications
TCP	Connection-oriented	Reliable, ordered, error-checked, slower	Where data integrity and delivery order are critical	Web browsing (HTTP/HTTPS), Email (SMTP/IMAP), File transfer (FTP/SFTP), Remote login (SSH/Telnet), Cloud sync
UDP	Connectionless	Unreliable, unordered, fast, low overhead	Where speed matters more than accuracy	Live streaming, Online gaming, VoIP (Skype/Zoom), DNS queries, IoT sensor data
IP	Connectionless (Network Layer)	Provides addressing and routing, unreliable	Delivering data across networks	Internet data routing, VPN/IPSec, Mobile IP, CCTV/IP cameras
SCTP	Connection-oriented	Reliable, supports multi-streaming and multi-homing	High-reliability, multi-stream communications	Telecom signaling (SS7), VoIP (next-gen), Financial trading, Redundant networks, Defense systems

TCP/IP State Transition Diagram

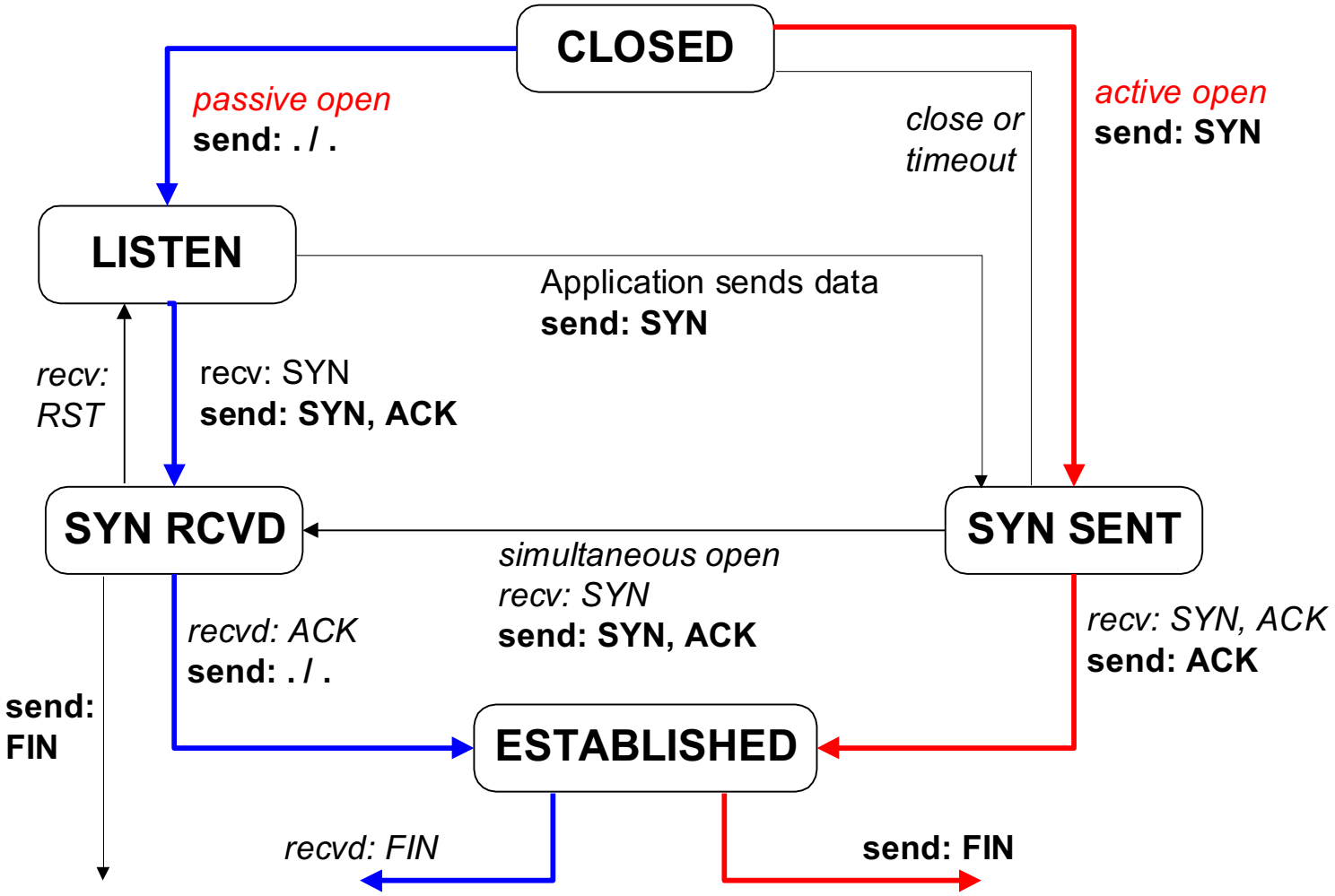
TCP States

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for Ack
SYN SENT	The client has started to open a connection
ESTABLISHED	Normal data transfer state
FIN WAIT 1	Client has said it is finished
FIN WAIT 2	Server has agreed to release
TIMED WAIT	Wait for pending packets (“2MSL wait state”)
CLOSING	Both Sides have tried to close simultanesously
CLOSE WAIT	Server has initiated a release
LAST ACK	Wait for pending packets

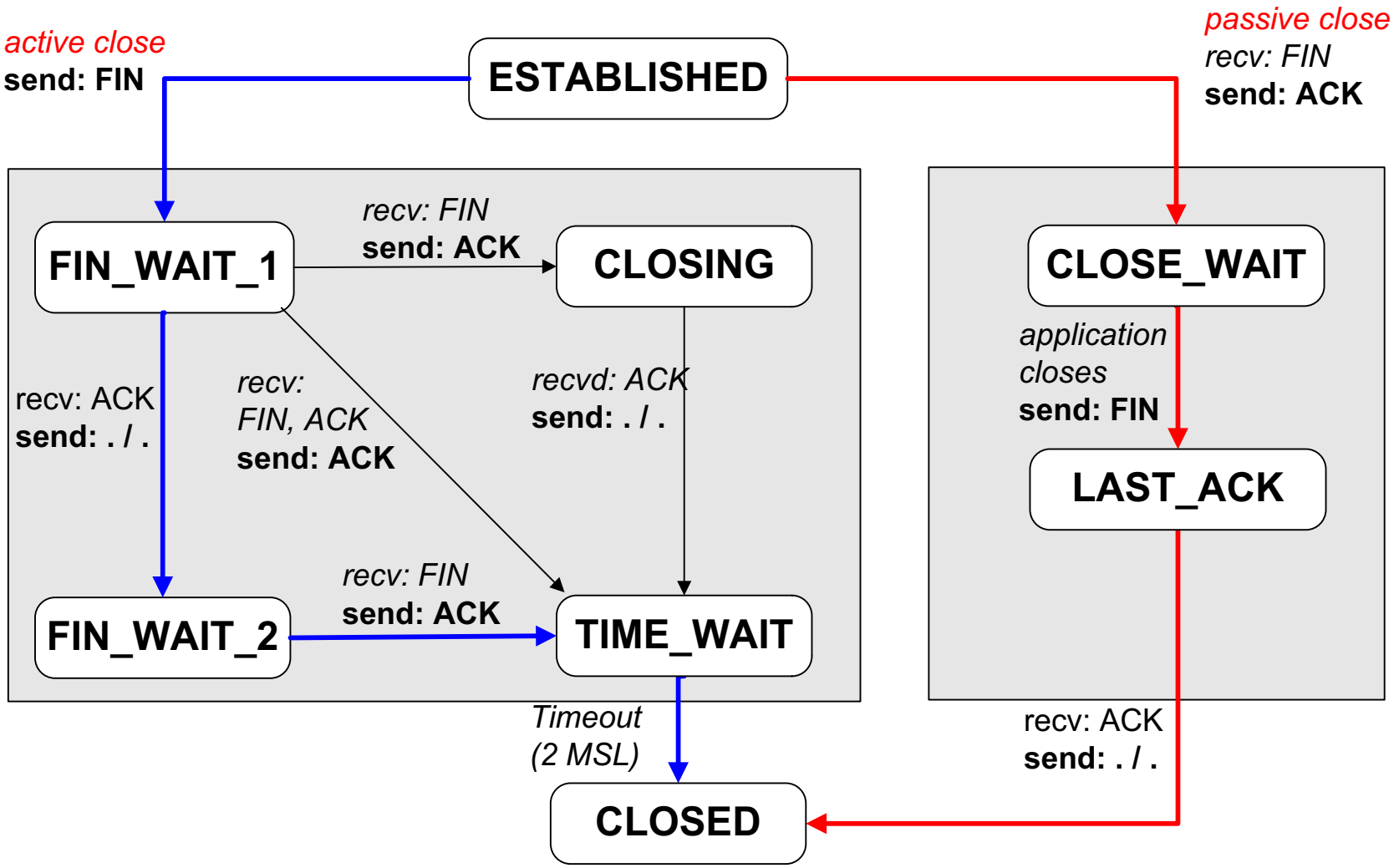
TCP States in "Normal" Connection Lifetime



