

Operating System Tutorial 7 - Socketing

Yonghao Lee

May 19, 2025

1 Communication Protocols

A communication protocol is a system of digital rules for data exchange between computer systems. Communication systems use well-defined formats for exchanging messages. A protocol defines the syntax, semantics, and synchronization of communication. Computer networking protocol implementation is called a protocol stack or network stack.

1.1 Communication Layers

Communication occurs in several layers:

- **Physical Layer:** Bytes are streamed physically, either through cables or wirelessly through the air.
- **Routing Layer:** Data is transferred from one entity to another, eventually reaching someone who can communicate directly with the intended recipient.
- **Management Layer:** Ensures that data indeed arrives, and handles interaction with the application through a convenient API; often includes compression, encryption, and authentication.
- **Application Layer:** The application understands the bytes by converting them to usable data (for example, ASCII text).

In this course, our focus will be primarily on the management layer. With this context, let's examine how data is organized for transmission.

2 Packets - Our Building Blocks

Allowing only messages of a fixed size is impractical, as some communications are long while others are short.

Very long messages present challenges for data integrity, as they are more prone to transmission errors. To address this, we use packets—the basic unit of data carried over networks. Packets have both minimum and maximum size constraints. When data exceeds the maximum packet size, it undergoes packetization—the process of breaking the original message into several smaller packets.

2.1 Packet Structure

A packet contains three main components, with the header and trailer representing transmission overhead:

- **Header** (overhead)
 - Source and destination addresses
 - Packet identification number (often included)

- Additional protocol-specific information
- **Payload/Data**
 - The actual information being transmitted
- **Trailer** (overhead, sometimes included)
 - Appended at the end of the packet, after the data
 - Often contains error-checking information such as checksums or CRC (Cyclic Redundancy Check) values.

The header and trailer are classified as *overhead* because they consume bandwidth but do not contribute to the actual content being transmitted. This overhead is necessary for proper network functionality, much like how packaging material is necessary for shipping physical goods but isn't part of the product itself.

2.2 Packetization

With the structure of packets established, we now turn to the process of creating and managing these packets.

Packetization tools are responsible for:

- Adding headers (and trailers) to packets.
- Breaking the message into packets.
- Sending the packets and receiving incoming packets.

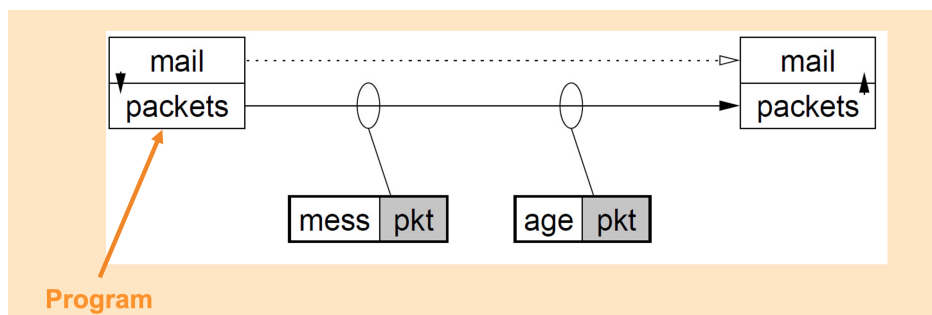


Figure 1: Packetization process showing logical view (dotted arrow) vs. physical reality (solid arrow).

Packetization acts as an interface between application programs and the network infrastructure. When a program needs to send data (like an email), the packetization process divides the complete message into smaller segments. Each segment becomes the payload for a packet, to which headers and possibly trailers are added.

As shown in Figure 1, there are two perspectives of data transmission:

- The **logical view** (shown by the dotted arrow at the top): Applications perceive sending a complete "mail" message directly from sender to receiver.
- The **physical reality** (shown by the solid arrow): The message is actually broken into separate packets that travel individually through the network.

In the diagram, "mess" and "age" represent parts of the original "message" that have been split. Each part is paired with "pkt" (representing packet headers/trailers). The boxes on the left and right labeled "packets" represent the packetization tools that handle this process at both the sending and receiving ends.

At the destination, the packetization tools receive these packets, strip away the headers and trailers, and reassemble the original message before delivering it to the receiving application. This process is largely transparent to the applications, which simply see complete messages being sent and received, while the underlying packetization handles all the complexity of network transmission.

