



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)



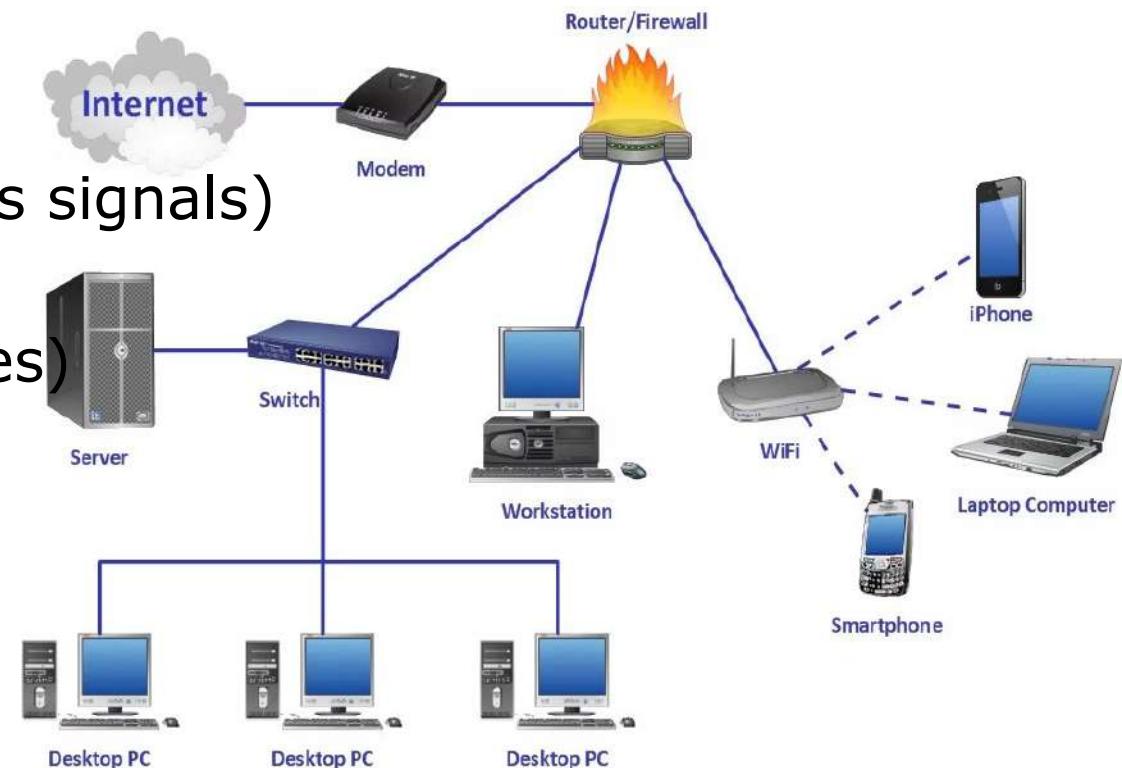
Outline

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 communicate 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.



INTRODUCTION TO NETWORKING AND NETWORK PROGRAMMING

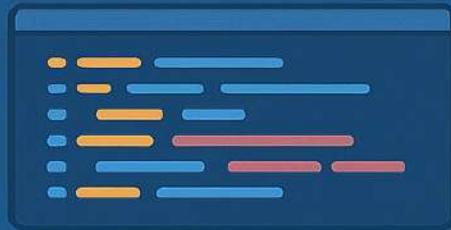


NETWORKING

Interconnection of computers and devices

IMPORTANCE OF NETWORKING

- Communication
- Resource Sharing
- Centralized Data Management
- Scalability
- Cloud Computing



NETWORK PROGRAMMING

Developing software for network communication

IMPORTANCE OF NETWORK PROGRAMMING

- Internet-based Applications
- Automation and Remote Control
- Client-Server Architecture
- Data Transmission
- Security Systems

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.		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.	Segment	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.	Packet	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.	Frame	Cabling/Network Interface Manages physical connections, interpretation of bit stream into electrical signals	Binary transmission, bit rates, voltage levels, Hubs
Bits			

OSI VS TCP/IP



DATA

Data

OSI MODEL

Application

Network Process to Application

Data

Presentation

Data Representation and Encryption

Data

Session

Inter host Communication

Segment

Transport

End to End connection and reliability

Packet

Network

Best path determination and IP (Logical) Addressing

Frame

Data Link

MAC and LLC (Physical Addressing)

Bits

Physical

Media, Signal and Binary Transmission

TCP MODEL

Application

Transport

Internet

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

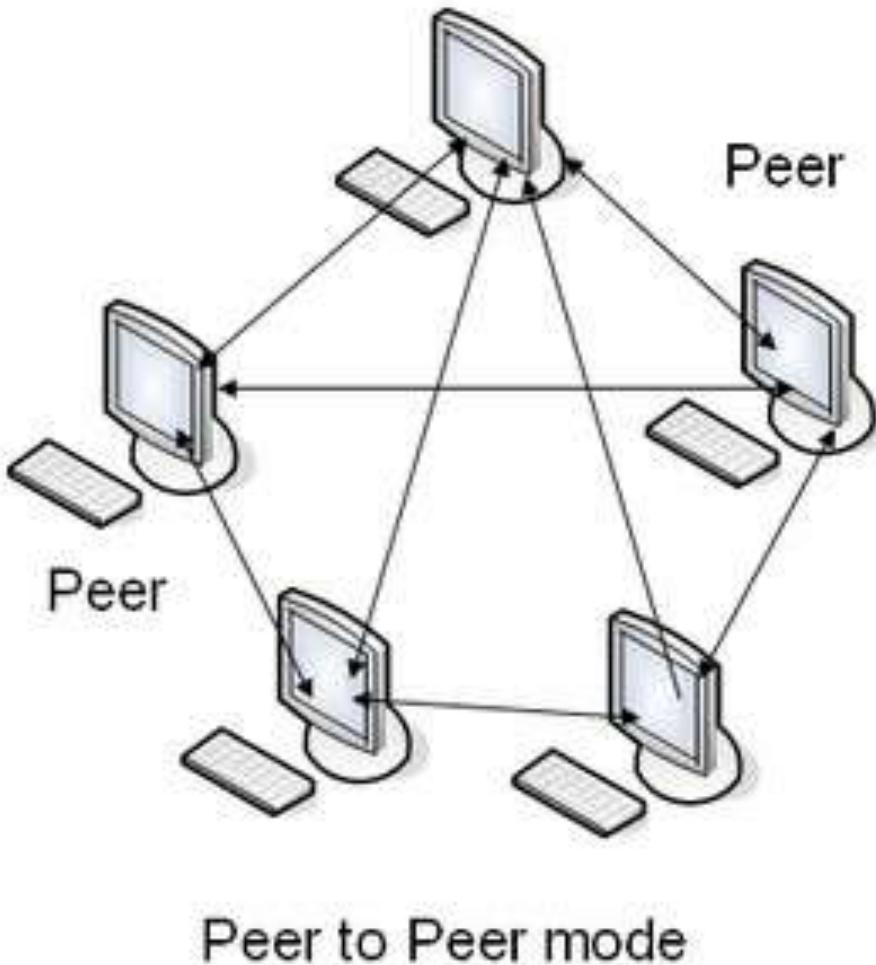
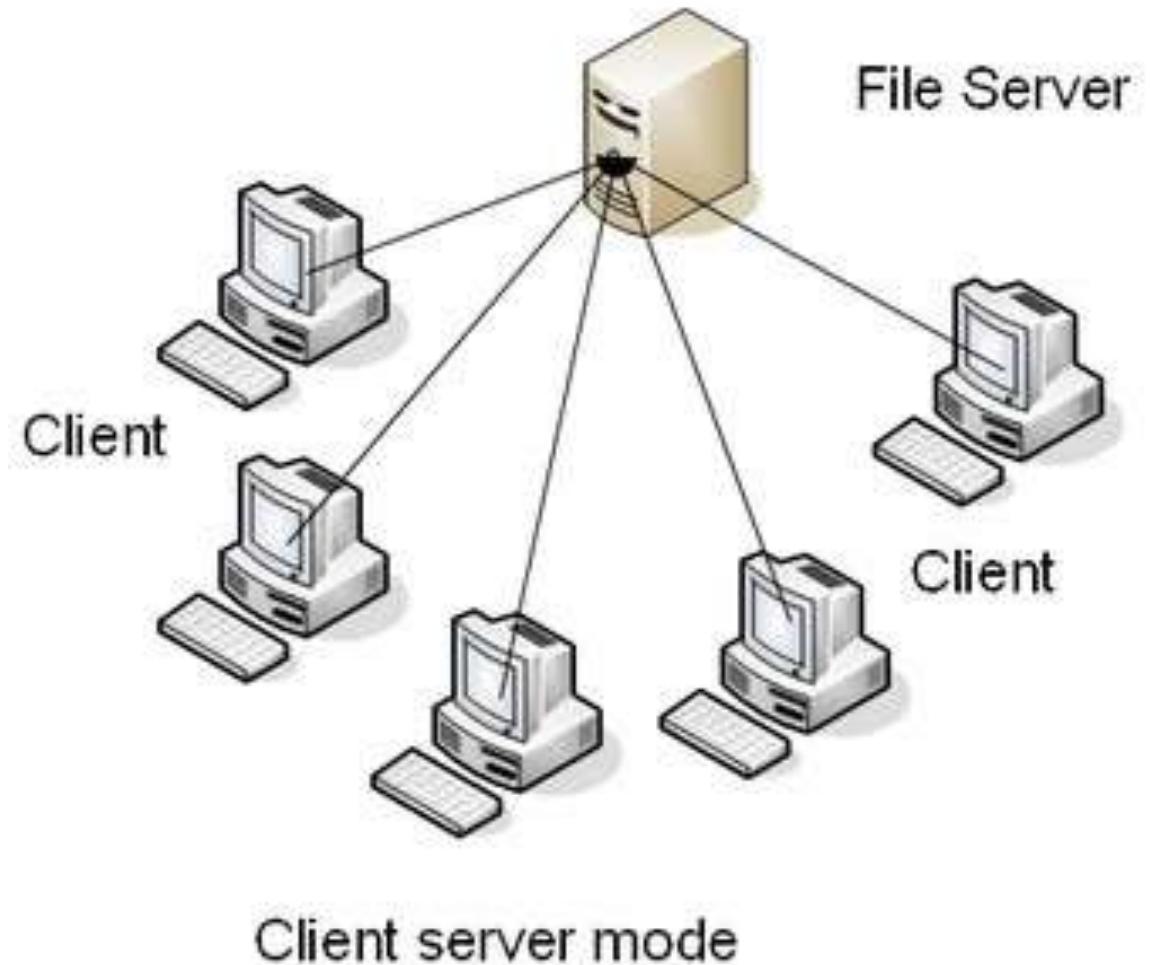
OSI

- 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





- 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:

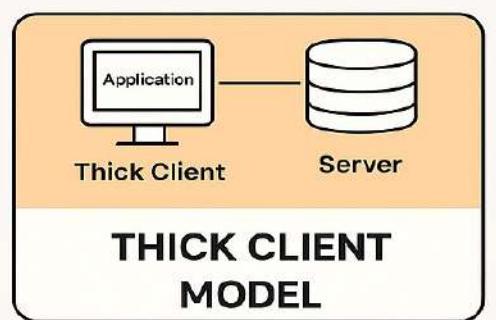
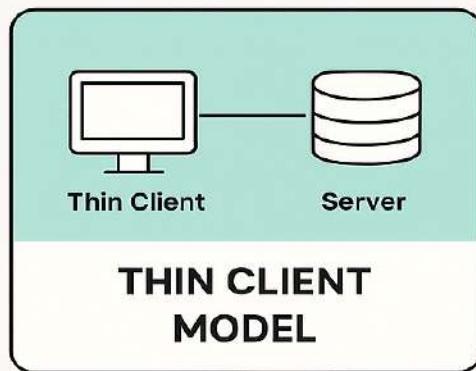
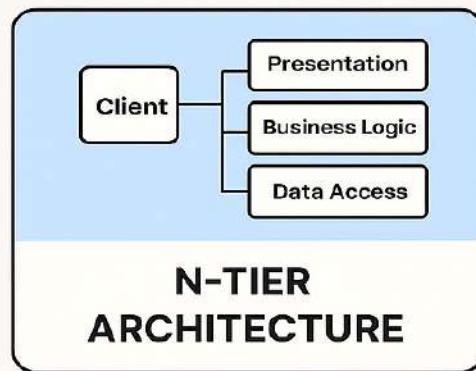
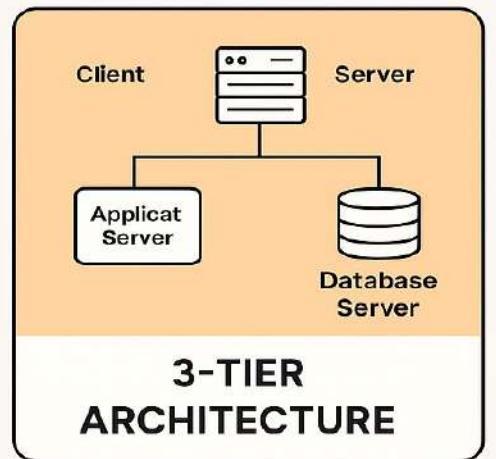
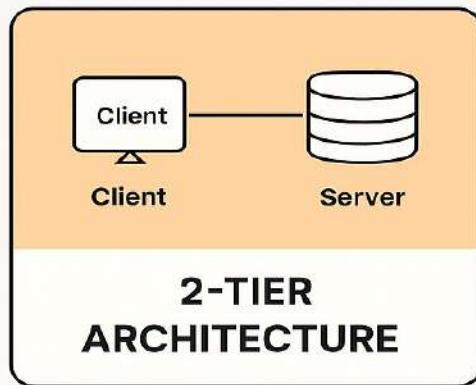
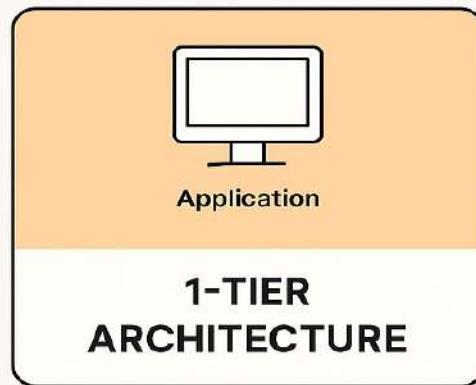
- Efficient for large networks with many users
- Central management of data and security-Servers manage data and authentication
- Easier to back up and maintain
- Scalable infrastructure- Servers can handle multiple clients

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

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

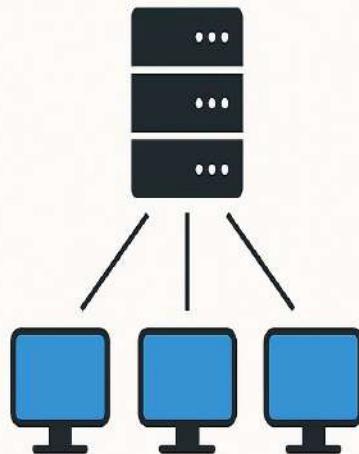
Type	Description	Examples	Pros	Cons
1. Pure P2P	Every peer has equal capabilities and responsibilities. No central server.	BitTorrent (initial design), Gnutella	Full decentralization, high fault tolerance	Poor search efficiency, redundant traffic
2. Hybrid P2P	A central server assists with functions 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



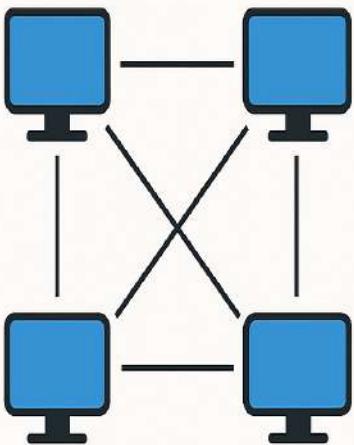
Feature	Client-Server	Peer-to-Peer
Control	Centralized	Decentralized
Scalability	Scales with server resources	Scales with number of peers
Examples	Web services, email, banking systems	File sharing, blockchain, Skype
Reliability	Depends on server	Depends on peers
Cost	High (infrastructure/server)	Low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

CLIENT-SERVER



- Clients request services or resources
- Server provides services or resources

PEER-TO-PEER



- Peers are both clients and servers
- Resources are shared among peers



Basic Network Protocols

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

Few Protocols and Description



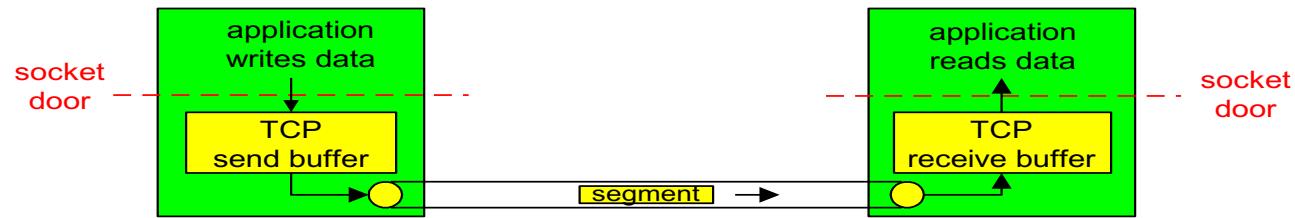
Protocol	Description
IPv4	Internet Protocol version 4. IPv4 uses 32-bit addresses and provides packet delivery service for TCP, UDP, SCTP, ICMP, and IGMP.
IPv6	Internet Protocol version 6. IPv6 uses 128-bit addresses.
TCP	Transmission Control Protocol. TCP is a connection-oriented protocol that provides 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)



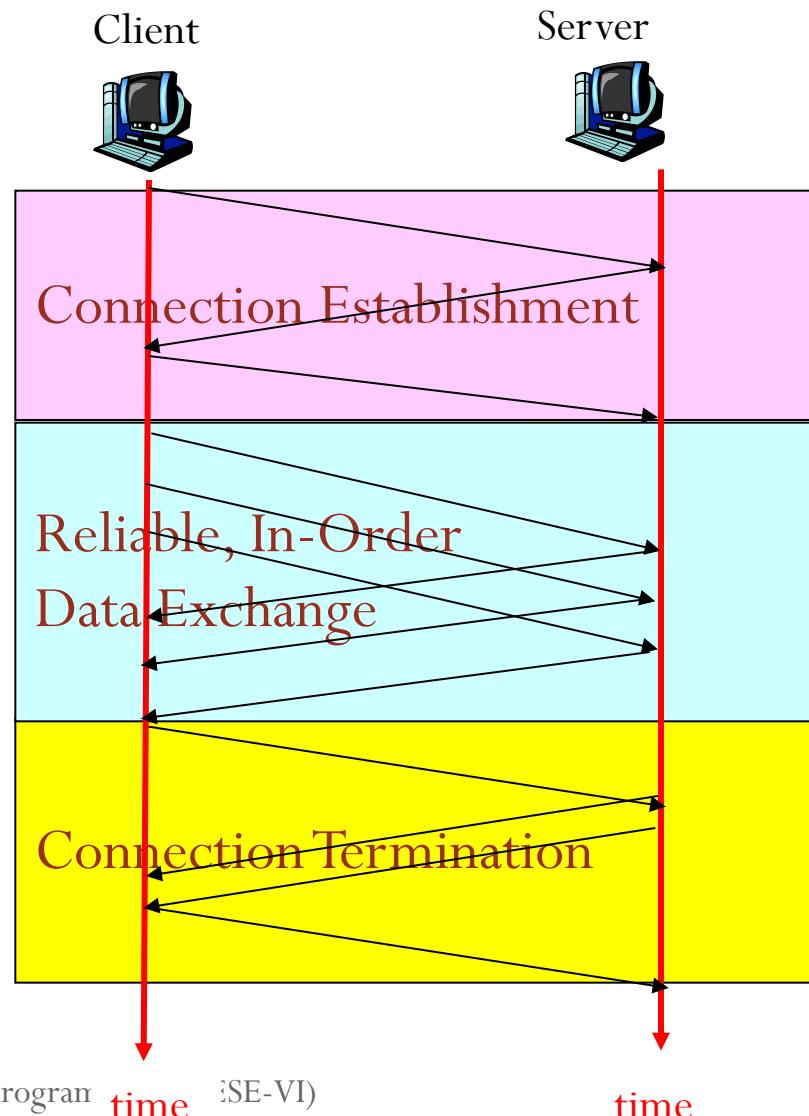
1. **Connection:** TCP provides connections between clients and servers. A TCP client establishes a connection with a server, exchanges data across the connection, and then terminates the connection.
2. **Reliability:** TCP requires acknowledgment when sending data. If an acknowledgment is not received, 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:

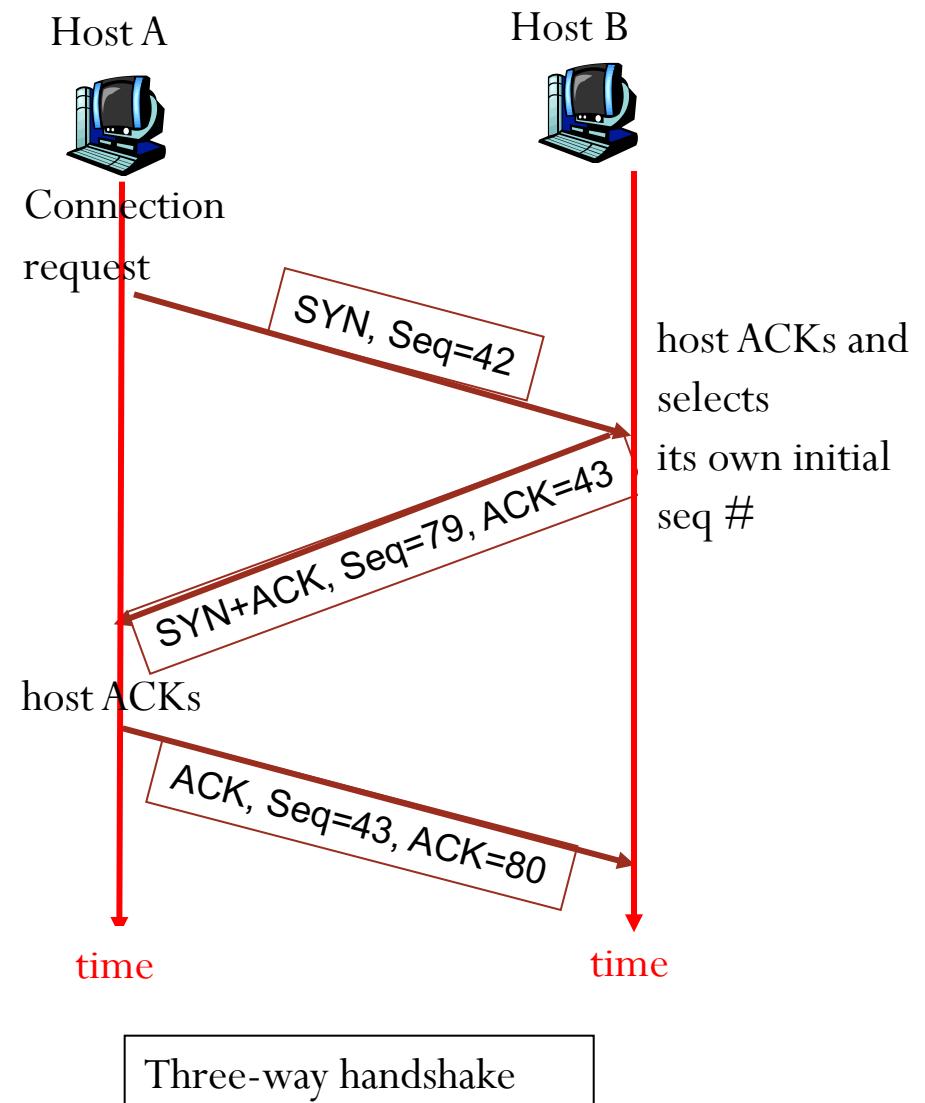
Step 1: client host sends TCP SYN segment to server

- 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



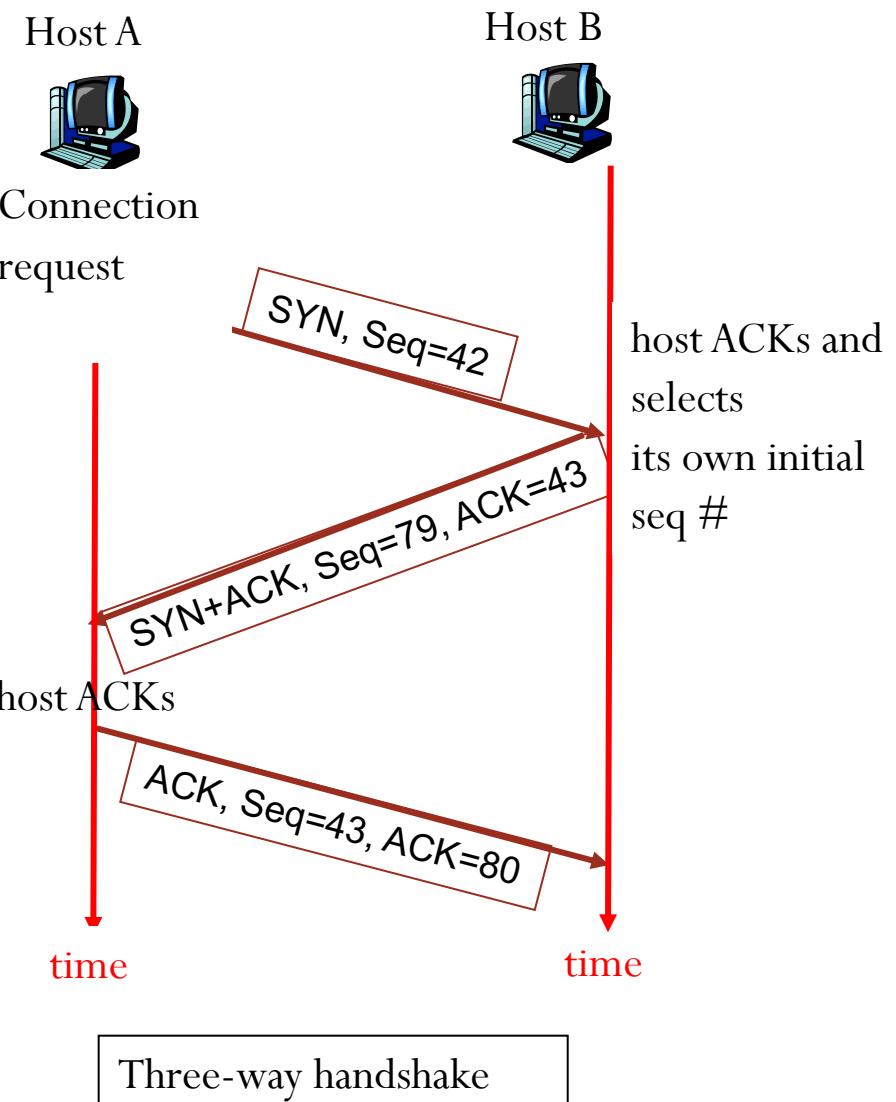
Connection Establishment (cont)

Seq. #'s:

- byte stream “number” of first byte in segment’s data

ACKs:

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



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

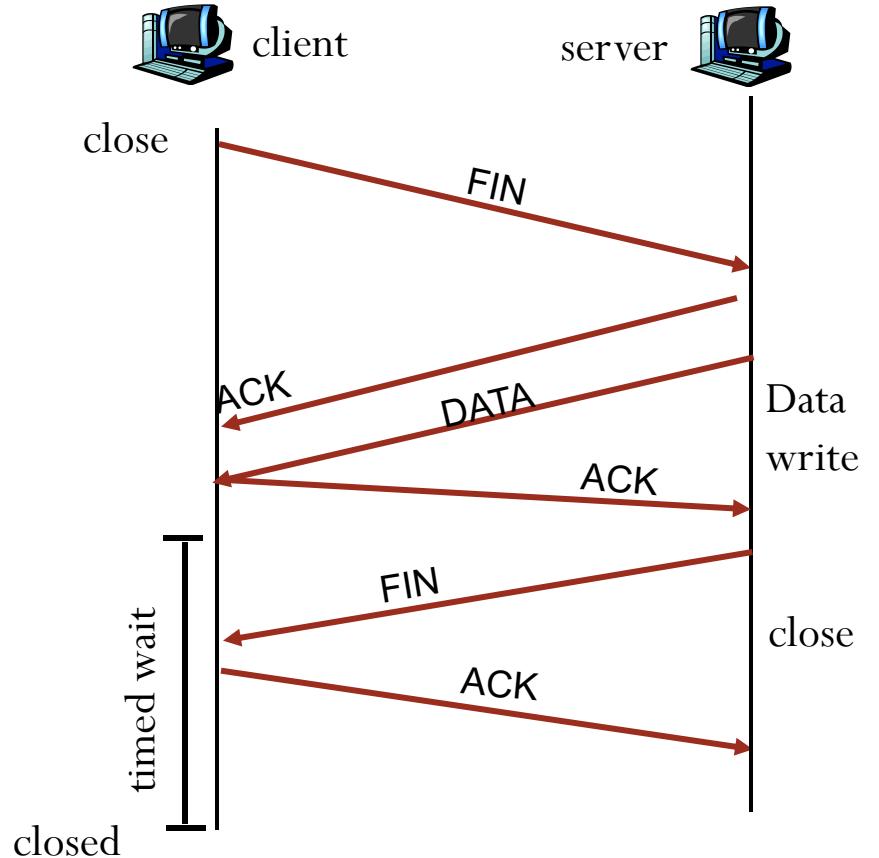
Closing a connection:

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.



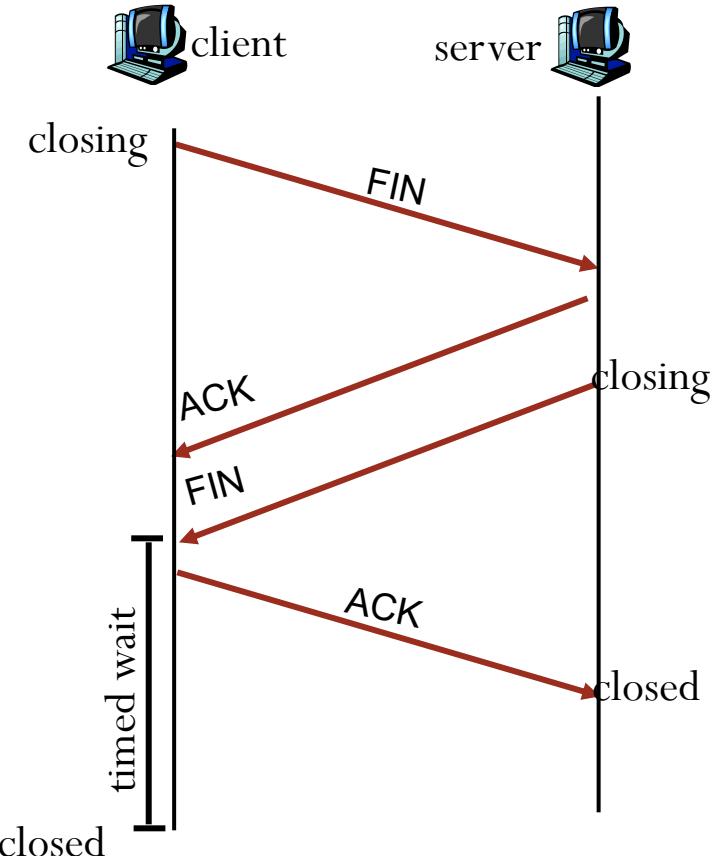
TCP Connection Termination

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
idle 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



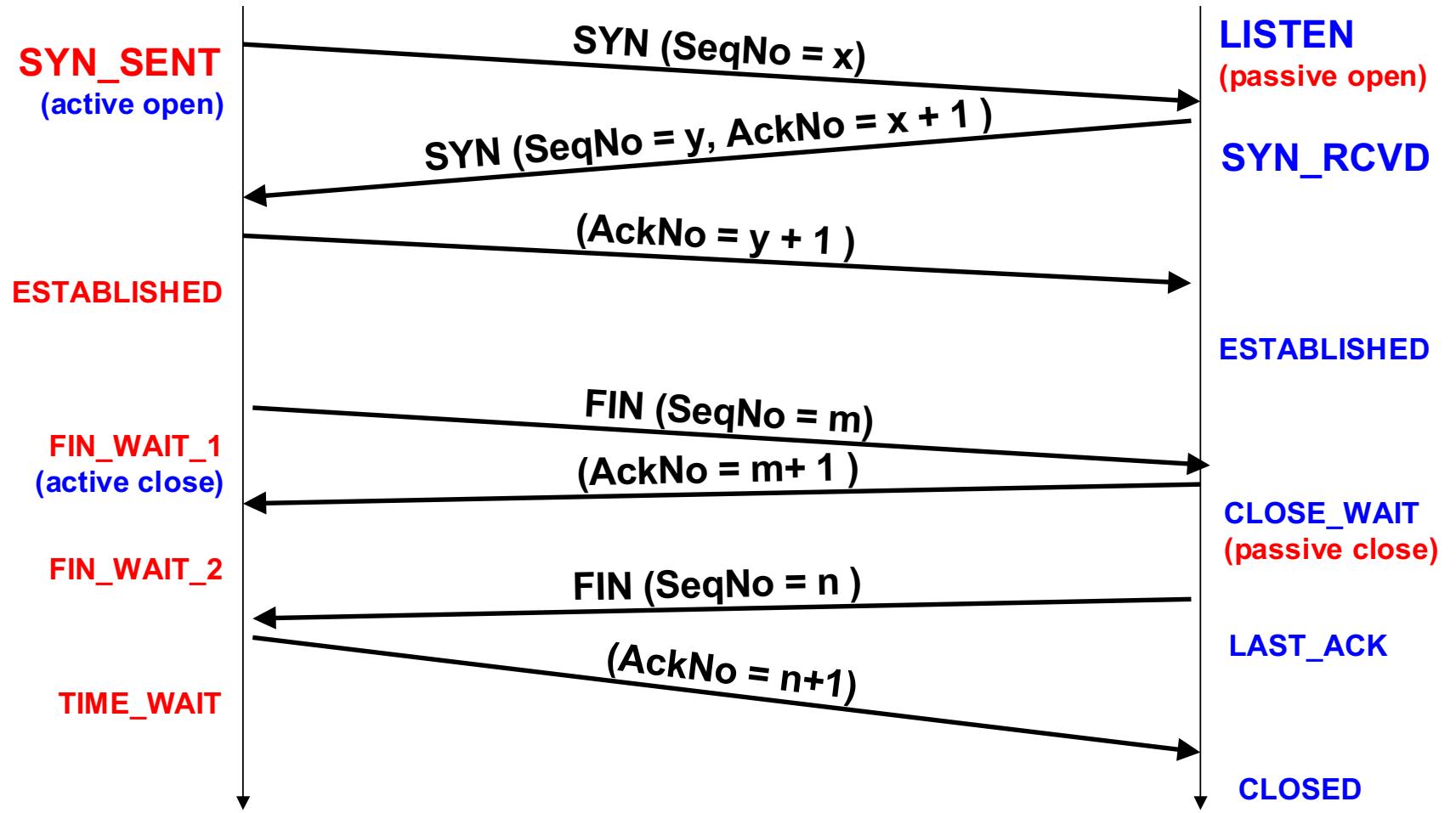
TCP/IP State Transition Diagram



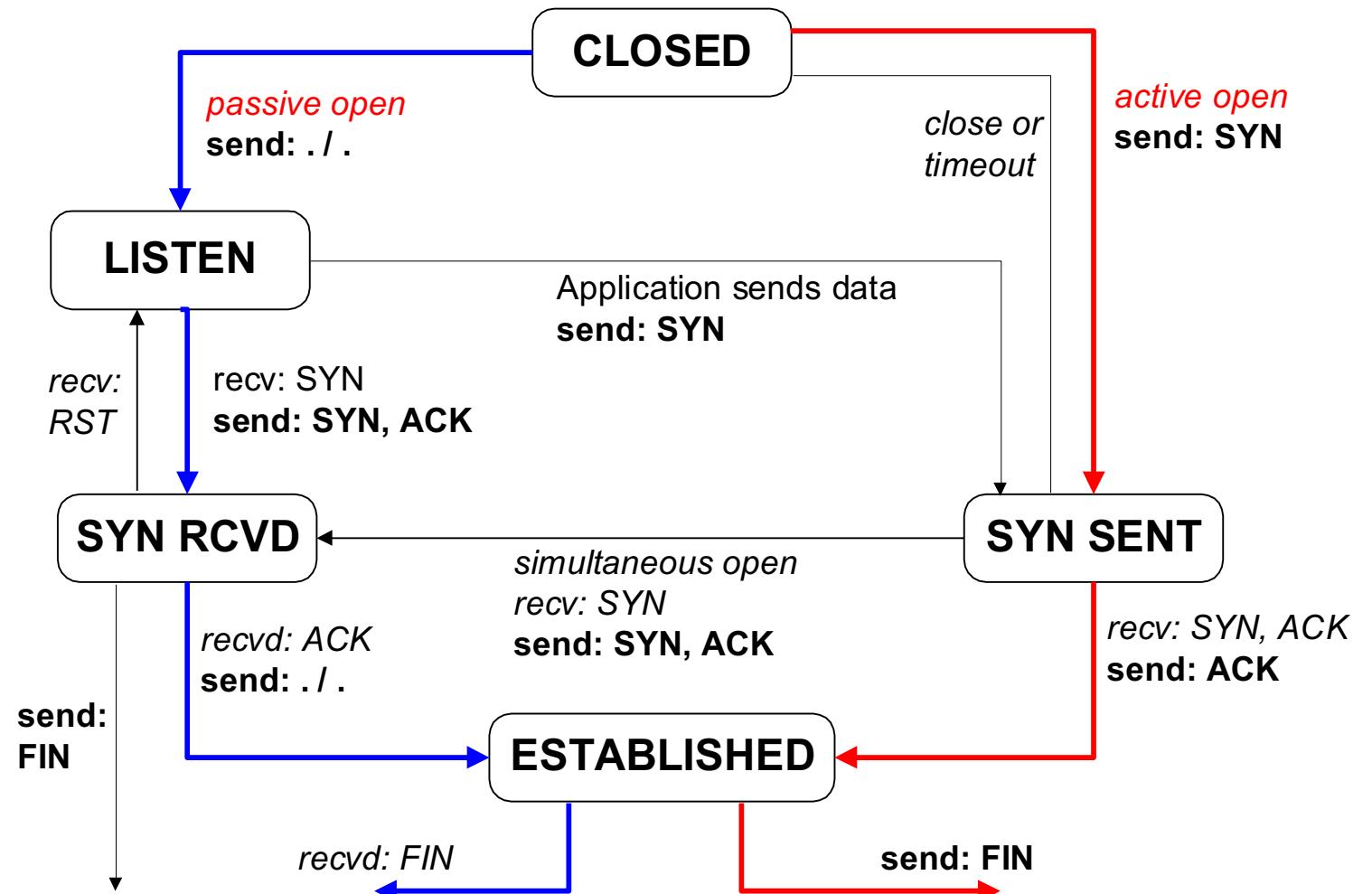
TCP States

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 simultaneously
CLOSE WAIT	Server has initiated a release
LAST ACK	Wait for pending packets

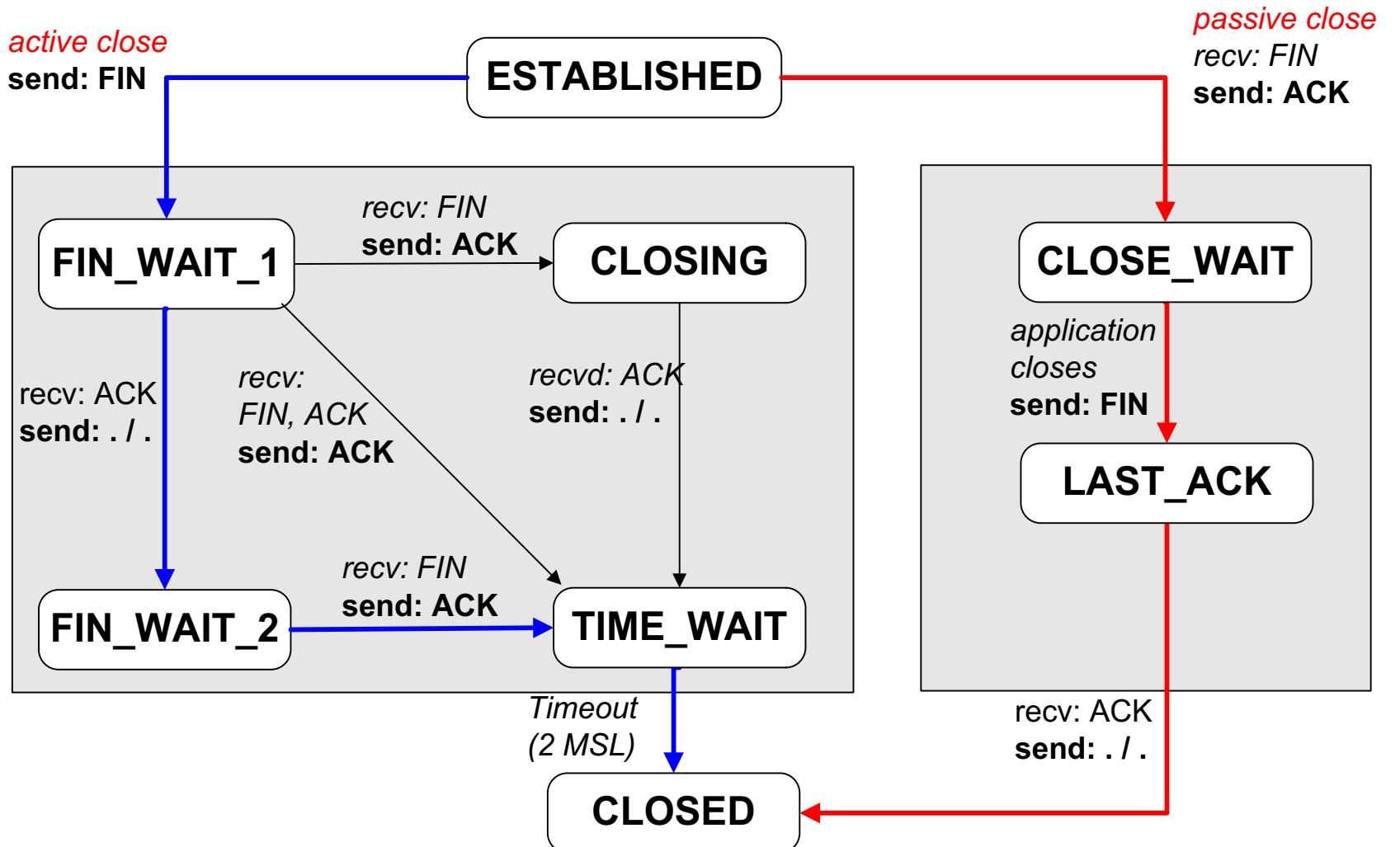
TCP States in “Normal” Connection Lifetime



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

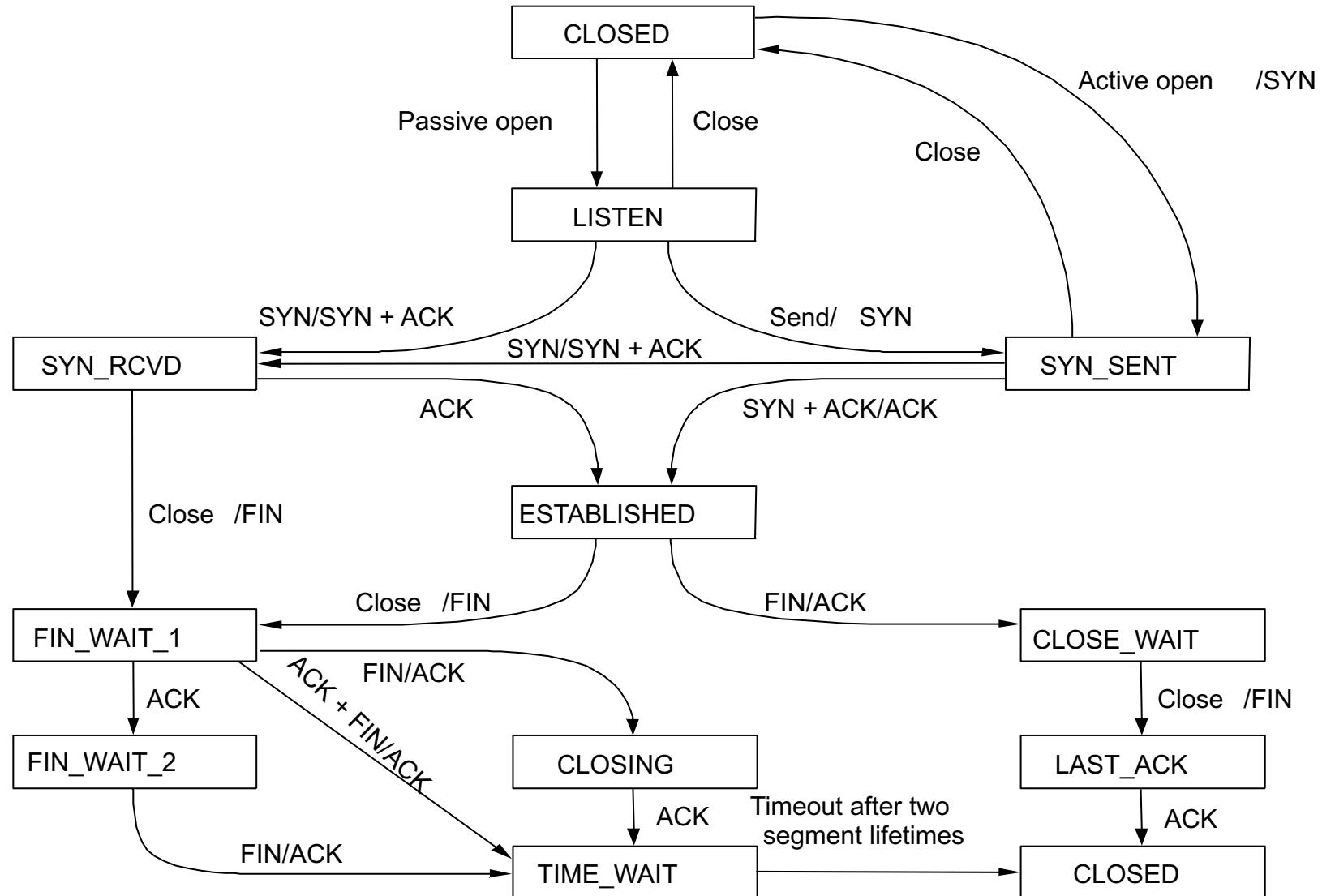
- Why?
TCP is given a chance to re resent the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)
- The MSL is set to 2 minutes or 1 minute or 30 seconds.