TCP State Transition Diagram Opening A Connection



TCP State Transition Diagram Closing A Connection



2MSL Wait State

2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection **must stay in in the TIME_WAIT state for twice the maximum segment lifetime.**

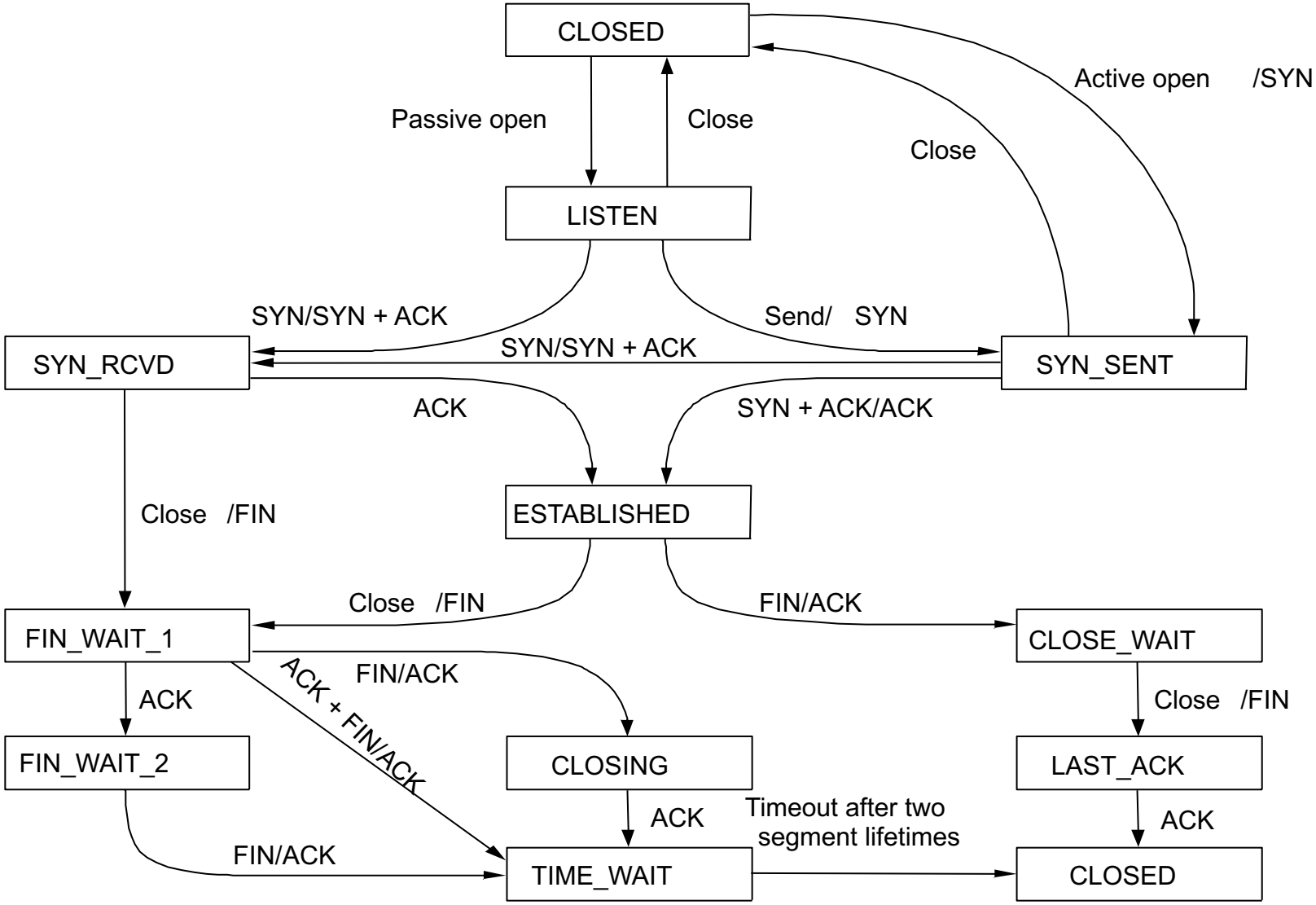
2MSL= 2 * Maximum Segment Lifetime

- Why?
  TCP is given a chance to resent the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)
- The MSL is set to 2 minutes or 1 minute or 30 seconds.