2.3 End-to-End Control

Once packets are created, we need mechanisms to ensure they reach their destination correctly.

During network transmission, various errors can occur:

- Lost packets (packets never arrive at the destination).
- Incorrect transmission of packets (bytes are corrupted during transmission).

To address these issues, communication protocols often implement end-to-end control mechanisms that ensure reliable transmission. These mechanisms:

- Verify that data has successfully arrived at the destination.
- Handle packets arriving in arbitrary order by implementing reordering functionality.
- Ensure data accuracy and integrity upon arrival.
 - This may include correcting errors using error-correction codes (ECCs).

2.4 Routing

After understanding how packets are created and verified, we need to consider how they travel through networks.

What about packet transmission in networks?

- As you probably know, computers are not all directly connected to one another.
- Packets travel through the network and pass through "intermediate stations" (routers).
- This process is called **routing**.

- Specialized software/hardware (routers) is responsible for routing, sending packets through intermediate stations until they reach their destination.

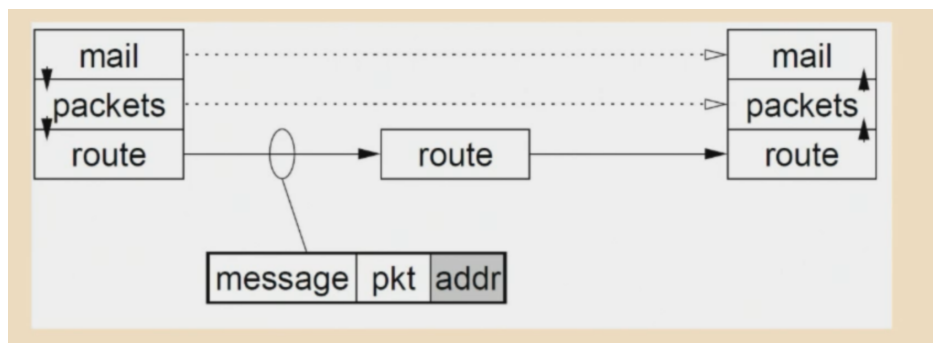


Figure 2: Multi-level view of routing showing application, packetization, and network levels.

As illustrated in Figure 2, there are three perspectives of network communication:

- At the application level (top dotted arrow): Programs simply see complete messages being sent.
- At the packetization level (middle dotted arrow): The system handles breaking messages into packets.
- At the routing level (bottom solid arrow): Each packet travels through intermediate stations to reach its destination.

Note that packets include not only the message content (“message”) and protocol information (“pkt”), but also addressing information (“addr”) that enables proper routing.

3 The TCP/IP Model

Now that we understand the fundamentals of packet-based communication, let’s examine how these concepts are organized into a standardized model.

In the TCP/IP model, communication is performed in 4 distinct layers:

- **Application Layer**
 - The way the application “understands” the data.
 - Protocol examples: HTTP(S), FTP, DNS.
- **Transport Layer**
 - Ensures communication is done smoothly between applications.
 - Protocols: TCP, UDP.
- **Network/Internet Layer**

- Handles transmission of the data itself across networks and routing.
- Primary protocol: IP (Internet Protocol).
- **Link/Physical Layer**
 - Manages actual byte transfer over the physical medium.
 - Examples: WiFi, Ethernet.

3.1 TCP/IP Simplified Demonstration

To better understand how these layers interact, Figure 3 visualizes the data transmission process.