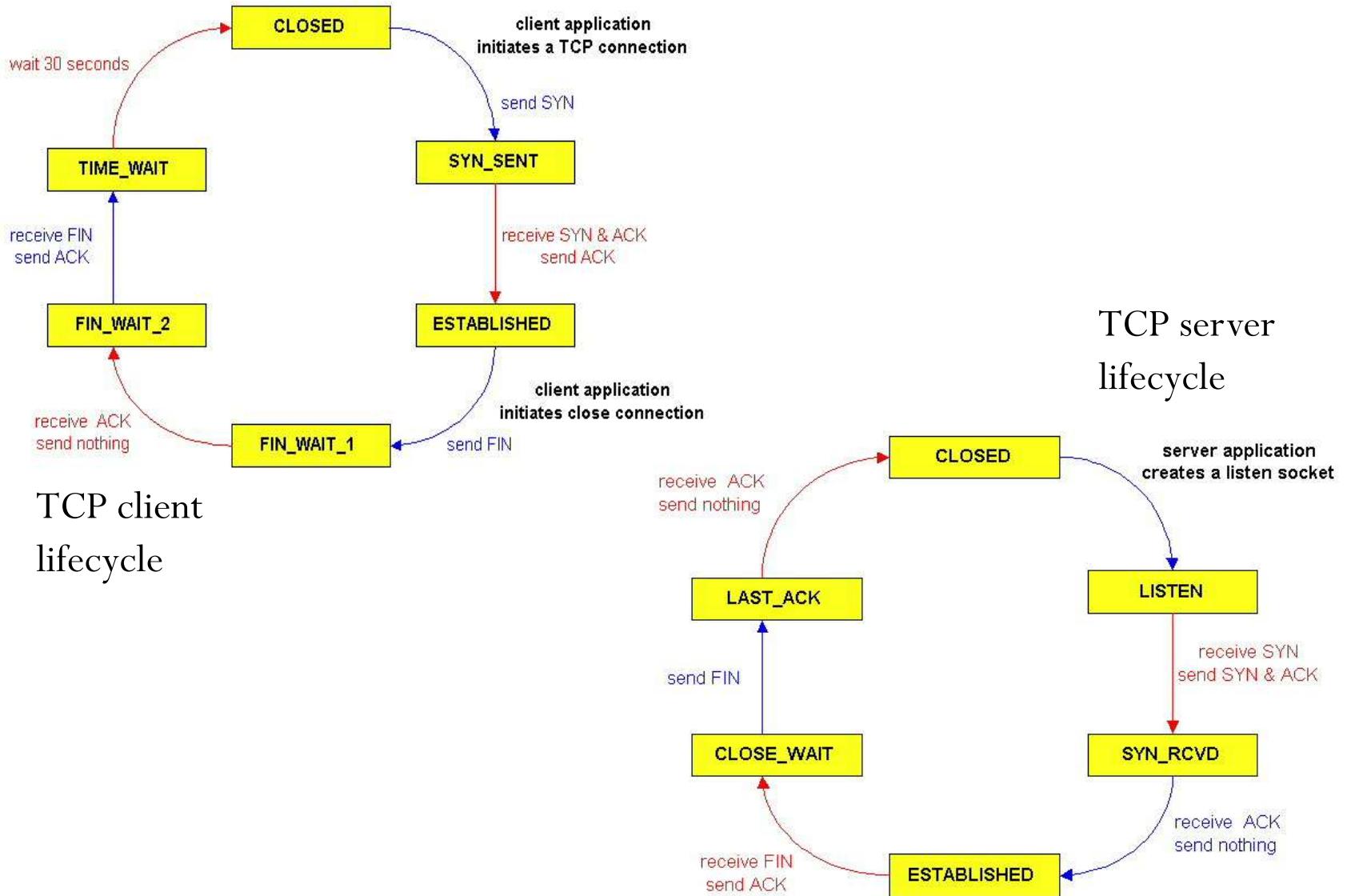
Resetting Connections

- 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





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*.

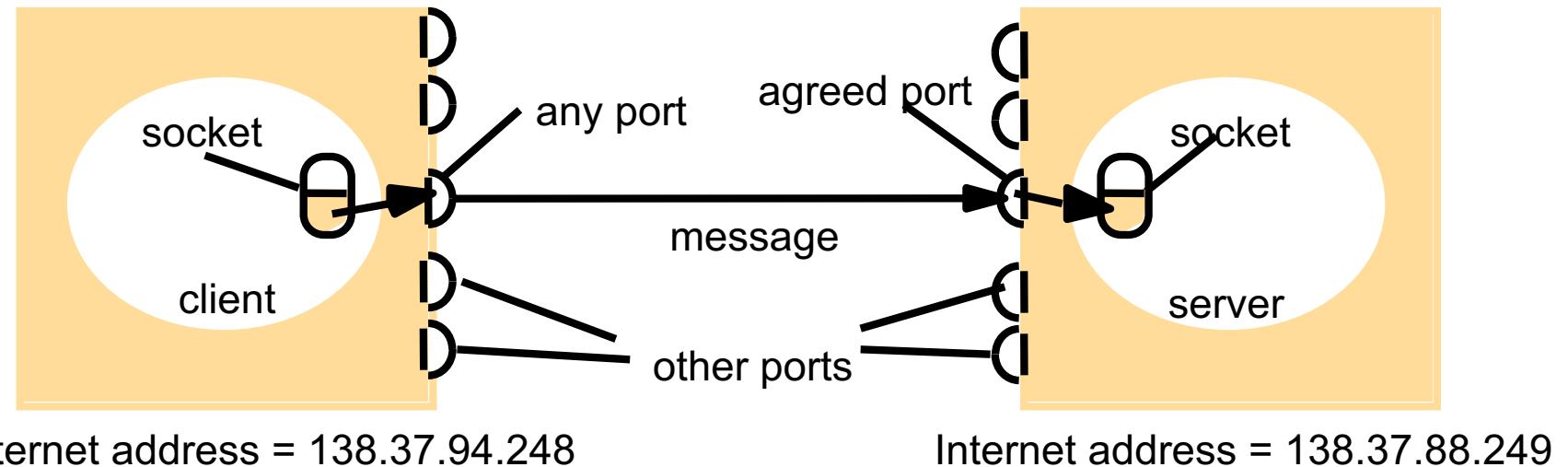
Key Properties of Sockets:

- **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



Network Programming

BESE-VI – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np



Chapter 2:

Basics of Unix Network Programming

(12 hrs)

Outline



1. Overview of Unix OS and Network Administration
2. Unix socket introduction
3. Socket Address Structures
 - i. Genric socket: struct sockaddr
 - ii. IPv4: struct sockaddr_in
 - iii. IPv6: struct sockaddr_in6
 - iv. Unix Domain Sockets: struct sockaddr_un
 - v. Storage: struct sockaddr_storage
4. Comparisions of various socket address structure
5. Value result arguments and system calls
6. Byte ordering and manipulation functions
7. Hostname and Network Name Lookups
4. Basic socket system calls
 - i. Elementary TCP Sockets
 - ii. Elementary UDP Sockets
5. Unix Domain Socket
 - i. Passing file descriptors
6. Signal Handling in Unix
 - i. Introduction to signals
 - ii. Handling signals in network applications (e.g., SIGINT, SIGIO)
 - iii. Signal functions (signal(), sigaction())
7. Daemon Processes
 - i. Characteristics of daemon processes
 - ii. Converting a program into a daemon
 - iii. Daemonizing techniques in Unix

Overview of Unix OS



- Unix is a *powerful, multiuser, multitasking* operating system originally developed in the 1970s at Bell Labs.

History and Evolution:

- Is created by Ken Thompson and Dennis Ritchie at AT&T's Bell Labs.
- Written originally in assembly, later rewritten in C.
- Basis for many open-source and commercial systems.
- It has served as the foundation for many modern operating systems such as **Linux**, **macOS**, **Solaris**, and **BSD**.
- Unix operating systems are known for their robust features, including multi-user capabilities, multitasking, a hierarchical file structure, and built-in networking.
- Network administration on Unix involves managing network devices, configuring services, and ensuring network security.

Overview of Unix OS



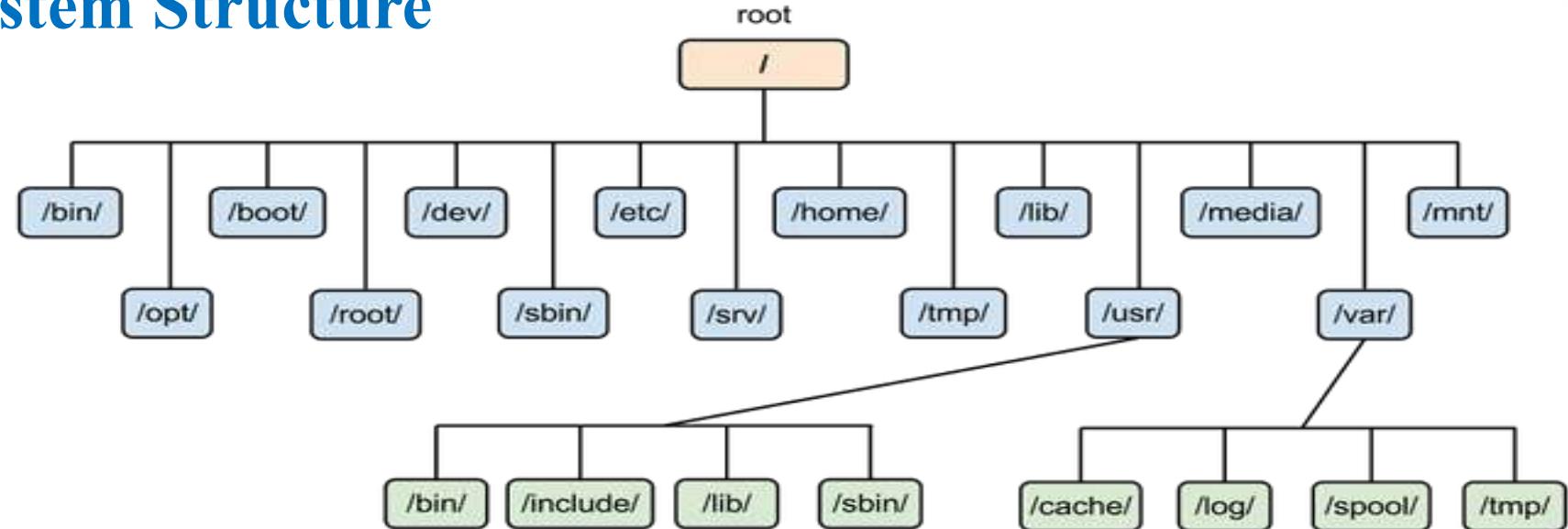
Key features of Unix OS

1. *Multiuser Support*: Allows multiple users to access the system simultaneously without interfering with each other.
2. *Multitasking*: Ability to run multiple processes at once.
3. *Portability*: Written in C, making it adaptable across hardware platforms.
4. *Security*: File permission and user management systems.
5. *Stability and Efficiency*: Minimal crashes, efficient memory and process handling.
6. *Shell Interface*: CLI and scripting for task automation.

Overview of Unix OS



Unix File System Structure



./bin: short for binaries, this is the directory where many commonly used executable commands reside

./dev: contains device-specific files

./etc: contains system configuration files

./home: contains user directories and files

./lib: contains all library files

./mnt: contains device files related to mounted devices

./proc: contains files related to system processes

./root: the root users' home directory (note this is different than ` / `)

./sbin: binary files of the system reside here.

./tmp: storage for temporary files that are periodically removed from the file system

./usr: It is the directory holding user home directories, its use has changed, and it also contains executable commands

./var: It is a short form for 'variable', a place for files that may often change



Overview of Unix OS

Basic Unix Commands

Command	Description
ls	Lists files in the current directory
cd	Changes the directory
cp, mv, rm	Copy, move, and delete files
chmod	Change file permissions
chown	Change file owner
ps, top	View active processes
kill	Terminate a process
man	Display manual for commands
cron	Schedule jobs/tasks periodically

Overview of Unix Network Administration



- Network administration involves *managing, configuring, monitoring*, and *troubleshooting* network components to ensure smooth communication between systems.
- Unix systems provide many powerful tools for network administration.

Network Configuration in Unix

- **Interfaces:** *eth0, wlan0, lo*
- **Configuration files:**
 - */etc/network/interfaces* (Debian/Ubuntu legacy)
 - */etc/netplan/*.yaml* (Ubuntu 18.04+)
 - */etc/sysconfig/network-scripts/* (RHEL/CentOS)
- **Tools:**
 - *ifconfig* – Legacy tool to view and configure interfaces.
 - *ip addr, ip link* – Modern tools for networking tasks.
 - *nmcli* – Command-line tool for NetworkManager.

Overview of Unix Network Administration

Essential Networking Tools



Tool/Command	Purpose
ping	Tests connectivity to a remote host
traceroute	Shows the path packets take to a host
netstat / ss	Displays active connections & ports
nslookup, dig	DNS query tools
scp, rsync	Secure file transfer between systems

Common Network Services on Unix

- **SSH (Secure Shell):** Remote login access (ssh user@host)
- **FTP / SFTP:** File transfer protocols
- **HTTP / HTTPS:** Web server services (Apache, Nginx)
- **NFS / SMB:** Network file sharing
- **Mail Services:** SMTP, POP3, IMAP using tools like Postfix or Dovecot

Overview of Unix Network Administration



Firewall and Security Tools

- **iptables**: Powerful firewall for setting packet filtering rules.
- **ufw (Uncomplicated Firewall)**: Simpler frontend for iptables.
- **Example:**
 - `sudo ufw allow 22/tcp` : Allows incoming TCP traffic on port 22 (the default port for SSH). This enables secure remote login to the system via SSH and is essential when configuring a server for remote access.
 - `sudo ufw enable`: Activates the firewall with the configured rules.
 - `sudo ufw status` : Displays the current firewall rules and their status.
- **fail2ban**: Scans logs and bans IPs with suspicious activity (e.g., failed SSH logins).
- **SELinux/AppArmor**: Kernel-level security modules (in RHEL/Ubuntu respectively).
- **Log Monitoring**: Check logs in `/var/log/` for intrusion or service errors.

Monitoring and Troubleshooting

- **Log Files:**
 - `/var/log/syslog`: System messages
 - `/var/log/auth.log`: Authentication attempts
 - `/var/log/messages`: General log messages
- **System Monitoring Tools:**
 - `top, htop`: Real-time process monitoring
 - `uptime, vmstat, iostat`: System performance
 - `Network`: `iftop, nload, iptraf`

Overview of Unix Network Administration



Practical Admin Task

Task	Tool / Command
Set a static IP address	ip, nmcli, or config files
Start and enable SSH server	systemctl enable --now sshd
Add new user with access	useradd, passwd, usermod
Secure the server	Configure ufw, disable root SSH
Set up cron jobs	crontab -e
Monitor service status	systemctl status <service>
View open ports	ss -tuln, netstat -tulnp

Summary of Unix Network Administration



- *Unix* provides a robust platform for secure and efficient computing.
- *System administration* requires understanding the filesystem, permissions, and process control.
- *Network administration* in Unix includes configuring interfaces, running services like SSH or FTP, monitoring traffic, and securing the network using firewalls and logging tools.
- Mastery of the *CLI and shell scripting* is essential for automation and system management.



Unix socket introduction

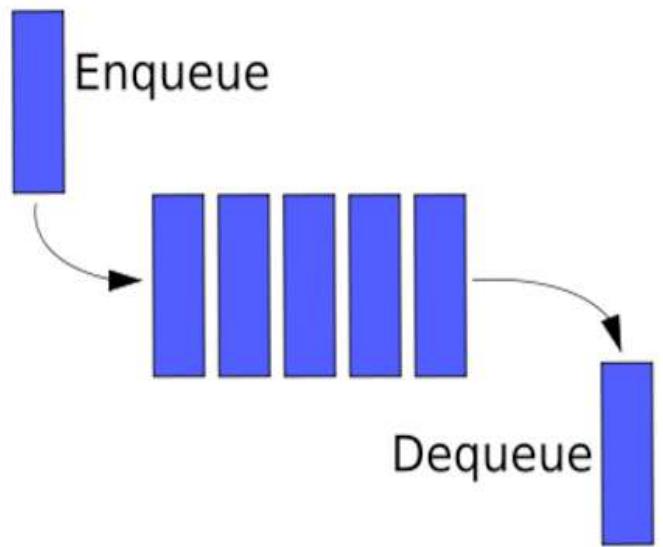
Characteristic of IPC



- Message passing between a pair of processes supported by two communication operations: **send** and **receive**
 - Defined in terms of destinations and messages.
- In order for one process A to communicate with another process B:
 - A sends a message (sequence of bytes) to a destination
 - Another process at the destination (B) receives the message.
- This activity involves the communication of data from the **sending** process to the **receiving** process and may involve the **synchronization** of the two processes

Sending and Receiving

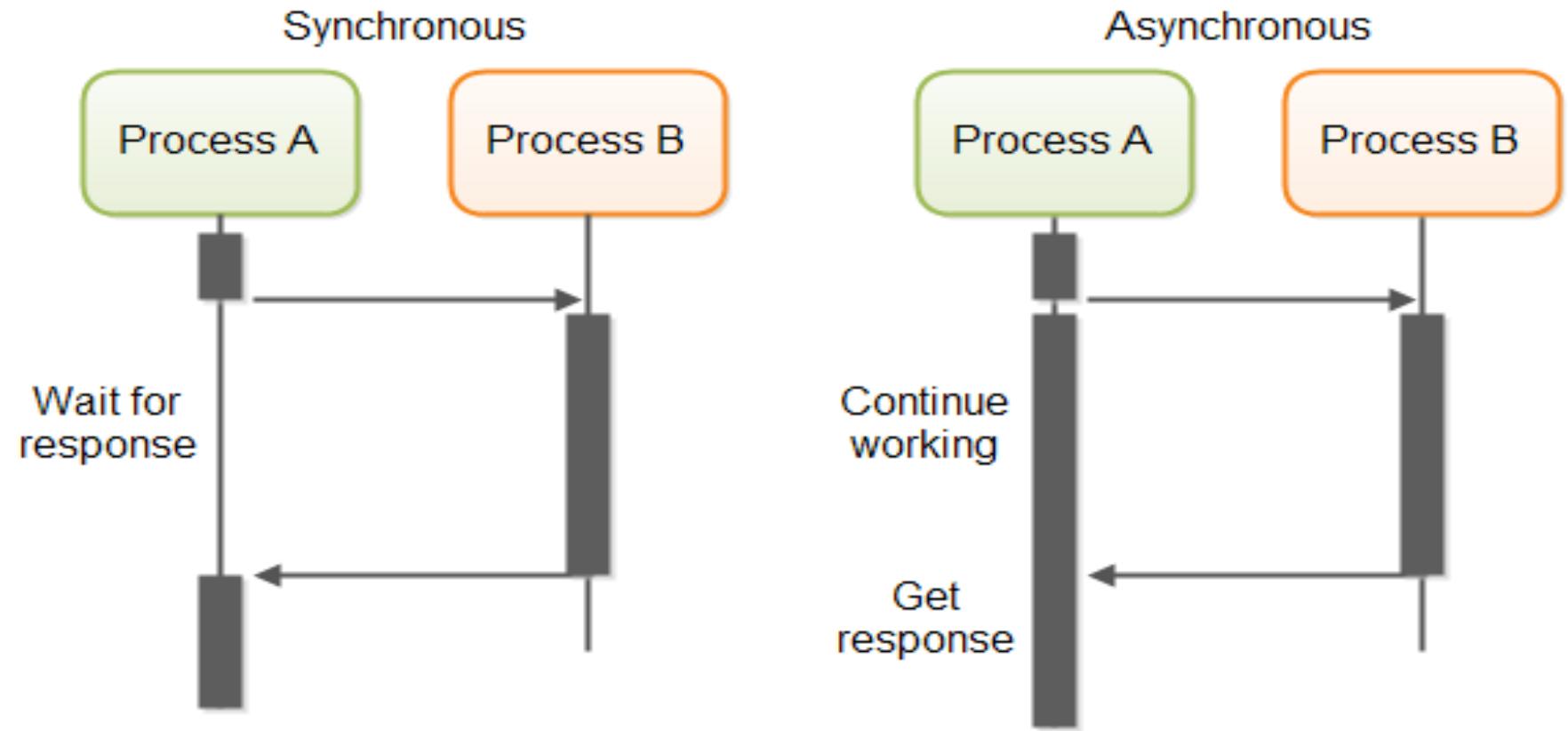
- A queue is associated with each message destination.
- Sending processes cause messages to be added to remote queues.
- Receiving processes remove messages from local queues.
- Communication between the sending and receiving process may be either **Synchronous** or **Asynchronous**.



Synchronous and asynchronous Communication



Synchronous Communication	Asynchronous Communication
<ul style="list-style-type: none"> ➤ The sending and receiving processes synchronize at every message. ➤ In this case, both send and receive are blocking operations: <ul style="list-style-type: none"> ➤ whenever a send is issued the sending process is blocked until the corresponding receive is issued; ➤ whenever a receive is issued the receiving process blocks until a message arrives. 	<ul style="list-style-type: none"> ➤ The send operation is non-blocking: <ul style="list-style-type: none"> ➤ the sending process is allowed to proceed as soon as the message has been copied to a local buffer; ➤ The transmission of the message proceeds in parallel with the sending process. ➤ The receive operation can have blocking and non-blocking variants: <ul style="list-style-type: none"> ➤ [non-blocking] the receiving process proceeds with its program after issuing a receive operation; ➤ [blocking] receiving process blocks until a message arrives.





Network Programming Introduction

- Typical applications today consist of many cooperating processes either on the **same host** or on **different hosts**.
- For example, consider a client-server application. How to share (large amounts of) data?
- Share files? How to avoid **contention**? What kind of system support is available?
- We want a general mechanism that will work for processes irrespective of their location.



IPC using sockets

Purposes of IPC

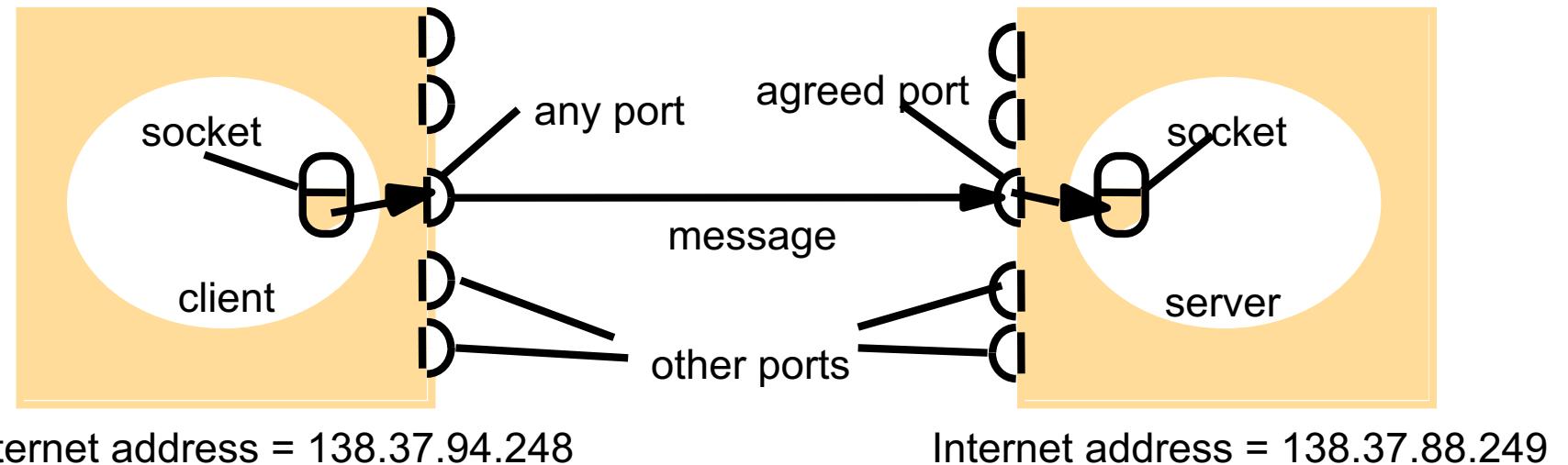
- Data transfer
 - Sharing data
 - Event notification
 - Process control
- What if wanted to communicate between processes that have no common ancestor? Ans: **sockets**
- IPC for processes that are not necessarily on the **same host**.
- Sockets use names to refer to one another: Means of network IO.



What are sockets?

- Socket is an **abstraction** for an **end point of communication** that can be manipulated with a **file descriptor**.
 - **File descriptor:** File descriptors are **nonnegative** integers that the kernel uses to identify the file being accessed by a particular process. Whenever the **kernel** opens an existing file or creates a new file it returns the file descriptor.
- A socket is an abstraction which provides an endpoint for communication between **processes**.
- A **socket address** is the combination of an **IP address** (the location of the computer) and a **port** (a specific service) into a **single identity**.
- **Interprocess communication** consists of transmitting a message between a **socket** in **one process** and a socket in **another process**.
- A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. Example: UNIX domain, internet domain.

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.



IPC using sockets

- **IP address and port number:** In IPv4 about 2^{16} ports are available for use by user processes.
- **Socket** is associated with a protocol.
- IPC is transmitting a message between a socket in **one process** to a socket in **another process**.
- Messages sent to particular IP and port# can be received by the process whose socket is associated with that IP and port#.
- Processes cannot share ports with other processes within the computer. Can receive messages on diff ports.

Sockets and Sockets API



Socket

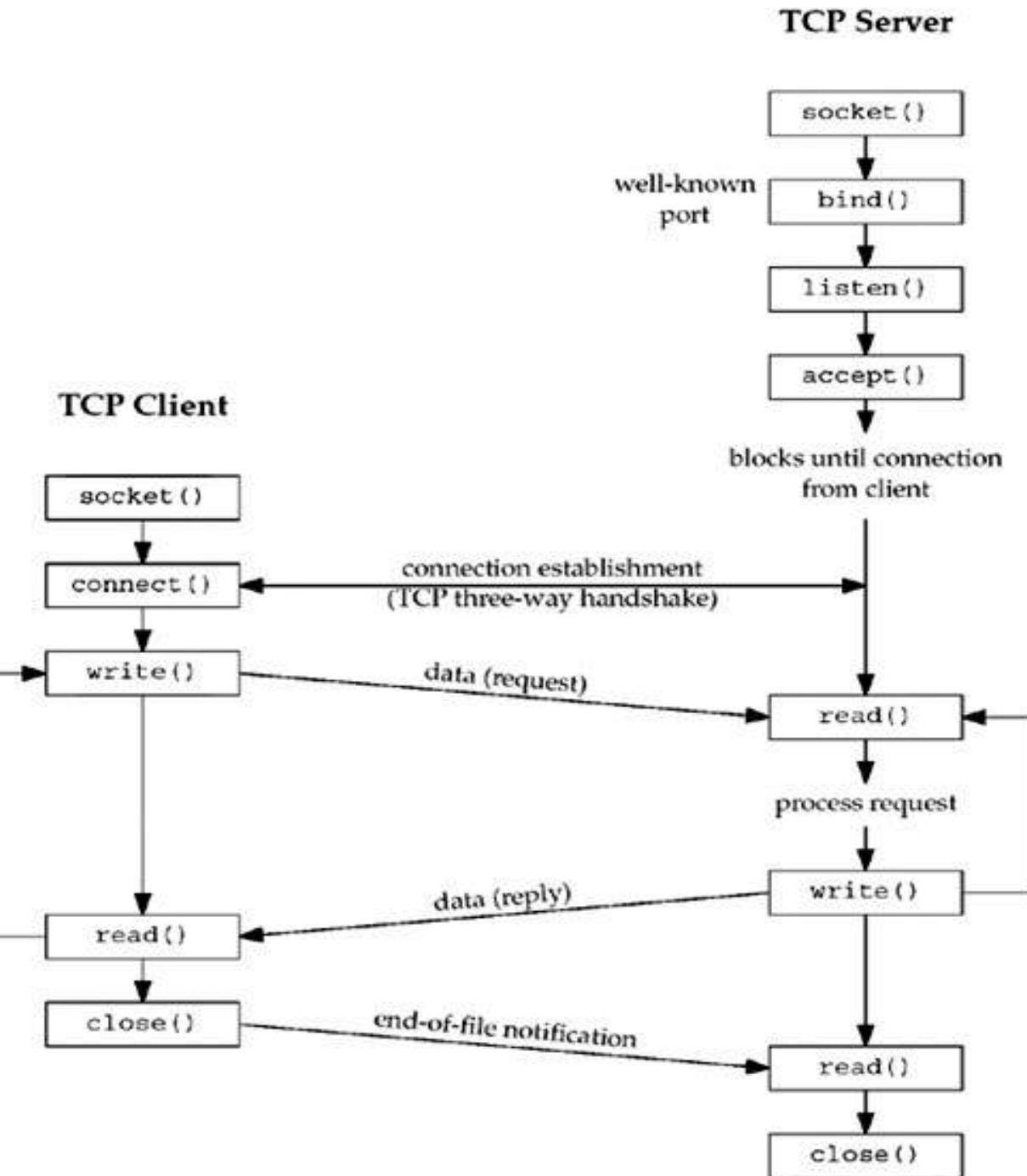
- ❖ To the **kernel**, a **socket is an endpoint of communication**.
- ❖ To an **application**, a socket is a **file descriptor that lets the application read/write from/to the network**.
- ❖ Remember: All Unix I/O devices, including networks, are modeled as files.
- **Clients** and **servers** communicate with each by reading from and writing to socket descriptors.
- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.

Socket API

- A socket API is an **application-programming interface** (API), usually provided by the **operating system**, that allows application programs to control and use network sockets.
- Internet socket APIs are usually based on the **Berkeley sockets** standard.
- In the **Berkeley sockets** standard, sockets are a form of **file descriptor** (a file handle)
- In **inter-process communication**, each end will generally have its own socket, but these may use different APIs: they are abstracted by the network protocol.
- A socket address is the combination of an **IP address** and a **port number**.



Socket Functions for TCP Client/Server





Socket Address Structure

Socket Address Structures



- Various structures are used in Unix Socket Programming to hold information about the address and port, and other information.
- Most socket functions require a **pointer to a socket address structure** as an argument.
- Generic socket address structure to hold socket information. It is passed in most of the socket function calls.
- Unix Socket Programming provides IP version specific socket address structures.
- The name of socket address structures begin with ***sockaddr_*** and end with a unique suffix for each protocol suite.

Socket Address Structures



Generic socket address structure (`struct sockaddr`):

- A socket address structures is always passed by reference when passed as an argument to any socket functions
 - For address arguments to `connect()`, `bind()`, and `accept()`.
- A generic socket address structure in the `<sys/socket.h>` header:

```
struct sockaddr {  
    uint8_t sa_len;  
    sa_family_t sa_family; /* address family: AF_xxx value */  
    char sa_data[14]; /* protocol-specific address */ };
```

- From an *application programmer's perspective*, the only use of these generic socket address structures is to cast pointers to protocol-specific structures.
- From the *kernel's perspective*, kernel must take the caller's pointer, cast it to a `struct sockaddr *`, and then look at the value of `sa_family` to determine the type of the structure.

Socket Address Structures



- **Internet-specific socket address (IPv4 socket address structure)**(*struct sockaddr_in*):
 - Called an "**Internet socket address structure**" and is defined in the <netinet/in.h> header.
 - Must cast (*sockaddr_in* *) to (*sockaddr* *) for connect, bind, and accept.

```
struct sockaddr_in {
    unsigned short sin_family; /* address family (always AF_INET) */
    unsigned short sin_port; /* 16 bit port num in network byte order */
    struct in_addr sin_addr; /* 32 bit IPv4 address in network byte order */
    unsigned char sin_zero[8]; /* pad to sizeof(struct sockaddr), unused */
};
```

```
struct in_addr {
    unsigned long s_addr; // this holds IPv4 address
};
```

- **POSIX** requires only three members in the structure: **sin_family**, **sin_addr**, and **sin_port**. Almost all implementations add the **sin_zero** member so that all socket address structures are at least 16 bytes in size.
- The **in_addr_t** datatype must be an **unsigned integer** type of at least **32 bits**, **in_port_t** must be an unsigned integer type of at least **16 bits**, and **sa_family_t** can be any unsigned integer type.
- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The **sin_zero** member is unused. By convention, we always set the entire structure to 0 before filling it in.
- Socket address structures are used only on a given host: The structure itself is not communicated between different hosts

Socket Address Structures



IPv6 socket address Structure (*struct sockaddr_in6*)

```
struct sockaddr_in6 {
    u_int16_t      sin6_family; // AF_INET6
    u_int16_t      sin6_port; // this holds port number
    u_int32_t      sin6_flowinfo; // this holds IPv6 flow information
    struct in6_addr sin6_addr; // used to hold IPv6 address
    u_int32_t      sin6_scope_id;// scope id
};

struct in6_addr {
    unsigned char s6_addr[16]; // this holds IPv6 address
};
```

- The **IPv6** family is **AF_INET6**, whereas the **IPv4** family is **AF_INET**
- The members in this structure are ordered so that if the **sockaddr_in6** structure is 64-bit aligned, so is the 128-bit **sin6_addr** member.
- The **sin6_flowinfo** member is divided into two fields:
 - The low-order 20 bits are the flow label
 - The high-order 12 bits are reserved
- The **sin6_scope_id** identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address

Socket Address Structure



Storage socket address structure (*struct sockaddr_storage*)

- It is a new generic socket address structure and defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct *sockaddr*.
- Unlike the struct *sockaddr*, the new struct *sockaddr_storage* is large enough to hold any socket address type supported by the system.
- The *sockaddr_storage* structure is defined by including the *<netinet/in.h>* header:

```
struct sockaddr_storage {
    uint8_t ss_len; /* length of this struct (implementation dependent) */
    sa_family_t ss_family; /* address family: AF_XXX value */
    /* implementation-dependent elements to provide:
     * a) alignment sufficient to fulfill the alignment requirements of * all socket address
     * types that the system supports. *
     * b) enough storage to hold any type of socket address that the * system supports. */
};
```

- Different from struct ***sockaddr*** in two ways
 1. If any socket address structures that the system supports have alignment requirements, the *sockaddr_storage* provides the strictest alignment requirement.
 2. The *sockaddr_storage* is large enough to contain any socket address structure that the system supports.
- The *sockaddr_storage* must be cast or copied to the appropriate socket address structure for the address given in *ss_family* to access any other fields.

Socket Address Structure



Unix Domain socket address structure (*struct sockaddr_un*)

- The UNIX-domain protocol family is a collection of protocols that provides local (on-machine) interprocess communication through the normal socket mechanisms.
- UNIX-domain addresses are variable-length filesystem pathnames of at most 104 characters.
- The include file *<sys/un.h>* defines this address:

```
struct sockaddr_un {  
    sa_family_t sun_family; /* Always AF_UNIX */  
    char sun_path[108]; /* Null-terminated socket pathname */  
};
```

- We can't bind a socket to an existing pathname (bind() fails with the error **EADDRINUSE**).
- It is common to bind a socket to an absolute pathname for a fixed location. Relative pathnames are rare, as they require the client to know the server's working directory.
- A socket may be bound to only one pathname; conversely, a pathname can be bound to only one socket.
- We can't use open() to open a socket.
- When the socket is no longer required, its pathname entry can (and generally should) be removed using unlink() (or remove()).

Comparison of various socket address structure



IPv4	IPv6	Unix	Datalink	Storage
sockaddr_in{}	sockaddr_in6{}	sockaddr_un{}	sockaddr_dl{}	sockaddr_storage{}
length AF_INET 16-bit port# 32-bit IPv4 address (unused) fixed-length (16 bytes)	length AF_INET6 16-bit port# 32-bit flow label 128-bit IPv6 address 32-bit scope ID fixed-length (28 bytes)	length AF_LOCAL pathname (up to 104 bytes) variable-length	length AF_LINK interface index type name len addr len sel len interface name and link-layer address variable-length	length AF_XXX (opaque) longest on system
Figure 3.1	Figure 3.4	Figure 15.1	Figure 18.1	

- Socket address structures all contain a one-byte length field
- The family field also occupies one byte
- Any field that must be at least some number of bits is exactly that number of bits.
- To handle variable-length structures, whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument

Comparison of various socket address structure



Structure	Header File	Used For	Address Family	Commonly Used Fields / Purpose
sockaddr	<sys/socket.h>	Generic socket container	AF_UNSPEC	<i>sa_family, sa_data</i> ; used for casting to specific types
sockaddr_in	<netinet/in.h>	IPv4 sockets	AF_INET	<i>sin_family, sin_port, sin_addr</i>
sockaddr_in6	<netinet/in.h>	IPv6 sockets	AF_INET6	<i>sin6_family, sin6_port, sin6_flowinfo, sin6_addr, sin6_scope_id</i>
sockaddr_un	<sys/un.h>	Unix domain sockets	AF_UNIX	<i>sun_family, sun_path</i> (up to 104 bytes)
sockaddr_storage	<sys/socket.h>	Generic storage (all types)	Depends	Large, aligned structure to hold any socket type (opaque)



Datatypes required by POSIX

Data Type	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>



Value-Result Arguments



Value-Result Arguments

- When a **socket address structure** is passed to any socket function, it is always passed by **reference**. That is, a **pointer to the structure** is passed.
- The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed:
 - From the process to the kernel
 - From the kernel to the process

From process to kernel

`bind()`, `connect()`, and `sendto()` functions pass a **socket address structure** from the process to the kernel

Arguments to these functions:

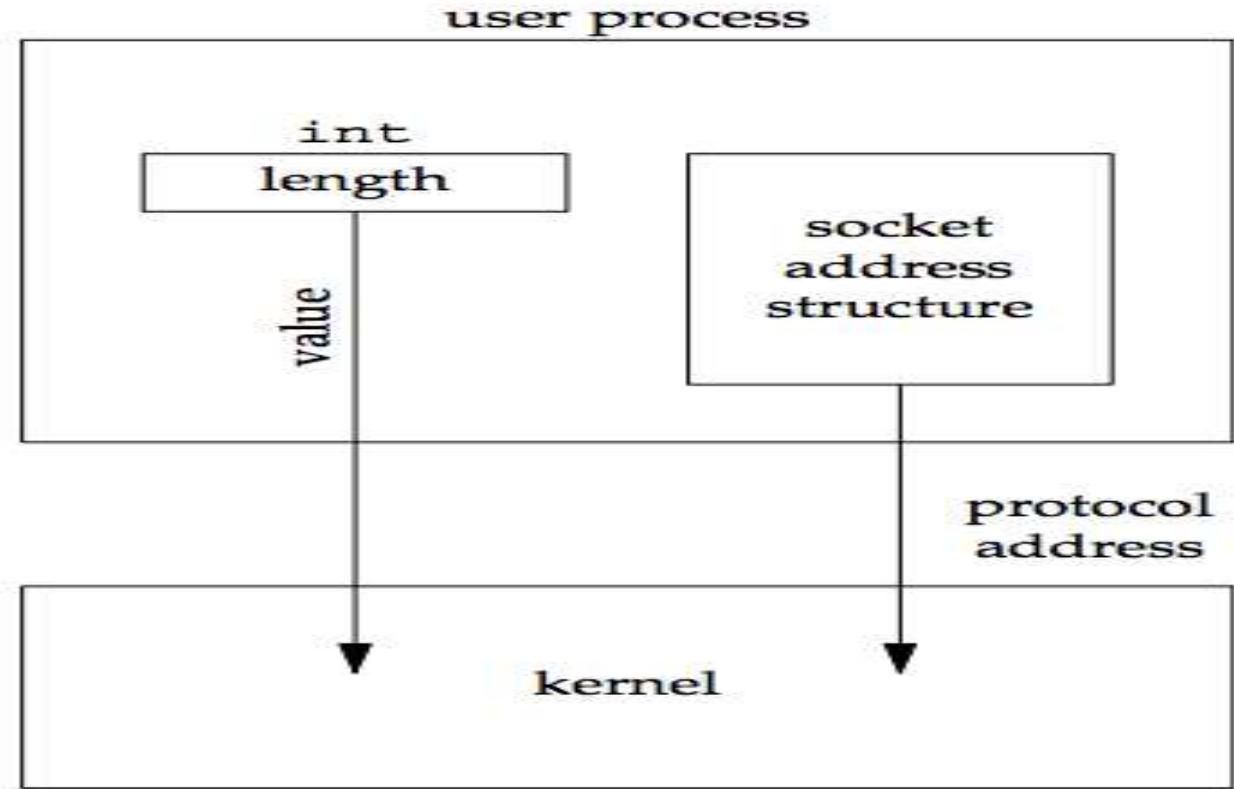
The pointer to the socket address structure

The integer size of the structure

```
struct sockaddr_in serv;
/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

- The datatype for the size of a socket address structure is actually `socklen_t` and not `int`, but the POSIX specification recommends that `socklen_t` be defined as `uint32_t`

Value-Result Arguments



Value-Result Arguments



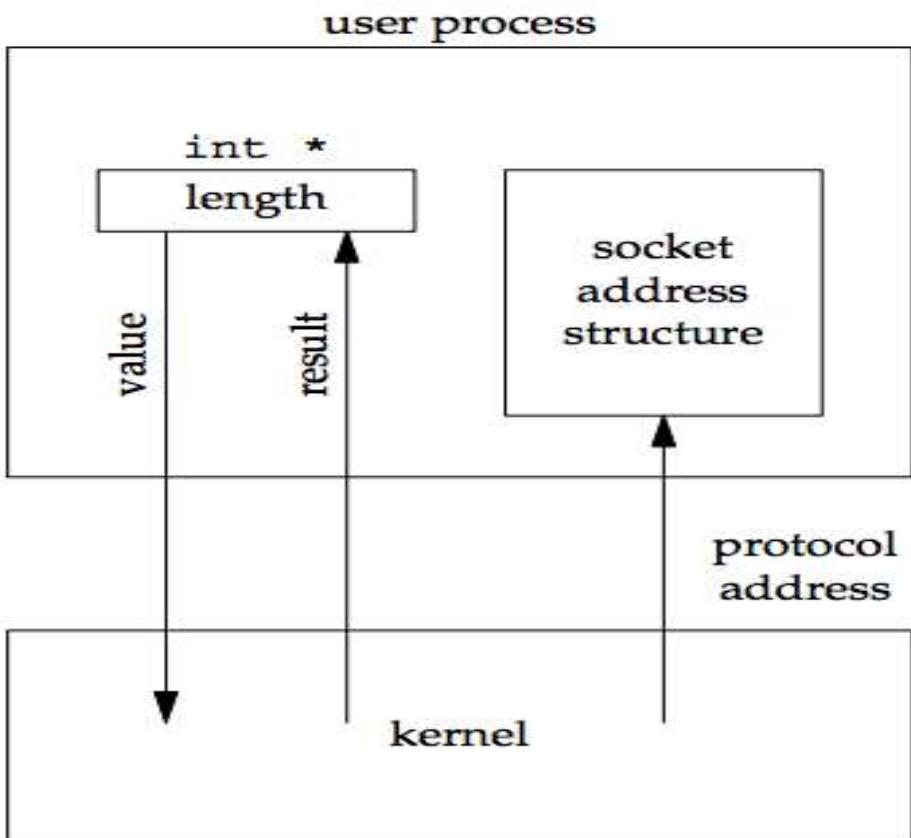
- **Kernel to Process**

accept(), *recvfrom()*, *getsockname()*, and *getpeername()* functions pass a **socket address structure** from the kernel to the process.

Arguments to these functions:

- The pointer to the socket address structure
- The pointer to an integer containing the size of the structure

```
struct sockaddr_un cli; /* Unix domain */
socklen_t len;
len = sizeof(cli); /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```



Value-Result Arguments



- Value-only: ***bind()*, *connect()*, *sendto()*** (from process to kernel)
- Value-Result: ***accept()*, *recvfrom()*, *getsockname()*, *getpeername()*** (from kernel to process, pass a pointer to an integer containing size)
 - *Tells process how much information kernel actually stored*
- As a **value**: it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in
- As a **result**: it tells the process how much information the kernel actually stored in the structure
- If the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: *16 for an IPv4 sockaddr_in* and *28 for an IPv6 sockaddr_in6*. But with a variable-length socket address structure (e.g., a Unix domain *sockaddr_un*), the value returned can be less than the maximum size of the structure.
- Though the most common example of a value-result argument is the length of a returned socket address structure.