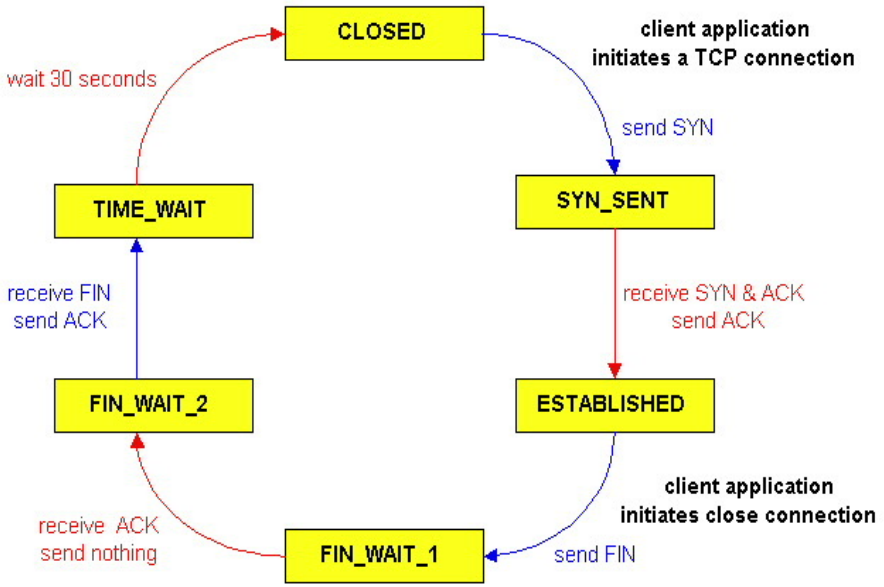
Resetting Connections

- Resetting connections is done by setting the RST flag
- **When is the RST flag set?**
  - Connection request arrives and no server process is waiting on the destination port
  - Abort (Terminate) a connection
    Causes the receiver to throw away buffered data. Receiver does not acknowledge the RST segment

TCP State-Transition Diagram

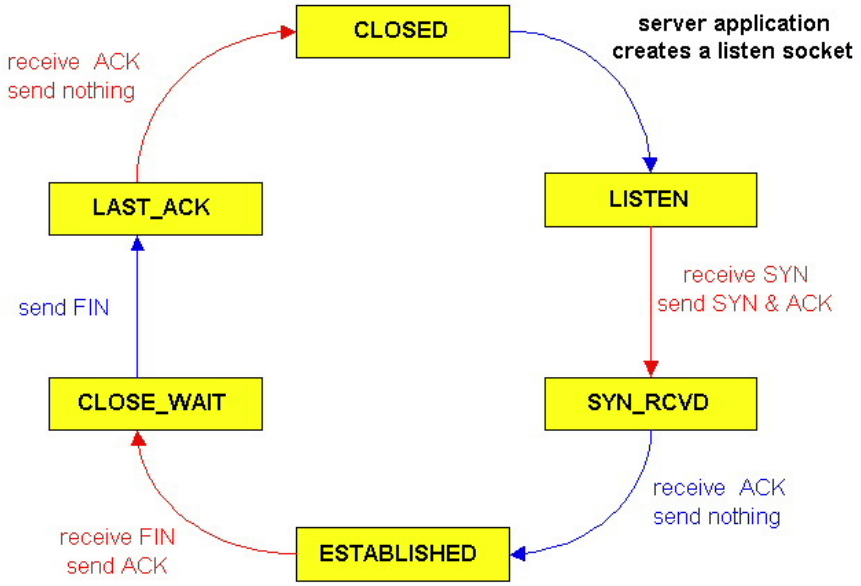


Typical TCP Client/Server Transitions



TCP client lifecycle

TCP server lifecycle



Introduction to Sockets

- Concept of sockets in network communication
- Types of sockets: Stream (TCP) vs Datagram (UDP)

Introduction to Sockets

- A **socket** is an *endpoint for communication* between two machines over a network. It provides an interface for applications to send and receive data using standard protocols like *TCP/IP* or *UDP*.