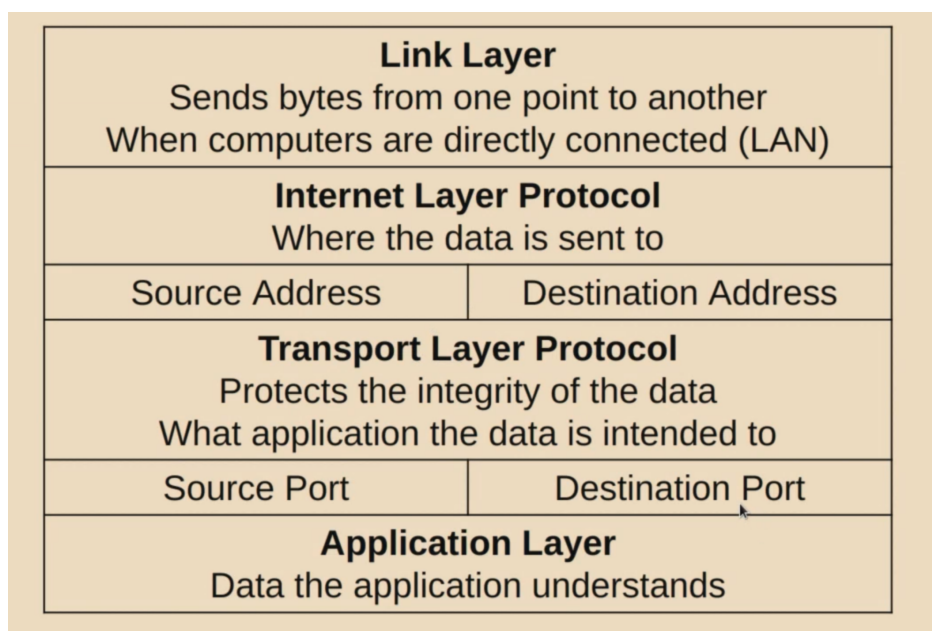


Figure 3: TCP/IP layered structure showing the role of each layer in communication.

3.2 The Transport Layer

Let's look more closely at one of the most critical layers in the TCP/IP model.

The Network/Internet Layer (primarily using IP) is responsible for packet transmission from computer to computer, but this communication is inherently unreliable. IP packets may be lost, duplicated, or arrive out of order without any built-in IP mechanisms to detect or correct these issues.

This is why the Transport Layer is essential. It is responsible for application-to-application (or process-to-process) transmission of packets and can add reliability mechanisms missing from the Network Layer. Depending on the protocol used (e.g., TCP), it may supply guaranteed delivery, error detection, packet reordering, and flow control.

This layer also enables multiple clients to connect to a single server (or multiple applications on one host to communicate simultaneously) through a mechanism called multiplexing, which directs incoming data to the correct application using port numbers.

Two main protocols in this layer are TCP and UDP, which offer different trade-offs:

- **TCP (Transmission Control Protocol)** - Provides reliable, connection-oriented service with guaranteed delivery, in-order arrival, and error detection.
- **UDP (User Datagram Protocol)** - Provides faster, connectionless service without reliability guarantees, suitable for applications where speed is more important than perfect reliability (e.g., streaming, online gaming, DNS).

3.2.1 Ports

The Network/Internet layer protocol (IP) is responsible for data transmission from one machine to another. However, once data arrives at a machine, we need a way to direct it to the correct application among potentially many running applications. This is where ports come in.

A port is a unique number that identifies a specific communication endpoint for an application or process on a host machine. The transport layer protocol (TCP or UDP) uses port numbers to distinguish between different applications.

The packet header at the transport layer contains source and destination port numbers. This is how the OS knows which application the incoming data is associated with. For example, if a web server application is "listening" on port 80, any incoming TCP packets destined for port 80 on that machine's IP address will be directed to the web server application.

Ports allow multiple applications on the same machine to use network resources simultaneously. Each active TCP connection or UDP communication is uniquely identified by a combination of: (Source IP, Source Port, Destination IP, Destination Port, Protocol). By default, a specific port number on a given IP address using a specific protocol (TCP or UDP) can only be bound to one socket (application process) at a time.

3.2.2 Common Ports

- Ports are numbers, typically ranging from 0 to 65535.
 - Ports 0-1023 are "well-known ports" or "system ports," often requiring administrative privileges to use.
 - Ports 1024-49151 are "registered ports."
 - Ports 49152-65535 are "dynamic" or "private" or "ephemeral ports."
- As mentioned before, protocols of the *Application Layer* use *Transport Layer* protocols (TCP or UDP) and often have conventional port numbers associated with them. Here are some examples:
 - FTP (File Transfer Protocol): Port 21 (Control), Port 20 (Data) - TCP
 - SSH (Secure Shell): Port 22 - TCP

- SMTP (Simple Mail Transfer Protocol): Port 25 - TCP
- HTTP (Hypertext Transfer Protocol): Port 80 - TCP
- HTTPS (HTTP Secure): Port 443 - TCP
- DNS (Domain Name System): Port 53 - Primarily UDP (sometimes TCP for large transfers)
- Many more!