Byte Ordering and Manipulation Functions

Byte Ordering Functions

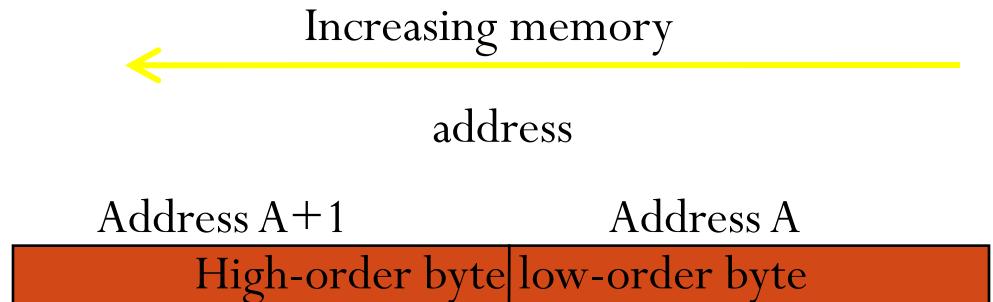


- Two ways to store 2 bytes (16-bit integer) in memory
 - Low-order byte at starting address → little-endian byte order
 - High-order byte at starting address → big-endian byte order
- *In a big-endian computer* → store 4F52
 - Stored as 4F52 → 4F is stored at storage address 1000, 52 will be at address 1001, for example
- *In a little-endian system* → store 4F52
 - it would be stored as 524F (52 at address 1000, 4F at 1001)
- Byte order used by a given system known as *host byte order*
- Network programmers use *network byte order*
- Internet protocol uses big-endian byte ordering for integers (port number and IP address)

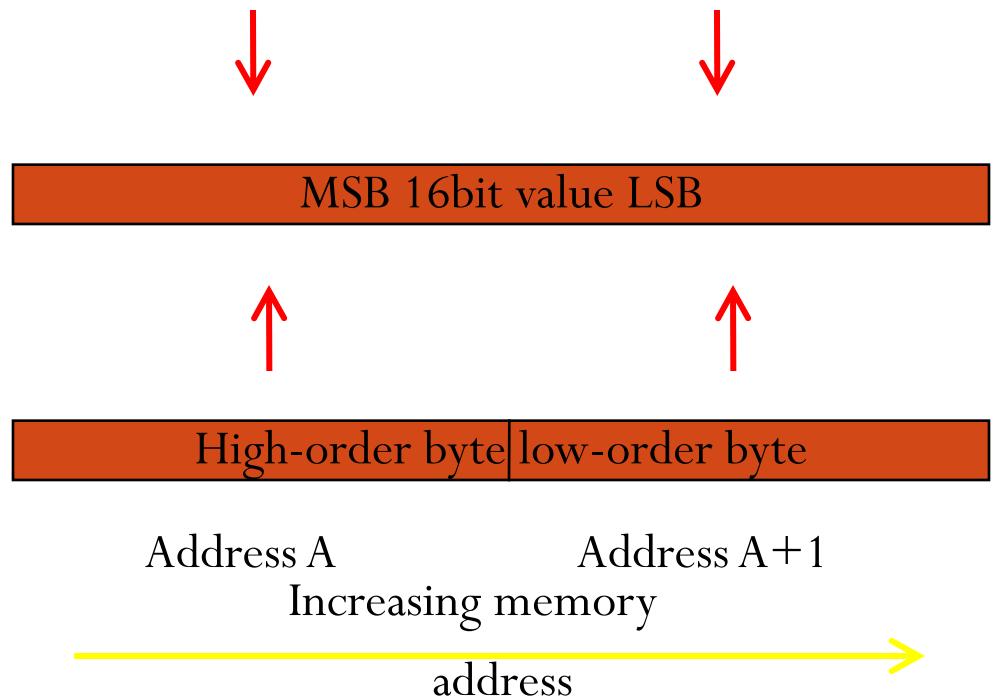
Byte Ordering Functions



Little-endian byte order:



big-endian byte order:



Byte Ordering Functions



```
#include      "unp.h"
int main(int argc, char **argv)
{
    union {
        short s;
    char c[sizeof(short)];
    } un;

    un.s = 0x0102;
    //printf("%s: ", CPU_VENDOR_OS);
    if (sizeof(short) == 2) {
        if (un.c[0] == 1 && un.c[1] == 2)
            printf("big-endian\n");
        else if (un.c[0] == 2 && un.c[1] == 1)
            printf("little-endian\n");
        else
            printf("unknown\n");
    } else
        printf("sizeof(short) = %d\n", sizeof(short));

    exit(0);
}
```

- Sample program to figure out little-endian or big-endian machine
- Source code in [byteorder.c](#)

Byte Ordering Functions



- Converts between **host byte order** and **network byte order**
 - 'h' = host byte order
 - 'n' = network byte order
 - 'l' = long (4 bytes), converts IP addresses
 - 's' = short (2 bytes), converts port numbers

```
#include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
/* Both return: value in network byte order */
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
/* Both return: value in host byte order */
```

- Must call appropriate function to convert between host and network byte order
- On systems that have the same ordering as the Internet protocols, four functions usually defined as null macros
 - `servaddr.sin_addr.s_addr = htonl(INADDR_ANY);`
 - `servaddr.sin_port = htons(13);`



Byte Manipulation Functions (from BSD)

```
#include <strings.h>
#include <strings.h>
void bzero(void *dest, size_t nbytes);
/*sets specified number of bytes to 0 in the destination */

void bcopy(const void *src, void *dest, size_t nbytes);
/* moves specified number of bytes from source to destination */

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
/* Returns: 0 if equal, nonzero if unequal */
```

- The memory pointed to by the *const* pointer is read but not modified by the function.



Byte Manipulation Functions (from ANSI C)

```
#include <string.h>
void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t nbytes);
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
/* Returns: 0 if equal, <0 or >0 if unequal
```

- **memset** sets the specified number of bytes to the value **c** in the destination.
- **memcpy** is similar to **bcopy**, but the order of the two pointer arguments is swapped.
 - One way to remember the order of the two pointers for *memcpy* is to remember that they are written in the same left-to-right order as an assignment statement in C:
 - *dest* = *src*;
- **memcmp** compares two arbitrary byte strings and returns 0 if they are identical. If not identical, the return value is either greater than 0 or less than 0, depending on whether the first unequal byte pointed to by *ptr1* is greater than or less than the corresponding byte pointed to by *ptr2*. The comparison is done assuming the two unequal bytes are unsigned chars.

Address Conversion Functions



- These functions convert Internet addresses between ASCII strings ("128.2.35.50") and network byte ordered binary values (values that are stored in socket address structures).

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
```

/* Returns: 1 if string was valid, 0 on error */

- Converts the C character string pointed to by *strptr* into its 32-bit binary network byte ordered value, which is stored through the pointer *addrptr*

```
in_addr_t inet_addr(const char *strptr);
```

/* Returns: 32-bit binary network byte ordered IPv4 address;
INADDR_NONE if error */

- does the same conversion, returning the 32-bit binary network byte ordered value as the return value. It is deprecated and any new code should use *inet_aton* instead

```
char *inet_ntoa(struct in_addr inaddr);
```

/* Returns: pointer to dotted-decimal string */

- converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

Address Conversion Functions



To handle both IPv4 and IPv6 addresses

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);
/* Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error */

• converts the string pointed to by strptr, storing the binary result through the pointer addrptr.
```

```
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
/* Returns: pointer to result if OK, NULL on error */

• does the reverse conversion, from numeric (addrptr) to presentation (strptr).
```

Examples:

```
if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
    err_quit("inet_pton error for %s", argv[1]);
ptr = inet_ntop (AF_INET, &addr.sin_addr, str, sizeof(str));

struct sockaddr_in6 addr6;
inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

- The family argument for both functions is either **AF_INET** or **AF_INET6**. If family is not supported, both functions return an error with **errno** set to **EAFNOSUPPORT**.



Dealing with IP Addresses

- IP Addresses are commonly written as strings ("128.2.35.50"), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;  
  
srv.sin_addr.s_addr = inet_addr("128.2.35.50");  
if(srv.sin_addr.s_addr == (in_addr_t) -1) {  
    fprintf(stderr, "inet_addr failed!\n");  
    exit(1);  
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;  
char *t = inet_ntoa(srv.sin_addr);  
if(t == 0) {  
    fprintf(stderr, "inet_ntoa failed!\n");  
    exit(1);  
}
```



Hostname and Network Name Lookups



Name and Address Conversion: Domain Name System(DNS)

- The DNS is used primarily to map between hostnames and IP addresses.
- These lookups are typically handled through system calls and library functions provided by the OS.
- A hostname can be either a simple name, such as `solaris` or **freebsd**, or a fully qualified domain name (FQDN), such as `solaris.unpbook.com`.
- Entries in the DNS are known as resource records (RRs).
- Domain name is converted to IP address by contacting a DNS server by calling functions in a library known as the resolver or local configuration files like **/etc/hosts**.



gethostbyname() Function

- This function is used to convert hostname to IP address.

```
#include <netdb.h>

struct hostent * gethostbyname(const char * hostname);
/* Returns: non-null pointer if OK, NULL on error with h_errno set */
```

- The non-null pointer returned by this function points to the following **hostent** structure.

```
struct hostent {
    char *h_name; /* official (canonical) name of host */
    char **h_aliases; /* pointer to array of pointers to alias names */
    int h_addrtype; /* host address type: AF_INET */
    int h_length; /* length of address: 4 */
    char **h_addr_list; /* ptr to array of ptrs with IPv4 addrs */
};
```

- This function can return only IPv4 addresses.

gethostbyname() Function...



- **Gethostbyname()** differs from the other socket functions that it does not set **errno** when an error occurs. Instead, it sets the global integer **h_errno** to one of the following constants defined by including **<netdb.h>**:

Error	Description
HOST_NOT_FOUND	No such host is known
TRY AGAIN	usually a temporary error indicating the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	Some unexpected server failure was encountered. This is a non-recoverable error
NO_DATA	The specified name is valid, but it does not have an A record.



gethostbyaddr() Function

- The function **gethostbyaddr()** takes a binary IPv4 address and tries to find the host-name corresponding to that address. This is the reverse of **gethostbyname**.

```
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
/*Returns: non-null pointer if OK, NULL on error with h_errno set */
```

- The **addr** argument is not a **char ***, but is really a pointer to an *in_addr* structure containing the IPv4 address.
- The **len** is the size of this structure: 4 for an IPv4 address. The family argument is **AF_INET**.

Print out the hostname associated with a specific IP address



```
const char *ipstr = "127.0.0.1"; //8.8.8.8
struct in_addr ip;
struct hostent *hp;

if (!inet_aton(ipstr, &ip))
errx(1, "can't parse IP address %s", ipstr);

if ((hp = gethostbyaddr((const void *)&ip, sizeof ip, AF_INET)) == NULL)
errx(1, "no name associated with %s", ipstr);

printf("name associated with %s is %s\n", ipstr, hp->h_name);
```

getservbyname() and getservbyport() Functions



- Services, like hosts, are often known by names, too.

```
#include <netdb.h>

struct servent *getservbyname (const char *servname, const char *proto);
>Returns: non-null pointer if OK, NULL on error
```

This function returns a pointer to the following structure.

```
struct servent {
    char *s_name; /* official name of service */
    char **s_aliases; /* alias list */
    int s_port; /* port service resides at */
    char *s_proto; /* protocol to use */
};
```

- The service name *servname* must be specified. If a protocol is also specified (proto is a non-null pointer), then the entry must also have a matching protocol.



getservbyport()

Getservbyport(), looks up a service given its port number and an optional protocol.

```
#include <netdb.h>

struct servent *getservbyport (int port, const char *proto name);
/* Returns: non-null pointer if OK, NULL on error */
```

- The *port* value must be network byte ordered. Typical calls to this function could be as follows:

```
struct servent *sptr;
sptr = getservbyport (htons (53), "udp"); /* DNS using UDP */
sptr = getservbyport (htons (21), "tcp"); /* FTP using TCP */
sptr = getservbyport (htons (21), NULL); /* FTP using TCP */
sptr = getservbyport (htons (21), "udp"); /* this call will fail */
*/
```

getaddrinfo() Function



- The gethostbyname and gethostbyaddr functions only support IPv4.
- The getaddrinfo supports both IPv4 and IPv6.
- The getaddrinfo function handles both name-to-address and service-to-port translation, and returns sockaddr structures instead of a list of addresses.
- These sockaddr structures can then be used by the socket functions directly.

```
#include <netdb.h>

int getaddrinfo (const char *hostname, const char *service, const struct addrinfo
*hints, struct addrinfo **result) ;

/* Returns: 0 if OK, nonzero on error */
```

- This function returns through the result pointer a pointer to a linked list of addrinfo structures, which is defined by including <netdb.h>.

```
struct addrinfo {
    int ai_flags; /* input flags */
    int ai_family; /* protocol family for socket */
    int ai_socktype; /* socket type */
    int ai_protocol; /* protocol for socket */
    socklen_t ai_addrlen; /* length of socket-address */
    struct sockaddr *ai_addr; /* socket-address for socket */
    char *ai_canonname; /* canonical name for service location */
    struct addrinfo *ai_next; /* pointer to next in list */
};
```

- The hostname is either a hostname or an address string (dotted-decimal for IPv4 or a hex string for IPv6). The service is either a service name or a decimal port number string. hints is either a null pointer or a pointer to an addrinfo structure that the caller fills in with hints about the types of information the caller wants returned.

gai_strerror Function



- The non zero error return values from getaddrinfo have specific names and meanings.
- The function gai_strerror takes one of these values (int) as an argument and returns a pointer to the corresponding error string.

```
#include <netdb.h>
const char * gai_strerror(int ecode);
/* Returns: pointer to string describing error message */
```

Nonzero error return constants from `getaddrinfo`.

Constant	Description	Constant	Description
EAI_AGAIN	Temporary failure in name resolution	EAI_OVERFLOW	User argument buffer overflowed (getnameinfo() only)
EAI_BADFLAGS	Invalid value for ai_flags	EAI_SERVICE	service not supported for ai_socktype
EAI_FAIL	Unrecoverable failure in name resolution	EAI_SOCKTYPE	ai_socktype not supported
EAI_FAMILY	ai_family not supported	EAI_SYSTEM	System error returned in errno
EAI_MEMORY	Memory allocation failure		
EAI_NONAME	hostname or service not provided, or not known		

freeaddrinfo Function



- All the storage returned by getaddrinfo is obtained dynamically. This storage is returned by calling freeaddrinfo.

```
#include <netdb.h>

void freeaddrinfo (struct addrinfo *ai);
```

- ai should point to the first *addrinfo* structure returned by *getaddrinfo*.
- All the structures in the linked list are freed.

/etc/hosts File

- A local file that maps hostnames to IP addresses, checked before DNS by the resolver.

127.0.0.1 localhost

192.168.1.10 myserver.local myserver

DNS Resolution

- If the name is not found in /etc/hosts, the system uses **DNS**, defined in /etc/resolv.conf.



Summary of Hostname and Network Name Lookups

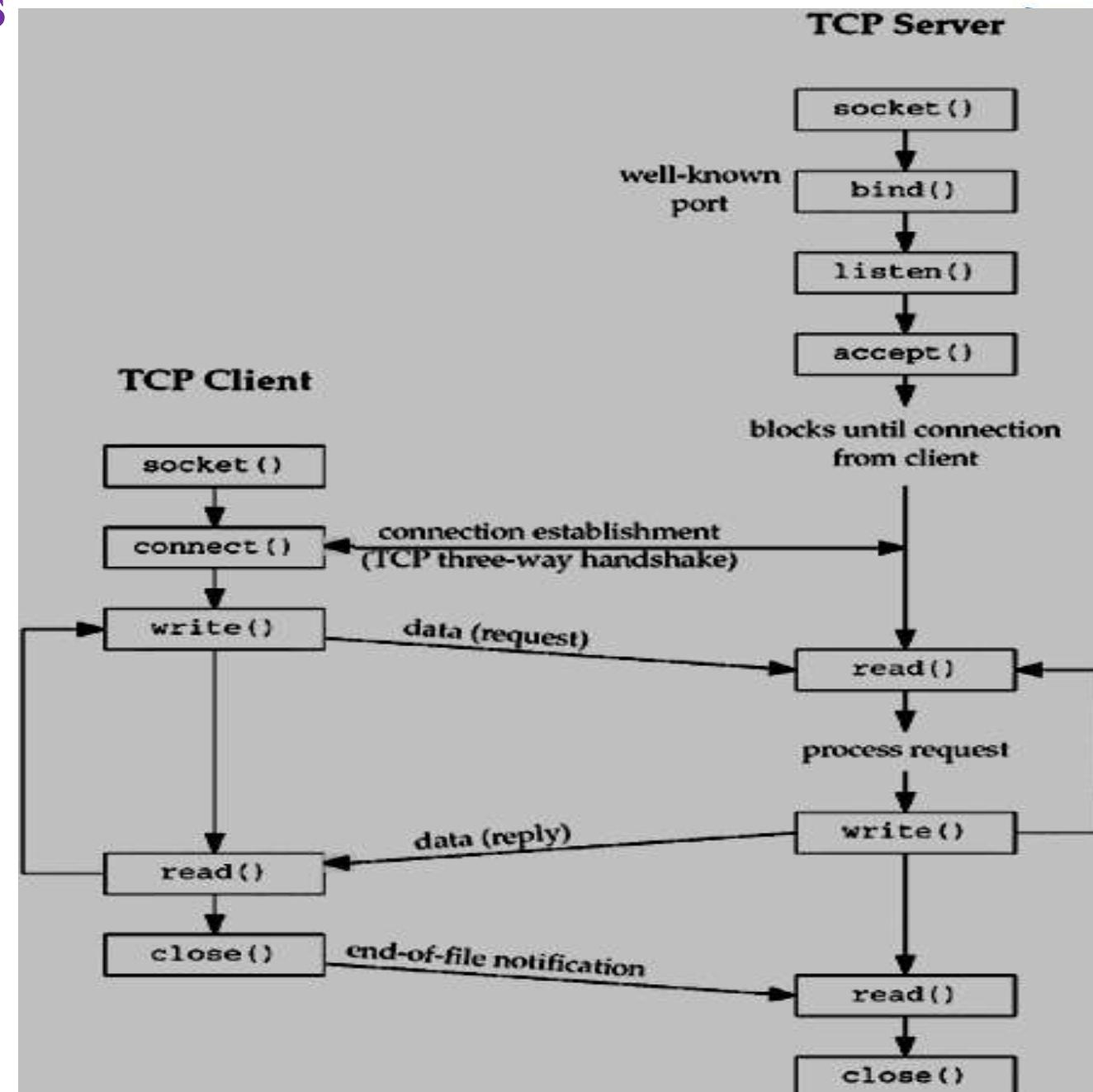
Function	Purpose
gethostname()	Hostname to IP (IPv4 only)
gethostbyaddr()	IP to Hostname
getservbyname()	Get service by matching protocol
getservbyport()	Get service given by its port number
getaddrinfo()	Hostname/service to sockaddr (IPv4/IPv6)
/etc/hosts	Local static name lookup
DNS	Dynamic network-wide name resolution



Basic Socket System Calls

1. Elementary TCP Sockets
2. Elementary UDP Sockets

Elementary TCP Sockets



socket()



- To perform network I/O, the first thing a process must do is call the **socket** function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

```
int socket(int family, int type, int protocol);  
/* Returns: non-negative descriptor if OK, -1 on error */
```

- family is one of
 - AF_INET** (IPv4), **AF_INET6** (IPv6), **AF_LOCAL** (local Unix),
 - AF_ROUTE** (access to routing tables), **AF_KEY** (new, for encryption)
- type is one of
 - SOCK_STREAM** (TCP), **SOCK_DGRAM** (UDP)
 - SOCK_RAW** (for special IP packets, PING, etc. Must be root)
 - SOCK_SEQPACKET** (Sequenced packet socket)
- Protocol is one of
 - IPPROTO_TCP**
 - IPPROTO_UDP**
 - IPPROTO_SCTP**
- protocol* is **0** (used for some raw socket options)

socket()



- Not all combinations of socket *family* and *type* are valid. The table below shows the valid combinations, along with the actual protocols that are valid for each pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP/SCTP	TCP/SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

- The key socket, AF_KEY, is newer than the others. It provides support for cryptographic security. Similar to the way that a routing socket (AF_ROUTE) is an interface to the kernel's routing table, the key socket is an interface into the kernel's key table.

bind()



- The **bind** function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
/* Returns: 0 if OK, -1 on error */
```

- *sockfd* is socket descriptor from `socket()`
- *myaddr* is a pointer to *address* `struct sockaddr` with:
 - *port number* and *IP address*
 - if port is 0, then host will pick ephemeral port
 - not usually for server (exception RPC port-map)
 - IP address != INADDR_ANY (unless multiple nics)
- *addrlen* is length of structure
- returns 0 if ok, -1 on error
 - **EADDRINUSE** ("Address already in use")



- Calling **bind** lets us specify the IP address, the port, both, or neither. The following table summarizes the values to which we set **sin_addr** and **sin_port**, or **sin6_addr** and **sin6_port**, depending on the desired result.

IP address	Port	Result
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

- If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called.
- If we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).



Wildcard Address and INADDR_ANY

- With IPv4, the *wildcard* address is specified by the constant **INADDR_ANY**, whose value is normally 0. This tells the kernel to choose the IP address.

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY); /* wildcard */
```

- While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure.

```
struct sockaddr_in6 serv;
serv.sin6_addr = in6addr_any; /* wildcard */
```

- The system allocates and initializes the **in6addr_any** variable to the constant **IN6ADDR_ANY_INIT**.
- The value of **INADDR_ANY (0)** is the same in either network or host byte order, so the use of **htonl** is not really required. But, since all the **INADDR_constants** defined by the [`<netinet/in.h>`](#) header are defined in host byte order, we should use **htonl** with any of these constants.

listen()



- The connect function is used by a TCP client to establish a connection with a TCP server

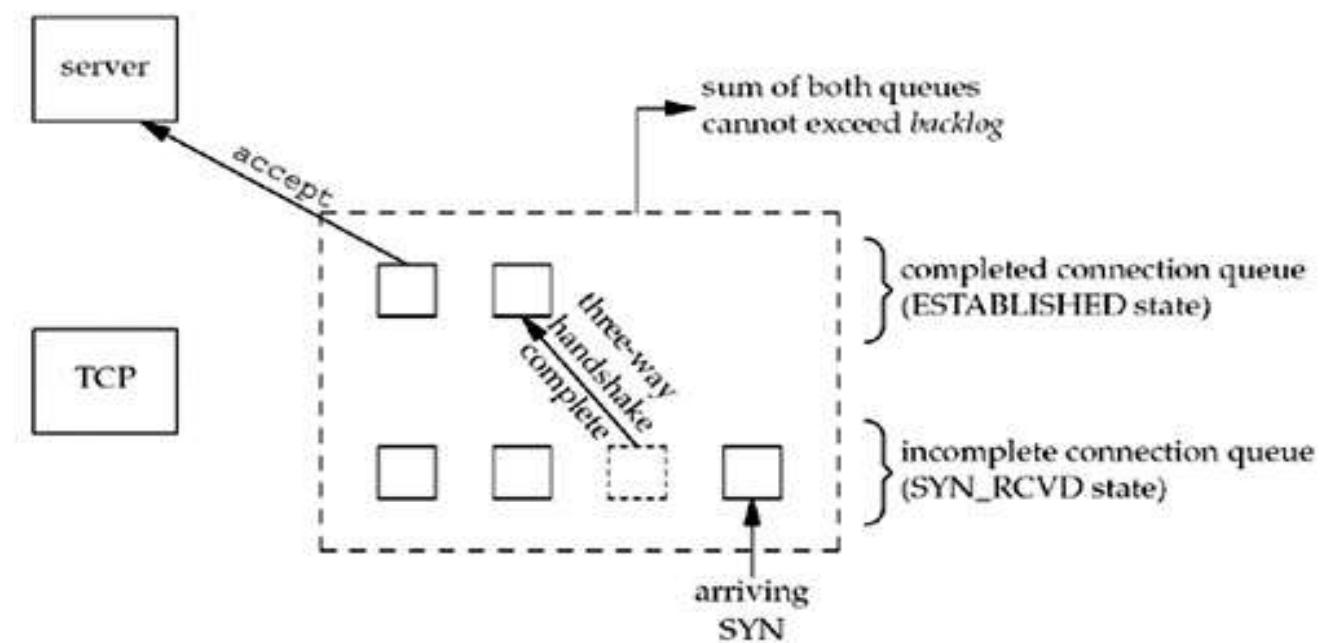
```
int listen(int sockfd, int backlog);
```

- Change socket state for TCP server.
- *sockfd* is socket descriptor from `socket()`
- *backlog* is maximum number of *incomplete* connections
 - historically 5
 - rarely above 15 on even moderate Web server!
- Sockets default to active (for a client)
 - change to passive so OS will accept connection
- An *incomplete connection queue*, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake.
- A *completed connection queue*, which contains an entry for each client with whom the TCP three-way handshake has completed.
- In terms of the TCP state transition diagram, the call to `listen` moves the socket from the CLOSED state to the LISTEN state.

Connection queues



- For ***backlog*** argument, we must realize that for a given listening socket, the kernel maintains two queues:
 - An **incomplete connection queue**, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.
 - A **completed connection queue**, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.



connect()



```
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);  
/* returns socket descriptor if ok, -1 on error */
```

- *sockfd* is socket descriptor from socket()
- *servaddr* is a pointer to a structure with:
 - *port number* and *IP address*
 - must be specified (unlike bind())
- *addrlen* is length of structure
- The client does not have to call bind before calling connect, the kernel will choose both an ephemeral port and the source IP address if necessary.
- **ETIMEDOUT**: host doesn't exist (connection timed out)
- **ECONNREFUSED**: no process is waiting for connections on the server host at the port specified
- **EHOSTUNREACH**: no route to host

connect()



- In the case of a TCP socket, the connect function initiates TCP's three-way handshake.
- The function returns only when the connection is established or an error occurs.

Possible Error:

1. If the client TCP receives no response to its SYN segment, **ETIMEDOUT** is returned.
 - connect moves from the **CLOSED** state (the state in which a socket begins when it is created by the socket function) to the **SYN_SENT** state, and then, on success, to the **ESTABLISHED** state.
 - If connect fails, the socket is no longer usable and must be closed. We cannot call connect again on the socket.
2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (the server process is probably not running). This is a **hard error** and the error **ECONNREFUSED** is returned to the client as soon as the RST is received. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are:
 - When a SYN arrives for a port that has no listening server.
 - When TCP wants to abort an existing connection.
 - When TCP receives a segment for a connection that does not exist.
3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a **soft error**. The client kernel saves the message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either **EHOSTUNREACH** or **ENETUNREACH**.



accept()

```
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Return next completed connection.

- *sockfd* is socket descriptor from socket()
 - *cliaddr* and *addrlen* return protocol address from client
 - returns brand new descriptor, created by the kernel. This new descriptor refers to the TCP connection with the client.
 - The **listening socket** is the first argument (*sockfd*) to accept (the descriptor created by socket and used as the first argument to both bind and listen).
 - The **connected socket** is the return value from accept the connected socket.
 - A given server normally creates only one listening socket, which then exists for the lifetime of the server.
 - The kernel creates one connected socket for each client connection that is
 - When the server is finished serving a given client, the connected socket is closed.
-
- **This function returns up to three values:**
 - An integer return code that is either a new socket descriptor or an error indication,
 - The protocol address of the client process (through the *cliaddr* pointer),
 - The size of this address (through the *addrlen* pointer).



Sending and Receiving

```
int recv(int sockfd, void *buff, size_t mbytes, int flags);  
int send(int sockfd, void *buff, size_t mbytes, int flags);
```

- Same as read() and write() but for *flags*
 - MSG_DONTWAIT (this send non-blocking)
 - MSG_OOB (out of band data, 1 byte sent ahead)
 - MSG_PEEK (look, but don't remove)
 - MSG_WAITALL (don't give me less than max)
 - MSG_DONTROUTE (bypass routing table)



close()

```
int close(int sockfd);
```

Close socket for use.

- *sockfd* is socket descriptor from socket()
- closes socket for reading/writing
 - returns (doesn't block)
 - attempts to send any unsent data
 - socket option **SOLINGER**
 - block until data sent
 - or discard any remaining data
 - *returns -1 if error*
 - The default action of close with a TCP socket is to mark the socket as closed and return to the process immediately.
 - The socket descriptor is no longer usable by the process. It cannot be used as an argument to read or write.
 - But ,TCP will try to send any data that is already queued to be sent to the other end, and after this occurs, the normal TCP connection termination sequence takes place.



getsockname and getpeername Functions

```
#include <sys/socket.h>
int getsockname (int sockfd, struct sockaddr* localaddr, socklen_t * addrlen)
Int getpeername (int sockfd, struct sockaddr* peeraddr, socklen_t * addrlen)
```

- **getsockname** returns local protocol address associated with a socket
- **getpeername** returns the foreign protocol address associated with a socket
- **getsockname** will return local IP/Port if unknown (TCP client calling connect without a bind, calling a bind with port 0, after accept to know the connection local IP address, but use connected socket)



getsockname and getpeername Functions

Why **getsockname()** and **getpeername()** is required?

- After connect successfully returns in a TCP client that does not call bind, **getsockname()** returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number), **getsockname** returns the local port number that was assigned.
- **getsockname** can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address, once a connection is established with a client, the server can call **getsockname** to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is executed by the process that calls accept, the only way the server can obtain the identity of the client is to call **getpeername**.



Example

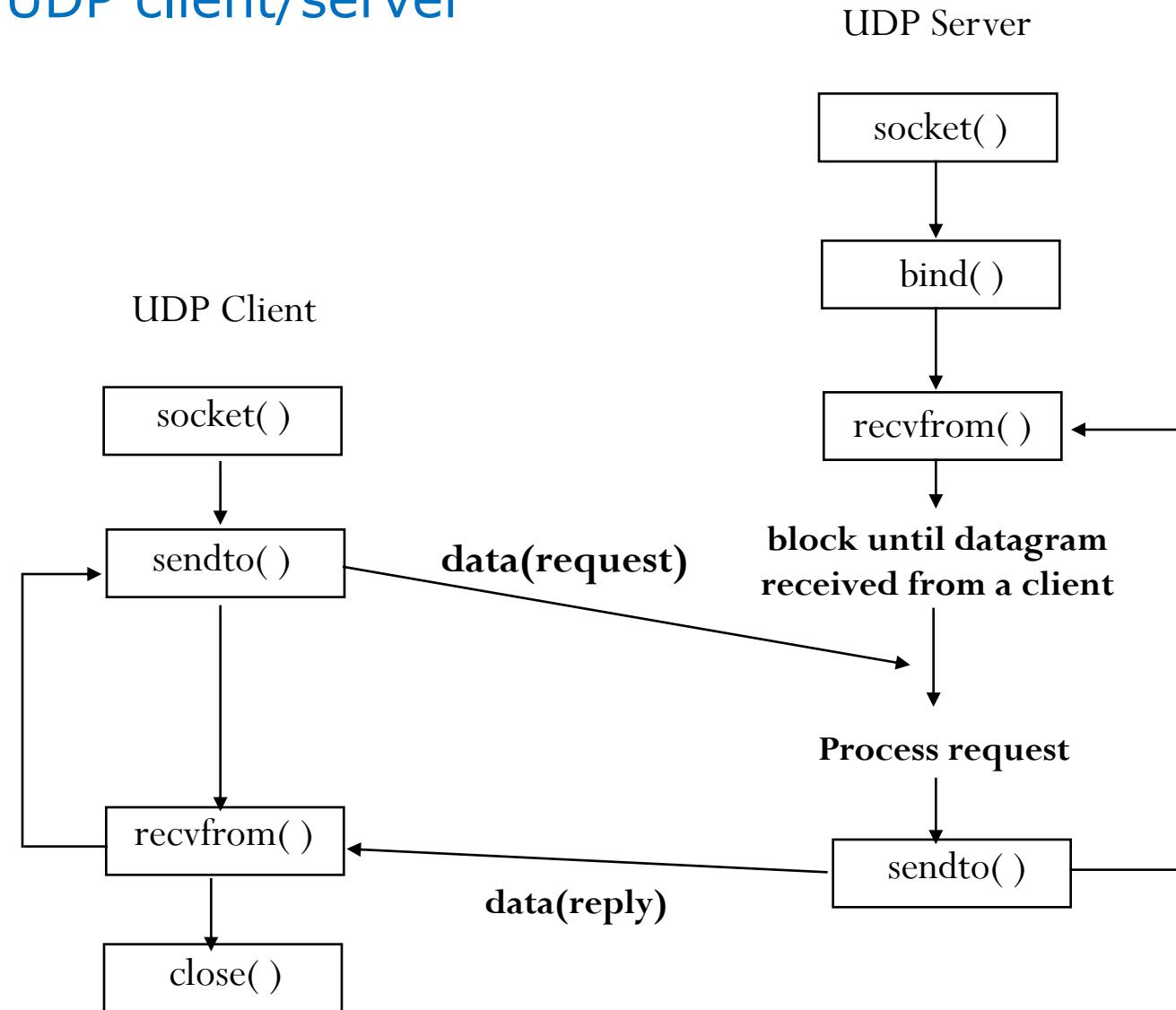
- ```
int sockfd_to_family(int sockfd) {
 struct sockaddr_storage ss;
 socklen_t len;
 len = sizeof(ss);
 if (getsockname(sockfd, (SA *) &ss, &len) < 0)
 return (-1);
 return (ss.ss_family);
}
```
- **Allocate room for largest socket address structure.** Since we do not know what type of socket address structure to allocate, we use a `sockaddr_storage` value, since it can hold any socket address structure supported by the system.
- **Call `getsockname`.** We call `getsockname` and return the address family. The POSIX specification allows a call to `getsockname` on an unbound socket.



## Example programs

- Daytime server and daytime client - DONE
- RWServer and RWClient – DONE

## Socket functions for UDP client/server





# Introduction

- UDP is transport layer protocol which is *is a **connectionless, unreliable,datagram protocol.***
- Some popular applications are built using **UDP: DNS, NFS, and SNMP** etc.
- The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the **sendto** function, which requires the address of the destination (the server) as a parameter.
- Similarly, the server does no accept a connection from a client. Instead, the server just calls the **recvfrom** function, which waits until data arrives from some client.
- **recvfrom** returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

# recvfrom and sendto Function



```
#include<sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
 struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
 const struct sockaddr *to, socklen_t addrlen);
/* Both return: number of bytes read or written if OK, -1 on error */
```

- The first three arguments, *sockfd*, *buff*, and *nbytes*, are identical to the first three arguments for **read** and **recv**: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.
- The ***to*** argument for **sendto** is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent.
- The final argument to **sendto** is an integer value, while the final argument to **recvfrom** is a pointer to an integer value (a value-result argument).
- The final two arguments to **recvfrom** are similar to the final two arguments to **accept**: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to **sendto** are similar to the final two arguments to **connect**: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).
- Both functions return the length of the data that was read or written as the value of the function. In the typical use of **recvfrom**, with a datagram protocol, the return value is the amount of user data in the datagram received.

## recvfrom and sendto Function



- Writing a datagram of length 0 is acceptable. In the case of UDP, this results in an IP datagram containing an IP header (*normally 20 bytes for IPv4 and 40 bytes for IPv6*), an 8-byte UDP header, and no data.
- This also means that a return value of 0 from **recvfrom** is acceptable for a datagram protocol: It does not mean that the peer has closed the connection, as does a return value of 0 from **read** on a TCP socket. Since UDP is connectionless, there is no such thing as closing a UDP connection.
- If the from argument to **recvfrom** is a null pointer, then the corresponding length argument (addrlen) must also be a null pointer, and this indicates that we are not interested in knowing the protocol address of who sent us data.

## Connect Function With UDP



- We can call connect for a UDP socket. The kernel just checks for any immediate errors (e.g. an obviously unreachable destination), records the IP address and port number of the peer, and returns immediately to the calling process. Obviously, there is no three-way handshake.
- With this capability, we must now distinguish between
  - An unconnected UDP socket, the default when we create a UDP socket
  - A connected UDP socket, the result of calling connect on a UDP socket
- With a connected UDP socket, three things change, compared to the default unconnected UDP socket:
  - We can no longer specify the destination IP address and port for an output operation. We do not use **sendto**, but **write** or **send** instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by connect.
  - Similar to TCP, we can call **sendto** for a connected UDP socket, but we cannot specify a destination address. The fifth argument to **sendto** (the pointer to the socket address structure) must be a null pointer, and the sixth argument (the size of the socket address structure) should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.

## Connect Function with UDP...



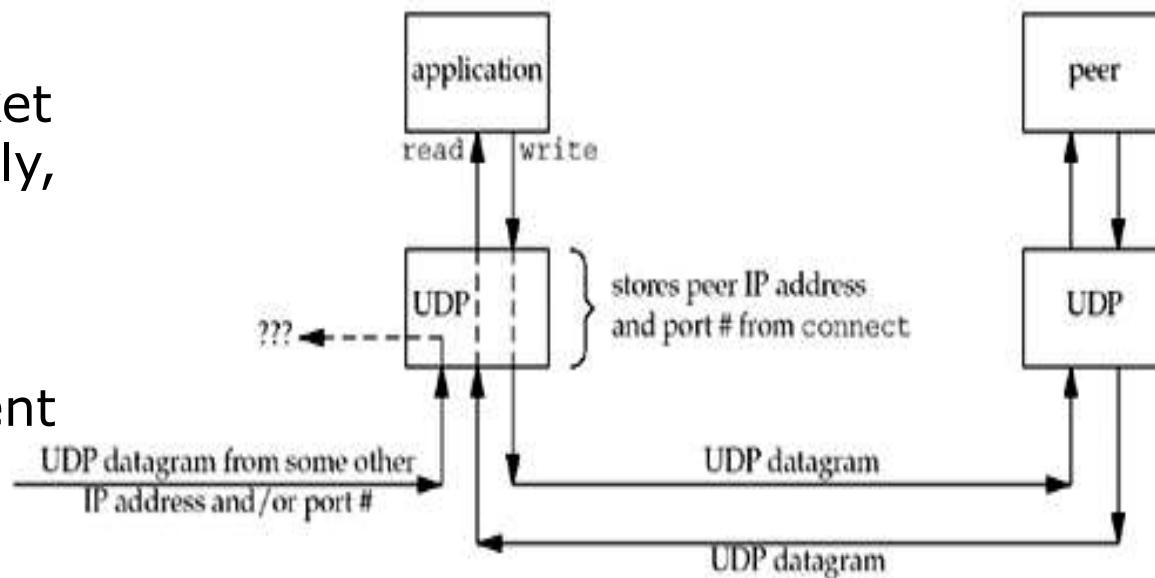
- We do not need to use **recvfrom** to learn the sender of a datagram, but **read**, **recv**, or **recvmsg** instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in connect. Datagrams destined to the connected UDP socket's local protocol address (e.g., IP address and port) but arriving from a protocol address other than the one to which the socket was connected are not passed to the connected socket.
  - Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to connect to a multicast or broadcast address.
  - Asynchronous errors are returned to the process for connected UDP sockets.

| Type of Socket                  | <b>write()</b> or <b>send()</b>                    | <b>sendto()</b> without destination                | <b>sendto()</b> with destination   |
|---------------------------------|----------------------------------------------------|----------------------------------------------------|------------------------------------|
| <b>TCP Socket</b>               | OK                                                 | OK                                                 | EISCONN (Error: Already connected) |
| <b>UDP Socket (Connected)</b>   | OK                                                 | OK                                                 | EISCONN (Error: Already connected) |
| <b>UDP Socket (Unconnected)</b> | EDESTADDRREQ (Error: Destination address required) | EDESTADDRREQ (Error: Destination address required) | OK                                 |

## Connect Function With UDP...



- The application calls connect, specifying the IP address and port number of its peer. It then uses read and write to exchange data with the peer.
- Datagrams arriving from any other IP address or port (??? in Figure) are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is connected. These datagrams could be delivered to some other UDP socket on the host. If there is no other matching socket for the arriving datagram, UDP will discard it and generate an ICMP "port unreachable" error.
- In summary, UDP client or server can call connect only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls connect, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP); in this case, both the client and server can call connect.





## Example programs

- UDP Client Server Program - DONE



# UNIX Domain Socket

## Unix Domain Socket



- The Unix domain protocols are not an actual protocol suite, but a way of performing client/server communication on a single host using the same API that is used for clients and servers on different hosts.
- Two types of sockets are provided in the Unix domain: **stream sockets** (similar to TCP) and **datagram sockets**(similar to UDP).

### **Unix domain sockets are used for three reasons:**

- On Berkeley-derived implementations, Unix domain sockets are often twice as fast as a TCP socket when both peers are on the same host.
- Unix domain sockets are used when passing descriptors between processes on the same host.
- Newer implementations of Unix domain sockets provide the client's credentials to the server, which can provide additional security checking.

## Unix Domain Socket Address Structure



The Unix domain socket address structure, which is defined by including the `<sys/un.h>` header, is:

```
struct sockaddr_un {
 sa_family_t sun_family; /* AF_LOCAL */
 char sun_path[104]; /* null-terminated pathname */
};
```

- The pathname stored in the `sun_path` array must be null-terminated. The macro `SUN_LEN` is provided and it takes a pointer to a `sockaddr_un` structure and returns the length of the structure, including the number of non-null bytes in the pathname.
- The unspecified address is indicated by a null string as the pathname, that is, a structure with `sun_path[0]` equal to 0. This is the Unix domain equivalent of the IPv4 `INADDR_ANY` constant and the IPv6 `IN6ADDR_ANY_INIT` constant.

## Example: bind of Unix Domain Socket



```
int main(int argc, char ** argv[]) {
 int sockfd;
 socklen_t len;
 struct sockaddr_un addr1, addr2;
 if (argc != 2)
 err_quit("usage: unixbind <pathname>");
 sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
 unlink(argv[1]); // OK if this fails
 bzero(&addr, sizeof(addr1));
 addr1.sun_family = AF_LOCAL;
 strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) - 1);
 bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));
 len = sizeof(addr2);
 getsockname(sockfd, (SA *) &addr2, &len);
 printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);
 exit(0);
}
```



## socketpair() Function

- The **socketpair()** function creates two sockets that are connected together. This function applies only to Unix domain sockets.