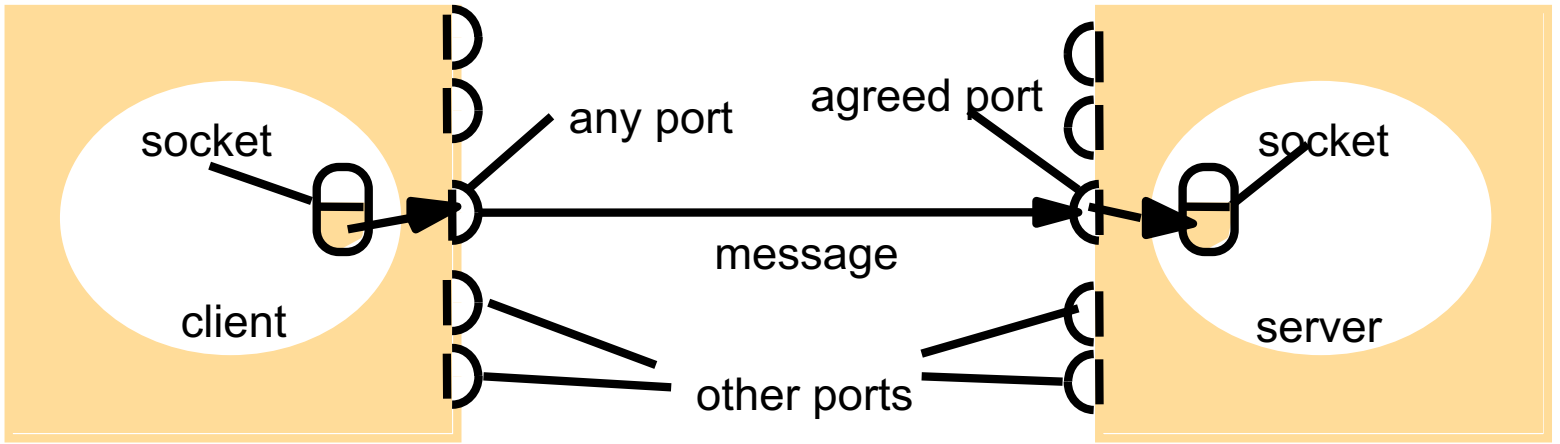
Key Properties of Sockets:

- IP Address + Port Number: Unique identifier for communication.
  - Example: 192.168.1.10:8080 (IP + Port)
- Bi-directional: Can send and receive data.
- OS-Managed: Created and controlled via system calls(*socket(),bind(),connect(),.*).
- Supports Multiple Protocols: TCP (reliable), UDP (fast), etc.

Socket Workflow (Simplified):

1. Server creates a socket, binds it to an IP:Port, and listens (*listen()*).
2. Client creates a socket and connects (*connect()*) to the server.
3. Data exchange happens via *send()/recv().*
4. Connection closes (*close()*).

Sockets and ports



- Messages sent to a particular **Internet address** and **port number** can be received only by a process whose **socket** is associated with that **Internet address** and **port number**.
- Processes may use the same **socket** for **sending** and **receiving** messages.
- Any process may make use of **multiple ports** to receive messages, BUT a process cannot share ports with other processes on the same computer.
- Each socket is associated with a particular protocol, either UDP or TCP.

Types of Sockets

Sockets are categorized based on the *transport layer protocol* they use.

1. Stream Sockets (TCP)

•**Connection-oriented**: Requires connect() before communication.
•**Reliable**: Guarantees delivery (ACKs, retransmissions).
•**In-order delivery**: Data arrives in the same sequence sent.
•**Flow control**: Adjusts speed to avoid congestion.

Use Cases:

✓ Web Browsing (HTTP/HTTPS)
✓ File Transfers (FTP)
✓ Email (SMTP)

2. Datagram Sockets (UDP)

•**Connectionless**: No connect() needed; just sendto() /recvfrom().
•**Unreliable**: No retransmissions (packets may be lost).
•**No ordering**: Data may arrive out of sequence.
•**Low overhead**: Faster than TCP (8-byte header vs TCP's 20-60 bytes).

Use Cases:

➢ Video Streaming (e.g., Zoom, Twitch)
➢ Online Gaming (real-time updates)
➢ DNS Queries

When to Use Which?

•**Use TCP Sockets** when:
- Data integrity is critical (e.g., file downloads).
- You need ordered delivery (e.g., web pages).

•**Use UDP Sockets** when:
- Speed > reliability (e.g., live video).
- You handle loss/ordering at the app layer (e.g., VoIP).

End of Chapter 1