3.2.3 UDP - User Datagram Protocol

- The simplest transport layer protocol.
- The header contains:
 - Source Port and Destination Port.
 - Length of the UDP datagram (header + data).
 - Checksum (an optional integrity-check method for the header and data).
- Unreliable:
 - Duplication/Loss of packets is possible. UDP itself does not retransmit lost packets.
 - Out-of-order delivery of datagrams is possible.
- Connectionless: No prior "handshake" or connection setup is required before sending data. Each datagram is handled independently.

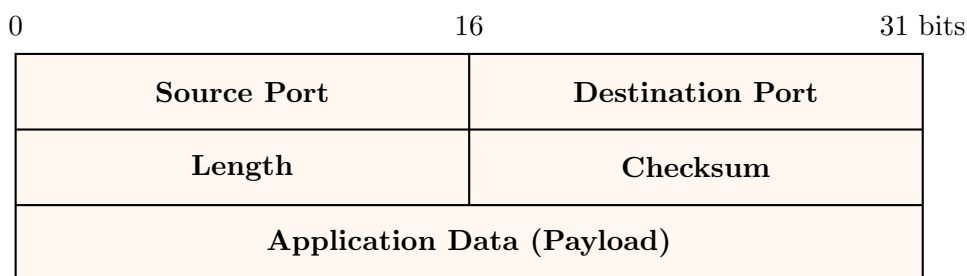


Figure 4: UDP Header Structure. The UDP header is a fixed 8 bytes.

- Each of the four fields in the UDP header (Source Port, Destination Port, Length, Checksum) is 16 bits (2 bytes) long.
- The entire header is only 8 bytes, contributing to UDP's low overhead compared to TCP.
- UDP provides no guarantees for message delivery or protection against duplication. Reliability must be handled by the application layer if needed.

3.2.4 TCP – Transmission Control Protocol

- A reliable, stream-oriented transport protocol.
- Connection-oriented:
 - A three-way ”handshake” (SYN, SYN-ACK, ACK) is mandatory before communication can begin to establish a connection.
 - Sender and receiver agree on parameters like sequence numbers during this handshake.
 - A formal connection termination (e.g., using FIN segments) is also performed.
- Takes care of:
 - Lost/Duplicated packets (using acknowledgments and retransmissions).
 - Out-of-order packets (using sequence numbers to reassemble the data stream correctly).
 - Flow control (to prevent a fast sender from overwhelming a slow receiver).
 - Congestion control (to manage network traffic).
 - Data buffering.

3.2.5 Transport Layer Summary

Property	UDP	TCP
Reliable?	X (No)	✓ (Yes)
Connection type	Connectionless	Connection-oriented
Flow control	X (No)	✓ (Yes)
Latency	Low	High
Applications	VOIP, Most games	HTTP, HTTPS, FTP, SMTP, Telnet, SSH

Table 1: Comparison of UDP and TCP Properties

3.2.6 TLS (Transport Layer Security)

- Until now, we’ve discussed data transport, but not necessarily secure transport.
 - Protocols like TLS (Transport Layer Security) and its predecessor SSL (Secure Sockets Layer) are responsible for providing secure communication channels.
 - TLS operates above a reliable transport-layer protocol, typically TCP.
 - The TLS layer encrypts (and often authenticates) the application data, forming a TLS record (or packet).
 - This entire TLS record becomes the payload (data) of a TCP segment.
 - The TCP segment (including the TLS record as its data) is then encapsulated within an IP packet for network transmission.
- TLS requires its own handshake process (distinct from the TCP handshake) to establish security parameters, exchange certificates, and agree on cryptographic keys before application data is transmitted securely.

3.2.7 QUIC (Quick UDP Internet Connections)

- Using TLS over TCP (common for HTTPS) requires two separate handshakes: one for TCP to establish the connection, and another for TLS to establish the secure session. This can introduce latency.
- QUIC is a relatively new transport layer network protocol designed to address this and other limitations. It is standardized by the IETF.
 - QUIC is built on top of UDP to leverage its speed and avoid TCP's head-of-line blocking issues, while implementing reliability and congestion control mechanisms itself.
 - Crucially, QUIC integrates TLS encryption directly into its own handshake process. This means it can establish a secure, reliable connection with a single handshake, reducing connection setup time.
 - It supplies both TLS-level security and TCP-like reliability features (e.g., stream multiplexing, per-stream flow control, loss recovery).