```
#include <sys/socket.h>

int socketpair(int family, int type, int protocol, int sockfd[2]);

/* Returns: nonzero if OK, -1 on error */
```

- Family: **AF\_LOCAL** or **AF\_UNIX** and the protocol must be 0.
- Type: either **SOCK\_STREAM** or **SOCK\_DGRAM**. The two socket descriptors that are created are returned as **sockfd[0]** and **sockfd[1]**
- The two created sockets are unnamed; that is; there is no implicit bind involved.
- The result of **socketpair()** with a type of **SOCK\_STREAM** is called a stream pipe. The stream pipe is full-duplex; that is, both descriptors can be read and written.

## Socket function



### Differences and restrictions in the socket functions when using Unix domain sockets.

- The default file access permissions for a pathname created by bind should be 0777 (read, write, and execute by user, group, and other), modified by the current umask value.
- The pathname associated with a Unix domain socket should be an absolute pathname, not a relative pathname.
- The pathname specified in a call to connect must be a pathname that is currently bound to an open Unix domain socket of the same type (stream or datagram).
- Unix domain stream sockets are similar to TCP sockets. They provide a byte stream interface to the process with no record boundaries.
- If a call to connect for a Unix domain stream socket finds that the listening socket's queue is full, ECONNREFUSED is returned immediately.
- Unix domain datagram sockets are similar to UDP sockets. They provide an unreliable datagram service that preserves record boundaries.
- Sending a datagram on an unbound Unix domain datagram socket does not bind a pathname to the socket. Calling connect for a Unix domain datagram socket does not bind a pathname to the socket.

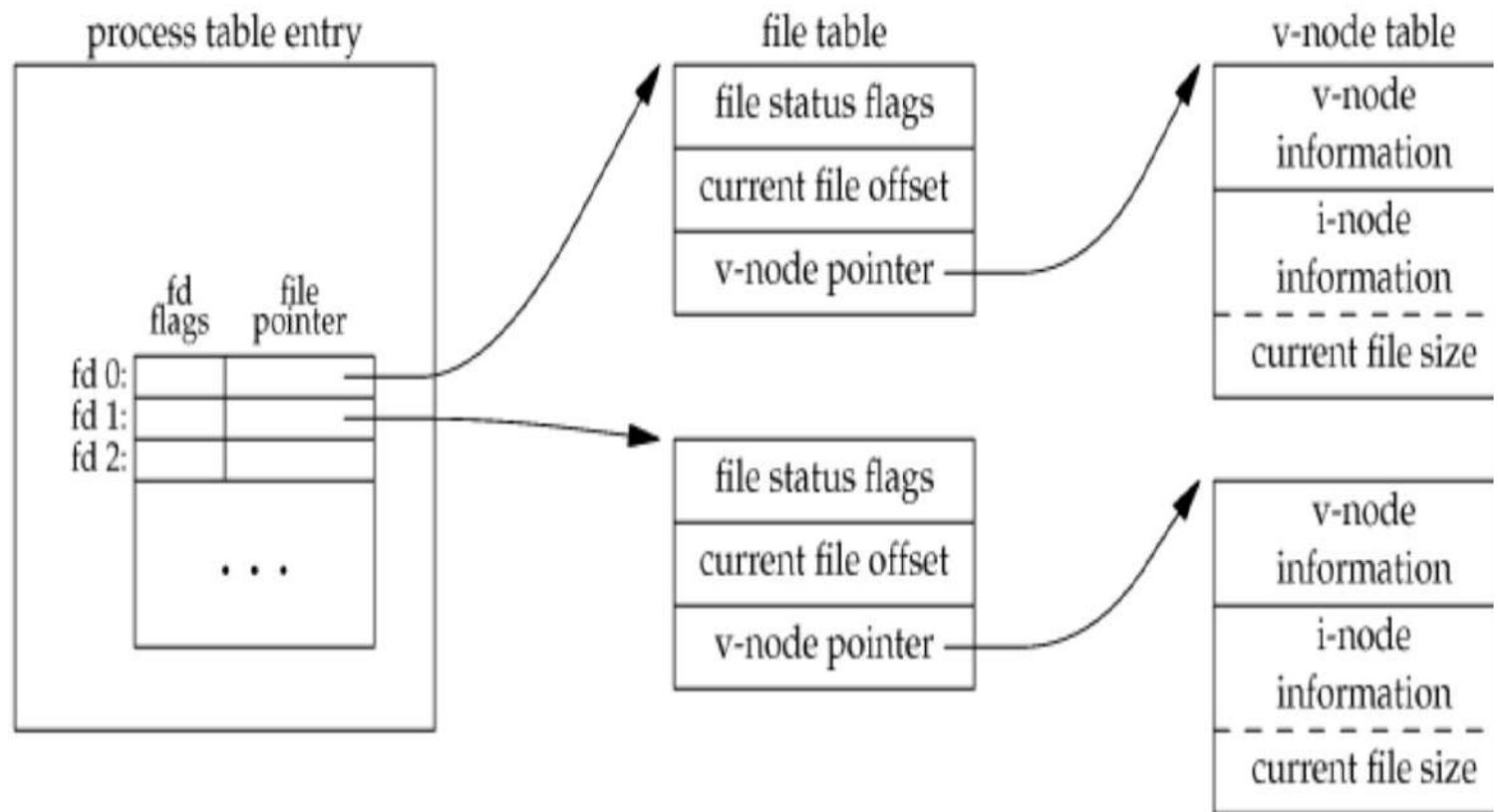


# Passing (File) Descriptors

# Open File in Unix system



- The kernel uses three data structures to represent an open file.
  1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
    - The file descriptor flags
    - A pointer to a file table entry
  2. The kernel maintains a file table for all open files. Each file table entry contains
    - The file status flags for the file, such as read, write, append, sync, and nonblocking;
    - The current file offset
    - A pointer to the v-node table entry for the file
  3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, etc.



The figure shows a pictorial arrangement of these three tables for a single process that has two different files open.

# Passing a file descriptor



- Passing an open descriptor from one process to another takes place as
  - A child sharing all the open descriptors with the parent after a call to fork
  - All descriptors normally remaining open when exec is called.
- Current Unix systems provide a way to pass any open descriptor from one process to any other process.
- The technique requires us to first establish a Unix domain socket between the two processes and then use **sendmsg** to send a special message across the Unix domain socket.
- This message is handled specially by the kernel, passing the open descriptor from the sender to the receiver.

## Steps involved in passing a descriptor between two processes

- Create a Unix domain socket, either a stream socket or a datagram socket.
  - If the goal is to fork a child and have the child open the descriptor and pass the descriptor back to the parent, the parent can call **socketpair** to create a stream pipe that can be used to exchange the descriptor.
  - If the processes are unrelated, the server must create a Unix domain stream socket and bind a pathname to it, allowing the client to connect to that socket.



- **Steps involved in passing a descriptor between two processes(contd...)**
- One process opens a descriptor by calling any of Unix functions that returns a descriptor. Any type of descriptor can be passed from one process to another.
- The sending process builds a **msghdr** structure containing the descriptor to be passed. The sending process calls **sendmsg** to send the descriptor across the Unix domain socket.
  - At this point, the descriptor is “in flight”. Even if the sending process closes the descriptor after calling **sendmsg**, but before the receiving process calls **recvmsg**, the descriptor remains open for the receiving process. Sending a descriptor increments the descriptor’s reference count by one.
- The receiving process calls **recvmsg** to receive the descriptor on the Unix domain socket. It is normal for the descriptor number in the receiving process to differ from the descriptor number in the sending process. Passing a descriptor involves creating a new descriptor in the receiving process that refers to the same file table entry within the kernel as the descriptor that was sent by the sending process.
-



# Signal Handling in Unix

# (Posix) Signal Handling



- A signal is a notification to a process that an event has occurred. Signals are sometimes called software interrupts. Signals usually occur asynchronously.
- Signals can be sent
  - By one process to another process (or to itself)
  - By the kernel to a process
- The **SIGCHLD** signal is one that is sent by the kernel whenever a process terminates, to the parent of the terminating process.
- Every signal has a disposition, which is also called the action associated with the signal. We set the disposition of a signal by calling the **sigaction** function. We have three choices for the disposition.
  1. We can provide a function that is called whenever a specific signal occurs. This function is called a signal handler and the action is called catching a signal. The two signals **SIGKILL** & **SIGSTOP** cannot be caught.
  2. We can ignore a signal by setting its disposition to **SIG\_IGN**. The two signals **SIGKILL** and **SIGSTOP** cannot be ignored.
  3. We can set the default disposition for a signal by setting its disposition to **SIG\_DFL**. There are few signals whose default disposition is to be ignored. **SIGCHLD** and **SIGURG**.



## How to catch a signal?

- use the `signal()` or `sigaction()` system calls.
- Simple example with `signal()`:

```
#include <stdio.h>
#include <signal.h>

void handler(int signum) {
 printf("Caught signal %d\n", signum);
}

int main() {
 signal(SIGINT, handler); // Register handler for Ctrl+C
 while (1) {
 printf("Running...\n");
 sleep(1);
 }
 return 0;
}
```

- When we press **Ctrl+C**, instead of killing the program immediately, it **prints a message**.
- `signal(SIGINT, handler)` means: "When SIGINT arrives, call handler."



## sigaction() — the modern and safer way

- `signal()` is considered **old and unreliable** sometimes because of race conditions.  
**sigaction()** provides more control:

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
void handler(int signum) {
 printf("Caught signal %d\n", signum);
}
int main() {
 struct sigaction sa;
 memset(&sa, 0, sizeof(sa));
 sa.sa_handler = handler;
 sigaction(SIGINT, &sa, NULL);
 while (1) {
 printf("Running safely...\n");
 sleep(1);
 }
 return 0;
}
```

- We can set flags like `SA_RESTART` (restart interrupted system calls) or `SA_SIGINFO` (for getting detailed info).

### Sending Signals

- We can send signals using:
  - `kill(pid, signal)` — send a signal to a process.
  - `raise(signal)` — send a signal to **own**.
  - `raise(SIGTERM);` // Current process sends itself SIGTERM
  - From Terminal :`kill -SIGINT 1234` (1234 is pid)

#### Notes:

**SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Signals can **interrupt** `read()`, `write()`, `sleep()`, etc. unless `SA_RESTART` is used.

Use **signal masks** (`sigprocmask()`) to block/unblock signals temporarily.



## sigaction function

**sigaction** returns the old action for the signal as the return value of the **signal** function.

```
#include<signal.h>
int sigaction(int sig, const struct sigaction *restrict act, struct sigaction
*restrict oact);
```

- The **sig** is the signal to be captured. The **act** is the information about signal handling function, masked signal and flags. The **oact** is the information about the previous signal handling function, masked signal and flags.
- The **sa\_mask** in **struct sigaction** takes the signal to be masked when the handler function is called on arrival of the signal. The **sa\_flags** in **struct sigaction** sets flags. E.g., Setting **sa\_flags** to **SA\_SIGINFO** uses **POSIX** signal handler function.



| <b>Signal Name</b> | <b>Default Action</b> | <b>Description</b>                                                                                     |
|--------------------|-----------------------|--------------------------------------------------------------------------------------------------------|
| SIGHUP             | Terminate             | Hangup detected on controlling terminal or death of controlling process. Often used to reload configs. |
| SIGINT             | Terminate             | Interrupt from keyboard (Ctrl+C).                                                                      |
| SIGQUIT            | Core Dump             | Quit from keyboard (Ctrl+), generates core dump.                                                       |
| SIGILL             | Core Dump             | Illegal instruction (usually due to corrupted program image or stack).                                 |
| SIGABRT            | Core Dump             | Abort signal from abort() function.                                                                    |
| SIGFPE             | Core Dump             | Floating-point exception (e.g., divide by zero).                                                       |
| SIGKILL            | Terminate             | Kill signal. Cannot be caught, blocked, or ignored.                                                    |
| SIGSEGV            | Core Dump             | Segmentation fault (invalid memory reference).                                                         |
| SIGPIPE            | Terminate             | Broken pipe: write to pipe with no readers.                                                            |
| SIGALRM            | Terminate             | Timer signal from alarm() function.                                                                    |
| SIGTERM            | Terminate             | Termination signal (used for graceful termination).                                                    |
| SIGUSR1            | Terminate             | User-defined signal 1.                                                                                 |
| SIGUSR2            | Terminate             | User-defined signal 2.                                                                                 |
| SIGCHLD            | Ignore                | Sent to parent when child stops or terminates.                                                         |
| SIGCONT            | Continue              | Continue if stopped.                                                                                   |
| SIGSTOP            | Stop                  | Stop the process. Cannot be caught or ignored.                                                         |

| <b>Signal Name</b> | <b>Default Action</b> | <b>Description</b>                                   |
|--------------------|-----------------------|------------------------------------------------------|
| SIGTSTP            | Stop                  | Stop typed at terminal (Ctrl+Z).                     |
| SIGTTIN            | Stop                  | Background process attempting read from terminal.    |
| SIGTTOU            | Stop                  | Background process attempting write to terminal.     |
| SIGBUS             | Core Dump             | Bus error (e.g., misaligned memory access).          |
| SIGPOLL            | Terminate             | Pollable event (Sys V). Synonym for SIGIO.           |
| SIGPROF            | Terminate             | Profiling timer expired.                             |
| SIGSYS             | Core Dump             | Bad argument to a system call.                       |
| SIGTRAP            | Core Dump             | Trace/breakpoint trap.                               |
| SIGURG             | Ignore                | Urgent condition on socket (e.g., out-of-band data). |
| SIGVTALRM          | Terminate             | Virtual alarm clock (process time).                  |
| SIGXCPU            | Core Dump             | CPU time limit exceeded.                             |
| SIGXFSZ            | Core Dump             | File size limit exceeded.                            |
| SIGIO              | Terminate             | I/O now possible (asynchronous I/O).                 |
| SIGWINCH           | Ignore                | Window size change (used in terminal apps).          |



## POSIX Signal Semantics

- Once a signal handler is installed, it remains installed.
- While a signal handler is executing, the signal being delivered is blocked. Furthermore, any additional signals that were specified in the **sa\_mask** signal set passed to **sigaction** when the handler was installed are also blocked.
- If a signal is generated one or more times while it is blocked, it is normally delivered only one time after the signal is unblocked. That is, by default, Unix signals are not queued.
- It is possible to selectively block and unblock a set of signals using the **sigprocmask** function.



# Daemon Processes



## Daemon Process

- A daemon is a process that runs in the background and is not associated with a controlling terminal. Unix systems typically have many processes that are daemons, running in the background, performing different administrative tasks.
  
- The lack of a controlling terminal is typically a side effect of being started by a system initialization script. But if a daemon is started by a user typing to a shell prompt, it is important for the daemon to disassociate itself from the controlling terminal to avoid any unwanted interaction with job control, terminal session management, or simply to avoid unexpected output to the terminal from the daemon as it runs in the background.

# Characteristics of daemon processes in Unix



| Characteristic                             | Description                                                                                                                |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <b>Runs in background</b>                  | Daemons do not interact with users directly and run as background services.                                                |
| <b>No controlling terminal</b>             | A daemon is detached from the terminal that launched it. It won't be affected by terminal I/O or hangups.                  |
| <b>Independent of user session</b>         | It usually survives beyond the life of a user login session.                                                               |
| <b>Starts at boot</b>                      | Many daemons are started during system boot and run until shutdown (e.g., sshd, cron).                                     |
| <b>Parent process is init or systemd</b>   | After forking, a daemon often calls setsid() and its parent exits, so the daemon is reparented to PID 1 (init or systemd). |
| <b>Uses umask(0)</b>                       | To avoid file permission issues, a daemon resets the file mode creation mask.                                              |
| <b>Changes working directory</b>           | Typically changes to the root directory (/) to avoid locking mount points.                                                 |
| <b>Closes file descriptors</b>             | Standard input/output/error (0, 1, 2) are closed or redirected to avoid terminal I/O.                                      |
| <b>Logs activity</b>                       | Daemons log errors or events using syslog or log files, instead of writing to the terminal.                                |
| <b>Implements proper signal handling</b>   | Daemons handle signals like SIGHUP, SIGTERM, or SIGCHLD for graceful behavior.                                             |
| <b>Often runs with elevated privileges</b> | Some daemons start as root but drop privileges for security.                                                               |

## How to start daemon process?



- During system startup, many daemons are started by the system initialization scripts. Daemons started by these scripts begin with superuser privileges.
- Many network servers are started by the **inetd superserver**. The **inetd** itself is started from one of the scripts in Step 1. The inetd listens for network requests, and when a request arrives, it invokes the actual server.
- The execution of programs on a regular basis is performed by the **cron** daemon, and programs that it invokes run as daemons. The cron daemon itself is started in Step 1 during system startup.
- The execution of a program at one time in the future is specified by the "**at**" command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.
- Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason.
- Since a daemon does not have a controlling terminal, it needs some way to output messages when something happens, either normal informational messages or emergency messages that need to be handled by an administrator.

# How to create a daemon in Unix ( how to daemonize a process)



## 1. fork

We first call fork and then the parent terminates, and the child continues. If the process was started as a shell command in the foreground, when the parent terminates the shell think the command is done. This automatically runs the child process in the background.

```
pid_t pid = fork();
if (pid < 0) exit(EXIT_FAILURE);
if (pid > 0) exit(EXIT_SUCCESS); // Parent exits
```

## 2. setsid

**setsid** is a POSIX function that creates a new session. The process becomes the session leader of the new session, becomes the process group leader of a new process group, and has no controlling terminal.

```
if (setsid() < 0) exit(EXIT_FAILURE);
```

## 3. Ignore SIGHUP and fork again

We ignore SIGUP and call fork again. When this function returns, the parent is really the first child and it terminates, leaving the second child running. The purpose of this second fork is to guarantee that the daemon cannot automatically acquire a controlling terminal should it open a terminal device in the future. We must ignore SIGHUP because when the session leader terminates (the first child), all processes in the session (our second child) receive the SIGHUP signal.

## 4. Change working directory

We change the working directory to the root directory. The file system cannot be un-mounted if working directory is not changed.

```
chdir("/");
```



## 5. Close any open descriptors

We open any open descriptors that are inherited from the process that executed the daemon (normally a shell).

```
for (int x = sysconf(_SC_OPEN_MAX); x >= 0; x--) {
 close(x);
}
```

## 6. Redirect stdin, stdout, and stderr to /dev/null

We open /dev/null for standard input, standard output, and standard error. This guarantees that these common descriptors are open, and a read from any of these descriptors returns 0 (EOF), and the kernel just discards anything written to them.

## 7. Use syslogd for errors

The syslogd daemon is used to log errors.

```
FILE *log = fopen("/tmp/mydaemon.log", "a+");
fprintf(log, "Daemon started...\n");
fflush(log);
```



## Demo Programs

- daemon\_process.c
- daemon\_mac.c

# Integrate daemon program with systemd



- After integrating daemon program with *systemd* it can run as a managed service on modern Linux systems (like Ubuntu, Debian, CentOS, etc.).

## 1. Step 1: Install the Daemon Binary

```
gcc -o mydaemon mydaemon.c
```

**Copy it to a system location:** sudo cp mydaemon /usr/local/bin/

## 2. Step 2: Create a systemd Service File: sudo nano /etc/systemd/system/mydaemon.service

Paste the following:

```
[Unit]
Description=My Custom Daemon
After=network.target
[Service]
Type=simple
ExecStart=/usr/local/bin/mydaemon
Restart=on-failure
RestartSec=5
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=mydaemon
User=nobody
Group=nogroup
[Install]
WantedBy=multi-user.target
```

### Note:

**Type=simple** (daemon runs in the foreground).

**StandardOutput=syslog** (systemd captures logs and can see them with journalctl).

**User=nobody** (Run as a non-privileged user)

*Don't double-fork inside the code when using Type=simple.*

# Integrate daemon program with systemd



## 3. Step 3: Reload systemd and Start the Daemon

- sudo systemctl daemon-reexec
- sudo systemctl daemon-reload
- sudo systemctl enable mydaemon.service
- sudo systemctl start mydaemon.service

## 4. Step 4: Check Status and Logs

- Check if the service is running: `systemctl status mydaemon.service`
- View logs: `journalctl -u mydaemon.service`



# End of Chapter 2



# Network Programming

**BESE-VI – Pokhara University**

**Prepared by:**

**Assoc. Prof. Madan Kadariya (NCIT)**

**Contact: [madan.kadariya@ncit.edu.np](mailto:madan.kadariya@ncit.edu.np)**



# Chapter 3:

## Advance Unix Network Programming

### (12 hrs)

# Outline



1. I/O Models in Unix
  - i. Blocking I/O
  - ii. Non-blocking I/O
  - iii. I/O multiplexing (select(), pselect() poll())
  - iv. Signal-driven I/O
  - v. Asynchronous I/O
2. Concurrent Server Design
  - i. Overview of process and threads
  - ii. Fork() and exec() function
  - iii. Using fork() to handle multiple clients
  - iv. Using select() to handle multiple socket descriptors
  - v. Multithreading model using pthreads
3. Implementing broadcast and multicast communication
4. Socket Options
  - i. Using setsockopt(), getsockopt(), fcntl() and ioctl() to modify socket behavior
  - ii. Common options: SO\_REUSEADDR, SO\_BROADCAST, SO\_KEEPALIVE, SO\_LINGER etc.
5. Logging in Unix
  - i. Introduction to Syslog
  - ii. Logging messages from network applications
  - iii. Configuring and using syslog(), openlog(), and closelog()
6. Socket operations
7. Introduction to P2P programming
8. P2P Socket fundamentals
9. Overview Network Security Programming
  - i. Defining Security
  - ii. Challenges of Security
  - iii. Securing by Hostname or Domain Name
  - iv. Identification by IP Number
  - v. Wrapper program to implement simple security policy



# IO Models in Unix

# IO Models in Unix



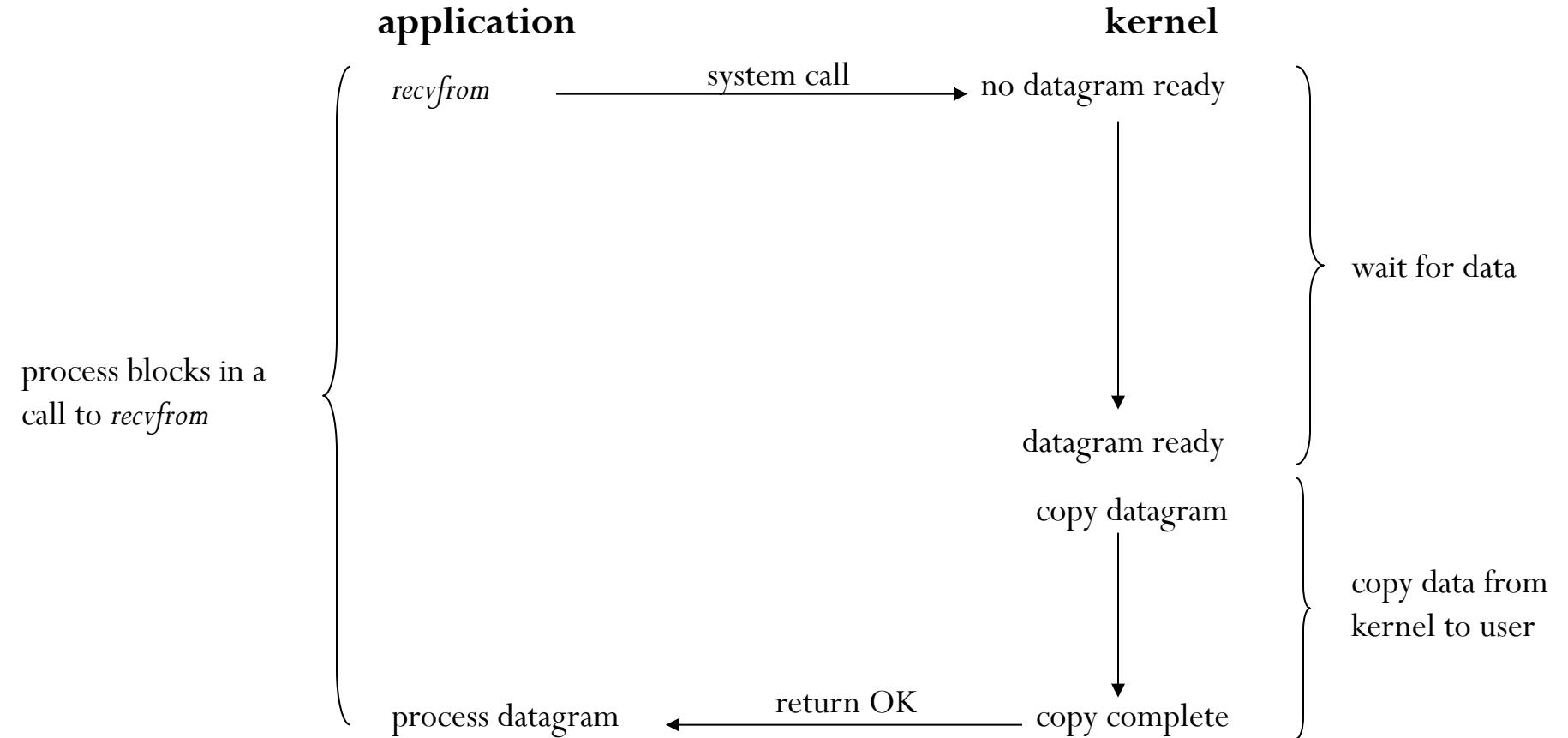
- TCP echo client is handling two inputs at the same time: standard input and a TCP socket
  - when the client was blocked in a call to read, the server process was killed
  - server TCP sends FIN to the client TCP, but the client never sees FIN since the client is blocked reading from standard input
    - We need the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready.
    - I/O multiplexing (**select**, **poll**, or newer **pselect** functions)
- Scenarios for I/O Multiplexing
  - client is handling multiple descriptors (interactive input and a network socket).
  - Client to handle multiple sockets (rare)
  - TCP server handles both a listening socket and its connected socket.
  - Server handle both TCP and UDP.
  - Server handles multiple services and multiple protocols



## Models

1. Blocking I/O
  2. Nonblocking I/O
  3. I/O multiplexing(**select** and **poll**)
  4. Signal driven I/O (**SIGIO**)
  5. Asynchronous I/O
- Two *distinct phases* for an input operation
    - Waiting for the data to be ready (for a socket, wait for the data to arrive on the network, then copy into a buffer within the kernel)
    - Copying the data from the kernel to the process (from kernel buffer into application buffer)

# Blocking I/O Model

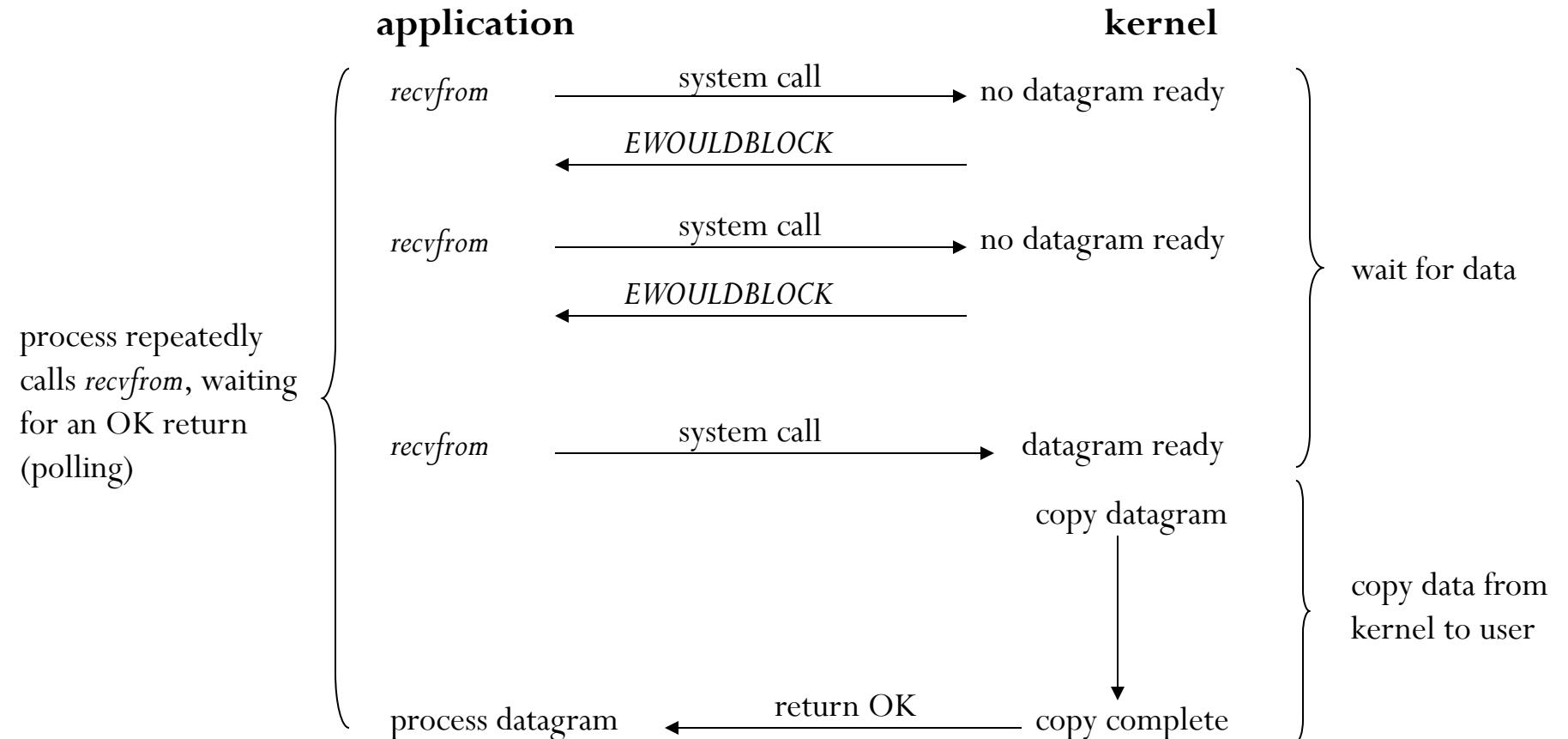




## Blocking I/O model...

- ❖ By default, all sockets are blocking.
- ❖ The process calls **recvfrom** and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.
- ❖ We say that our process is blocked the entire time from when it calls **recvfrom** until it returns.
- ❖ When **recvfrom** returns successfully, our application process the datagram.

# Non-blocking I/O Model

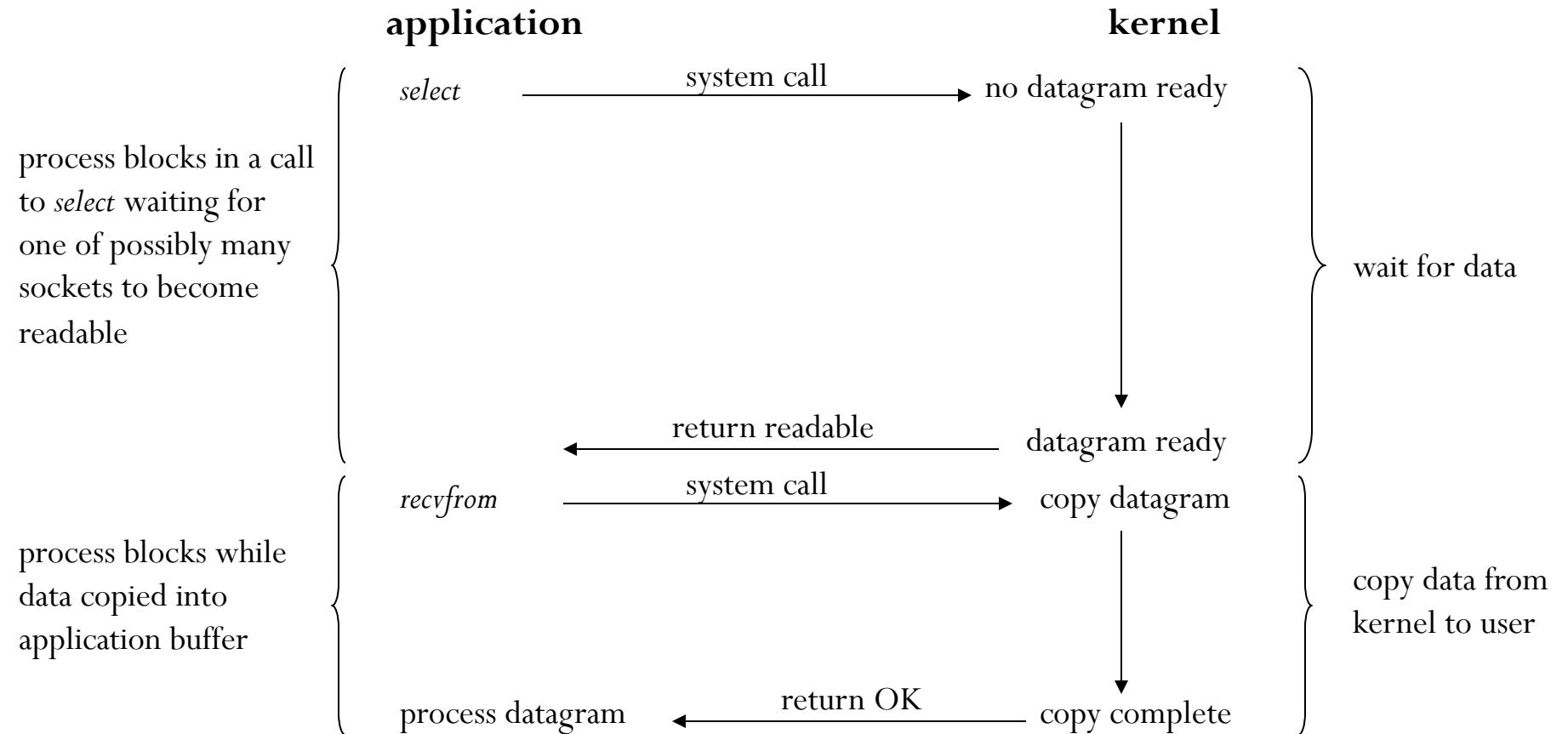




## Non-blocking I/O Model...

- ❖ When a socket is non-blocking, It instruct the kernel as “when an I/O operation that the process requests cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead.”
- ❖ The first two times(as in fig) that we call **recvfrom**, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead.
- ❖ The third time we call **recvfrom**, a datagram is ready, it is copied into our application buffer, and **recvfrom** returns successfully.
- ❖ We then process data. When an application sits in a loop calling **recvfrom** on a non-blocking descriptor like this, it is called polling.
- ❖ The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time.

# I/O Multiplexing Model

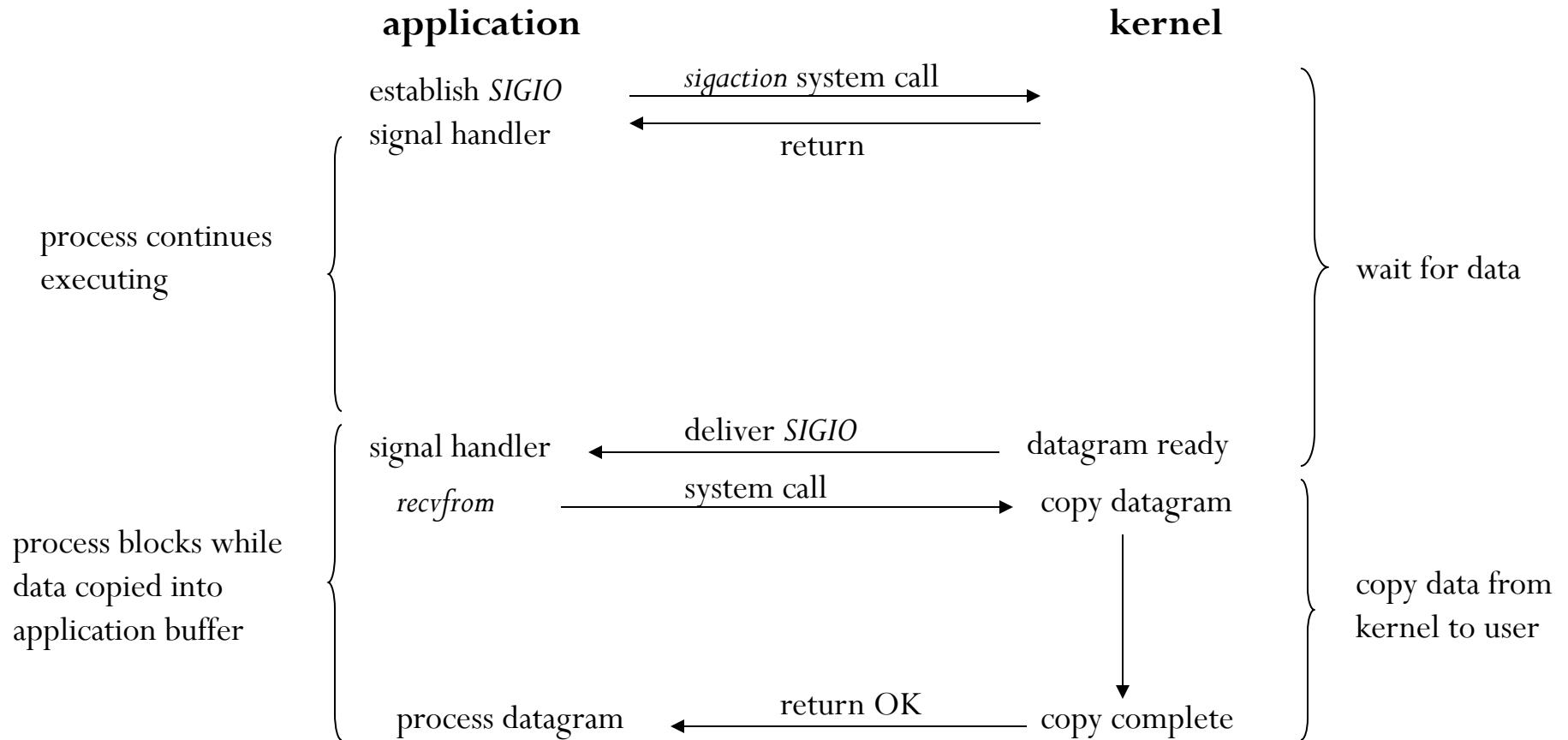




## I/O Multiplexing Model...

- ❖ With I/O multiplexing, we call **select** or **poll** and block in one of these two system calls, instead of blocking in the actual I/O system call.
- ❖ We block in a call to **select**, waiting for the datagram socket to be readable.
- ❖ When **select** returns that the socket is readable, we then call **recvfrom** to copy the datagram into our application buffer.
- ❖ With **select**, we can wait for more than one descriptor to be ready.

# Signal Driven I/O Model

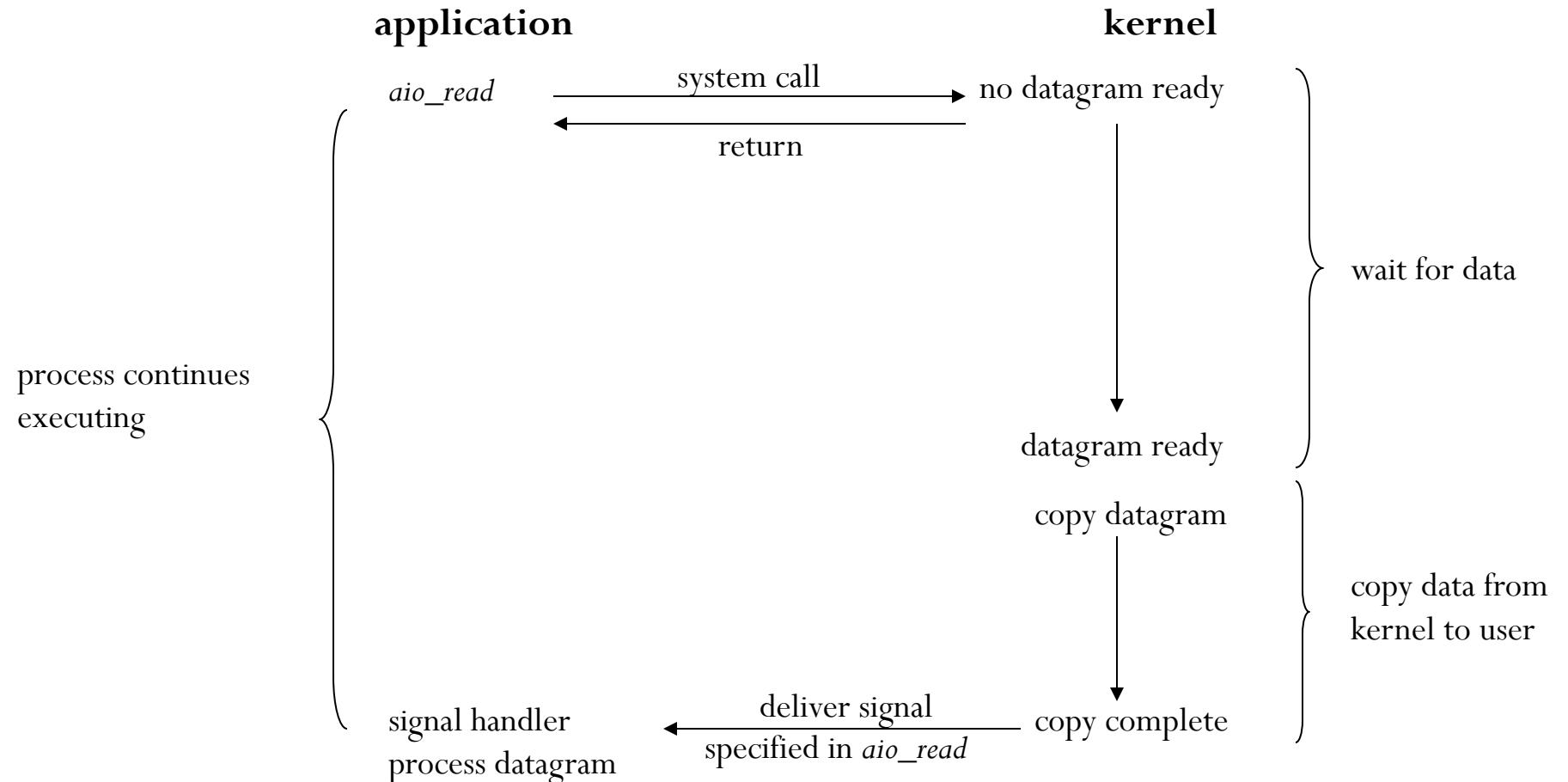


# Signal Driven I/O Model...



- ❖ A **SIGIO signal** is used to tell the kernel when the descriptor is ready. We call this signal-driven I/O.
- ❖ We first enable the socket for the signal-driven I/O and install a signal handler using the **sigaction** system call.
- ❖ The return from this system call is immediate and our process continues; it is not blocked.
- ❖ When the datagram is ready to be read, the **SIGIO** signal is generated for our process. We can then read the data.
- ❖ The advantage of this model is that we are not blocked while waiting for the datagram to arrive.
- ❖ The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

# Asynchronous I/O Model

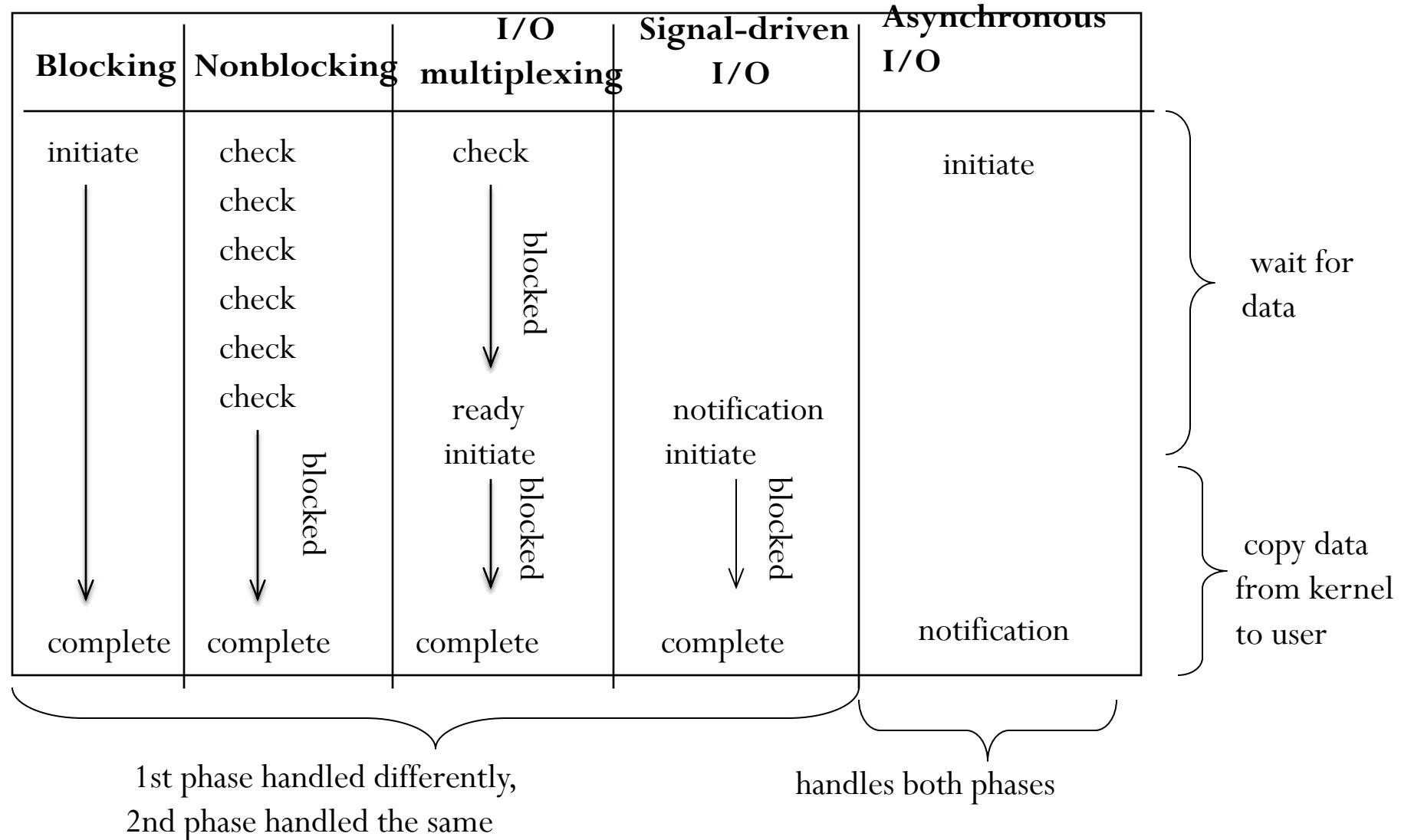




## Asynchronous I/O Model

- ❖ Asynchronous I/O is defined by the POSIX specification.
- ❖ These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.
- ❖ The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete.
- ❖ We call **aio\_read** and pass the kernel the descriptor, buffer pointer, buffer size, file offset, and how to notify us when the entire operation is complete.
- ❖ This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

# Comparison of the I/O Models





# Synchronous I/O and Asynchronous I/O

- **Synchronous I/O**
  - causes the requesting process to be blocked until that I/O operation (recvfrom) completes. (blocking, nonblocking, I/O multiplexing, signal-driven I/O)
- **Asynchronous I/O**
  - does not cause the requesting process to be blocked



## select function

- ❖ Allows the process to instruct the kernel to *wait for any one of multiple events to occur* and to wake up the process only when one or more of these events occurs or *when a specified amount of time has passed*.
- ❖ What descriptors we are interested in (readable ,writable , or exception condition) and how long to wait?