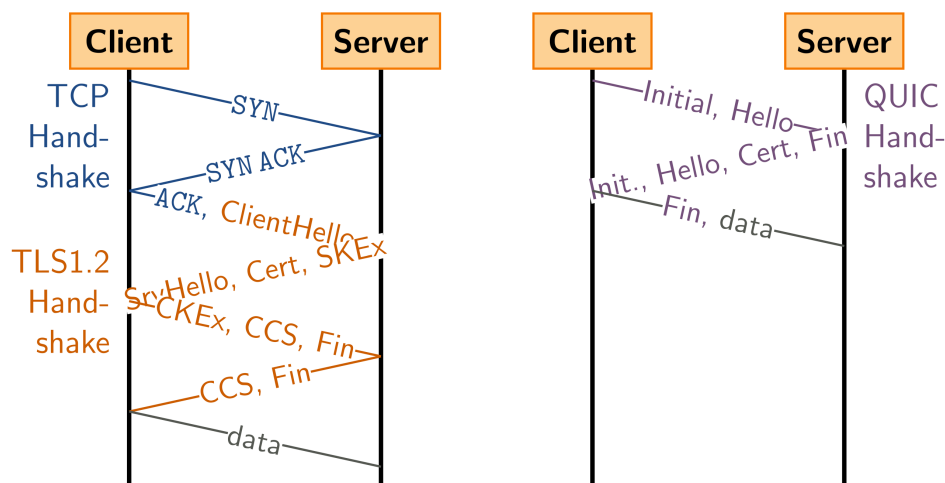


Figure 5: Comparison of TCP+TLS Handshake Latency vs. QUIC's Integrated Handshake.

4 Sockets

We have discussed protocols, but how do applications actually use them? How does the Operating System (OS) enable programs to utilize TCP and UDP for network communication? The answer lies in the Sockets API.

The Sockets API is a set of system calls (functions provided by the OS) that offer a standardized interface for processes to perform network communication.

Definition 4.1 (Socket): A **socket** is a communication endpoint (often visualized as a door or gate). More formally, a socket is a host-local, application-created, OS-controlled interface through which an application process can both send and receive messages to or from another application process, either on the same machine or across a network.

Sockets support the client/server paradigm and can be used with various transport protocols, most commonly TCP (for reliable, stream-based communication) and UDP (for unreliable, datagram-based communication).

4.1 Sockets vs. Ports

Sockets and ports are related but distinct concepts:

- A **Port** is a numerical identifier (0-65535) within a host's operating system that is used by transport protocols (like TCP and UDP) to differentiate between multiple applications or services running on that host. It's essentially an address for a specific application process on a machine.
- A **Socket** is an instance of a communication endpoint that is bound to a specific IP address and port number, using a specific protocol (TCP or UDP). An application creates a socket to initiate or listen for network communication. You can think of a socket as one end of a two-way communication link between two programs on the network.

So, while a port is just a number, a socket is a more concrete software object that an application uses to send and receive data.

4.2 Client/Server Socket Model

The client/server model is a common architectural pattern for networked applications, facilitated by sockets:

4.2.1 Basic Architecture

- Typically, a single server process (or a set of server processes) provides services to multiple client processes.
- Clients request services, and the server responds to these requests.
- Communication occurs through socket endpoints on both the client and server machines.

4.2.2 Connection Protocol (Conceptual for TCP)

1. The server application creates a socket, binds it to a well-known IP address and port number on the server machine, and then "listens" for incoming connection attempts from clients.
2. A client application creates a socket and initiates a connection to the server's IP address and port number.

3. When the server receives a client's connection request (for connection-oriented protocols like TCP), it typically "accepts" the connection. This often results in the server creating a new, dedicated socket specifically for communication with that particular client.

- This new socket on the server side uniquely identifies the connection with that specific client.
- The server's original "listening" socket remains available to accept new connection requests from other clients.