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
 *exceptset, const struct timeval *);
//Returns: +ve count of ready descriptors, 0 on timeout, -1 on error
struct timeval{
 long tv_sec; /* seconds */
 long tv_usec; /* microseconds */
}
```

- ❖ The final argument, timeout, tells the kernel how long to wait for one of the specified file descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.



## Possibilities for select function

1. **Wait forever** : return only when descriptor (s) is ready (specify **timeout** argument as NULL)
2. **wait up to a fixed amount of time:** Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.
3. **Do not wait at all** : return immediately after checking the descriptors(called Polling) (specify **timeout** argument as pointing to a **timeval** structure where the timer value is 0)
  - ❖ The wait is normally interrupted if the process catches a signal and returns from the signal handler
    - **select** might return an error of **EINTR**
    - Actual return value from function = -1

## Return value of select



- ❖ **Select()** returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred.
- ❖ If the time limit expires, **select()** returns 0.
- ❖ If **select()** returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified and the global variable **errno** will be set to indicate the error.

| <b>errno Constant</b> | <b>Description</b>                                                                            |
|-----------------------|-----------------------------------------------------------------------------------------------|
| EBADF                 | One or more of the file descriptors is not valid.                                             |
| EINTR                 | A signal was caught during the <b>select()</b> call.                                          |
| EINVAL                | One of the arguments is not valid (e.g., nfds is negative, or the timeout struct is invalid). |
| ENOMEM                | Unable to allocate memory for internal tables. Rare.                                          |



## select function Descriptor Arguments

- **readset** → descriptors for checking readable
- **writeset** → descriptors for checking writable
- **exceptset** → descriptors for checking exception conditions  
(2 exception conditions)
  - ✓ arrival of out of band data for a socket
  - ✓ the presence of control status information to be read from the master side of a pseudo terminal (Ignore)
- If you pass the 3 arguments as NULL, you have a high precision timer than the sleep function

# Descriptor Sets



- Array of integers : each bit in each integer correspond to a descriptor (**fd\_set**)
- 4 macros

- void FD\_ZERO(fd\_set \*fdset); /\* clear all bits in fdset \*/
- void FD\_SET(int fd, fd\_set \*fdset); /\*turn on the bit for fd in fdset \*/
- Void FD\_CLR(int fd, fd\_set \*fdset); /\* turn off the bit for fd in fdset\*/
- int FD\_ISSET(int fd, fd\_set \*fdset);/\* is the bit for fd on in fdset ? \*/

## Example of Descriptor sets Macros

**fd\_set rset;**

```
FD_ZERO(&rset); /*all bits off : initiate*/
FD_SET(1, &rset); /*turn on bit fd 1*/
FD_SET(4, &rset); /*turn on bit fd 4*/
FD_SET(5, &rset); /*turn on bit fd 5*/
```

## maxfdp1 argument to select function



- ❖ specifies the number of descriptors to be tested.
- ❖ Its value is the maximum descriptor to be tested, plus one. (hence maxfdp1)
  - Descriptors 0, 1, 2, up through and including **maxfdp1**-1 are tested
  - example: interested in **fds** 1,2, and 5 → **maxfdp1** = 6
  - Your code has to calculate the **maxfdp1** value constant **FD\_SETSIZE** defined by including **<sys/select.h>**
  - is the number of descriptors in the **fd\_set** datatype. (often = 1024)

## Value-Result arguments in select function

- ❖ Select modifies descriptor sets pointed to by **readset**, **writeset**, and **exceptset** pointers
  - On function call : Specify value of descriptors that we are interested in
  - On function return : Result indicates which descriptors are ready
  - Use **FD\_ISSET** macro on return to test a specific descriptor in an **fd\_set** structure
    - Any descriptor not ready will have its bit cleared
    - You need to turn on all the bits in which you are interested on all the descriptor sets each time you call **select**



## Condition for a socket to be ready for select

| Condition                                                                                                            | Readable?   | writable? | Exception? |
|----------------------------------------------------------------------------------------------------------------------|-------------|-----------|------------|
| <i>Data to read</i><br><i>read-half of the connection closed</i><br><i>new connection ready for listening socket</i> | •<br>•<br>• |           |            |
| <i>Space available for writing</i><br><i>write-half of the connection closed</i>                                     |             | •<br>•    |            |
| <i>Pending error</i>                                                                                                 | •           | •         |            |
| <i>TCP out-of-band data</i>                                                                                          |             |           | •          |

## pselect() function



```
#include <sys/select.h>
#include <signal.h>
#include <time.h>
int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
*exceptset, const struct timespec *timeout, const sigset_t
*sigmask);
/* Returns: count of ready descriptors, 0 on timeout, -1 on error */
```

- pselect contains two changes from the normal select function:
- **pselect** uses the **timespec** structure (another POSIX invention) instead of the **timeval** structure. The **tv\_nsec** member of the newer structure specifies nanoseconds, whereas the **tv\_usec** member of the older structure specifies microseconds.

```
struct timespec {
 time_t tv_sec; /* seconds */
 long tv_nsec; /* nanoseconds */
};
```

- **pselect** adds a sixth argument: a pointer to a signal mask.
- This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call pselect, telling it to reset the signal mask.



```

if (intr_flag)
 handle_intr(); /* handle the signal */
/* signals occurring in here are lost */
if ((nready = select(...)) < 0) {
 if (errno == EINTR) {
 if (intr_flag)
 handle_intr();
 }
...
}

```

- The problem is that between the test of **intr\_flag** and the call to **select**, if the signal occurs, it will be lost if **select** blocks forever.

```

sigset_t newmask, oldmask, zeromask;
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
/* block SIGINT */
if (intr_flag)
 handle_intr(); /* handle the signal */
if ((nready = pselect (... , &zeromask)) < 0) {
 if (errno == EINTR) {
 if (intr_flag) handle_intr ();
 }
...
}

```

Before testing the **intr\_flag** variable, we block SIGINT. When **pselect** is called, it replaces the signal mask of the process with an empty set (i.e., **zeromask**) and then checks the descriptors, possibly going to sleep. But when **pselect** returns, the signal mask of the process is reset to its value before **pselect** was called (i.e., SIGINT is blocked).



## poll() Function

```
#include <poll.h>
 int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- returns count of ready descriptors, 0 on timeout, -1 on error.
- poll() examines a set of file descriptors to see if some of them are ready for I/O or if certain events have occurred on them.
- The **fds** argument is a pointer to an array of **pollfd** structures.
- The **nfds** argument specifies the size of the **fds** array.
- Each element is a **pollfd** structure that specifies the conditions to be tested for a given descriptor, fd.

```
struct pollfd {
 int fd; /* descriptor to check */
 short events; /* events of interest on fd */
 short revents; /* events that occurred on fd */
};
```



- **fd**: File descriptor to poll.
- **events**: Events to poll for.
- **revents**: Events which may occur or have occurred.
- The event bitmasks in **events** and **revents** have the following bits:
  - **POLLERR** : An exceptional condition has occurred on the device or socket. This flag is output only, and ignored if present in the input events bitmask.
  - **POLLHUP**: The device or socket has been disconnected. This flag is output only, and ignored if present in the input events bitmask. Note that POLLHUP and POLLOUT are mutually exclusive and should never be present in the revents bitmask at the same time.
  - **POLLIN** : Data other than high priority data may be read without blocking.
  - **POLLNVAL** : The file descriptor is not open. This flag is output only, and ignored if present in the input events bitmask.
  - **POLLOUT** : Normal data may be written without blocking. This is equivalent to POLLWRNORM.
  - **POLLPRI** : High priority data may be read without blocking.
  - **POLLRDBAND** : Priority data may be read without blocking.
  - **POLLRDNORM** : Normal data may be read without blocking.
  - **POLLWRBAND** : Priority data may be written without blocking.
  - **POLLWRNORM** : Normal data may be written without blocking.

## Poll() function...



- With regard to TCP and UDP sockets, the following conditions cause poll to return the specified **revent**.
  - All regular TCP data and all UDP data is considered normal.
  - TCP's out-of-band data is considered priority band.
  - When the read half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
  - The presence of an error for a TCP connection can be considered either normal data or an error (POLLERR). In either case, a subsequent read will return -1 with errno set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
  - The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
  - The completion of a nonblocking connect is considered to make a socket writable.



- If timeout is greater than zero, it specifies a maximum interval (in milliseconds) to wait for any file descriptor to become ready.
- If timeout is zero, then poll() will return without blocking.
- If the value of timeout is -1, the poll blocks indefinitely.
- RETURN VALUES
  - poll() returns the number of descriptors that are ready for I/O, or -1 if an error occurred.
  - If the time limit expires, poll() returns 0.
  - If poll() returns with an error, including one due to an interrupted call, the fds array will be unmodified and the global variable errno will be set to indicate the error.
- **poll() will fail if:**
  - [EAGAIN] :Allocation of internal data structures fails. A subsequent request may succeed.
  - [EFAULT] : fds points outside the process's allocated address space.
  - [EINTR] : A signal is delivered before the time limit expires and before any of the selected events occurs.
  - [EINVAL]:The nfds argument is greater than OPEN\_MAX or the timeout argument is less than -1.

# Comparison of select(), pselect() and poll()



| Feature                      | <b>select()</b>                        | <b>pselect()</b>                        | <b>poll()</b>                                 |
|------------------------------|----------------------------------------|-----------------------------------------|-----------------------------------------------|
| <b>Header File</b>           | <sys/select.h>                         | <sys/select.h>                          | <poll.h>                                      |
| <b>Descriptor Limit</b>      | Limited by FD_SETSIZE (usually 1024)   | Same as select()                        | No hard limit — supports arbitrary fd numbers |
| <b>Input Parameter Type</b>  | Bitmask sets (fd_set)                  | Bitmask sets (fd_set)                   | Array of struct pollfd                        |
| <b>Output Mechanism</b>      | Modifies fd_set in place               | Modifies fd_set in place                | Sets revents field in each pollfd             |
| <b>Signal Safety</b>         | Interrupted by signals (EINTR)         | Atomically blocks/unblocks signals      | Interrupted by signals (EINTR)                |
| <b>Signal Masking Option</b> | No                                     | Yes — takes sigset_t *sigmask           | No                                            |
| <b>Precision of Timeout</b>  | struct timeval(microseconds)           | struct timespec(nanoseconds)            | int in milliseconds                           |
| <b>Timeout Modification</b>  | select() may modify the timeout struct | pselect() may modify the timeout struct | Timeout value not modified                    |
| <b>Portability</b>           | Widely portable                        | Less portable (POSIX.1-2001)            | Widely portable                               |
| <b>Scalability</b>           | Poor for high fd numbers               | Same as select()                        | Better scalability than select()              |
| <b>Error Values(errno)</b>   | EBADF, EINTR, EINVAL                   | Same as select()                        | EBADF, EINTR, ENOMEM, EINVAL                  |



## Summary of comparison

- Use **poll()** for better scalability and cleaner code when handling many file descriptors.
- Use **pselect()** if we need to **atomically block/unblock signals** during the wait (e.g., avoid race conditions).
- Avoid **select()** for large-scale applications due to its limitations with FD\_SETSIZE and fd\_set.



# Concurrent Server Design



- **Process** and **thread** are fundamental concepts in operating systems, representing units of execution. Understanding their differences, relationships, and uses is essential in systems programming and OS design.

## Process

- A *process* is an independent program in execution. It has its own:
  - *Address space* (memory), *Code*, *data*, *heap*, and *stack*, *Open file descriptors*, *Execution context* (registers, program counter)
- Processes are managed by the operating system and are isolated from each other, providing *security and stability*.

## Key Characteristics:

- Created using *fork()*, *exec()* in Unix/Linux.
- Switching between processes requires a *context switch*, which is relatively expensive.
- A process may contain one or more threads.



## Thread

- A *thread* (also called a *lightweight process*) is the smallest unit of CPU execution. A process can have *multiple threads*, which:
  - Share the *same address space*
  - Share *code, data, heap, and file descriptors*
  - Have their own *stack, program counter, and registers*
- Threads allow *concurrent execution* within a single process.

## Key Characteristics

- Created using APIs like *pthread\_create()* in POSIX systems or *std::thread* in C++.
- Faster context switching compared to processes.
- Ideal for *multitasking* within the same application (e.g., web server handling multiple clients).

## Comparison: Process vs Thread

| Feature           | Process                                     | Thread                                             |
|-------------------|---------------------------------------------|----------------------------------------------------|
| Memory Space      | Separate for each process                   | Shared within a process                            |
| Creation Time     | Slower (fork(), exec())                     | Faster (pthread_create())                          |
| Context Switching | Expensive (due to full state switch)        | Lightweight and faster                             |
| Communication     | Through IPC (pipes, sockets, etc.)          | Shared memory (easier but risky)                   |
| Failure Isolation | Safer — process crash doesn't affect others | Less safe — thread crash affects the whole process |
| Use Case          | Independent apps (e.g., browser, editor)    | Concurrent tasks (e.g., I/O, GUI + logic)          |

### Use Cases:

- **Processes:** Isolated apps, security-sensitive operations, separate services.
- **Threads:** Parallel tasks in the same application, I/O-bound or compute-bound tasks, performance optimization.

## Fork and Exec Functions



- **fork()** is called once but it returns twice.
- The creation of a new process is done using the **fork()** system call.
- A new program is run using the **exec(l,lp,le,v, vp)** family of system calls.
- These are two separate functions which may be used independently.
- A call to **fork()** will create a completely separate sub-process which will be exactly the same as the parent.
- The process that initiates the call to **fork** is called the **parent process**.
- The new process created by **fork** is called the **child process**.
- The child gets a copy of the parent's text and memory space.
- They do not share the same memory .



## Fork return values

- **fork()** system call returns an integer to both the parent and child processes:
  - -1 this indicates an **error** with no child process created.
  - A value of **zero** indicates that the child process code is being executed.
  - Positive integers represent the **child's process identifier** (PID) and the code being executed is in the parent's process.

```
if ((pid = fork()) == 0)
 printf("I am the child\n");
else
 printf("I am the parent\n");
```



# The exec() System Call

- Calling one of the **exec()** family will terminate the currently running program and starts executing a new one which is specified in the parameters of exec in the context of the existing process. The process id is not changed.

## EXEC Family of Functions

- int **execl**( const char \*path, const char \*arg, ...);
- int **execle**( const char \*path, const char \*arg , ..., char \* const envp[]);
- int **execv**( const char \*path, char \*const argv[]);
- int **execv**( const char \*path, char \*const argv[], char \* const envp[]);
- int **execlp**( const char \*file, const char \*arg, ...);
- int **execvp**( const char \*file, char \*const argv[]);

- The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a filename argument is specified:
  - if filename contains a slash, it is taken as a pathname.
  - otherwise the executable file is searched for in the directories specified by the PATH environment variable.

# Simple Execp Example



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
 /* fork a child process */
 pid = fork();
 if (pid < 0){ /* error occurred */
 fprintf(stderr, "Fork Failed");
 exit(-1);
 }
 else if (pid == 0){ /* child process */
 execl("/bin/ls","ls",NULL);
 }
 else { /* parent process */
 /* parent will wait for child to complete */
 wait(NULL);
 printf("Child Complete");
 exit(0);
 }
}
```

## Fork() and exec()



- When a program wants to have another program running in parallel, it will typically first use **fork**, then the child process will use **exec** to actually run the desired program.
- The **fork-and-exec** mechanism switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, and environment variables.

### Purpose of exec() functions

- When a process calls one of the **exec** functions that process is completely replaced by the new program.
- The new program starts execution from **main** function.
- The process does not change across an exec because a new process is not created.
- But this function replaces the current process with new program from disk.



## Fork() and exec()

### Problems of fork() function

- **Fork** is expensive. Because memory and all descriptors are duplicated in the child.
- **Inter process communication** is required to pass information between the **parent** and the **child** after the fork.
- A **descriptor** in the child process can affect a subsequent read or write by the parent.
- This **descriptor** copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.



# wait() and waitpid()

## Wait() / Waitpid() :

- Either of **wait** or **waitpid** can be used to remove zombies.
- **wait** (and **waitpid** in it's blocking form) temporarily suspends the execution of a parent process while a child process is running.
- Once the **child** has finished, the **waiting parent** is restarted.

## Declarations:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc); /* returns process ID if OK, or -1 on error */
pid_t waitpid(pid_t pid, int *statloc, int options); /*returns
process ID :if OK,
0 : if non-blocking option && no zombies around
-1 : on error
```

- The *statloc* argument can be one of two values:
- *NULL pointer*: the argument is simply ignored
- *pointer to an integer*: when **wait** returns, the integer this describes will contain status information of the terminated process.

# wait() and waitpid()



## Wait()

**wait** blocks the caller until a child process terminates

if more than one **child** is running then **wait()** returns the first time one of the parent's offspring exits

## Waitpid()

**waitpid** can be either **blocking** or **non-blocking**:

- If **options** is 0, then it is **blocking**
- If **options** is **WNOHANG**, then it is **non-blocking**

**waitpid** is more flexible:

- If **pid == -1**, it waits for any child process. In this respect, **waitpid** is equivalent to **wait**
- If **pid > 0**, it waits for the child whose process ID equals **pid**
- If **pid == 0**, it waits for any child whose process group ID equals that of the calling process
- If **pid < -1**, it waits for any child whose process group ID equals that absolute value of **pid**



# Process Related function

Functions those who are return a **process ID** are:

- **getpid()** : returns the current **process ID**.
- **getppid()** : returns the parent **process ID** of the **calling process**.
- **getuid()** : returns the real **user ID** of the **calling process**.
- **geteuid()**: returns the effective **user ID** of the calling process.
  - effective user ID gives the process additional permissions during execution of "set-user-ID" mode processes
- **getgid()**: returns the **real group ID** of the calling process
- **getegid()**: returns the **effective group ID** of the calling process.

## Process group

- A process group is a collection of one or more processes.
- Each process group has a unique **process ID**.
- A function **getpgrp()** returns the process **group id** of the calling process.
- Each process group have a **leader**.
- The leader is identified by having its **process group ID equal its process ID**.
- A process joins an existing process group or creates a new process group by calling **setpgid()**.

# Concurrent Server



- When a client request can take longer to service, we do not want to tie up a single server (*Iterative/sequential*) with one client; we want to handle multiple clients at the same time. The server that handles multiple clients simultaneously is a concurrent server.
- A concurrent server uses *processes*, *threads*, or *non-blocking I/O* to serve multiple clients in parallel - improving responsiveness and scalability.

## Key Goals of a Concurrent Server:

- Accept multiple client connections.
- Handle each client independently.
- Avoid blocking the entire server while waiting on I/O for one client
- Efficiently use system resources

# Common Approaches to Building Concurrent Servers



| Approach                           | Description                                                                | Tools/Functions Used               |
|------------------------------------|----------------------------------------------------------------------------|------------------------------------|
| Forking<br>(Process-based)         | Spawn a new process for each client connection                             | fork(), waitpid(), signal()        |
| Threading                          | Spawn a new thread for each client                                         | pthread_create(), pthread_join()   |
| Event-driven<br>(I/O multiplexing) | Use a single process/thread to monitor many clients using non-blocking I/O | select(), poll(), epoll()(Linux)   |
| Preforked/Thread Pool              | Pre-create a pool of worker processes/threads to handle connections        | fork()/pthread_create() in advance |



## Workflow of a Concurrent Server

1. *Socket Creation*: Create a listening socket using `socket()`.
2. *Binding*: Bind the socket to an IP and port using `bind()`.
3. *Listening*: Listen for incoming connections using `listen()`.
4. *Accepting Clients*: Use `accept()` in a loop.
5. *Concurrency*: For each connection:
  - i. Fork a child `process`, or
  - ii. Create a `thread`, or
  - iii. Register the socket in an event loop (e.g., `select()`).
6. *Serve the client*: Communicate via `read()/write()` or `recv()/send()`.
7. *Cleanup*: Close client sockets and terminate the worker/thread as needed.

## Pros and Cons of Each Model



| Model        | Pros                                | Cons                                         |
|--------------|-------------------------------------|----------------------------------------------|
| Forking      | Simple to implement, good isolation | High overhead, not scalable for many clients |
| Threading    | Lower overhead than processes       | Needs careful synchronization, shared memory |
| Event-driven | Scales well, uses fewer resources   | More complex logic, harder to debug          |
| Thread Pool  | Predictable resource usage          | Adds management overhead                     |

### Real-world Usage:

- Apache (prefork or worker MPM) – process/thread pool models
- Nginx – event-driven (epoll)
- Node.js – single-threaded event loop
- Chat servers, web servers, game servers - all use concurrency patterns

# Outline of concurrent server (forking model)

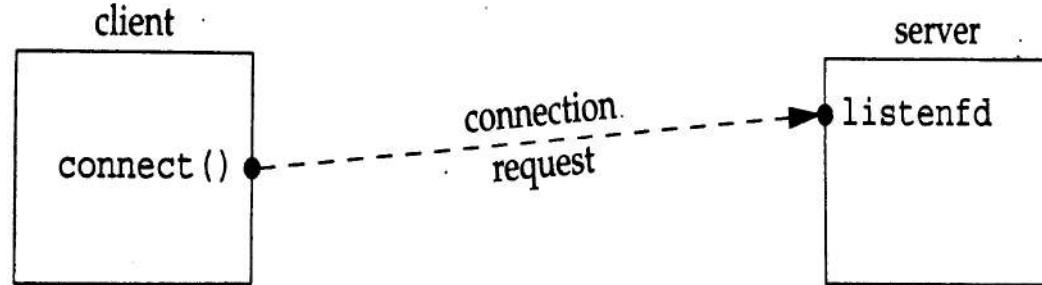


```
pid_t pid;

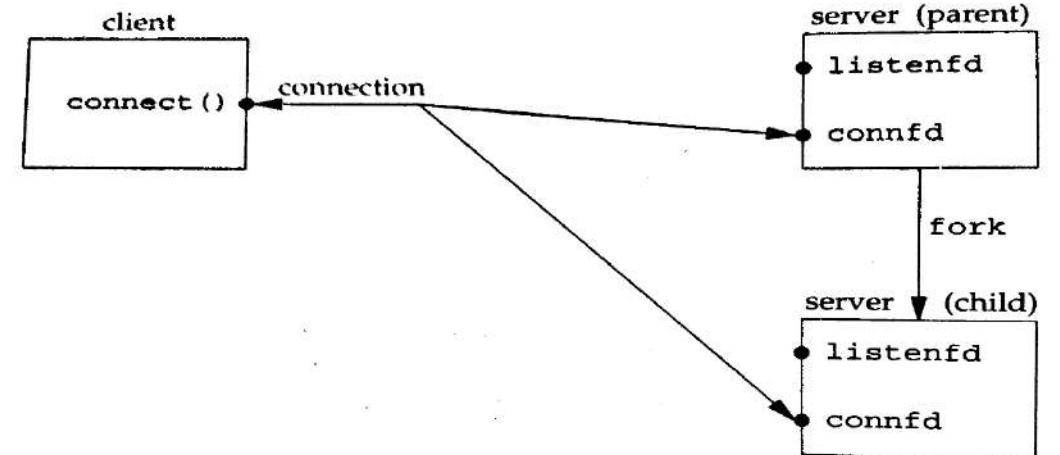
int listenfd, connfd;
listenfd = socket(...);
/* fill in sockaddr_in{ } with server's well-known port */
bind(listenfd, ...);
listen(listenfd, LISTENQ);
for(;;) {
 connfd = accept(listenfd, ...); // probably blocks
 if((pid = Fork()) ==0) {
 close(listenfd); // child closes listening socket
 doit(connfd); // process the request
 close(connfd); // done with this client
 exit(0); //child terminates
 }
 close(connfd);
}
```



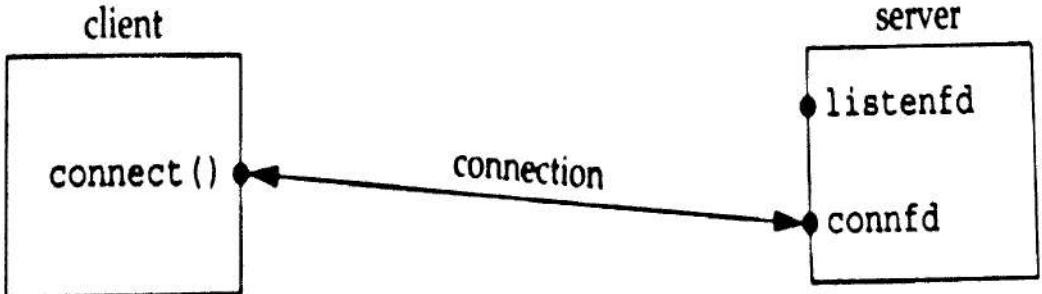
- When a connection is established, accept returns, the server calls fork, and the child process services the client (on *connfd*, the connected socket) and the parent process waits for another connection (on *listenfd*, the listening socket).
- The parent closes the connected socket since the child handles the new client. The function *doit* does whatever is required to service the client.
- Calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. However, the close of *connfd* by the parent (in the outline) doesn't terminate its connection with the client. This is because every file or socket has a reference count.
- The reference count is maintained in the file table entry. This is a count of the number of descriptors that are currently open that refer to this file or socket. In the outline, after socket returns, the file table entry associated with *listenfd* has a reference count of 1.
- After accept returns, the file table entry associated with *connfd* has a reference count of 1. But, after fork returns, both descriptors are shared between the parent and child, so the file table entries associated with both sockets now have a reference count of 2.
- Therefore, when the parent closes *connfd*, it just decrements the reference count from 2 to 1 and that is all. The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0. This will occur at some time later when the child closes *connfd*.



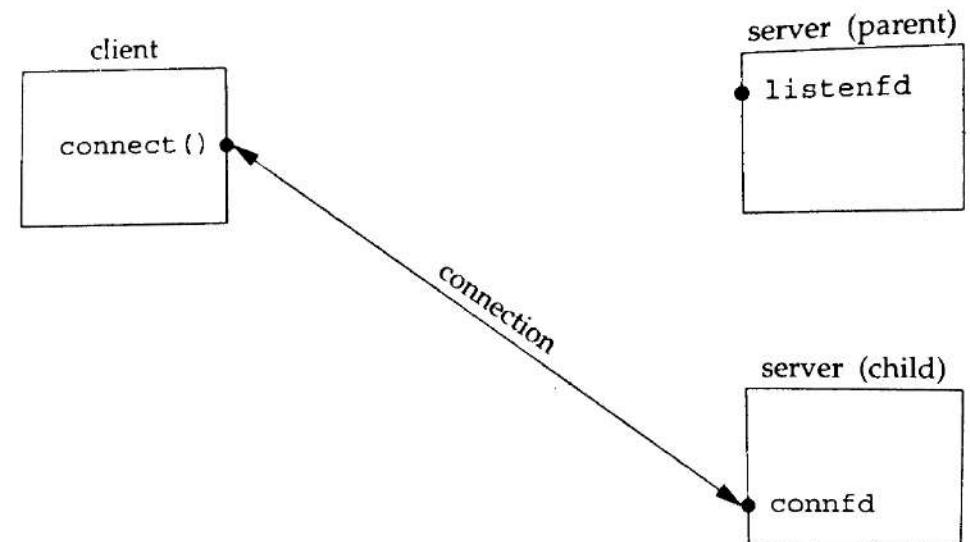
**Figure 4.14** Status of client/server before call to accept returns.



**Figure 4.16** Status of client/server after `fork` returns.



**Figure 4.15** Status of client/server after return from accept.



**Figure 4.17** Status of client/server after parent and child close appropriate sockets.

# Concurrent Server Using Thread Model



- A *thread-based concurrent server* creates a new thread for each incoming client connection. This allows the server to handle *multiple clients in parallel*, each served independently in its own thread of execution.
- Threads share the same memory space (unlike processes), so they can access shared resources easily but this also means *thread safety and synchronization* (e.g., using mutexes) are important if shared data is accessed.
- This model is efficient for I/O-bound applications (like web or chat servers), where most threads are blocked on I/O waiting for client input.

## Advantages:

- Lower overhead than fork-based (no memory duplication).
- Easier communication between threads (shared memory).
- More scalable for moderate number of clients.

## Disadvantages:

- Requires thread-safety handling (e.g., mutexes).
- Threads crashing can affect the whole process.

# Outline of concurrent server (Multithreaded model)



```
#include <...> // Standard headers
#define PORT ...
#define MAX_CLIENTS ...
void *handle_client(void *arg) {
 read();
 write();
 close();
 pthread_exit();
}
int main() {
 socket();
 bind();
 listen();
 while (1) {
 accept();
 malloc(); // Allocate memory for client socket
 pthread_create(); // Create a new thread
 pthread_detach(); // Detach thread
 }
 close(); // Close server socket (if reached)
 return 0;
}
```

- The **main thread** creates a socket, binds it to a port, listens for incoming connections, and continuously accepts clients.
- For each client, it:
  - Allocates memory for the client socket
  - Spawns a **new thread** using `pthread_create()` to handle the client
  - Detaches the thread so resources are auto-reclaimed
- The handler thread performs `read()`, `write()`(to echo or process client data), and then closes the connection.
- Each client is handled concurrently in its own thread.

Example program: `conserver_pthread.c`



## Concurrent Server Using I/O Multiplexing (using select)

- A select()-based server uses a single thread/process to monitor multiple client sockets simultaneously.
- It waits until one or more sockets become "ready" for I/O (read/write).
- This is more scalable than creating one thread per client and avoids the overhead of context switching.
- It is ideal for:
  - Moderate concurrency
  - Simple I/O-bound servers
- No need for multi-threading or synchronization

# Outline of concurrent server (I/O Multiplexing)



```
#include <...> // Required headers
#define PORT ...
#define MAX_CLIENTS ...
int main() {
 socket(); //Create server socket
 bind(); //Bind to IP and port
 listen(); //Start listening
 FD_ZERO(); //Clear the fd_set
 FD_SET(); //Add server socket to the set
 select(); //Wait for activity on sockets
 while (1) {
 select(); // Monitor multiple sockets
 if (FD_ISSET(server_socket)) {
 accept(); //New connection
 FD_SET(); //Add new client socket to set
 }
 }
}
```

Example Program: conserver\_select.c

```
for (each client socket) {
 if(FD_ISSET(client_socket)) {
 read(); // Handle client input
 write(); // Respond if needed
 if (connection closed) {
 close(); // Close client socket
 FD_CLR(); // Remove from fd_set
 }
 }
}
close(); // Close server socket (if reached)
return 0;
}
```

- This is a single-threaded concurrent server that uses `select()` to monitor all sockets:
- When the listening socket is ready, a new client is accepted.
- When a client socket is ready, data is read/written.
- `FD_SET`, `FD_CLR`, and `FD_ISSET` manage sockets in the monitored set.



# Implementing broadcast and multicast communication

# Broadcast, multicast and anycast



- **Unicast** sends packets from one sender to one receiver-the most common form.
- **Broadcast** delivers a packet to all hosts on a subnet simultaneously.
- **Multicast** sends a packet to a specific group of receivers (one-to-many).
- **Anycast** sends a packet to the nearest or most available receiver from a group offering the same service.

| Communication | Destination       | Scope       | Use Case                |
|---------------|-------------------|-------------|-------------------------|
| Unicast       | One specific host | One-to-one  | HTTP, SSH, FTP          |
| Broadcast     | All on subnet     | One-to-all  | DHCP, ARP, service ping |
| Multicast     | Group subscribers | One-to-many | IPTV, conferencing      |

# Broadcast



- Sends packet to **all hosts on a subnet**
- Only works with UDP
- IP address: **Subnet broadcast address** (e.g., 192.168.1.255)
- Socket must be explicitly enabled for broadcast using SO\_BROADCAST.
- TCP does not support broadcast (since it's connection-oriented).

## Limitations of Broadcast

| Issue                        | Description                                     |
|------------------------------|-------------------------------------------------|
| <b>Network load</b>          | Every host receives the packet, even if unused. |
| <b>Security</b>              | Can be abused for DoS (e.g., smurf attack).     |
| <b>No delivery guarantee</b> | UDP is unreliable-packets may be dropped.       |
| <b>Not routable</b>          | Broadcasts do not cross routers.                |

# Broadcast



## Broadcast Programming Steps

1. Create UDP socket
2. Enable broadcast: `setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, ...)`
3. Set destination address: Broadcast IP
4. Send message using `sendto()`
5. Receiver binds to `INADDR_ANY`

# Broadcast



## Sender Code Outline

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
// Enable broadcast option
int broadcast = 1;

setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast, sizeof(broadcast));

// Set destination to broadcast address
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = inet_addr("192.168.1.255");

// Send broadcast message
sendto(sockfd, message, strlen(message), 0, (struct sockaddr *)&addr, sizeof(addr));
```

# Broadcast



## Receiver Code Outline

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = INADDR_ANY;

bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));

// Receive from any sender
recvfrom(sockfd, buffer, sizeof(buffer), 0, NULL, NULL);
```

## Network Configuration Requirements

- Host must be connected to a subnet with broadcast enabled.
- Firewall and router must **allow UDP broadcast traffic** on the desired port.
- Proper subnet mask must be set to determine the correct broadcast address.

Example Program: `broadcast_sender.c` and `broadcast_receiver.c`

# Multicast



- Multicast is a form of one-to-many communication.
- A sender sends a message to a multicast group address, and only subscribed receivers get the message.
- Efficient for group communication like IPTV, conferencing, and live data feeds.

## Multicast IP Address Range (IPv4)

- *Class D addresses:* 224.0.0.0 – 239.255.255.255
- Common examples:
  - 224.0.0.1: All systems on subnet
  - 224.0.0.9: RIP routers
- These addresses are not assigned to specific hosts but to **groups**.

## Characteristics of Multicast

|                    |                                                       |
|--------------------|-------------------------------------------------------|
| Protocol           | UDP only                                              |
| Delivery           | To subscribed members only                            |
| Group join/leave   | Managed via IGMP (Internet Group Management Protocol) |
| TTL (Time To Live) | Controls how far packets can travel                   |
| Loopback           | Sender can (optionally) receive its own message       |

# Multicast



## Programming Steps

### Receiver:

1. Create UDP socket
2. Set SO\_REUSEADDR (optional but useful for multiple receivers)
3. Bind to port and IP INADDR\_ANY
4. Join multicast group with IP\_ADD\_MEMBERSHIP

### Sender:

1. Create UDP socket
2. Set IP\_MULTICAST\_TTL if needed
3. Send packet to multicast group address + port

### Note:

- Routers must support **IGMP** and allow multicast forwarding.
- On Linux, loopback interfaces may not forward multicast unless manually enabled.

Example Program: [multicast\\_sender.c](#) and [multicast\\_receiver.c](#)



# Socket option

# Socket Options



- **Socket options** are configuration parameters that modify the behavior of sockets at runtime.
- They allow fine control over:
  - Timeouts
  - Buffer sizes
  - Address reuse
  - Multicast behavior
  - TCP-level features (e.g., Nagle's algorithm)
- There are 3 ways to get and set options affecting sockets -
  - the *getsockopt* and *setsockopt* functions.
  - the *fcntl* function
  - the *ioctl* function

# getsockopt() and setsockopt()



```
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname, void * optval, socklen_t * optlen);
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

*Both functions return 0 if OK else -1 on error.*

- The *sockfd* must refer to an open socket descriptor.
- The *level* indicates whether the socket option is *general* or *protocol-specific* socket.
- The *optval* is a pointer to a variable from which the new value of the option is fetched by *setsockopt*, or into which the current value of the option is stored by *getsockopt*.
- The size of this variable is specified by the final argument, as a value for *setsockopt* and as a value-result for *getsockopt*.



## getsockopt() and setsockopt()

- There are two basic types of options that can be queried by *getsockopt* or set by *setsockopt*: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values).
- When calling *getsockopt* for the flag options, *\* optval* is an integer.
- The value returned in *\*optval* is *zero* if the option is disabled, or nonzero if the option is enabled.
- Similarly, *setsockopt* requires a *nonzero \*optval* to turn the option on, and a *zero* value to turn the option off.
- For non-flag options, the option is used to pass a value of the specified datatype between the user process and the system.



## Socket Options

- Two basic type of options -
  - Flags - binary options that enable or disable a feature.
  - Values - options that fetch and return specific values.
- Not supported by all implementations.
- Socket option fall into 4 main categories -
  - Generic socket options
    - SO\_RCVBUF, SO\_SNDBUF, SO\_BROADCAST, etc.
  - IPv4
    - IP\_TOS, IP\_MULTICAST\_IF, etc.
  - IPv6
    - IPv6\_HOPLIMIT, IPv6\_NEXTHOP, etc.
  - TCP
    - TCP\_MAXSEG, TCP\_KEEPALIVE, etc.

# Common Socket Options

| Level       | Option                   | Purpose                            |
|-------------|--------------------------|------------------------------------|
| SOL_SOCKET  | SO_REUSEADDR             | Allow reuse of local address       |
| SOL_SOCKET  | SO_RCVBUF, SO_SNDBUF     | Set receive/send buffer size       |
| SOL_SOCKET  | SO_RCVTIMEO, SO_SNDTIMEO | Set timeout for recv/send          |
| SOL_SOCKET  | SO_BROADCAST             | Enable sending broadcast datagrams |
| IPPROTO_IP  | IP_TTL                   | Set IP time-to-live                |
| IPPROTO_IP  | IP_MULTICAST_TTL         | TTL for multicast packets          |
| IPPROTO_IP  | IP_ADD_MEMBERSHIP        | Join a multicast group             |
| IPPROTO_TCP | TCP_NODELAY              | Disable Nagle's algorithm          |

## Socket States



- Options have to be set or fetched depending on the state of a socket.
- Some socket options are inherited from a listening socket to the connected sockets on the server side.
  - E.g. **SO\_RCVBUF** and **SO\_SNDBUF**

These options have to be set on the socket before calling *listen()* on the server side and before calling *connect()* on the client side.

# Generic Socket Options



## ➤ SO\_BROADCAST

- Enables or disables the ability of a process to send broadcast messages.
- It is supported only for datagram sockets.
- Its default value is **off**.

## ➤ SO\_ERROR

- *Pending Error* - When an error occurs on a socket, the kernel sets the **so\_error** variable.
- The process can be notified of the error in two ways -
  - If the process is blocked in **select** for either read or write, it returns with either or both conditions set.
  - If the process is using signal driven I/O, the **SIGIO** signal is generated for the process.

# Generic Socket Options contd...



## ➤ **SO\_KEEPALIVE**

- Purpose of this option is to detect if the peer host crashes. The **SO\_KEEPALIVE** option will detect half-open connections and terminate them.
- If this option is set and no data has been exchanged for 2 hours, then TCP sends **keepalive** probe to the peer.
  - Peer responds with **ACK**. Another probe will be sent only after 2 hours of inactivity.
  - Peer responds with **RST** (has crashed and rebooted). Error is set to **ECONNRESET** and the socket is closed.
  - No response. 8 more probes are sent after which the socket's pending error is set to either **ETIMEDOUT** or **EHOSTUNREACH** and the socket is closed.

## ➤ Receive Low Water Mark -

- Amount of **data** that must be in the socket receive buffer for a socket to become ready for *read*.

## ➤ Send Low Water Mark -

- Amount of **space** that must be available in the socket send buffer for a socket to become ready for *write*.

## ➤ **SO\_RCVLOWAT** and **SO SNDLOWAT**

- These options specify the receive low water mark and send low water mark for TCP and UDP sockets.



## ➤ **SO\_RCVTIMEO** and **SO\_SNDFTIMEO**

- These options place a timeout on socket receives and sends.
- The timeout value is specified in a ***timeval*** structure.

```
struct timeval {
 long tv_sec ;
 long tv_usec ;
}
```

- To disable a timeout, the values in the ***timeval*** structure are set to 0.

## ➤ **SO\_REUSEADDR**

- It allows a listening server to restart and bind its well known port even if previously established connections exist.
- It allows multiple instances of the same server to be started on the same port, as long as each instance binds a different local IP address.
- It allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.
- It allows completely duplicate bindings only for UDP sockets (broadcasting and multicasting).



## ➤ SO\_LINGER

- Specifies how *close* operates for a connection-oriented protocol
- The following structure is used:

```
struct linger {
 int l_onoff;
 int l_linger;
}

// l_onoff - 0=off; nonzero=on
// l_linger specifies seconds
```

### Three scenarios:

1. If ***l\_onoff*** is 0, ***close*** returns immediately. If there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. The value of ***l\_linger*** is ignored.
2. If ***l\_onoff*** is **nonzero** and ***linger*** is 0, TCP aborts the connection when ***close*** is called. TCP discards data in the send buffer and sends **RST** to the peer.



## Generic Socket Options Contd...

### 3. SO\_LINGER (cont)

If **I\_onoff** is nonzero and **linger** is nonzero, the kernel will linger when *close* is called.

- If there is any data in the send buffer, the process is put to sleep until either:
  - the data is sent and acknowledged
  - Or
  - the linger time expires (for a nonblocking socket the process will not wait for *close* to complete)
- When using this feature, the return value of *close* must be checked. If the linger time expires before the remaining data is sent and acknowledged, *close* returns **EWOULDBLOCK** and any remaining data in the buffer is ignored.

## Generic Socket Options Contd...



### ➤ **SO\_SNDBUF/SO\_RCVBUF**

- Level: SOL\_SOCKET
- Get/Set supported
- Non-flag option
- Datatype of optval: int
- Description: Send buffer size/Receive buffer size

### ➤ **TCP\_NODELAY**

- Level: IPPROTO\_TCP
- Get/Set supported
- Flag option i.e. enable or disable Nagle algorithm
- Description: Disable/Enable Nagle algorithm

# Socket Options Summary



| Socket Option     | Level       | Use Case                           |
|-------------------|-------------|------------------------------------|
| SO_REUSEADDR      | SOL_SOCKET  | Reuse port                         |
| SO_RCVBUF         | SOL_SOCKET  | Tune performance                   |
| SO_BROADCAST      | SOL_SOCKET  | Enable UDP broadcast               |
| SO_RCVTIMEO       | SOL_SOCKET  | Set recv() timeout                 |
| TCP_NODELAY       | IPPROTO_TCP | Disable Nagle (for real-time apps) |
| IP_TTL            | IPPROTO_IP  | Set packet TTL                     |
| IP_MULTICAST_TTL  | IPPROTO_IP  | Set multicast range                |
| IP_ADD_MEMBERSHIP | IPPROTO_IP  | Join multicast group               |

# fentl()function



- *fcntl()* stands for “file control” and this function performs various descriptor control operations.
- The fcntl function provides the following features related to network programming.
  - *Non-blocking I/O* – We can set the *O\_NONBLOCK* file status flag using the *F\_SETFL* command to set a socket as non-blocking.
  - *Signal-driven I/O* – We can set the *O\_ASYNC* file status flag using the *F\_SETFL* command, which causes the SIGIO signal to be generated when the status of a socket changes.
  - The *F\_SETOWN* command lets us set the socket owner (the process ID or process group ID) to receive the *SIGIO* and *SIGURG* signals. The former signal is generated when the signal-driven I/O is enabled for a socket and the latter signal is generated when new out-of-band data arrives for a socket. The *F\_GETOWN* command returns the current owner of the socket.

# fcntl(...)



```
int fcntl(int fd, int cmd, long arg);
```

- Each descriptor has a set of file flags that is fetched with the *F\_GETFL* command and set with the *F\_SETFL* command. The two flags that affect a socket are
  - *O\_NONBLOCK* - non-blocking I/O
  - *O\_ASYNC*-signal-driven I/O
  - Miscellaneous file control operations
    - Non-blocking I/O (*O\_NONBLOCK*, *F\_SETFL*)
    - Signal-driven I/O (*O\_ASYNC*, *F\_SETFL*)
    - Set socket owner (*F\_SETOWN*)

## fcntl and ioctl

| Operation                           | fcntl               | ioctl                  | Routing socket | Posix.1g   |
|-------------------------------------|---------------------|------------------------|----------------|------------|
| set socket for nonblocking I/O      | F_SETFL, O_NONBLOCK | FIONBIO                |                | fcntl      |
| set socket for signal-driven I/O    | F_SETFL, O_ASYNC    | FIOASYNC               |                | fcntl      |
| set socket owner                    | F_SETOWN            | SIOCSPGRP or FIOSETOWN |                | fcntl      |
| get socket owner                    | F_GETOWN            | SIOCGPGRP or FIOGETOWN |                | fcntl      |
| get #bytes in socket receive buffer |                     | FIONREAD               |                |            |
| test for socket at out-of-band mark |                     | SIOCATMARK             |                | socketmark |
| obtain interface list               |                     | SIOCGIFCONF            | sysctl         |            |
| interface operations                |                     | SIOC[GS]IFxxx          |                |            |
| ARP cache operations                |                     | SIOCxARP               | RTM_xxx        |            |
| routing table operations            |                     | SIOCxxxRT              | RTM_xxx        |            |

**Figure 7.15** Summary of fcntl, ioctl, and routing socket operations.



## fcntl() function

- *fcntl* provides the following features related to network programming
- **Nonblocking I/O** (be aware of error-handling in the following code)

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

- **Signal driven I/O**

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_ASYNC;
fcntl(fd, F_SETFL, flags);
```

- Set socket owner to receive SIGIO signals

```
fcntl(fd, F_SETOWN, getpid());
```

## fcntl() function



- The signals **SIGIO** and **SIGURG** are generated for a socket only if the socket has been assigned an owner with the **F\_SETOWN** command. The integer arg value for the **F\_SETOWN** command can be either positive integer, specifying the process ID to receive the signal, or a negative integer whose absolute value is the process group ID to receive the signal.
- The **F\_GETOWN** command returns the socket owner as the return value from the fcntl function, either the process ID (a positive return value) or the process group ID (a negative value other than -1).
- The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group to receive the signal.

# ioctl operations



- The common use of *ioctl* by network programs (typically servers) is to obtain information on all the host's interfaces when the program starts: the interface addresses, whether interface supports broadcasting, whether the interface supports multicasting, and so on.

## ➤ **Ioctl() Function**

- This function affects an open file referenced by the **fd** argument.

```
#include <unistd.h>
int ioctl(int fd, int request, .../* void *arg */);
```

Returns: 0 if OK, -1 on error

- The third argument is always a pointer, but the type of pointer depends on the request.



## Example ( Convert to Non blocking)

```
int flags = fcntl(fd, F_GETFL, 0);
int status = ioctl(fd, FIONBIO, &flags);
if (status < 0) {
 perror("ioctl");
 exit(EXIT_FAILURE);
}
```

## ioctl() Function



- We can divide the requests related to networking into six categories.
  1. Socket operations
  2. File operations
  3. Interface operations
  4. ARP cache operations
  5. Routing table operations
  6. STREAMS system

Note that not only do some of the **ioctl** operations overlap some of the **fcntl** operations (e.g. setting a socket to non-blocking), but there are also some operations that can be specified more than one way using **ioctl** (e.g., setting the process group ownership of a socket).

| Category  | request         | Description                                  | Datatype       |
|-----------|-----------------|----------------------------------------------|----------------|
| Socket    | SIOCATMARK      | At out-of-band mark ?                        | int            |
|           | SIOCSPGRP       | Set process ID or process group ID of socket | int            |
|           | SIOCGPGRP       | Get process ID or process group ID of socket | int            |
| File      | FIONBIO         | Set/clear nonblocking flag                   | int            |
|           | FICASYNC        | Set/clear asynchronous I/O flag              | int            |
|           | FIONREAD        | Get # bytes in receive buffer                | int            |
|           | FIOSETOWN       | Set process ID or process group ID of file   | int            |
|           | FIOGETOWN       | Get process ID or process group ID of file   | int            |
| Interface | SIOCGLIFCONF    | Get list of all interfaces                   | struct ifconf  |
|           | SIOCSIFADDR     | Set interface address                        | struct ifreq   |
|           | SIOCGLIFADDR    | Get interface address                        | struct ifreq   |
|           | SIOCSIFFLAGS    | Set interface flags                          | struct ifreq   |
|           | SIOCGLIFFLAGS   | Get interface flags                          | struct ifreq   |
|           | SIOCSIFDSTADDR  | Set point-to-point address                   | struct ifreq   |
|           | SIOCGLIFDSTADDR | Get point-to-point address                   | struct ifreq   |
|           | SIOCGLIFBRDADDR | Get broadcast address                        | struct ifreq   |
|           | SIOCSIFBRDADDR  | Set broadcast address                        | struct ifreq   |
|           | SIOCGLIFNETMASK | Get subnet mask                              | struct ifreq   |
|           | SIOCSIFNETMASK  | Set subnet mask                              | struct ifreq   |
|           | SIOCGLIFMETRIC  | Get interface metric                         | struct ifreq   |
|           | SIOCSIFMETRIC   | Set interface metric                         | struct ifreq   |
|           | SIOCGLIFMTU     | Get interface MTU                            | struct ifreq   |
|           | SIOCXXX         | (many more; implementation-dependent)        | struct ifreq   |
| ARP       | SIOCSARP        | Create/modify ARP entry                      | struct arpreq  |
|           | SIOCGRARP       | Get ARP entry                                | struct arpreq  |
|           | SIOCDARP        | Delete ARP entry                             | struct arpreq  |
| Routing   | SIOCADDRT       | Add route                                    | struct rtentry |
|           | SIOCDELRT       | Delete route                                 | struct rtentry |
| STREAMS   | I_XXX           | (see Section 31.5)                           |                |

# Socket operations



- Three **ioctl** requests are explicitly used for sockets. All three require that the third argument to **ioctl** be a pointer to an integer.
- **SIOCATMARK**: Return through the integer pointed to by the third argument a non-zero value if the socket's read pointer is currently at the out-of-band mark, or a zero value if the read pointer is not at the out-of-band mark.
- **SIOCGRPGRP**: Return through the integer pointed to by the third argument either the process ID or the process group ID that is set to receive **SIGIO** or SIGURG signal for this socket. This request is identical to an fcntl of **F\_GETOWN**, note that POSIX standardizes the fcntl.
- **SIOCSPGRP**: Set either the process ID or process group ID to receive the **SIGIO** or **SIGURG** signal for this socket from the integer pointed to by the third argument. This request is identical to an fnctl of **F\_SETOWN**, note that POSIX standardizes the **fcntl**.



# File Operations

- The next group of requests begin with FIO and may apply to certain types of files, in addition to sockets.

|                  |                                                                                                                                                                                                                                                                                                                                                                           |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>FIONBIO</b>   | The nonblocking flag for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This request has the same effect as the <b>O_NONBLOCK</b> file status flag, which can be set and cleared with the <b>F_SETFL</b> command to the fcntl function.                                            |
| <b>FIOASYNC</b>  | The flag that governs the receipt of asynchronous I/O signals (SIGIO) for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This flag has the same effect as the <b>O_ASYNC</b> file status flag, which can be set and cleared with the <b>F_SETFL</b> command to the fcntl function. |
| <b>FIONREAD</b>  | Return in the integer pointed to by the third argument to ioctl the number of bytes currently in the socket receive buffer. This feature also works for files, pipes, and terminals.                                                                                                                                                                                      |
| <b>FIOSETOWN</b> | Equivalent to SIOCSPGRP for a socket.                                                                                                                                                                                                                                                                                                                                     |
| <b>FIOGETOWN</b> | Equivalent to SIOCGPGRP for a socket.                                                                                                                                                                                                                                                                                                                                     |

# Interface Operations



- The **SIOCGIFCONF** request returns the name and a socket address structure for each interface that is configured. Many of requests use a socket address structure to specify or return an IP address or address mask with the application.

|                |                                                                                                                                                                                                                                                                                      |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SIOCGIFADDR    | Return the unicast address in the <code>ifr_addr</code> member.                                                                                                                                                                                                                      |
| SIOCSIFADDR    | Set the interface address from the <code>ifr_addr</code> member. The initialization function for the interface is also called.                                                                                                                                                       |
| SIOCGIFFLAGS   | Return the interface flags in the <code>ifr_flags</code> member. The names of the various flags are <code>IFF_xxx</code> and are defined by including the <code>&lt;net/if.h&gt;</code> header.                                                                                      |
| SIOCSIFFLAGS   | Set the interface flags from the <code>ifr_flags</code> member.                                                                                                                                                                                                                      |
| SIOCGIFDSTADDR | Return the point-to-point address in the <code>ifr_dstaddr</code> member.                                                                                                                                                                                                            |
| SIOCSIFDSTADDR | Set the point-to-point address from the <code>ifr_dstaddr</code> member.                                                                                                                                                                                                             |
| SIOCGIFBRDADDR | Return the broadcast address in the <code>ifr_broadaddr</code> member. The application must first fetch the interface flags and then issue the correct request: <code>SIOCGIFBRDADDR</code> for a broadcast interface or <code>SIOCGIFDSTADDR</code> for a point-to-point interface. |
| SIOCSIFBRDADDR | Set the broadcast address from the <code>ifr_broadaddr</code> member.                                                                                                                                                                                                                |
| SIOCGIFNETMASK | Return the subnet mask in the <code>ifr_addr</code> member.                                                                                                                                                                                                                          |
| SIOCSIFNETMASK | Set the subnet mask from the <code>ifr_addr</code> member.                                                                                                                                                                                                                           |
| SIOCGIFMETRIC  | Return the interface metric in the <code>ifr_metric</code> member. The interface metric is maintained by the kernel for each interface but is used by the routing daemon <code>routed</code> . The interface metric is added to the hop count (to make an interface less favorable). |
| SIOCSIFMETRIC  | Set the interface routing metric from the <code>ifr_metric</code> member.                                                                                                                                                                                                            |

## ARP Cache Operations



- On some systems, the ARP cache is also manipulated with the **ioctl** function. Systems that use routing sockets usually use routing sockets instead of **ioctl** to access the ARP cache. These requests use an **arpreq** structure, shown below and defined by including the <net/if\_arp.h> header.

```
<net/if_arp.h>

struct arpreq {
 struct sockaddr arp_pa; /* protocol address */
 struct sockaddr arp_ha; /* hardware address */
 int arp_flags; /* flags */
};

#define ATF_INUSE 0x01 /* entry in use */
#define ATF_COM 0x02 /* completed entry (hardware addr valid) */
#define ATF_PERM 0x04 /* permanent entry */
#define ATF_PUBL 0x08 /* published entry (respond for other host) */
```

# ARP Cache Operations



- The third argument to ioctl must point to one of these structures. The following three *requests* are supported

|                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>SIOCSARP</b> | Add a new entry to the ARP cache or modify an existing entry. arp_pa is an Internet socket address structure containing the IP address, and arp_ha is a generic socket address structure with sa_family set to AF_UNSPEC and sa_data containing the hardware address (e.g., the 6-byte Ethernet address). The two flags, ATF_PERM and ATF_PUBL, can be specified by the application. The other two flags, ATF_INUSE and ATF_COM, are set by the kernel. |
| <b>SIOCDARP</b> | Delete an entry from the ARP cache. The caller specifies the Internet address for the entry to be deleted.                                                                                                                                                                                                                                                                                                                                              |
| <b>SIOCGARP</b> | Get an entry from the ARP cache. The caller specifies the Internet address, and the corresponding Ethernet address is returned along with the flags.                                                                                                                                                                                                                                                                                                    |

# Routing Table Operations



- On some systems, two **ioctl** requests are provided to operate on the routing table. These two requests require that the third argument to **ioctl** be a pointer to an **rtentry** structure, which is defined by including the <net/route.h> header. These requests are normally issued by the route program. Only the superuser can issue these requests. On systems with routing sockets, these requests use routing sockets instead of ioctl.

|           |                                         |
|-----------|-----------------------------------------|
| SIOCADDRT | Add an entry to the routing table.      |
| SIOCDELRT | Delete an entry from the routing table. |

- There is no way with **ioctl** to list all the entries in the routing table. This operation is usually performed by the **netstat** program when invoked with the -r flag. This program obtains the routing table by reading the kernel's memory (/dev/kmem).

# IOCTL summary



The ioctl commands that are used in network programs can be divided into six categories:

1. Socket operations (Are we at the out-of-band mark?)
2. File operations (set or clear the nonblocking flag)
3. Interface operations (return interface list, obtain broadcast address)
4. ARP table operations (create, modify, get, delete)
5. Routing table operations (add or delete)
6. STREAMS system



# Logging in Unix



## syslogd Daemon

- Unix systems normally start a daemon named **syslogd** from one of the system initialization scripts, and it runs as long as the system is up. The **syslogd** perform the following actions on startup.
  - The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log messages that the daemon can receive.
  - A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).
  - A UDP socket is created and bound to port 514 (the syslog service).
  - The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.
- The **syslogd** daemon runs in an infinite loop that calls **select**, waiting for any one of its three descriptors (last three of above bullets) to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the SIGUP signal, it reads its configuration file

## syslog function



- Since a daemon does not have a controlling terminal, it cannot just **fprintf** to **stderr**. The common technique for logging messages from a daemon is to call the **syslog** function.

```
#include <syslog.h>
void syslog(int priority, const char * message, ...);
```

- The priority argument is a combination of a level and a facility. The message is like a format string to **printf**, with addition of a **%m** specification, which is replaced with error message corresponding to the current value of **errno**.
- For example, the following call could be issued by a daemon when a call to the **rename** function unexpectedly fails:

```
syslog(LOG_INFO | LOG_LOCAL2, "rename (%s, %s) : %m", file1, file2);
```

# Common Implementations



| Daemon Name | Description                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------|
| syslogd     | Traditional BSD syslog daemon. Lightweight and basic.                                              |
| rsyslogd    | Default on most Linux distros now. Adds filtering, remote logging, and database output.            |
| syslog-ng   | Advanced syslog daemon with better performance and security features.                              |
| journald    | Systemd-based logging (used in modern Linux systems), replaces traditional syslog in many distros. |

## Common Syslog Log Files

| File Path           | Purpose                       |
|---------------------|-------------------------------|
| /var/log/syslog     | General system messages       |
| /var/log/messages   | General log (on some distros) |
| /var/log/auth.log   | Authentication events         |
| /var/log/daemon.log | Logs from daemons             |
| /var/log/kern.log   | Kernel messages               |

# Testing syslog



```
#include <syslog.h>

int main() {
 openlog("myapp", LOG_PID | LOG_CONS, LOG_DAEMON);
 syslog(LOG_INFO, "This is a test log from my daemon.");
 closelog();
 return 0;
}
```

# Log Priorities (Severity Levels)



| Priority Level       | Constant for syslog() | Constant for os_log() | Description                                                                                           |
|----------------------|-----------------------|-----------------------|-------------------------------------------------------------------------------------------------------|
| <b>Emergency</b>     | LOG_EMERG             | OS_LOG_TYPE_FAULT     | System is unusable. Highest severity. Alerts for catastrophic system failures or panic.               |
| <b>Alert</b>         | LOG_ALERT             | OS_LOG_TYPE_ERROR     | Immediate action required. Severe system issues, like disk failure, that require immediate attention. |
| <b>Critical</b>      | LOG_CRIT              | OS_LOG_TYPE_ERROR     | Critical conditions like hardware failures, but not as urgent as "alert."                             |
| <b>Error</b>         | LOG_ERR               | OS_LOG_TYPE_ERROR     | Non-critical issues like application errors, system errors, or recoverable failures.                  |
| <b>Warning</b>       | LOG_WARNING           | OS_LOG_TYPE_INFO      | Conditions that might indicate potential problems (e.g., low disk space, deprecated API usage).       |
| <b>Notice</b>        | LOG_NOTICE            | OS_LOG_TYPE_INFO      | Normal but significant events that are important to log (e.g., routine logins, background tasks).     |
| <b>Informational</b> | LOG_INFO              | OS_LOG_TYPE_INFO      | Informational messages that track the normal operation of the system (e.g., service started).         |
| <b>Debug</b>         | LOG_DEBUG             | OS_LOG_TYPE_DEBUG     | Detailed debug-level information for troubleshooting purposes (e.g., verbose output for developers).  |



# Introduction to P2P programming



# P2P programming

## ► What is Peer-to-Peer (P2P) Programming?

- Peer-to-Peer (P2P) is a decentralized network model in which each participant (peer) can act as both a client and a server.
- Unlike client-server, there is no central server.
- Each node shares and consumes resources (files, messages, etc.).
- Common in file sharing (BitTorrent), communication tools (Skype), blockchain, etc.

## Key Characteristics of P2P

| Feature                 | Description                                |
|-------------------------|--------------------------------------------|
| <b>Decentralization</b> | No single point of failure                 |
| <b>Scalability</b>      | More peers → more capacity                 |
| <b>Resource sharing</b> | Bandwidth, storage, and data shared        |
| <b>Fault tolerance</b>  | Network can continue despite node failures |



# Flow in a P2P Application

## Peer A

socket()

bind()

listen()

connect()

send()/recv() ←exchange()→

close()

## Peer B

socket()

bind()

listen()

connect()

send()/recv()

close()

# Challenges in P2P Programming



- **Peer Discovery:** How does one peer find another? (e.g., bootstrap servers)
- **NAT Traversal:** Many peers are behind routers; connecting across networks may require STUN/TURN.
- **STUN** stands for **Session Traversal Utilities for NAT**. It is a **network protocol** that allows a device behind a NAT (Network Address Translation) or firewall to discover its **public IP address and port** as seen by the outside world.
  - The client (behind NAT) sends a request to a **STUN server** on the public internet.
  - The STUN server replies with:
    - The **public IP and port** it sees for the request.
  - The client now knows:
    - "This is how I appear to the outside world."
  - TURN is used when:
    - STUN fails to establish a direct connection (e.g., due to **symmetric NAT, firewalls, or corporate networks**).
    - Peers can't see or connect to each other directly.
    - A fallback relay is needed to route the data.
    - Without TURN, applications like video calls, chat, or file sharing would **fail in restrictive networks**.
- **Security:** No central control, so encryption, authentication, and trust are vital.
- **Concurrency:** A peer may serve and connect to multiple peers at once → use threads or `select()`.



# Overview Network Security Programming



- **Network Security Programming** involves writing software that ensures **confidentiality, integrity, and availability** of data transmitted over computer networks. It implements mechanisms to **detect, prevent, and respond** to unauthorized access or attacks.

## Goals of Network Security

| Goal                   | Description                                        |
|------------------------|----------------------------------------------------|
| <i>Confidentiality</i> | Ensure that only authorized parties can read data  |
| <i>Integrity</i>       | Prevent unauthorized data modification             |
| <i>Availability</i>    | Ensure services are accessible and usable          |
| <i>Authentication</i>  | Verify identity of communicating parties           |
| <i>Non-repudiation</i> | Prevent denial of actions (e.g., sending messages) |

## Common Threats in Network Communication

- *Eavesdropping (Sniffing)* – Reading data in transit
- *Man-in-the-Middle (MitM)* – Intercepting/modifying communication
- *IP Spoofing* – Pretending to be another host
- *Denial-of-Service (DoS)* – Overloading a service to make it unavailable
- *Replay Attacks* – Re-sending captured data



## Key Concepts & Techniques

| Concept                  | Description                             |
|--------------------------|-----------------------------------------|
| <i>Encryption</i>        | Secure data in transit using ciphers    |
| <i>Digital Signature</i> | Ensure data integrity and authenticity  |
| <i>TLS/SSL</i>           | Encrypts TCP connections (e.g., HTTPS)  |
| <i>VPNs</i>              | Secure tunneling of network traffic     |
| <i>Firewalls</i>         | Filter network traffic                  |
| <i>Hashing</i>           | Generate fingerprint for data integrity |

## Programming Tools & APIs

| Library/Tool            | Usage                                  |
|-------------------------|----------------------------------------|
| <i>OpenSSL</i>          | Encryption, TLS, key generation        |
| <i>GnuTLS</i>           | Secure network communication           |
| <i>libpcap</i>          | Packet capture and monitoring          |
| <i>iptables (Linux)</i> | Firewall scripting                     |
| <i>Socket API</i>       | Core for implementing secure protocols |



1. Create TCP/UDP socket (`socket()`)
2. Setup encryption context (OpenSSL, TLS)
3. Authenticate peers (certificates, keys)
4. Encrypt and transmit data (`SSL_write`)
5. Decrypt and read data (`SSL_read`)
6. Handle errors and clean up (`SSL_free, close`)

## Securing by Hostname or Domain Name



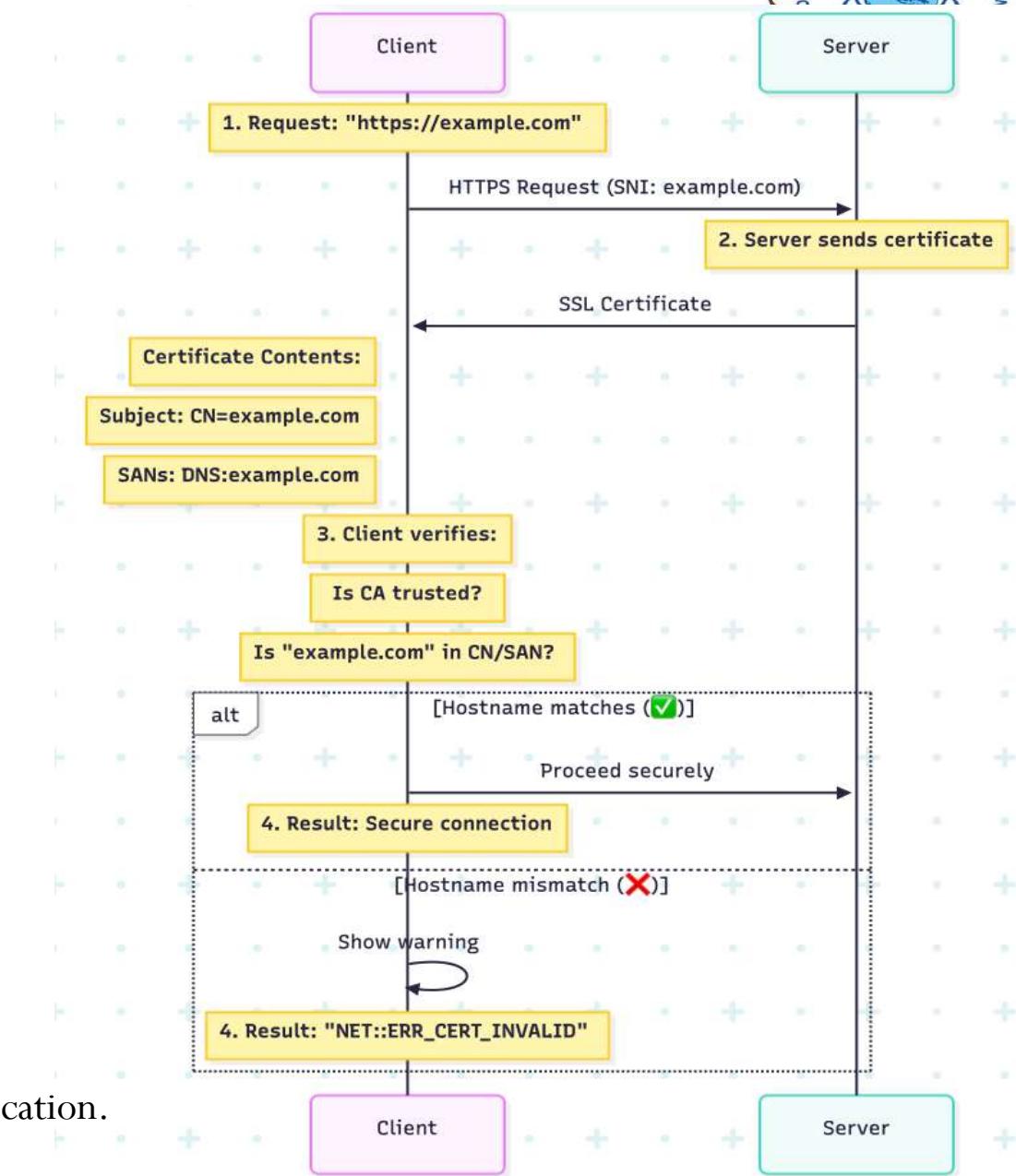
- Hostname verification ensures that a **server's certificate** matches the expected domain.
- Prevents **man-in-the-middle (MitM)** attacks where an attacker may impersonate the server.
- **Common Name (CN)** or **Subject Alternative Name (SAN)** in certificates must match the requested domain.
- Why is Hostname Verification Necessary?
  - **Risk of Impersonation:** Without verification, an attacker could impersonate a trusted server.
  - **Security Layer:** Prevents redirecting communication to a malicious server.
    - **Example:** Client requests <https://securebank.com>, but an attacker replies with a valid certificate for [attacker.com](http://attacker.com).
    - **Outcome:** The client would unknowingly trust the malicious server.

# Securing by Hostname or Domain Name



- Header:** How Does Hostname Verification Work?

- Client sends request:** `https://example.com`
- Server returns certificate:** Contains CN/SAN fields.
- Client compares:** Hostname against CN or SAN in the certificate.
- Connection succeeds:** If they match.
- Connection fails:** If they do not match (MitM prevention).



**Diagram:** The flow of the client-server communication with certificate verification.

Generated by: <https://www.mermaidchart.com/>  
Network Programming (NP)



- **Identification by IP number** refers to verifying the identity of a connecting client or server based on its IP address. This technique can be used as part of **access control** or **connection filtering**, especially in **server-side security** implementations.
- When to Use IP-Based Identification?
  - We control both client and server IP spaces
  - The network is private or internal
  - Simplicity is more important than flexibility
- Do not use it when:
  - Clients move between networks
  - Strong security/authentication is required
- In a real server, we would extract the client IP using `getpeername()` and convert it to string using `inet_ntop()`.
- *Program: [\*\*ip\\_filter\\_server.c\*\*](#)*

## What is a Wrapper Program?



- A **wrapper program**:
  - Runs **before** the actual service,
  - Performs checks like:
    - **IP address filtering**
    - **User ID / privilege checks**
    - **Time of day**
  - If all checks pass, it launches the target service using exec\*() family of system calls.
  - Program: *secure\_wrapper.c*

### Policy Features Can Add (Homework)

- Time of day: `is_allowed_time()`
- User identity: `getuid()`, `getpwuid()`
- Source IP (if used for network access): Check `getpeername()` if this is a socket-based wrapper
- Log every access: Log to a file using `syslog()` or `fprintf()`
- Limit usage frequency: Store access times in a file or memory



# End of Chapter 3



# Network Programming

**BESE-VI – Pokhara University**

**Prepared by:**

**Assoc. Prof. Madan Kadariya (NCIT)**

**Contact: [madan.kadariya@ncit.edu.np](mailto:madan.kadariya@ncit.edu.np)**



# Chapter 4:

# Basics of Winsock Network Programming

## (6 hrs)

# Outline



1. Introduction to Winsock
  - Overview of the Winsock API
  - Differences between Unix and Winsock programming
  - Winsock DLL
    - Setting Up Winsock Environment
    - Initialization: WSAStartup()
    - Clean-up: WSACleanup()
2. Basic Winsock API Functions
  - Socket creation and binding (socket(), bind())
  - Listening and accepting connections (listen(), accept())
  - Sending and receiving data (send(), recv())
  - Closing a socket (closesocket())
  - Functions for handling Blocked IO
  - Asynchronous IO functions
3. Creating Simple TCP/UDP Clients and Servers programs using Winsock



# Introduction to Winsock

# Introduction to Winsock

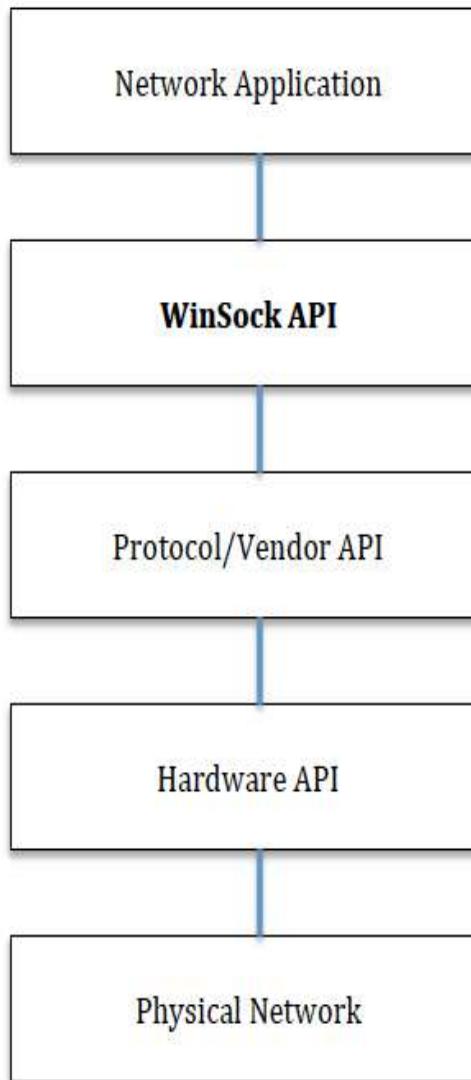


- The *Windows Sockets Application Programming Interface* (WinSock API) is a library of functions that implements the socket interface.
- **Winsock (Windows Sockets API)** is a specification for network programming on **Windows**, providing a **standard interface** for TCP/IP.
- Based on the **Berkeley Sockets API** (from UNIX), but includes Windows-specific extensions.

## Key Features

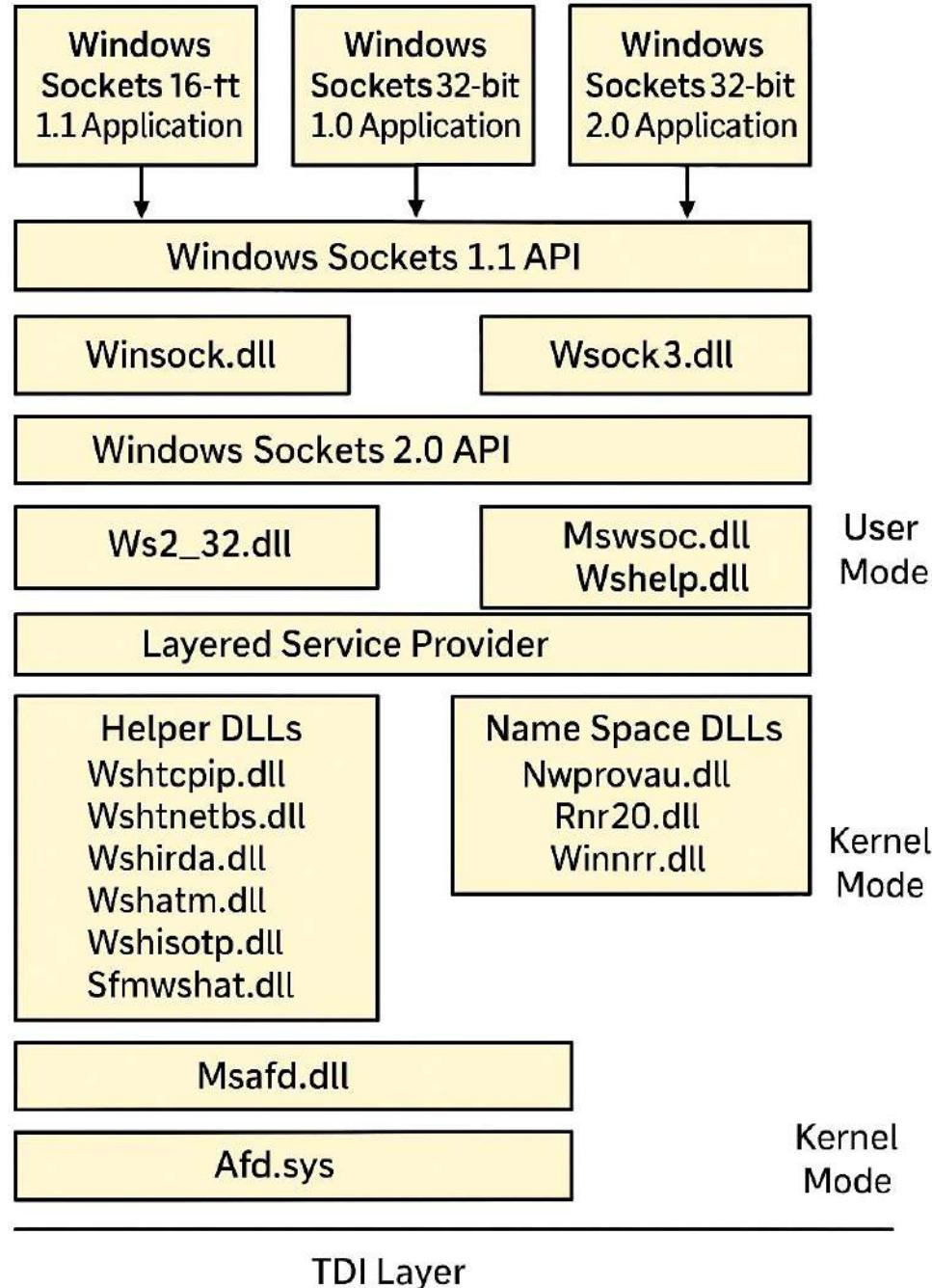
- Support for **TCP/UDP** networking.
- Uses **sockets** as abstractions for communication endpoints.
- Provides **asynchronous (non-blocking)** and **event-driven** models.
- Facilitates **client-server** communication over networks.

# Introduction to Winsock



- The WinSock specification standardizes the interface for TCP/IP stacks, enabling application developers to write code once and run it on any WinSock-compliant implementation.
- Before WinSock, developers had to link their applications to vendor-specific libraries, making it difficult to support multiple TCP/IP stacks due to differences in Berkeley sockets implementations.

# WinSock Architecture



## 1. Application Layer (Top Layer)

- Windows Sockets 16-bit 1.1 Application
- Windows Sockets 32-bit 1.1 Application
- Windows Sockets 32-bit 2.0 Application
  - These are user programs using Winsock for network communication.
  - Each application interfaces with the corresponding version of the Winsock API.

## 2. API Layer

### • Winsock 1.1 API:

- winsock.dll (for 16-bit apps)
- wsock32.dll (for 32-bit apps)

### • Winsock 2.0 API:

- ws2\_32.dll: Primary API for modern network apps
- mswock.dll, wshelp.dll: Extended and helper API support
- This layer provides the application programming interface (API) for sockets.



## 3. Service Provider Interface (SPI)

- **Layered Service Provider**

- Sits between API and protocol drivers.
- Can be extended/overridden by developers for filtering or custom network behavior.
- Communicates with protocol-specific helper DLLs.

## 4. Service Provider Layer

- ◆ **Helper DLLs (Protocol-Specific)**

- Examples:

- wshtcpip.dll – TCP/IP
- wshnetbs.dll – NetBIOS
- wshirda.dll – Infrared
- wshatm.dll, wshisn.dll, wshisotp.dll, sfmwshat.dll

- Implements protocol-specific functionality.

- ◆ **Namespace Provider DLLs (Name Resolution)**

- Examples:

- nwprovau.dll, rnr20.dll, winrnr.dll
- Used for resolving names to network addresses (like DNS or directory services).



## 5. Base Service Provider

- **Msafd.dll**
  - Microsoft Ancillary Function Driver: Bridges user-mode to kernel networking.
- **Afd.sys**
  - Kernel-mode driver for the Windows socket subsystem.

## 6. Kernel Layer (Bottom Layer)

- **TDI Layer (Transport Driver Interface)**
  - Interfaces with actual transport protocols like TCP/IP, NetBIOS, etc.
  - Now deprecated but crucial in legacy systems.

# WinSock Architecture

| Layer                  | Components                                 | Role                                         |
|------------------------|--------------------------------------------|----------------------------------------------|
| Application Layer      | Winsock Apps (16-bit, 32-bit, 2.0)         | Uses Winsock APIs                            |
| API Layer              | winsock.dll, wsock32.dll, ws2_32.dll, etc. | Provides socket functions                    |
| SPI Layer              | Layered Service Provider                   | Intermediary between API and protocols       |
| Service Provider Layer | Helper DLLs, Namespace DLLs                | Implements protocols and name resolution     |
| Base Provider Layer    | msafd.dll, afd.sys                         | Socket support and kernel bridge             |
| Kernel Layer           | TDI Layer                                  | Interfaces with transport protocols (legacy) |



# Dynamic Linking

# Dynamic Linking



- Dynamic linking refers to loading and linking libraries at **runtime** rather than **compile time**. It allows executables to use **shared code** from external modules.

## 1. Implicit Dynamic Linking (load-time dynamic linking)

- The application is **linked to DLLs during program startup**.
- DLLs are specified at **compile/link time** and automatically loaded by the OS when the application starts.

```
#include <windows.h>
#pragma comment(lib, "user32.lib") // linked at load-time
```

### Advantages:

- Simple to use.
- Errors due to missing functions are detected during startup.
- Less manual code to manage linking.

### Disadvantages:

- If DLL is missing or corrupted, the program fails to start.
- All DLLs are loaded even if not all functions are used.

# Dynamic Linking



## 2. Explicit Dynamic Linking(run-time dynamic linking)

- The application **loads the DLL manually at runtime** using functions like ***LoadLibrary()*** and ***GetProcAddress()***.

```
HMODULE hLib = LoadLibrary("user32.dll");
```

```
FARPROC msgBox = GetProcAddress(hLib, "MessageBoxA");
```

### Advantages:

- Load DLLs only when needed → reduces memory usage.
- Better error handling (can continue if DLL is missing).
- Enables plugin architecture and modular programs.

### Disadvantages:

- More complex code.
- Runtime errors if function names or DLL paths are incorrect.
- Performance overhead due to dynamic lookup.

# Dynamic Linking



## General Advantages of Dynamic Linking

- Saves **memory**: Shared libraries used by multiple processes.
- Saves **disk space**: No need to bundle libraries with every program.
- Allows **updates** without recompiling applications.
- Reduces **program size**.

## General Disadvantages of Dynamic Linking

- **Dependency issues**: If a required DLL is missing or incompatible → program fails.
- **Security risk**: Susceptible to DLL hijacking or injection attacks.
- **Performance hit**: Function lookups are slower than static linking.



# Winsock vs Berkeley Socket

# Winsock vs Berkeley Socket



| Feature / Aspect       | Unix Sockets (POSIX/BSD)                                      | Winsock (Windows Sockets API)                                                                           |
|------------------------|---------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Platform               | Unix, Linux, macOS                                            | Windows only                                                                                            |
| Initialization         | None                                                          | WSAStartup() / WSACleanup() required before/after any socket calls                                      |
| Header Files           | <sys/socket.h>, <netinet/in.h>, <arpa/inet.h>                 | <winsock2.h>, <ws2tcpip.h>                                                                              |
| Library Linking        | Standard C library                                            | Ws2_32.lib (and optionally Mswock.lib)                                                                  |
| Socket Descriptor Type | int                                                           | SOCKET (an opaque unsigned type); invalid value is INVALID_SOCKET                                       |
| Close Function         | close(fd)                                                     | closesocket(sock)                                                                                       |
| Error Reporting        | errno (e.g. EAGAIN, ECONNREFUSED)                             | WSAGetLastError() returns Winsock-specific codes (e.g. WSAEWOULDBLOCK, WSAECONNREFUSED)                 |
| Blocking / Nonblocking | fcntl() or ioctl(fd, FIONBIO, &flag)                          | ioctlsocket()                                                                                           |
| Asynchronous I/O       | select(), poll(), epoll, nonblocking + signals (SIGIO), aio_* | Overlapped I/O (WSASend, WSARcv + OVERLAPPED), WSAAsyncSelect(), WSAEventSelect(), I/O Completion Ports |

# Winsock vs Berkeley Socket



| Feature / Aspect          | Unix Sockets (POSIX/BSD)                                                             | Winsock (Windows Sockets API)                                                                             |
|---------------------------|--------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| select() Support          | Works on any file descriptor: sockets, pipes, files.                                 | <b>Only</b> on sockets. FD_SETSIZE default limit 64 (can be recompiled). No monitoring of stdin or pipes. |
| Socket Options API        | setsockopt(), getsockopt()                                                           | Same calls, but option names and levels (e.g. SOL_SOCKET, IPPROTO_TCP) can differ slightly.               |
| Name Resolution           | getaddrinfo(), gethostbyname()                                                       | getaddrinfo(), gethostbyname(), plus Win32 DNS APIs                                                       |
| IPv6 Support              | Native from Linux kernel ≥2.2 / BSD variants                                         | Supported since Winsock 2.0                                                                               |
| File I/O Semantics        | Sockets are file descriptors → can use read(), write(), dup(), select()              | Sockets are not normal file descriptors; must use Winsock calls (send(), recv())                          |
| Signals                   | SIGPIPE on write to closed socket (can be disabled with MSG_NOSIGNAL)                | No SIGPIPE; write to closed socket returns SOCKET_ERROR+ WSAGetLastError() == WSAECONNRESET               |
| IPC Mechanisms            | Sockets plus pipes / FIFOs, System V IPC (message queues, semaphores, shared memory) | Sockets plus Windows IPC: Named Pipes, Mailslots, LPC/ALPC, Shared Memory, RPC, COM                       |
| DLL Injection / Hijacking | N/A (shared libs loaded at launch)                                                   | Risk of DLL pre-loading/hijacking if path not fully qualified                                             |

# Winsock vs Berkeley Socket



## Behavior on Signal / Async Events

- **Unix:** Interrupted by signals (EINTR).
- **Winsock:** Not signal-driven; overlapped or window-message notifications via `WSAAAsyncSelect()`.

## Summary of Key Conceptual Differences:

- **Initialization:** Winsock needs manual startup/cleanup; Unix doesn't.
- **Error Reporting:** Unix uses global `errno`; Winsock uses function `WSAGetLastError()`.
- **Close Operation:** Unix uses `close()`, Winsock uses `closesocket()` to avoid interfering with file descriptors.
- **Handles:** Unix sockets are regular file descriptors; Winsock sockets are distinct from file I/O.



# Setting Up Winsock Environment

# Setting Up Winsock Environment



## Windows Socket Extension: Setup and Cleanup Function

- The WinSock functions the application needs are located in the dynamic library named **WINSOCK.DLL** or **WSOCK32.DLL** depending on whether the 16-bit or 32-bit version of Windows is being targeted.
- The application is linked with either **WINSOCK.LIB** or **WSOCK32.LIB** as appropriate.
- The include file where the WinSock functions and structures are defined is named **WINSOCK.H** for both the 16-bit and 32-bit environments.
- Before the application uses any WinSock functions, the application must call an initialization routine called **WSAStartup()**.
- Before the application terminates, it should call the **WSACleanup()** function.

- The *WSAStartup()* function initializes the underlying Windows Sockets Dynamic Link Library(WinSock DLL).
- The *WSAStartup()* function gives the TCP/IP stack vendor a chance to do any application-specific initialization that may be necessary.
- *WSAStartup()* is also used to confirm that the version of the WinSock DLL is compatible with the requirements of the application.
- *Header*: #include<winsock2.h>
- *Library*: link against *Ws2\_32.lib*

```
int WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData);
```

```
/* Returns 0 on success. On failure, returns a nonzero error
code (e.g. WSASYSNOTREADY, WSAVERNOTSUPPORTED, WSAEINPROGRESS) */
```

- *wVersionRequired*: The highest version of Winsock your application can use, encoded as *MAKEWORD(major, minor)*
- *lpWSAData*: Pointer to a WSADATA structure that will be filled in with details of the Winsock implementation loaded.

# WSADATA



```
#define WSADESCRIPTION_LEN 256
#define WSASYS_STATUS_LEN 128
typedef struct WSADATA {
 WORD wVersion; // Winsock version requested by the application
 WORD wHighVersion; // Highest version supported by the loaded DLL
 char szDescription[WSADESCRIPTION_LEN + 1]; // Description string
 char szSystemStatus[WSASYS_STATUS_LEN + 1]; // Additional status (often unused)
 unsigned short iMaxSockets; // Maximum number of sockets supported
 unsigned short iMaxUdpDg; // Maximum datagram size for UDP
 char *lpVendorInfo; // Pointer to vendor-specific info (can be `NULL`)
} WSADATA, *LPWSADATA;
```

- **wVersion:** Holds the Winsock version granted to the application (low byte = major, high byte = minor).
- **wHighVersion:** Highest version of Winsock the underlying DLL can support.
- **szDescription:** A null-terminated ANSI string describing the Winsock implementation
- **szSystemStatus:** A null-terminated ANSI string for implementation-specific status
- **iMaxSockets:** On Windows 9x, indicates the maximum simultaneous sockets supported.
- **iMaxUdpDg:** On Windows 9x, maximum size in bytes of a UDP datagram.
- **lpVendorInfo:** If non-NUL, points to vendor-specific data or version strings. Rarely used; can be treated as opaque.



# Usage Example

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

int main(void) {
 WSADATA wsaData;
 int err;
 // Initialize Winsock version 2.2
 err = WSAStartup(MAKEWORD(2,2), &wsaData);
 if (err != 0) {
 printf("WSAStartup failed: %d\n", err);
 return 1;
 }
 printf("Winsock initialized:\n");
 printf(" Version: %d.%d\n",
 LOBYTE(wsaData.wVersion),
 HIBYTE(wsaData.wVersion));
 printf(" Description: %s\n", wsaData.szDescription);
 // ... perform socket operations ...
 WSACleanup();
 return 0;
}
```

# WSACleanup()



- The *WSACleanup()* function is used to terminate an application's use of WinSock.
- For every call to *WSAStartup()* there has to be a matching call to *WSACleanup()*.
- *WSACleanup()* is usually called after the application's message loop has terminated.
- Purpose is to releases resources allocated by Winsock and decrements the per-process reference count of the Winsock DLL. When the reference count reaches zero, the Winsock implementation is unloaded.
- *Header*: #include<winsock2.h>
- *Library*: link against *Ws2\_32.lib*

```
int WSACleanup(void);
/* Returns 0 on success SOCKET_ERROR on failure; call WSAGetLastError() to retrieve
the error code (e.g. WSANOTINITIALISED, WSAENETDOWN). */
```

## WSAGetLastError()



- The *WSAGetLastError()* function doesn't deal exclusively with startup or shutdown procedures, but it needs to be addressed early. Its function prototype looks like

```
int WSAGetLastError(void);
```

- WSAGetLastError()* returns the last WinSock error that occurred. Because WinSock isn't really part of the operating system but is instead a later add-on, errno (like in UNIX and MS-DOS) couldn't be used.
- As soon as a WinSock API call fails, the application should call *WSAGetLastError()* to retrieve specific details of the error.
- Example:

```
int result = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
if (result == SOCKET_ERROR) {
 int err = WSAGetLastError(); // Handle or report 'err'
}
```

- Always call *WSAGetLastError()* immediately after a Winsock call returns an error.
- Do not call other Winsock functions before retrieving the error, as they may overwrite the error code.

# Few Error Codes



| Error Code        | Value | Description                                    |
|-------------------|-------|------------------------------------------------|
| WSAEINTR          | 10004 | Interrupted function call                      |
| WSAEBADF          | 10009 | Bad file/socket descriptor                     |
| WSAEACCES         | 10013 | Permission denied                              |
| WSAEFAULT         | 10014 | Bad address                                    |
| WSAEINVAL         | 10022 | Invalid argument                               |
| WSAEMFILE         | 10024 | Too many open sockets                          |
| WSAEWOULDBLOCK    | 10035 | Resource temporarily unavailable (nonblocking) |
| WSAEINPROGRESS    | 10036 | Operation now in progress                      |
| WSAENOTSOCK       | 10038 | Descriptor is not a socket                     |
| WSADESTADDRREQ    | 10039 | Destination address required                   |
| WSAEADDRINUSE     | 10048 | Address already in use                         |
| WSAECONNRESET     | 10054 | Connection reset by peer                       |
| WSAETIMEDOUT      | 10060 | Connection timed out                           |
| WSAECONNREFUSED   | 10061 | Connection refused                             |
| WSAHOST_NOT_FOUND | 10001 | Authoritative answer host not found            |



# Basic Winsock API Functions

## Socket Creation and Binding

- **socket()**

```
SOCKET socket(
 int af, // Address family (e.g. AF_INET, AF_INET6)
 int type, // Socket type (SOCK_STREAM, SOCK_DGRAM)
 int protocol // Protocol (IPPROTO_TCP, IPPROTO_UDP, or 0)
);
```

**Returns:** A *SOCKET* handle on success, or *INVALID\_SOCKET* on error.

**Errors:** Call *WSAGetLastError()* for codes like *WSAENRTDOWN*, *WSAEAFNOSUPPORT*. \*/

- **bind():** Assigns a local address (IP + port) to a socket.

```
int bind(SOCKET s, const struct sockaddr *addr, int addrlen);
```

**Returns:** 0 on success; *SOCKET\_ERROR* on failure

**Errors:** *WSAEADDRINUSE*, *WSAEINVAL*, etc \*/

## Example of socket() and bind()

```
SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in local = {0};
local.sin_family = AF_INET;
local.sin_port = htons(12345);
local.sin_addr.s_addr = INADDR_ANY;
if (bind(s, (SOCKADDR*)&local, sizeof(local)) == SOCKET_ERROR) {
 printf("bind failed: %d\n", WSAGetLastError());
 closesocket(s);
 WSACleanup();
 return 1;
}
```

## Establishing a Connection

- **connect():**

```
int connect(SOCKET s, const struct sockaddr *name, int namelen);
/* Returns: 0 on success (connection established).
 SOCKET_ERROR on failure; call WSAGetLastError() to get error code*/
```

## Listening and Accepting Connections

- **listen():** Marks a bound TCP socket as passive, ready to accept incoming connections.

```
int listen(SOCKET s, int backlog);
/* Returns: 0 on success; SOCKET_ERROR on failure
Errors: WSAEINVAL, WSAENOTSOCK. */
```

- **accept():** Extracts the first pending connection, returning a new connected socket.

```
SOCKET accept(SOCKET s, struct sockaddr *addr, int *addrlen);
/* Returns: New SOCKET on success; INVALID_SOCKET on failure.
Errors: WSAEWOULDBLOCK (nonblocking), WSAENOTSOCK, etc*/
```

## Example of listen() and accept()

```
listen(s, 5);
SOCKET client;
struct sockaddr_in peer;
int peerlen = sizeof(peer);
while ((client = accept(s, (SOCKADDR*) &peer, &peerlen)) != INVALID_SOCKET) {
 // handle client...
}
```

## Sending and Receiving Data

- **send()** and **recv()**:

```
int send(SOCKET s, const char *buf, int len, int flags);
int recv(SOCKET s, const char *buf, int len, int flags);
```

- flags is usually 0 or MSG\_OOB, MSG\_DONTROUTE

**/\*Returns:** Number of bytes sent/received(0=Connection closed),or SOCKET\_ERROR.

**Errors:** WSAEWOULDBLOCK, WSAECONNRESET, etc. \*/

## Closing a Socket

- **closesocket()**: Gracefully closes a socket handle, releases resources.

```
int closesocket(SOCKET s);
```

**/\*Returns:** 0 on success; SOCKET\_ERROR on failure.

**Errors:** WSAENOTSOCK, WSAENETDOWN, etc. \*/

### Note:

- Always call **closesocket()** before **WSACleanup()**
- After closing, the SOCKET handle is invalid and must not be reused.

### Example:

```
char buffer[512];
int bytes = recv(client, buffer, sizeof buffer, 0);
if (bytes > 0) {
 // process data
 send(client, buffer, bytes, 0);
}
closesocket(client);
```

# Flow of TCP Server and Client



## TCP Server

1. WSAStartup()
2. socket()
3. bind()
4. listen()
5. Loop:
  - accept()
  - recv()/send()
  - closesocket(client)
6. closesocket(listening)
7. WSACleanup()

## TCP Client

1. WSAStartup()
2. socket()
3. connect()
4. send()/recv()
5. closesocket()
6. WSACleanup()



# Functions for Handling Blocking I/O

- Winsock provides several mechanisms to control and react to blocking behavior on sockets.
- These let us multiplex many sockets or integrate with Windows' event and message systems.

- **ioctlsocket()** : Set Blocking/Nonblocking Mode

```
int ioctlsocket(SOCKET s,
 long cmd, // e.g. FIONBIO
 u_long *argp // 0 = blocking, nonzero = nonblocking
) ; s
```

- **FIONBIO**: turn nonblocking mode on/off.

```
/* Returns 0 on success, SOCKET_ERROR on failure
(WSAGetLastError() gives WSAEINVAL, WSAENOTSOCK, etc.). */
```

- **Example:**

```
u_long mode = 1; // nonblocking
ioctlsocket(sock, FIONBIO, &mode);
```

- **select()** – I/O Multiplexing

```
int select(
 int nfds, // ignored by Winsock; set FD_SETSIZE before compile
 fd_set *readfds, // sockets to check for readability
 fd_set *writefds, // sockets to check for writability
 fd_set *exceptfds, // sockets to check for errors/out-of-band data
 const struct timeval *timeout // NULL = block indefinitely
) ;
```

- **Monitors** multiple sockets at once for readiness.

**/\*Returns number of sockets ready, 0 on timeout, SOCKET\_ERROR on failure.\*/**

## Notes:

- Only sockets (not pipes/files).
- Default FD\_SETSIZE=64; raise by #define FD\_SETSIZE before including headers.

- **WSAAsyncSelect()** - Message-Driven Notification
- *WSAAsyncSelect()* is used to solve the problem of blocking socket function calls. It is a much more natural solution to the problem than using `ioctlsocket()` and `select()`.
- It works by sending a Windows message to notify a window of a socket event.

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, u_int wMsg, long lEvent);
```

- **s** is the socket descriptor for which event notification is required.
- **hWnd** is the Window handle that should receive a message when an event occurs on the socket.
- **wMsg** is the message to be received by **hWnd** when a socket event occurs on socket **s**. It is usually a user-defined message (WM\_USER + n).
- **lEvent** is a bitmask that specifies the events in which the application is interested.
- **WSAAsyncSelect()** returns 0 (zero) on success and **SOCKET\_ERROR** on failure. On failure, **WSAGetLastError()** should be called.

- *WSAAsyncSelect()* is capable of monitoring several socket events.

## Event Meaning

- FD\_READ Socket ready for reading
  - FD\_WRITE Socket ready for writing
  - FD\_OOB Out-of-band data ready for reading on socket
  - FD\_ACCEPT Socket ready for accepting a new incoming connection
  - FD\_CONNECT Connection on socket completed
  - FD\_CLOSE Connection on socket has been closed
- 
- The IEvent parameter is constructed by doing a logical OR on the events in which we are interested. To cancel all event notifications, call *WSAAsyncSelect()* with wMsg and IEvent set to 0.

- **WSAEventSelect()** – Event-Handle Notification

```
int WSAEventSelect (
 SOCKET s,
 WSAEVENT hEventObject, // Created by WSACreateEvent ()
 long lNetworkEvents // Same bitmask as WSAAsyncSelect
);
```

- Associates a Winsock event object with a socket.
- When any requested network event occurs, the event object is signaled.

**/\*Returns 0 on success, SOCKET\_ERROR on failure.\*/**

## Wait on the event with:

```
WSAWaitForMultipleEvents (
 DWORD cEvents,
 const WSAEVENT *lphEvents,
 BOOL fWaitAll,
 DWORD dwTimeout,
 BOOL fAlertable
);
```



# Asynchronous Database Functions

- The function of `gethostbyname()` is to take a host name and return its IP address.

```
struct hostent * gethostbyname(const char * name);
```

- Its asynchronous counterpart function is `WSAAsyncGetHostByName()`.

```
HANDLE WSAAsyncGetHostByName(HWND hwnd, u_int wMsg, const char *name, char *buf, int buflen);
```

- hWnd** is the handle to the window to which a message will be sent.
- wMsg** is the message that will be posted to **hWnd**
- name** is a pointer to a string that contains the host name
- buf** is a pointer to an area of memory that, on successful completion of the host name lookup, will contain the **hostent** structure for the desired host.
- buflen** is the size of the buf buffer. It should be MAXGETHOSTSTRUCT for safety's sake.
- If the asynchronous operation is initiated successfully, the return value is a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero), and `WSAGetLastError()` should be called to find out the reason for the error.

# WSACancelAsyncRequest()



- *WSACancelAsyncRequest()* can be used to cancel the asynchronous winsock calls.

```
int WSACancelAsyncRequest (HANDLE hAsyncTaskHandle);
```

- **hAsyncTaskHandle:** The asynchronous task handle returned by one of the *WSAAsyncGetXByY* functions (such as *WSAAasyncGetHostName*, *WSAAasyncGetHostByAddr*, *WSAAasyncGetServByName*, etc.).
- On success, this function returns 0 (zero).
- On failure, it returns SOCKET\_ERROR, and **WSAGetLastError()** can be called.

# WSAAsyncGetHostByAddr()

- The function of *gethostbyaddr()* is to take the IP address of a host and return its name.
- *struct hostent \*PASCAL gethostbyaddr(const char \* addr, int len, int type);*
- Its asynchronous counterpart function is *WSAAsyncGetHostByAddr()*

*HANDLE WSAAsyncGetHostByAddr (HWND hWnd, u\_int wMsg, const char\* addr, int len, int type, char\* buf, int buflen);*

- **hWnd** is the handle to the window to which a message.
- **wMsg** is the user defined message that will be posted to **hWnd**
- **addr** is a pointer to the IP address
- **len** is the length of the address to which **addr** points
- **buf** is a pointer to an area of memory that contains the hostent structure for the desired host.
- **buflen** is the size of the buf buffer.
- On Success returns a handle to the asynchronous task. On failure of initialization, the function returns 0 (zero) and WSAGetLastError() should be called to find out the reason for the error.

# WSAAsyncGetServByName()



- The **getservbyname()** function gets service information corresponding to a specific service name and protocol.

```
struct servent *getservbyname(const char* name,const char * proto);
```

- WSAAsyncGetServByName()** is its asynchronous counterpart to `getservbyname()`.

```
HANDLE WSAAsyncGetServByName(HWND hWnd,u_int wMsg,const char* name,
const char* proto,char* buf,int buflen);
```

- hWnd** is the handle to the window to which a message will be sent.
- wMsg** is the user defined message that will be posted to hWnd
- name** is a pointer to a service name about which you want service information.
- If **proto** is NULL, the first matching service is returned.
- buf** is a pointer to an area of memory
- buflen** is the size of the buf buffer.
- It should be MAXGETHOSTSTRUCT for safety's sake.

- The **getservbyport()** function gets service information corresponding to a specific port and protocol.

```
struct servent FAR* PASCAL getservbyport(int port, const char FAR* proto);
```

- **WSAAAsyncGetServByPort()** is the asynchronous counterpart to getservbyport().

```
HANDLE PASCAL FAR WSAAsyncGetServByPort (HWND hWnd, u_int wMsg, int port,
const char FAR* proto, char FAR *buf, int buflen);
```

- **hWnd** is the handle to the window to which a message will be sent.
- **wMsg** is the user defined message that will be posted to
- **port** is the service port in network byte order
- If **proto** is NULL, the first matching service is returned.
- **buf** is a pointer to an area of memory
- **buflen** is the size of the buf buffer
- If the asynchronous operation is initiated successfully, the return value of WSAAsyncGetServByName() is a handle to the asynchronous task. On failure, the function returns 0 (zero).

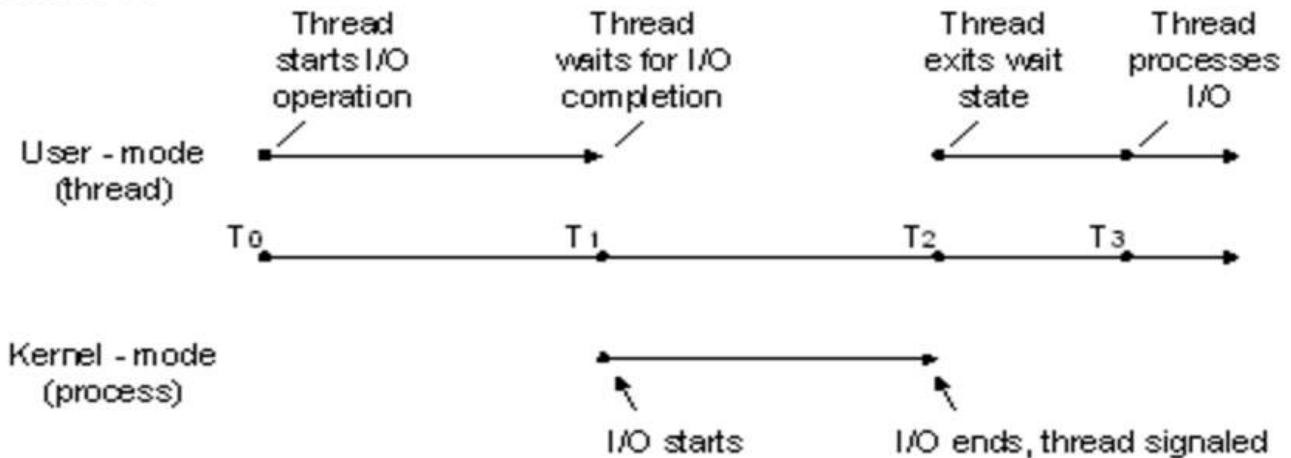


# Asynchronous IO Functions

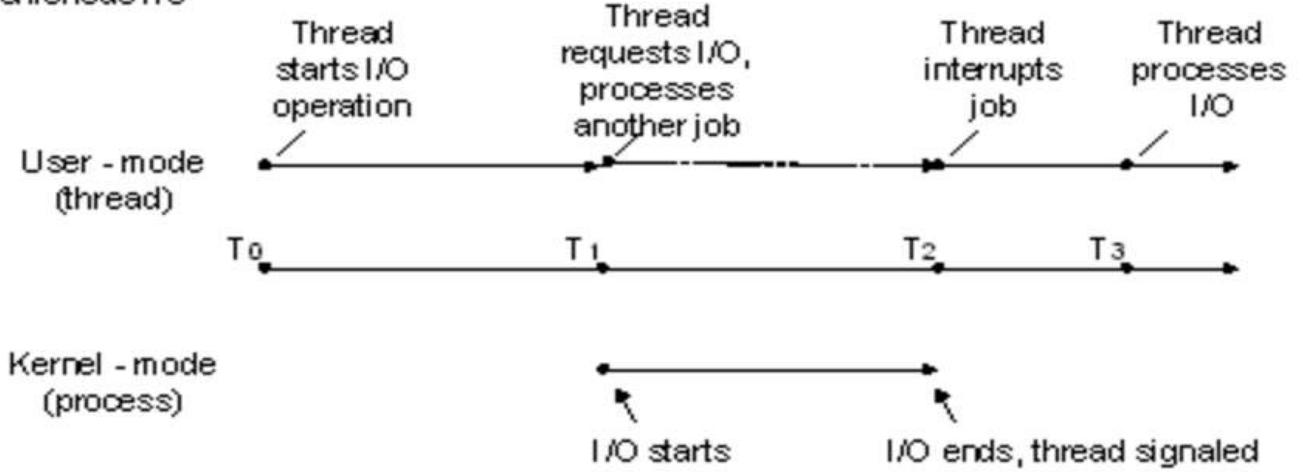
# Synchronous and asynchronous IO



## Synchronous I/O



## Asynchronous I/O



- **I/O Modes:** There are two I/O synchronization types: *synchronous I/O*, where a thread blocks until an operation completes, and *asynchronous (overlapped) I/O*, where a thread issues a request and continues working until notified of completion.
- **Asynchronous Workflow:** In overlapped I/O, a thread calls functions like *WSASend/WSARecv*; if the kernel accepts the request, the thread proceeds with other tasks until the kernel signals that the I/O is done.
- **Use Cases:** Asynchronous I/O boosts efficiency for long-running operations (e.g., large database backups or slow links) by avoiding idle waits.
- **Trade-Offs:** For short, fast I/O operations, the extra kernel signaling overhead may outweigh benefits, making synchronous I/O preferable.
- **Winsock Support:** Winsock APIs (*WSASend*, *WSARecv*, *WSASendTo*, *WSARecvFrom*, *WSAIoctl*, etc.) support both modes and *WSASocket*(..., *WSA\_FLAG\_OVERLAPPED*) is used to create sockets for overlapped operations.

- Create **overlapped** sockets necessary for high-performance asynchronous I/O
- Select a specific **service provider** via IpProtocolInfo.

```
SOCKET WSASocket(
 In int af,
 In int type,
 In int protocol,
 In LPWSAPROTOCOL_INFO lpProtocolInfo,
 In GROUP g,
 In DWORD dwFlags); // Used to create overlapped socket
```

- **af[in]** : address family-AF\_INET, AF\_INET6, AF\_UNSPEC
- **type[in]** : type of the new socket; SOCK\_STREAM, SOCK\_DGRAM, SOCK\_RAW
- **protocol[in]** : IPPROTO\_TCP, IPPROTO\_UDP, IPPROTO\_ICMP or 0
- **IpProtocolInfo[in]** : A pointer to a WSAPROTOCOL\_INFO structure that defines the characteristics of the socket to be created.
- **g [in]** : An existing socket group ID.
- **dwFlags[in]** : A set of flags used to specify additional socket attributes.

```
int WSASend(
 In SOCKET
 In LPWSABUF
 In DWORD
 Out LPDWORD
 In DWORD
 In LPWSAOVERLAPPED
 In LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

- **s[in]** : A descriptor that identifies a connected socket.
- **lpBuffers[in]** : A pointer to an array of WSABUF structures.
- **dwBufferCount[in]** :The number of WSABUF structures in the lpBuffers array.
- **lpNumberOfBytesSent[out]** :The pointer to the number, in bytes, sent by this call if the I/O operation completes immediately. NULL for overlapped socket.
- **dwFlags[in]** :The flags used to modify the behavior of the WSASend function call.
- **lpOverlapped[in]** :A pointer to a WSAOVERLAPPED structure. This parameter is ignored for non-overlapped sockets.
- **lpCompletionRoutine[in]** : A pointer to the completion routine called when the send operation has been completed. This parameter is ignored for non-overlapped sockets.



```
int WSASendTo (
 In SOCKET s,
 In LPWSABUF lpBuffers,
 In DWORD dwBufferCount,
 Out LPDWORD lpNumberOfBytesSent,
 In DWORD dwFlags,
 In const struct sockaddr *lpTo,
 In int iToLen,
 In LPWSAOVERLAPPED lpOverlapped,
 In LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

- **s[in]** : A descriptor identifying a socket.
- **lpBuffers[in]**: A pointer to an array of WSABUF structures.
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpOverlapped[in]**: A pointer to a WSAOVERLAPPED structure
- **lpCompletionRoutine[in]**:A pointer to the completion routine called when the send operation has been completed.
- **lpNumberofBytesSent[out]**:The pointer to the number.
- **dwFlags[in]** :The flags used to modify the behavior of the WSASendTo call.
- **lpTo[in]** : An optional pointer to the address of the target socket in the SOCKADDR structure.
- **iToLen [in]** :The size, in bytes, of the address in the lpTo parameter.

```
int WSARecv(
 In SOCKET
 Inout LPWSABUF
 In DWORD
 Out LPDWORD
 Inout LPDWORD
 In LPWSAOVERLAPPED
 In LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

- **s[in]**: A descriptor identifying a connected socket.
- **lpBuffers[in, out]** : A pointer to an array of WSABUF structures
- **dwBufferCount[in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberOfBytesRecvd[out]** : A pointer to the number data received in bytes.
- **lpFlags [in, out]** : A pointer to flags used to modify the behavior of the WSARecv function call.
- **lpOverlapped[in]** : A pointer to a WSAOVERLAPPED structure.
- **lpCompletionRoutine[in]** : A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets).

# WSARecvFrom()



```
int WSARecvFrom(
 In SOCKET s,
 Inout LPWSABUF lpBuffers,
 In DWORD dwBufferCount,
 Out LPDWORD lpNumberOfBytesRecvd,
 Inout LPDWORD lpFlags,
 Out struct sockaddr *lpFrom,
 Inout LPINT lpFromlen,
 In LPWSAOVERLAPPED lpOverlapped,
 In LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

- **s [in]** : A descriptor identifying a socket.
- **lpBuffers [in, out]** : A pointer to an array of WSABUF structures
- **dwBufferCount [in]** : The number of WSABUF structures in the lpBuffers array.
- **lpNumberOfBytesRecvd[out]** : A pointer to the number of bytes received by this call
- **lpFlags [in, out]** : A pointer to flags used to modify the behavior of this function.
- **lpFrom [out]** : An optional pointer to a buffer that will hold the source address
- **lpFromlen [in, out]** : A pointer to a WSAOVERLAPPED structure
- **lpCompletionRoutine [in]** : A pointer to the completion routine called when the WSARecvFrom operation has been completed

- Performs network I/O control operations on a socket—setting modes, querying status, or retrieving extension function pointers for overlapped I/O helpers.

```
int WSAIoctl(
 In SOCKET s,
 In DWORD dwIoControlCode,
 In LPVOID lpvInBuffer,
 In DWORD cbInBuffer,
 Out LPVOID lpvOutBuffer,
 In DWORD cbOutBuffer,
 Out LPDWORD lpcbBytesReturned,
 In LPWSAOVERLAPPED lpOverlapped,
 In LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

- **s [in]** : A descriptor identifying a socket.
- **dwIoControlCode [in]** : The control code of operation to perform
- **lpInBuffer [in]** : A pointer to the input buffer
- **cbInBuffer [in]** : The size, in bytes, of the input buffer
- **lpOutBuffer [out]** : A pointer to the output buffer
- **cbOutBuffer [in]** : The size, in bytes, of the output buffer
- **lpcbBytesReturned [out]** : A pointer to actual number of bytes of output
- **lpOverlapped [in]** : A pointer to a WSAOVERLAPPED structure
- **lpCompletionRoutine [in]** : A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets)



# Getting Started with WinSock



# Setting up Environment

## 1. Install a C compiler

1. Visual Studio (Community Edition) or
2. MinGW-w64 with GCC

## 2. Ensure Windows SDK is available

1. Contains headers (winsock2.h, ws2tcpip.h) and import library (Ws2\_32.lib).

## 3. Create a project

1. In Visual Studio: New → Win32 Console Application → blank project.
2. In MinGW/GCC: Create a .c file and compile via CLI.



# General model for creating a streaming TCP/IP Server and Client.

## Client

1. Initialize Winsock.
2. Create a socket.
3. Connect to the server.
4. Send and receive data.
5. Disconnect.

## Server

1. Initialize Winsock.
2. Create a socket.
3. Bind the socket.
4. Listen on the socket for a client.
5. Accept a connection from a client.
6. Receive and send data.
7. Disconnect.



## Include Headers & Link Libraries

// Before any <windows.h> include:

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
```

// Link against Ws2\_32.lib

// - Visual Studio: add Ws2\_32.lib in Linker → Input → Additional Dependencies  
// - MinGW/GCC: gcc myapp.c -lws2\_32 -o myapp.exe



## Include Headers & Link Libraries

// Before any <windows.h> include:

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
```

// Link against Ws2\_32.lib

// - Visual Studio: add Ws2\_32.lib in Linker → Input → Additional Dependencies  
// - MinGW/GCC: gcc myapp.c -lws2\_32 -o myapp.exe

# Initializing Winsock



1. Create a WSADATA object called wsaData.

```
WSADATA wsaData;
```

2. Call WSAStartup and return its value as an integer and check for errors.

```
int iResult;
// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
 printf("WSAStartup failed: %d\n", iResult);
 return 1;
}
```

- **MAKEWORD(2,2)** requests Winsock version 2.2.
- On success, wsaData holds implementation details.

# Creating a Socket for the Client



## 3. Create a socket

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == INVALID_SOCKET) {
 printf("socket() failed: %d\n", WSAGetLastError());
 WSACleanup();
 return 1;
}
```

- **AF\_INET** = IPv4, **SOCK\_STREAM** = TCP.
- **INVALID\_SOCKET** indicates failure.

## 4. Prepare Server Address

```
struct sockaddr_in serverAddr;
ZeroMemory(&serverAddr, sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080);
inet_pton(AF_INET, "127.0.0.1", &serverAddr.sin_addr);
```

**htons()** converts port to network byte order.  
**inet\_pton()** converts dotted -decimal string to binary.

## 5. Connect (Client) / Bind & Listen (Server)



### Client: connect()

```
if (connect(sock, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR)
{
 printf("connect() failed: %d\n", WSAGetLastError());
 closesocket(sock);
 WSACleanup();
 return 1;
}
printf("Connected to server!\n");
```

### Server: bind() + listen() + accept()

```
bind(sock, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
listen(sock, SOMAXCONN);

SOCKET client = accept(sock, NULL, NULL);
if (client != INVALID_SOCKET) {
 printf("Client connected!\n");
 // use 'client' for send/recv
}
```

## 6. Send and Receive Data

```
//send
char sendBuf[] = "Hello, Winsock!";
int sent = send(sock, sendBuf, (int)strlen(sendBuf), 0);
if (sent == SOCKET_ERROR) {
 printf("send() failed: %d\n", WSAGetLastError());
}

// Receive
char recvBuf[512];
int received = recv(sock, recvBuf, sizeof(recvBuf), 0);
if (received > 0) {
 recvBuf[received] = '\0';
 printf("Received: %s\n", recvBuf);
}
```



## 7. Clean Up

```
closesocket(sock);
WSACleanup();
```

- Always close sockets before calling WSACleanup().
- Match each WSAStartup() with one WSACleanup().



## Shutdown the connection

The shutdown function is used to shutdown for sending and receiving .

```
shutdown(socketfd, SD_SEND); //shutdown for sending
shutdown(socketfd, SD_RECV); //shutdown for receiving
shutdown(socketfd, SD_BOTH); //shutdown for both
```

**Note:** On Unix systems, a common programming technique for servers was for an application to listen for connections. When a connection was accepted, the parent process would call the fork function to create a new child process to handle the client connection, inheriting the socket from the parent. This programming technique is not supported on Windows, since the fork function is not supported. This technique is also not usually suitable for high-performance servers, since the resources needed to create a new process are much greater than those needed for a thread.

# shutdown() vs closesocket() (or Unix's close())

| Aspect         | shutdown()                                                                                                      | closesocket() / close()                                                                     |
|----------------|-----------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| Purpose        | Gracefully disable sends and/or receives on a socket                                                            | Release the socket handle and free all associated resources                                 |
| Prototype      | int shutdown(SOCKET s, int how);                                                                                | int closesocket(SOCKET s); (Win)<br>int close(int fd); (Unix)                               |
| how / howflags | SD_SEND (no more sends)<br>SD_RECEIVE (no more receives)<br>SD_BOTH (both directions)                           | N/A                                                                                         |
| Behavior       | Sends a FIN to peer (for SD_SEND/SD_BOTH) but socket remains open for the other direction until closed.         | Immediately tears down connection, discards unsent data, and invalidates the socket handle. |
| Use Cases      | Half-close a connection (e.g., signal “I’m done sending” while still reading).— Control TCP FIN/ACK sequencing. | Final cleanup when you’re completely done with the socket.— Release handle back to the OS.  |
| Return Value   | 0 on success; SOCKET_ERROR on failure (check WSAGetLastError())                                                 | Same as above                                                                               |
| After Calling  | Socket still valid; you must call closesocket() (or close()) to free it.                                        | Handle is invalid; no further Winsock (or POSIX) calls can use it.                          |

# Full Minimal Client Program



```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
int main() {
 WSADATA wsaData;
 if (WSAStartup(MAKEWORD(2,2), &wsaData) != 0)
 return 1;
 SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (sock == INVALID_SOCKET) { WSACleanup(); return 1; }
 struct sockaddr_in srv = {0};
 srv.sin_family = AF_INET;
 srv.sin_port = htons(8080);
 inet_pton(AF_INET, "127.0.0.1", &srv.sin_addr);

 if (connect(sock, (struct sockaddr*)&srv, sizeof(srv)) == SOCKET_ERROR) {
 printf("connect failed: %d\n", WSAGetLastError());
 } else {
 printf("Connected!\n");
 }
 closesocket(sock);
 WSACleanup();
 return 0;
}
```

# Full Minimal Echo Server Program



```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
// Link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

#define LISTEN_PORT "8080"
#define BUFFER_SIZE 512

int main(void) {
 WSADATA wsaData;
 SOCKET listenSock = INVALID_SOCKET, clientSock = INVALID_SOCKET;
 struct addrinfo hints = {0}, *res = NULL;
 struct sockaddr_storage clientAddr;
 int addrLen = sizeof(clientAddr);
 char buffer[BUFFER_SIZE];
 int bytes;

 // 1. Initialize Winsock
 if (WSAStartup(MAKEWORD(2,2), &wsaData) != 0) {
 fprintf(stderr, "WSAStartup failed\n");
 return 1;
 }

 // 2. Create a listening socket
 if ((listenSock = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET) {
 fprintf(stderr, "socket failed\n");
 return 1;
 }

 // 3. Bind the socket to a specific address and port
 if (bind(listenSock, (const struct sockaddr *) &clientAddr, addrLen) == -1) {
 fprintf(stderr, "bind failed\n");
 closesocket(listenSock);
 return 1;
 }

 // 4. Listen for incoming connections
 if (listen(listenSock, SOMAXCONN) == -1) {
 fprintf(stderr, "listen failed\n");
 closesocket(listenSock);
 return 1;
 }

 // 5. Accept a connection from a client
 if ((clientSock = accept(listenSock, (const struct sockaddr *) &clientAddr, (socklen_t *) &addrLen)) == INVALID_SOCKET) {
 fprintf(stderr, "accept failed\n");
 closesocket(listenSock);
 return 1;
 }

 // 6. Read data from the client
 if (recv(clientSock, buffer, BUFFER_SIZE, 0) == -1) {
 fprintf(stderr, "recv failed\n");
 closesocket(clientSock);
 return 1;
 }

 // 7. Echo the data back to the client
 if (send(clientSock, buffer, strlen(buffer), 0) == -1) {
 fprintf(stderr, "send failed\n");
 closesocket(clientSock);
 return 1;
 }

 // 8. Close the socket
 closesocket(clientSock);
}
```

# Minimal Echo Server Program...



```

// 2. Resolve local address and service (port)
hints.ai_family = AF_UNSPEC; // IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP
hints.ai_flags = AI_PASSIVE; // For bind
if (getaddrinfo(NULL, LISTEN_PORT, &hints, &res) != 0) {
 fprintf(stderr, "getaddrinfo failed\n");
 WSACleanup();
 return 1;
}
// 3. Create listening socket
listenSock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (listenSock == INVALID_SOCKET) {
 fprintf(stderr, "socket() failed: %d\n", WSAGetLastError());
 freeaddrinfo(res);
 WSACleanup();
 return 1;
}
// 4. Bind to the port
if (bind(listenSock, res->ai_addr, (int)res->ai_addrlen) == SOCKET_ERROR) {
 fprintf(stderr, "bind() failed: %d\n", WSAGetLastError());
 closesocket(listenSock);
 freeaddrinfo(res);
 WSACleanup();
 return 1;
}
freeaddrinfo(res);

```

# Minimal Echo Server Program...



```
// 5. Listen for incoming connections
if (listen(listenSock, SOMAXCONN) == SOCKET_ERROR) {
 fprintf(stderr, "listen() failed: %d\n", WSAGetLastError());
 closesocket(listenSock);
 WSACleanup();
 return 1;
}
printf("Server listening on port %s...\n", LISTEN_PORT);

// 6. Accept one client
clientSock = accept(listenSock, (struct sockaddr*)&clientAddr, &addrLen);
if (clientSock == INVALID_SOCKET) {
 fprintf(stderr, "accept() failed: %d\n", WSAGetLastError());
 closesocket(listenSock);
 WSACleanup();
 return 1;
}
printf("Client connected.\n");

// 7. Echo loop
while ((bytes = recv(clientSock, buffer, BUFFER_SIZE, 0)) > 0) {
 send(clientSock, buffer, bytes, 0);
}
```

# Minimal Echo Server Program...



```
// 8. Cleanup
closesocket(clientSock);
closesocket(listenSock);
WSACleanup();
printf("Server shut down.\n");
return 0;
}
```



# End of Chapter 4



# Network Programming

**BESE-VI – Pokhara University**

**Prepared by:**

**Assoc. Prof. Madan Kadariya (NCIT)**

**Contact: [madan.kadariya@ncit.edu.np](mailto:madan.kadariya@ncit.edu.np)**



# Chapter 5:

## Advance Winsock Programming

### (5 hrs)

# Outline



## 1. Asynchronous I/O and Nonblocking operations in Winsock

- Error handling functions
- Using non blocking sockets
- Select in conjunction with accept, select with recv/recvfrom and send/sendto
- Overlapped I/O: Basics and implementation
- Event-driven programming
  - Using WSAEventSelect()
  - Completion routines

## 2. Winsock Extensions

- Advanced polling mechanisms: WSAPoll()
- Event management with WSAEventSelect()

## 3. Implementation of basic cross platform application



# Asynchronous I/O and Nonblocking operations in Winsock

# Asynchronous IO and Non-blocking operation



- Windows' socket API (WinSock 2.2 and later) supports two primary models for avoiding thread-blocking on network operations:

## **1. Non-Blocking I/O**, where calls return immediately with an error if they cannot complete.

- Non-blocking I/O configures a socket so that every send/recv call returns immediately—either with data or with an error indicating “would block”—instead of stalling the calling thread.
- Enabling non-blocking mode:**

### **1. ioctlsocket()**

```
int ioctlsocket(
 SOCKET s,
 long cmd,
 u_long *argp
);
// Example: turn on non-blocking
u_long mode = 1; // 1 = non-blocking, 0 = blocking
if (ioctlsocket(s, FIONBIO, &mode) == SOCKET_ERROR) {
 // handle error: WSAGetLastError()
}
```

# Non-blocking operation



## 2. **WSAAsyncSelect()** (event-driven alternative)

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, u_int wMsg, long lEvent);
// Example: request FD_READ and FD_WRITE messages
if (WSAAsyncSelect(s, hWnd, WM_SOCKET, FD_READ|FD_WRITE) == SOCKET_ERROR) {
 // handle error
}
```

- Associates socket events with WM\_SOCKET messages to window procedure.

## Behavioral Characteristics

### i. Immediate Returns

- send() / recv() will never block.
- If no data (or unable to send), they return SOCKET\_ERROR and WSAGetLastError() yields WSAEWOULDBLOCK



## ii. Error Handling

- Must test for WSAEWOULDBLOCK and treat it as “no data right now” rather than a fatal error.

## • Application Responsibilities

- **Polling loop:** periodically retry I/O calls in a tight or timed loop.
- **Timer-driven:** use SetTimer/timeSetEvent to schedule retries at intervals.
- **Event loop:** integrate with select(), WSAPoll(), or WSAAsyncSelect() to be notified when the socket is ready.



```
// Assume 's' is a connected SOCKET
// 1. Enable non-blocking
u_long mode = 1;
ioctlsocket(s, FIONBIO, &mode);
// 2. Attempt to receive
char buf[512];
int bytes = recv(s, buf, sizeof(buf), 0);
if (bytes > 0) {
 // Data received
} else if (bytes == 0) {
 // Connection closed gracefully
} else {
 int err = WSAGetLastError();
 if (err == WSAEWOULDBLOCK) {
 // No data right now - retry later
 } else {
 // Real socket error - handle
appropriately
 }
}
// 3. Loop or wait for readiness before retrying
// e.g., select(), WSAPoll(), or
WSAAAsyncSelect() events
```

## Summary:

- **Non-blocking I/O** forces application to manage readiness notification and retry logic.
- It is **simpler** to set up than overlapped I/O but less efficient at scale due to polling overhead.
- Use **ioctlsocket(FIONBIO)** for raw non-blocking behavior, or **WSAAAsyncSelect()** to integrate with a Windows message loop.

# Asynchronous IO



- 2. Asynchronous (Overlapped) I/O**, where operations are handed off to the OS and completed in the background, with notification delivered via events, callbacks, or I/O completion ports.

## Prerequisites

### 1. Overlapped-capable socket

```
SOCKET s = WSASocket (AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, WSA_FLAG_OVERLAPPED);
```

- **OVERLAPPED structure**
  - Allocate and zero-initialize a WSAOVERLAPPED object to track each pending operation.
- **I/O functions**
  - Use WSARecv() and WSARecvFrom() (or their “From” variants) rather than recv()/.recvfrom().
- **Completion mechanism** (in one way from below)
  - **Event object** assigned to OVERLAPPED.hEvent
  - **Completion routine** (callback)
  - **I/O Completion Port (IOCP)**

**Functions:** *WSASocket()*, *WSASend()*, *WSARecv()* – Already discussed



```
//Assume 's' is an overlapped socket and WSASStartup() has been called.
char buffer[1024];
WSABUF wsabuf = { .len =sizeof(buffer), .buf = buffer };
DWORD flags = 0, bytesReceived = 0;
WSAOVERLAPPED overlapped = { 0 };
overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
int result=WSARecv(s,&wsabuf,1,&bytesReceived,&flags,&overlapped,NULL);

if (result == SOCKET_ERROR && WSAGetLastError() != WSA_IO_PENDING) {
 // Handle error
} else {
 // Wait for completion
 WaitForSingleObject(overlapped.hEvent, INFINITE);
 // bytesReceived now contains the number of bytes read
}
CloseHandle(overlapped.hEvent);
```



## 1. **WSAGetLastError():**

- If asynchronous function returns `SOCKET_ERROR`, and the specific error code retrieved by calling `WSAGetLastError()` is `WSA_ISO_PENDING`, then it means the overlapped operation has been successfully initiated and the completion will be indicated at a later time.
- Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.
- When the overlapped operation completes the amount of data transferred is indicated either through the `lpNumberOfBytesTransferred` parameter in **WSAGetOverlappedResult()**.

## 2. **WSAGetOverlappedResult()**

- `WSAGetOverlappedResult()` retrieves the results of an overlapped (asynchronous) operation on a socket. It allows an application to obtain the number of bytes transferred and any error status after an overlapped send or receive has completed.



- **WSAGetOverlappedResult()**

```
BOOL WSAGetOverlappedResult(
 SOCKET s,
 LPWSAOVERLAPPED lpOverlapped,
 LPDWORD lpNumberOfBytesTransferred,
 BOOL bWait,
 LPDWORD lpFlags
);
```

- **S:** The overlapped-capable socket on which the operation was issued.
- **lpOverlapped:** Pointer to the *WSAOVERLAPPED* structure used in the original *WSASend* or *WSARecv* call.
- **lpNumberOfBytesTransferred:** Receives the count of bytes sent or received.
- **bWait:** If TRUE, blocks until the overlapped operation has completed; if FALSE, returns immediately even if the operation is still pending.
- **lpFlags:** On receive operations, may be used to retrieve any flags that were set during the transfer (e.g., *MSG\_PARTIAL*). On send operations, this parameter is ignored.

# Error Handling Functions- Asynchronous Operation



- **Return Value**
- Returns nonzero (TRUE) if the overlapped operation completed successfully (either before or after the call, depending on bWait).
- Returns zero (FALSE) on failure; call WSAGetLastError() to retrieve the error code.
  - If the operation is still pending and bWait is FALSE, the error code will be WSA\_IO\_INCOMPLETE.



# Error Handling Functions- Asynchronous Operation

## Note:

### 1. Waiting vs. Polling

1. Use bWait = TRUE when we want to block until completion.
2. Use bWait = FALSE when we want to check status without blocking.

### 2. Error Handling

1. Always call WSAGetLastError() if WSAGetOverlappedResult returns FALSE.
2. Handle WSA\_IO\_INCOMPLETE specially if using polling.

### 3. Integration with Event Objects

1. If used an event in our OVERLAPPED.hEvent, we can WaitForSingleObject on that event, then call WSAGetOverlappedResult with bWait = FALSE to retrieve the final byte count and check for errors.

# Error Handling Functions- Asynchronous Operation



```
// s is an overlapped SOCKET
// overlapped is a WSAOVERLAPPED struct used in WSARecv()
// completionDone is signaled (via event or IOCP) when recv completes
DWORD bytesTransferred = 0;
DWORD flags = 0;
// Option A: Block until complete (if you haven't already waited on the event)
if (!WSAGetOverlappedResult(s, &overlapped, &bytesTransferred, TRUE, &flags)) {
 int err = WSAGetLastError();
 // Handle error...
} else {
 // bytesTransferred now contains the number of bytes received
}
// Option B: Poll for completion
if (!WSAGetOverlappedResult(s, &overlapped, &bytesTransferred, FALSE, &flags)) {
 int err = WSAGetLastError();
 if (err == WSA_IO_INCOMPLETE) {
 // Operation still pending—try again later
 } else {
 // Real failure
 }
} else {
 // Completed; process bytesTransferred
}
```

# select() in Conjunction with accept(), recv()/recvfrom(), and send()/sendto()



- The select() function in WinSock allows a program to monitor multiple sockets simultaneously to determine if they are ready for reading, writing, or if an exception occurred. It is commonly used with blocking sockets to prevent blocking on operations such as accept(), recv(), or send().

**Using select() with accept()**-To avoid blocking on accept(), use select() to check if the listening socket is ready for reading. If it is, that means there is at least one incoming connection waiting to be accepted.

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(listenSock, &readfds);
timeval timeout = {5, 0}; // wait up to 5 seconds
int ready = select(listenSock+1, &readfds, NULL, NULL, &timeout);
if (ready > 0 && FD_ISSET(listenSock, &readfds)) {
 SOCKET clientSock = accept(listenSock, NULL, NULL);
 // Now handle the new client
}
```

# select() in Conjunction with accept(), recv()/recvfrom(), and send()/sendto()



**Using select() with recv()/recvfrom()** - Use select() to check if a socket has incoming data. This prevents recv() or recvfrom() from blocking if no data is available.

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(s, &readfds);
timeval timeout = {5, 0}; // wait up to 5 seconds
int ready = select(s+1, &readfds, NULL, NULL, &timeout);
if (ready > 0 && FD_ISSET(s, &readfds)) {
 char buf[512];
 int bytes = recv(s, buf, sizeof(buf), 0);
 // or: recvfrom() if using UDP
}
```

# select() in Conjunction with accept(), recv()/recvfrom(), and send()/sendto()



**Using select() with send()/sendto()** - Use select() to determine if a socket is ready for writing (i.e., send buffer has space). This helps avoid blocking if the send buffer is full.

```
fd_set writefds;
FD_ZERO(&writefds);
FD_SET(s, &writefds);
timeval timeout = {3, 0}; // wait up to 3 seconds
int ready = select(s+1, NULL, &writefds, NULL, &timeout);
if (ready > 0 && FD_ISSET(s, &writefds)) {
 const char *msg = "Hello";
 send(s, msg, strlen(msg), 0);
 // or: sendto() for datagram
}
```

## select() Return Values and Timeout

- >0 → Number of sockets ready
- 0 → Timeout occurred, no socket is ready
- <0 → Error occurred (WSAGetLastError() for details)

## Overlapped IO Example:

### Overlapped I/O Program (TCP Client with Overlapped WSRecv)



```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib, "ws2_32.lib")
int main() {
 WSADATA wsaData;
 SOCKET sock;
 struct sockaddr_in serverAddr;
 char recvBuf[1024];
 WSABUF wsaBuf;
 DWORD bytesReceived = 0, flags = 0;
 WSAOVERLAPPED overlapped;
 HANDLE hEvent;
```

## Overlapped IO Example:

### Overlapped I/O Program (TCP Client with Overlapped WSRecv)



```
// 1. Initialize WinSock
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
 printf("WSAStartup failed\n");
 return 1;
}

// 2. Create Overlapped-capable socket
sock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, WSA_FLAG_OVERLAPPED);
if (sock == INVALID_SOCKET) {
 printf("WSASocket failed: %d\n", WSAGetLastError());
 WSACleanup();
 return 1;
}
// 3. Setup server address
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080); // Connect to localhost:8080
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

# Overlapped IO Example:

## Overlapped I/O Program (TCP Client with Overlapped WSARecv)



```
// 4. Connect to server
if(connect(sock, (struct sockaddr*)&serverAddr, sizeof(serverAddr))==SOCKET_ERROR)
{
 printf("Connect failed: %d\n", WSAGetLastError());
 closesocket(sock);
 WSACleanup();
 return 1;
}
// 5. Initialize buffer and OVERLAPPED struct
wsaBuf.buf = recvBuf;
wsaBuf.len = sizeof(recvBuf);
memset(&overlapped, 0, sizeof(overlapped));
hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
overlapped.hEvent = hEvent;
// 6. Post an overlapped receive
int result = WSARecv(sock, &wsaBuf, 1, &bytesReceived, &flags, &overlapped,NULL);
if (result == SOCKET_ERROR && WSAGetLastError() != WSA_IO_PENDING) {
 printf("WSARecv failed: %d\n", WSAGetLastError());
 closesocket(sock);
 WSACleanup();
 return 1;
}
```

## Overlapped IO Example:

### Overlapped I/O Program (TCP Client with Overlapped WSRecv)



```
printf("Waiting for data...\n");
// 7. Wait for completion
WaitForSingleObject(hEvent, INFINITE);

if (!WSAGetOverlappedResult(sock, &overlapped, &bytesReceived, FALSE, &flags)) {
 printf("WSAGetOverlappedResult failed: %d\n", WSAGetLastError());
} else {
 recvBuf[bytesReceived] = '\0'; // Null-terminate received string
 printf("Received %lu bytes: %s\n", bytesReceived, recvBuf);
}

// 8. Clean up
CloseHandle(hEvent);
closesocket(sock);
WSACleanup();
return 0;
}
```

# Overlapped IO Example: Overlapped I/O Program (TCP Server)



```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#pragma comment(lib, "ws2_32.lib")
int main() {
 WSADATA wsaData;
 SOCKET listenSock, clientSock;
 struct sockaddr_in serverAddr, clientAddr;
 int addrLen = sizeof(clientAddr);
 char recvBuf[1024];
 WSABUF wsaBuf;
 DWORD bytesReceived = 0, flags = 0;
 WSAOVERLAPPED overlapped;
 HANDLE hEvent;

 // 1. Initialize WinSock
 if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
 printf("WSAStartup failed\n");
 return 1;
 }
```

# Overlapped IO Example: Overlapped I/O Program (TCP Server)



```
// 2. Create overlapped-capable listening socket
listenSock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, WSA_FLAG_OVERLAPPED);
if (listenSock == INVALID_SOCKET) {
 printf("WSASocket failed: %d\n", WSAGetLastError());
 WSACleanup();
 return 1;
}
// 3. Bind and listen
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080);
serverAddr.sin_addr.s_addr = INADDR_ANY;
if (bind(listenSock, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) {
 printf("bind() failed: %d\n", WSAGetLastError());
 closesocket(listenSock);
 WSACleanup();
 return 1;
}
if (listen(listenSock, SOMAXCONN) == SOCKET_ERROR) {
 printf("listen() failed: %d\n", WSAGetLastError());
 closesocket(listenSock);
 WSACleanup();
 return 1;
}
printf("Server is listening on port 8080...\n");
```

# Overlapped IO Example: Overlapped I/O Program (TCP Server)



```

// 4. Accept a client (blocking accept for demo)
clientSock = accept(listenSock, (struct sockaddr*)&clientAddr, &addrLen);
if (clientSock == INVALID_SOCKET) {
 printf("accept() failed: %d\n", WSAGetLastError());
 closesocket(listenSock);
 WSACleanup();
 return 1;
}
printf("Client connected!\n");
// 5. Prepare buffer and overlapped struct
wsaBuf.buf = recvBuf;
wsaBuf.len = sizeof(recvBuf);
memset(&overlapped, 0, sizeof(overlapped));
hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
overlapped.hEvent = hEvent;
// 6. Perform overlapped receive
int result = WSARecv(clientSock, &wsaBuf, 1, &bytesReceived, &flags, &overlapped, NULL);
if (result == SOCKET_ERROR && WSAGetLastError() != WSA_IO_PENDING) {
 printf("WSARecv failed: %d\n", WSAGetLastError());
 closesocket(clientSock);
 closesocket(listenSock);
 WSACleanup();
 return 1;
}
printf("Waiting for client data (Overlapped) ... \n");

```

# Overlapped IO Example: Overlapped I/O Program (TCP Server)



```
// 7. Wait for I/O completion
WaitForSingleObject(hEvent, INFINITE);

if (!WSAGetOverlappedResult(clientSock, &overlapped, &bytesReceived, FALSE, &flags)) {
 printf("WSAGetOverlappedResult failed: %d\n", WSAGetLastError());
} else {
 recvBuf[bytesReceived] = '\0';
 printf("Received %lu bytes: %s\n", bytesReceived, recvBuf);
}

// 8. Cleanup
CloseHandle(hEvent);
closesocket(clientSock);
closesocket(listenSock);
WSACleanup();

return 0;
}
```

# Event-Driven Programming in Winsock



- Event-driven programming in WinSock enables applications to efficiently respond to network events such as readiness to read or write, connection requests, or disconnections—without blocking or polling continuously.
- This is ideal for GUI apps, single-threaded servers, or any application that needs to remain responsive.

## 1. Using WSAEventSelect()

- *WSAEventSelect()* configures a socket to notify the application via an **event object** when specified network events occur (e.g., FD\_READ, FD\_WRITE, FD\_ACCEPT).

```
int WSAEventSelect(SOCKET s,WSAEVENT hEventObject,long lNetworkEvents);
```

- s – the socket to monitor.
- hEventObject – a Win32 event handle created via WSACreateEvent().
- lNetworkEvents – a bitmask of events (e.g., FD\_READ | FD\_WRITE | FD\_CLOSE).



## Example of WSAEventSelect()

```
SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
WSAEVENT hEvent = WSACreateEvent();
// Monitor read and close events
WSAEventSelect(s, hEvent, FD_READ | FD_CLOSE);
```

## 2. Using Completion Routines (APC model)

- Completion routines (also called **asynchronous procedure calls**, or APCs) are a powerful event-driven mechanism used in **overlapped I/O**. Instead of waiting on an event or polling, the system **automatically invokes callback function** when the operation completes.

### Requirements:

- Use WSASend(), WSARcv(), or WSAIoctl() with:
  - OVERLAPPED structure
  - Pointer to the **completion routine**
- The calling thread must be in an **alertable wait state** using:
  - SleepEx()
  - WaitForSingleObjectEx()



```
void CALLBACK CompletionCallback(DWORD dwError, DWORD cbTransferred,
LPWSAOVERLAPPED lpOverlapped, DWORD dwFlags);
```

## Example:

```
void CALLBACK OnRecvComplete(DWORD dwError, DWORD cbTransferred,
 LPWSAOVERLAPPED lpOverlapped, DWORD dwFlags) {
 printf("Asynchronous receive complete: %lu bytes\n", cbTransferred);
}
WSARecv(s, &wsBuf, 1, NULL, &flags, &overlapped, OnRecvComplete);
// Enter alertable wait state
SleepEx(INFINITE, TRUE); // The callback runs when the I/O completes
```

- **WSAEVENTSELECT()** is easy to integrate with an event loop or GUI app.
- **Completion routines** are more efficient for asynchronous I/O and avoid explicit waiting.
- Both approaches are superior to manual polling and allow responsive, non-blocking I/O in event-driven applications.

# Winsock Extensions



- WinSock provides several advanced mechanisms beyond basic `select()` or blocking socket operations. These extensions are designed for higher performance, scalability, and finer control of asynchronous network events.

## 1. Advanced Polling Mechanism: `WSAPoll()`

- `WSAPoll()` is a scalable, thread-safe alternative to `select()` that checks multiple sockets for readiness without the limitations of `fd_set` size or destruction of input sets.

```
int WSAPoll(LPWSAPOLLFD fdArray, ULONG fds, INT timeout);
```

- **fdArray**: Pointer to an array of `WSAPOLLFD` structures.
- **fds**: Number of entries in the array.
- **timeout**: Timeout in milliseconds (-1 for infinite, 0 for non-blocking).

# Winsock Extensions



## WSAPOLLFD Structure

```
typedef struct _WSAPOLLFD {
 SOCKET fd;
 SHORT events; // Requested events (e.g., POLLRDNORM, POLLWRNORM)
 SHORT revents; // Returned events
} WSAPOLLFD;
```

## Events

| Constant   | Meaning                   |
|------------|---------------------------|
| POLLRDNORM | Ready for normal read     |
| POLLWRNORM | Ready for normal write    |
| POLLERR    | Error occurred            |
| POLLHUP    | Connection closed by peer |

# Winsock Extensions



## Example:

```
WSAPOLLFD fds[1];
fds[0].fd = s;
fds[0].events = POLLRDNORM;
int ret = WSAPoll(fds, 1, 5000); // 5-second timeout
if (ret > 0 && (fds[0].revents & POLLRDNORM)) {
 char buf[512];
 int len = recv(s, buf, sizeof(buf), 0);
 // process data
}
```

## Advantages over select()

- Handles thousands of sockets (unlike select's 64 or 1024 limit).
- Keeps input structures unchanged.
- Easier integration in multi-threaded and scalable systems.

## 2. WSAEventSelect() : Already Discussed

# Basic cross-platform networking application



- Creating a **basic cross-platform networking application** requires using libraries or APIs that abstract the underlying differences between Windows and Unix-like systems (Linux, macOS).
- The POSIX socket API is nearly identical across Unix platforms, and with some care, it can be made to work on Windows using conditional compilation.
- Conditional compilation** is a feature in C/C++ that allows you to **include or exclude parts of code** during compilation based on certain conditions. This is especially useful for writing **cross-platform code**, where different code is needed for different operating systems or compilers.
- Conditional compilation uses **preprocessor directives**, which are instructions to the compiler **before actual compilation** begins.

| Directive                 | Purpose                        |
|---------------------------|--------------------------------|
| #ifdef                    | If macro is defined            |
| #ifndef                   | If macro is <b>not</b> defined |
| #if, #elif, #else, #endif | If condition is true / false   |
| #define                   | Define a macro                 |
| #undef                    | Undefine a macro               |

# Basic cross-platform networking application [Server]



```
// cross_server.c
#ifndef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
typedef int SOCKET;
#endif

#include <stdio.h>
```

```
int main() {
#ifndef _WIN32
 WSADATA wsaData;
 WSASTARTUP(MAKEWORD(2, 2), &wsaData);
#endif
 SOCKET serverSock=socket(AF_INET, SOCK_STREAM,
0);
 if (serverSock == INVALID_SOCKET) {
 perror("socket failed");
 return 1;
 }
 struct sockaddr_in serverAddr = {0};
 serverAddr.sin_family = AF_INET;
 serverAddr.sin_addr.s_addr = INADDR_ANY;
 serverAddr.sin_port = htons(8080);
 if (bind(serverSock, (struct
sockaddr*)&serverAddr, sizeof(serverAddr)) ==
SOCKET_ERROR) {
 perror("bind failed");
 return 1;
 }
```

# Basic cross-platform networking application [Server]



```
listen(serverSock, 5);
printf("Server listening on port 8080...\n");
struct sockaddr_in clientAddr;
socklen_t clientLen = sizeof(clientAddr);
SOCKET clientSock = accept(serverSock, (struct sockaddr*)&clientAddr, &clientLen);
char buffer[1024];
int bytesRead;
while ((bytesRead = recv(clientSock, buffer, sizeof(buffer) - 1, 0)) > 0) {
 buffer[bytesRead] = '\0';
 printf("Client: %s\n", buffer);
 send(clientSock, buffer, bytesRead, 0); // echo back
}
#endif _WIN32
closesocket(clientSock);
closesocket(serverSock);
WSACleanup();
#else
close(clientSock);
close(serverSock);
#endif
return 0;
```

# Basic cross-platform networking application [Client]



```
// cross_client.c
#ifndef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "ws2_32.lib")
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
typedef int SOCKET;
#endif
#include <stdio.h>
int main() {
#ifdef _WIN32
 WSADATA wsaData;
 WSAStartup(MAKEWORD(2, 2),
 &wsaData);
#endif
}
```

```
SOCKET sock = socket(AF_INET, SOCK_STREAM);
if (sock == INVALID_SOCKET) {
 perror("socket failed");
 return 1;
}
struct sockaddr_in serverAddr = {0};
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(8080);
inet_nton(AF_INET, "127.0.0.1",
&serverAddr.sin_addr);
if (connect(sock, (struct
sockaddr*)&serverAddr, sizeof(serverAddr)) ==
SOCKET_ERROR) {
 perror("connect failed");
 return 1;
}
char msg[512];
char buf[1024];
```

# Basic cross-platform networking application [Client]



```
while (1) {
 printf("Enter message: ");
 fgets(msg, sizeof(msg), stdin);
 send(sock, msg, strlen(msg), 0);

 int bytes = recv(sock, buf, sizeof(buf) - 1, 0);
 if (bytes <= 0) break;
 buf[bytes] = '\0';
 printf("Echo: %s\n", buf);
}

#ifndef _WIN32
 closesocket(sock);
 WSACleanup();
#else
 close(sock);
#endif

 return 0;
}
```



# End of Chapter 5



# Network Programming

**BESE-VI – Pokhara University**

**Prepared by:**

**Assoc. Prof. Madan Kadariya (NCIT)**

**Contact: [madan.kadariya@ncit.edu.np](mailto:madan.kadariya@ncit.edu.np)**



# Chapter 6:

## Network utilities, Current Trends and Emerging Technologies in Network Programming

### (5 hrs)

# Outline



## 1. Network Utilities and Applications

- Introduction to ping, telnet, ip/ifconfig, iperf, netstat, remote login

## 2. Real-Time Communication Protocols

- WebSockets: Full-duplex communication over a single TCP connection
- gRPC: High-performance RPC framework

## 3. Security in Network Programming

- TLS/SSL: Introduction to secure socket programming

## 4. Software-Defined Networking (SDN)

- Overview of SDN concepts
- Introduction to OpenFlow protocol and SDN controllers

## 5. Introduction to P4 and frenetic programming



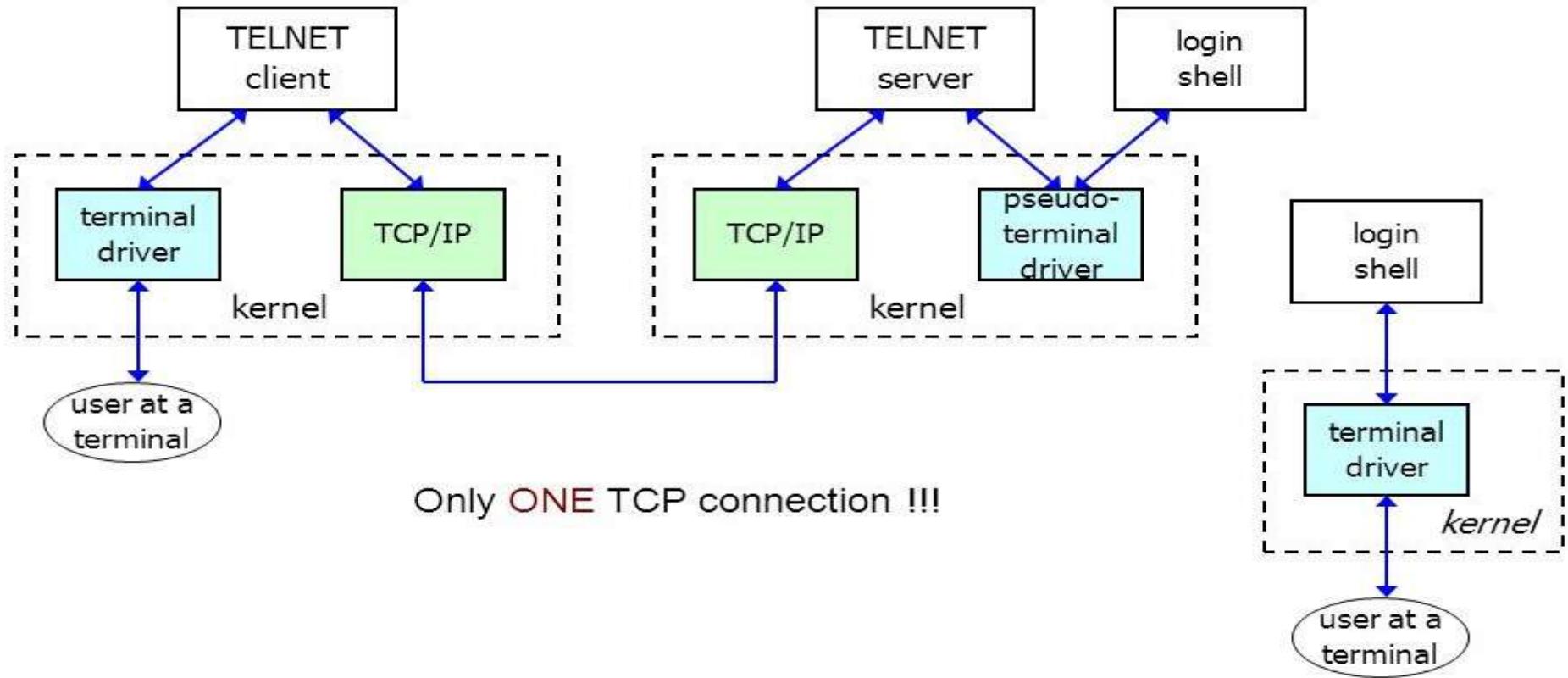
# Network Utilities and Applications



## Telnet and Rlogin: Remote Login

- ❖ Remote login is one of the most popular Internet applications. Instead of having a hardwired terminal on each host, we can login to one host and then remote login across the network to any other host (that we have an account on, of course).
  - ❖ Two popular applications provide remote login across TCP/IP internets.
1. **Telnet** is a standard application that almost every TCP/IP implementation provides. It works between hosts that use different operating systems. Telnet uses option negotiation between the client and server to determine what features each end can provide.
  2. **Rlogin** is from Berkeley Unix and was developed to work between Unix systems only, but it has been ported to other operating systems also.

- Remote login uses the client-server paradigm.
- Fig: Overview of Telnet client-server.





1. The Telnet client interacts with both the user at the terminal and the TCP/IP protocols. Normally everything we type is sent across the TCP connection, and everything received from the connection is output to our terminal.
  2. The Telnet server often deals with what's called a *pseudo-terminal* device, at least under Unix systems. This makes it appear to the login shell that's invoked on the server, and to any programs run by the login shell, that they're talking to a terminal device.
  3. Only a single TCP connection is used. Since there are times when the Telnet client must talk to the Telnet server (and vice versa) there needs to be some way to delineate commands that are sent across the connection, versus user data.
  4. In dashed boxes in Figure to note that the terminal and pseudo terminal drivers, along with the TCP/IP implementation, are normally part of the operating system kernel. The Telnet client and server, however, are often user applications.
  5. The login shell on the server host to reiterate that we have to login to the server. We must have an account on that system to login to it, using either Telnet or Rlogin.
- ❖ Remote login is not a high-volume data transfer application. Lots of small packets are normally exchanged between the two end systems. It is found that the ratio of bytes sent by the client (the user typing at the terminal) to the number of bytes sent back by the server is about 1:20. This is because we type short commands that often generate lots of output.

# Rlogin Protocol



- Rlogin appeared with 4.2BSD and was intended for remote login only between Unix hosts. This makes it a simpler protocol than Telnet, since option negotiation is not required when the operating system on the client and server are known in advance.
- **Application Startup:**
  - ❖ Rlogin uses a single TCP connection between the client and server. After the normal TCP connection establishment is complete, the following application protocol takes place between the client and server.
    1. The client writes four strings to the server; (a) a byte of 0, (b) the login name of the user on the client host terminated by a byte of 0, (c) the login name of the user on the server host, terminated by a byte of 0, (d) the name of the user's terminal type, followed by a slash, followed by the terminal speed, terminated by a byte of 0. Two login names are required because users aren't required to have the same login name on each system. The terminal type is passed from the client to the server because many full-screen applications need to know it. The terminal speed is passed because some applications operate differently depending on the speed. For example, the vi editor works with a smaller window when operating at slower speeds, so it doesn't take forever to redraw the window.
    2. The server responds with a byte of 0.

# Rlogin Protocol



3. The server has the option of asking the user to enter a password. This is handled as normal data exchange across the Rlogin connection-there is no special protocol. The server sends a string to the client (which the client displays on the terminal), often Password:. If the client does not enter a password within some time limit (often 60 seconds), the server closes the connection. We can create a file in our home directory on the server (named .rhosts) with lines containing a hostname and our username. If we login from the specified host with that username, we are not prompted for a password. If we are prompted by the server for a password, what we type is sent to the server as *cleartext*. Each character of the password that we type is sent as is. Anyone who can read the raw network packets can read the characters of our password. Newer implementations of the Rlogin client, such as 4.4BSD, first try to use Kerberos, which avoids sending cleartext passwords across the network. This requires a compatible server that also supports Kerberos.
4. The server normally sends a request to the client asking for the terminal's window size

# Rlogin Protocol



## ❖ Flow Control

- By default, flow control is done by the Rlogin client. The client recognizes the ASCII STOP and START characters (Control-S and Control-Q) typed by the user, and stops or starts the terminal output.

## ❖ Client Interrupt

- It is rare for the flow of data from the client to the server to be stopped by flow control. This direction contains only characters that we type. Therefore it is not necessary for these special input characters (Control-S or interrupt) to be sent from the client to the server using TCP's urgent mode.

## ❖ Window Size Changes

- With remote login, however, the change in the window size occurs on the client, but the application that needs to be told is running on the server. Some form of notification is required for the Rlogin client to tell the server that the window size has changed, and what the new size is.

- **Server to Client Commands :** the four commands that the Rlogin server can send to the client across the TCPconnection. The problem is that only a single TCP connection is used, so the server needs to mark these command bytes so the client knows to interpret them as commands, and not display the bytes on the terminal.

|      |                                                                                                                                                                                                                                                                                                   |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0x02 | Flush output. The client discards all the data received from the server, up through the command byte (the last byte of urgent data). The client also discards any pending terminal output that may be buffered. The server sends this command when it receives the interrupt key from the client. |
| 0x10 | The client stops performing flow control.                                                                                                                                                                                                                                                         |
| 0x20 | The client resumes flow control processing.                                                                                                                                                                                                                                                       |
| 0x80 | The client responds immediately by sending the current window size to the server, and notifies the server in the future if the window size changes. This command is normally sent by the server immediately after the connection is established.                                                  |

- **Client to Server Commands**

- Only one command from the client to the server is currently defined: sending the current window size to the server. Window size changes from the client are not sent to the server unless the client receives the command 0x80 from the server.



## Telnet Protocol

- ❖ Telnet was designed to work between any host (i.e., any operating system) and any terminal. Its specification defines the lowest common denominator terminal, called the *network virtual terminal* (NVT). The NVT is an imaginary device from which both ends of the connection, the client and server, map their real terminal to and from. That is, the client operating system must map whatever type of terminal the user is on to the NVT. The server must then map the NVT into whatever terminal type the server supports.
- ❖ The NVT is a character device with a keyboard and printer. Data typed by the user on the keyboard is sent to the server, and data received from the server is output to the printer. By default the client echoes what the user types to the printer.
- ❖ Telnet uses TCP to transmit data and telnet control information. The default port for telnet is TCP port 23. Telnet, however, predates TCP/IP and was originally run over Network Control Program (NCP) protocols.

## Telnet Commands

- ❖ Telnet uses in-band signaling in both directions. The byte 0xff (255 decimal) is called IAC, for "interpret as command." The next byte is the command byte. To send the data byte 255, two consecutive bytes of 255 are sent.

| Name  | Code<br>(decimal) | Description                                        |
|-------|-------------------|----------------------------------------------------|
| EOF   | 236               | end-of-file                                        |
| SUSP  | 237               | suspend current process (job control)              |
| ABORT | 238               | abort process                                      |
| EOR   | 239               | end of record                                      |
| SE    | 240               | suboption end                                      |
| NOP   | 241               | no operation                                       |
| DM    | 242               | data mark                                          |
| BRK   | 243               | break                                              |
| IP    | 244               | interrupt process                                  |
| AO    | 245               | abort output                                       |
| AYT   | 246               | are you there?                                     |
| EC    | 247               | escape character                                   |
| EL    | 248               | erase line                                         |
| GA    | 249               | go ahead                                           |
| SB    | 250               | suboption begin                                    |
| WILL  | 251               | option negotiation ( <a href="#">Figure 26.9</a> ) |
| WONT  | 252               | option negotiation                                 |
| IX)   | 253               | option negotiation                                 |
| DONT  | 254               | option negotiation                                 |
| IAC   | 255               | data byte 255                                      |



## • Option Negotiation

- ❖ Although Telnet starts with both sides assuming an NVT, the first exchange that normally takes place across a Telnet connection is option negotiation. The option negotiation is symmetric - either side can send a request to the other. Either side can send one of four different requests for any given option.
- 1. **WILL**. The sender wants to enable the option itself.
- 2. **DO**. The sender wants the receiver to enable the option.
- 3. **WONT**. The sender wants to disable the option itself.
- 4. **DONT**. The sender wants the receiver to disable the option.
- Since the rules of Telnet allow a side to either accept or reject a request to enable an option (cases 1 and 2 above), but require a side to always honor a request to disable an option (cases 3 and 4 above), these four cases lead to the six scenarios shown as follows.

|    | Sender | Receiver | Description |                                                                                      |
|----|--------|----------|-------------|--------------------------------------------------------------------------------------|
| 1. | WILL   | -><br>-< | DO          | <b>sender</b> wants to <b>enable</b> option<br>receiver says OK                      |
| 2. | WILL   | -><br>-< | DONT        | <b>sender</b> wants to <b>enable</b> option<br>receiver says NO                      |
| 3. | DO     | -><br>-< | WILL        | <b>sender</b> wants <b>receiver</b> to <b>enable</b> option<br>receiver says OK      |
| 4. | DO     | -><br>-< | WONT        | <b>sender</b> wants <b>receiver</b> to <b>enable</b> option<br>receiver says NO      |
| 5. | WONT   | -><br>-< | DONT        | <b>sender</b> wants to <b>disable</b> option<br>receiver must say OK                 |
| 6. | DONT   | -><br>-< | WONT        | <b>sender</b> wants <b>receiver</b> to <b>disable</b> option<br>receiver must say OK |

- Table: Six scenarios for Telnet option negotiation.



## netstat

- ❖ It means network statistics. The netstat is a command-line network utility tool to display the information about network connections. It can
  - **1) display the routing table**
  - **netstat -r** : shows routing table i.e. destination, gateway, genmask, flags, network interface, etc.
  - **2) display interface statistics**
  - **netstat -i** : displays statistics for the network interfaces currently configured. If the **-a** option is also given, it prints all interfaces present in the kernel, not only those that have been configured currently.
  - **3) display network connections**
  - **netstat -p tcp**: displays the information about TCP connections. The netstat provides statistics for the following: proto, local address, foreign address, TCP states.



# The ifconfig/ipconfig

- ❖ **ipconfig** – Windows
- ❖ **ifconfig** – Unix and Unix-like systems
- ❖ The ifconfig stands for “interface configuration”. It is used to assign an address to a network interface and/or configure network interface parameters.

- **Examples**

- ● **List interfaces (only active)**

- ifconfig

- ● **List all interfaces**

- ifconfig -a

- ● **Display the configuration of device eth0 only**

- ifconfig eth0

- ● **Enable and disable an interface**

- sudo ifconfig eth1 up
    - sudo ifconfig eth1 down

- ● **Configure an interface**

- sudo ifconfig eth0 inet 192.168.0.10 netmask 255.255.255.0



- *iperf* is a commonly used **network testing tool** that measures **network bandwidth and performance** between two hosts. It's useful for diagnosing issues, benchmarking throughput, or stress-testing network infrastructure.

## Key Features of *iperf*:

- Measures **TCP** and **UDP** bandwidth.
- Supports **IPv4** and **IPv6**.
- Reports **jitter**, **packet loss**, and **retransmissions**.
- Allows **server-client** mode.
- Supports **multiple parallel streams**.
- Works cross-platform (Linux, Windows, macOS).



## Common Options:

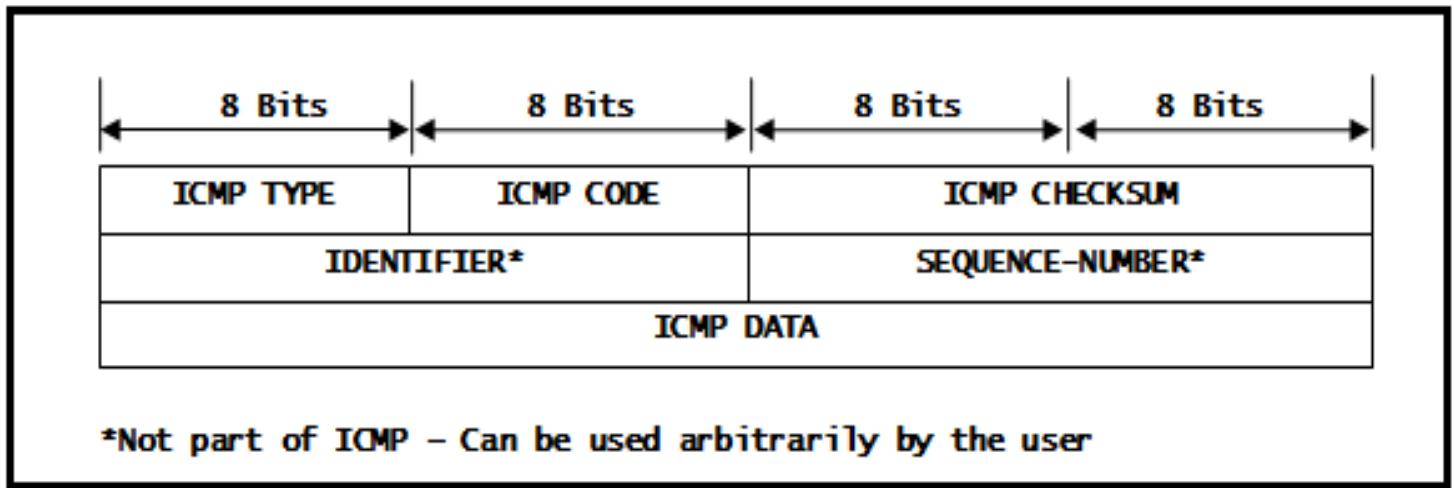
| Option | Description                                    |
|--------|------------------------------------------------|
| -s     | Run in server mode                             |
| -c     | Run in client mode (followed by server IP)     |
| -u     | Use UDP instead of TCP                         |
| -t     | Duration of test in seconds (default: 10 sec)  |
| -p     | Specify port number                            |
| -b     | Bandwidth to send at (UDP only, e.g., -b 100M) |
| -P     | Number of parallel client streams              |
| -R     | Reverse test (server sends to client)          |



## The ping

- ❖ Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network.
- ❖ It measures the round-trip time for messages sent from the originating host to a destination computer and echoed back to the source.
- ❖ Ping operates by sending Internet Control Message Protocol (ICMP) Echo Request packets to the target host and waiting for an ICMP Echo Reply.
- ❖ The results of the test usually include a statistical summary of the results, including the minimum, maximum, the mean, round-trip time, and usually standard deviation of the mean.
- ❖ Example: ping -c 4 www.google.com // c = packet counts

## Frame format - ICMP

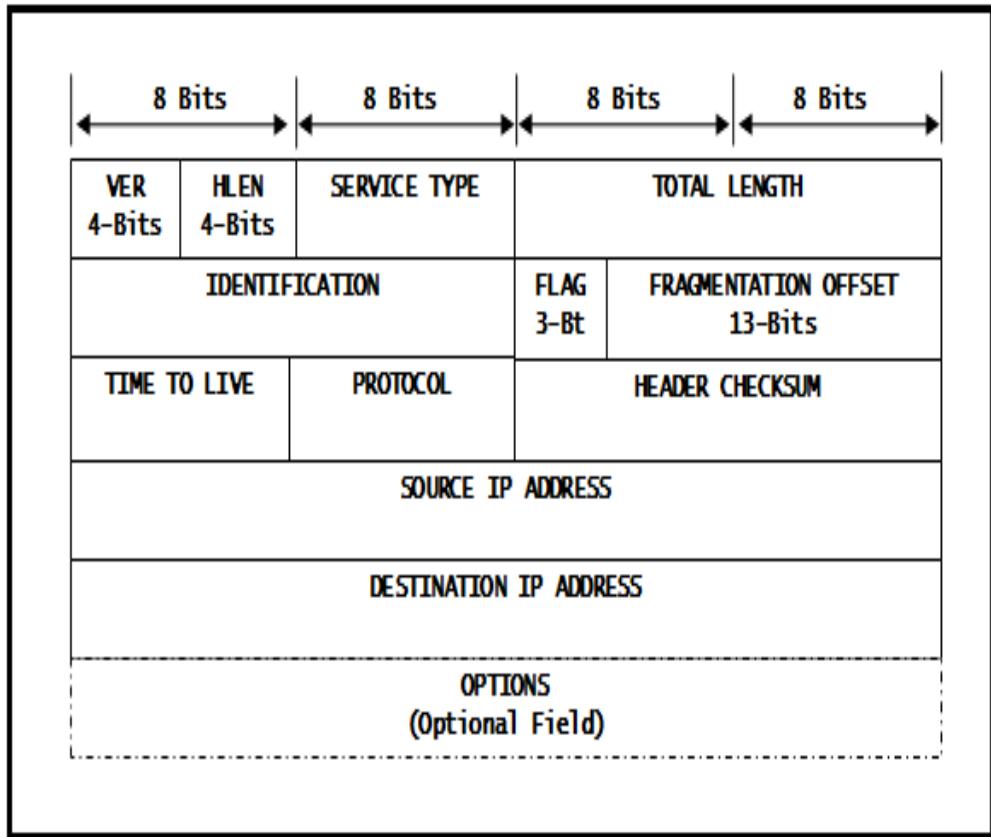


1. **ICMP TYPE** shall be set to **0x08** since this is an ‘Echo-Request’ message
2. **ICMP CODE** shall always be **0x00** for PING message
3. **ICMP CHECKSUM** is for header and data and is '**0xA5, 0x51**' for our message
4. **ICMP DATA** is “PING data to be sent” defined above

- ❑ Before sending the ICMP encapsulated PING command to MAC layer, the messages should be encapsulated into IP datagrams. Below is the frame-format of IPv4 packet:



Frame format - IP



- VER** shall be set to **4** since it's a IPv4 packet
- HLEN** is header length in 'lwords' - It shall be set to 5 here since the header length is 20Bytes
- SERVICE TYPE** shall be set to **0x00** since ICMP is a normal service
- TOTAL LENGTH** shall be set as '**0x00 0x54**' bytes, which includes header and data length
- IDENTIFICATION** is the unique identity for all datagrams sent from this source IP – Let's set it as '**0x96, 0xA1**' for our packet
- FLAG** and **FRAGMENTATION OFFSET** is set to **0x00, 0x00** since we don't intent to fragment the packet. **Reason:** The packet that we are sending here is smaller than a WLAN frame size
- TIME TO LIVE** shall be set to **0x40** since we want to discard the packet after 64 hops
- PROTOCOL** is set to **0x01** since it's an ICMP packet
- Next 2Bytes is **HEADER CHECKSUM** which shall be set to **0x57, 0xFA** in our case
- SOURCE IP ADDRESS** is set to **0xc0, 0xa8, 0x01, 0x64** (which is 192.168.1.100)
- DESTINATION IP ADDRESS** is set to **0xc0, 0xa8, 0x01, 0x65** (which is 192.168.1.101)
- OPTIONS** field is left blank



# Real-Time Communication Protocols

# WebSockets: Full-duplex communication over a single TCP connection



- WebSockets provide a **full-duplex, bidirectional communication** channel between a client (usually a browser) and a server over a single long-lived TCP connection.
- This is ideal for applications that require real-time updates, such as:
  - Chat applications
  - Online gaming
  - Live sports updates
  - Stock trading dashboards
  - Collaborative editing tools (e.g., Google Docs)
- WebSockets allow:
  - Client and server to send messages independently and simultaneously.
  - Both parties can communicate without waiting for a request or response.



## How it works? (Connection Establishment)

- **Handshake Process:**

1. **Client sends an HTTP request** with headers:

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

2. **Server replies if supported:**

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZRbK+xOo=
```

3. After this, the TCP connection stays open and the protocol switches to WebSocket frames.



## Advantages of WebSockets

- Real-time, low-latency communication
- Less overhead than HTTP polling
- Efficient for apps needing instant updates

### Demo:

Start server: node server.js  
Open client.html in browser

## Limitations / Considerations

- Needs proxy/firewall/WebSocket support
- Not cacheable like HTTP
- Slightly complex to implement securely (use wss://)

## Comparison: HTTP vs WebSocket

| Feature               | HTTP                           | WebSocket               |
|-----------------------|--------------------------------|-------------------------|
| Communication         | Half-duplex (request-response) | Full-duplex (both ways) |
| Persistent connection | No                             | Yes                     |
| Protocol              | Text-based                     | Binary or text frames   |
| Overhead              | High                           | Low (after upgrade)     |

# gRPC: High-Performance RPC Framework



- **gRPC (Google Remote Procedure Call)** is a modern open-source RPC framework designed for high-performance, low-latency communication between distributed systems and microservices.
- Built on top of HTTP/2 (*HTTP/2 is a major revision of the HTTP network protocol, designed to improve web page load times and efficiency by reducing latency*)
- Uses Protocol Buffers (protobuf) for efficient serialization
- Supports multi-language clients & servers
- Enables streaming, authentication, and load balancing

| Term                    | Description                                                                |
|-------------------------|----------------------------------------------------------------------------|
| <b>RPC</b>              | Remote Procedure Call – function call between systems over a network.      |
| <b>Protocol Buffers</b> | Efficient binary format to define service contracts.                       |
| <b>Stub</b>             | Client-side code auto-generated from .proto file that talks to the server. |
| <b>IDL</b>              | Interface Definition Language – .proto file in gRPC.                       |



## How gRPC Works?

1. We define **service** and **messages** in a .proto file.
2. Use the protoc compiler to generate client and server code.
3. Server implements the service.
4. Client uses the stub to call the server method like a local function.

## Types of gRPC Calls

| Type                    | Description                            |
|-------------------------|----------------------------------------|
| Unary RPC               | Single request and response (as above) |
| Server streaming        | Server sends a stream of responses     |
| Client streaming        | Client sends a stream of requests      |
| Bidirectional streaming | Both send streams concurrently         |

## Why Use gRPC?

- Fast, due to HTTP/2 and Protobuf
- Full duplex streaming
- Built-in authentication
- Great for microservices



## HTTP vs gRPC Comparison

| Feature          | HTTP REST         | gRPC                    |
|------------------|-------------------|-------------------------|
| Transport        | HTTP/1.1          | HTTP/2                  |
| Message Format   | JSON              | Protocol Buffers        |
| Speed            | Slower            | Faster                  |
| Streaming        | Hard to implement | Native support          |
| Language Support | Manual SDKs       | Auto-generated for many |



## Demo

### Setup:

1. npm init -y
2. npm install @grpc/grpc-js @grpc/proto-loader
3. create greet.proto (Contents are in file)
4. create greet\_server.js (Contents are in file)
5. create greet\_client.js (Contents are in file)
6. run
  - 6.1. node greet\_server.js
  - 6.2. node greet\_client.js



# Security in Network Programming

# TLS/SSL: Introduction to secure socket programming



- TLS(Transport Layer Security) and its predecessor SSL(Secure Sockets Layer) are cryptographic protocols that secure communication over a network, ensuring:
  - Confidentiality (data is encrypted)
  - Integrity (data is not tampered with)
  - Authentication (server or client identity is verified)
  - TLS is used in HTTPS, FTPS, SMTPS, and secure socket programming

## TLS vs SSL

| Feature     | SSL (Deprecated) | TLS (Modern)     |
|-------------|------------------|------------------|
| Versions    | SSLv2, SSLv3     | TLS 1.0–1.3      |
| Security    | Weak             | Strong           |
| Current Use | ✗ No longer safe | ✓ TLS 1.2 or 1.3 |

## Why Use TLS in Socket Programming?

- Prevents eavesdropping(spy), MITM attacks(On-path Attack), data forgery(falsification).
- Essential for financial apps, login systems, APIs.

*After certificate:*  
 secure\_server.py  
 secure\_client.py



# Software Defined Network (SDN)



## Overview of SDN Concepts

- Software-Defined Networking (SDN) is a network architecture approach that separates:
  - Control Plane (decision-making logic)
  - Data Plane (packet forwarding)
- Traditional networks combine both. SDN decouples them for centralized control, programmability, and agility.

## Key SDN Concepts:

| Component      | Description                                                                   |
|----------------|-------------------------------------------------------------------------------|
| Controller     | Centralized “brain” of the network, makes decisions                           |
| Data Plane     | Switches and routers that follow instructions from controller                 |
| Southbound API | Communication between controller and devices (e.g., OpenFlow)                 |
| Northbound API | Apps communicating with controller (e.g., for monitoring, policy enforcement) |



## Benefits of SDN

- **Centralized Control** : The SDN controller manages the entire network from one place, simplifying decision-making and visibility.
- **Dynamic Configuration**: Network behavior (e.g., routing, policies) can be changed on the fly via software, without manual device reconfiguration.
- **Network Automation**: Common tasks like provisioning, traffic routing, and security enforcement can be scripted and automated.
- **Easier Experimentation**: Developers can test new protocols or policies in real-time without changing physical hardware.
- **Scalability & Flexibility**: Networks can grow or adapt quickly by simply updating the controller logic, not the underlying hardware.

# Introduction to OpenFlow & SDN Controllers



- OpenFlow is a standard protocol that allows **SDN** controllers to instruct switches how to handle network packets.
- It defines a flow table on each switch.

## Flow Table Entry Example

| Match Fields | Actions        | Counters   |
|--------------|----------------|------------|
| IP Src = A   | Forward Port 2 | 50 packets |

## How It Works:

1. Packet enters a switch.
2. If it matches a rule in the flow table → apply action (forward, drop, modify).
3. If no match → switch contacts the controller for instructions.

## Popular SDN Controllers

| Controller   | Language | Notes                     |
|--------------|----------|---------------------------|
| Ryu          | Python   | Great for beginners       |
| ONOS         | Java     | Carrier-grade, scalable   |
| OpenDaylight | Java     | Modular, enterprise-ready |
| Floodlight   | Java     | Simple, good for labs     |

# Introduction to OpenFlow & SDN Controllers



**Demo (try yourself in ubuntu)-my machine doesn't support natively**

## 1. Install Mininet

```
sudo apt update
```

```
sudo apt install mininet -y
```

## 2. Install Ryu

```
sudo apt install python3-pip -y
```

```
pip3 install ryu
```

## 3. Start Ryu Controller (local)

```
ryu-manager ryu.app.simple_switch_13
```

## 4. Launch Mininet with Remote Controller (localhost)

```
sudo mn --controller=remote,ip=127.0.0.1 --topo=tree,depth=2
```

## 5. Test Connectivity

```
mininet> pingall
```



# Introduction to P4 and Frenetic Programming

## P4 Language: Programming Protocol-Independent Packet Processors



- P4 = Programming Protocol-Independent Packet Processors
- Used to program the data plane, not just control logic.
- Focuses on defining how packets are parsed, matched, and processed.
- Can be used with software switches (e.g., BMv2) and programmable hardware (Tofino).

### P4 Program Structure:

```
parser MyParser {
 extract(ethernet);
 extract(ipv4);
}

control MyIngress {
 if (ipv4.dst == 10.0.0.1) {
 forward to port 1;
 }
}
```

### Frenetic Language:

- Functional programming language for writing SDN applications.
  - Higher abstraction than OpenFlow.
  - Developed by Cornell/Princeton researchers.
- ```
# Frenetic-style logic (Python-like pseudocode)
on_packet(packet):
    if packet.dst_ip == "10.0.0.1":
        forward(packet, port=1)
```



P4 Language: Programming Protocol-Independent Packet Processors

Summary Comparison

Technology	Role	Language	Use Case
OpenFlow	Southbound protocol	N/A	Switch-controller link
Ryu	SDN Controller	Python	Manage flow rules
P4	Data Plane language	P4	Define packet processing
Frenetic	High-level SDN lang	OCaml/Py	Write network policies



End of Chapter 6

(End of syllabus of Network Programming)