The creation of separate sockets (or handling mechanisms) for each client connection allows the server to:

- Handle multiple clients concurrently (e.g., by using threads, processes, or asynchronous I/O with the new socket).
- Maintain isolated communication sessions with different clients.
- Keep the main listening socket free to accept new incoming connections.

4.3 TCP Client/Server Socket Communication Flow

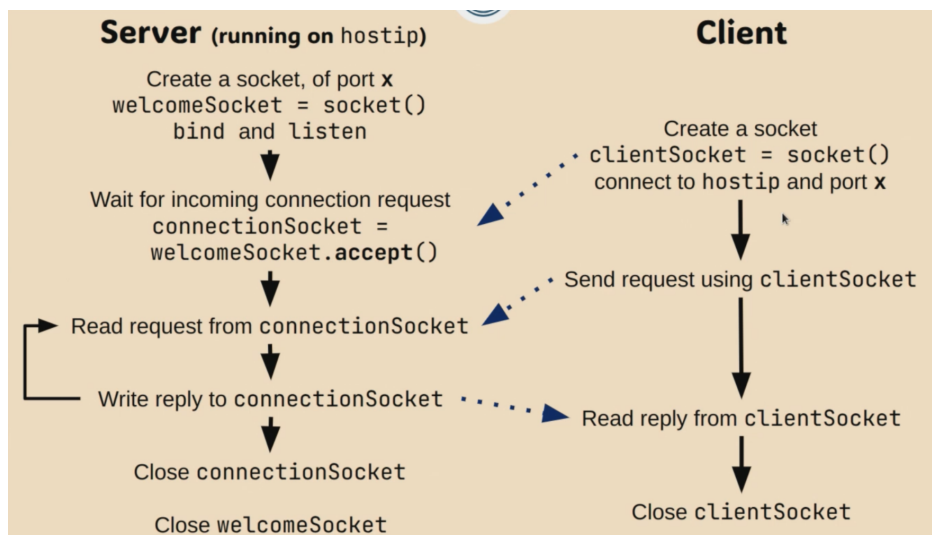


Figure 6: The Client/Server Model - TCP socket operations sequence.

The diagram in Figure 6 illustrates the typical sequence of socket system calls and operations in a TCP client/server interaction:

4.3.1 Server Side Operations

1. `welcomeSocket = socket()`: Creates an initial socket (the "listening" or "welcome" socket).

2. `bind(welcomeSocket, ...)` and `listen(welcomeSocket, ...)`: Assigns an address (IP and port `x`) to the `welcomeSocket` and marks it as passive, ready to accept incoming connection requests.
3. `connectionSocket = welcomeSocket.accept()`: Waits (blocks) for an incoming connection request. When a client connects, `accept()` creates a new socket (`connectionSocket`) dedicated to this specific client and returns its descriptor.
4. `read(connectionSocket, ...)`: Reads the client's request from the `connectionSocket`.
5. `write(connectionSocket, ...)`: Processes the request and writes the reply to the `connectionSocket`.
6. `close(connectionSocket)`: Closes the socket dedicated to this client after the transaction is complete.
7. Optionally, `close(welcomeSocket)`: Closes the listening socket if the server is shutting down.

4.3.2 Client Side Operations

1. `clientSocket = socket()`: Creates the client's socket.
2. `connect(clientSocket, server_hostip, server_port_x)`: Initiates a connection to the server at its IP address (`hostip`) and port (`x`). This triggers the TCP three-way handshake.
3. `write(clientSocket, ...)`: Sends a request to the server using `clientSocket`.
4. `read(clientSocket, ...)`: Reads the server's reply from `clientSocket`.
5. `close(clientSocket)`: Closes the client's socket when communication is complete.

4.3.3 Key Concepts Illustrated

- The server typically maintains two types of sockets in this model:
 - The `welcomeSocket` (or listening socket): Dedicated to listening for and accepting new incoming connections.
 - One or more `connectionSocket(s)`: Each created by `accept()` to handle communication with a specific, connected client.
- The client uses a single socket (`clientSocket`) for the entire communication session with the server.
- The dotted arrows in the diagram represent the logical flow of data transmission between the client and server over the established connection.
- The solid arrows show the sequence of system calls or operations performed on each side (server and client).

5 Socket Addressing in Network Programming

Effective network communication relies on correctly specifying the addresses of the communicating endpoints.

5.1 Socket Address Structure Basics

- Socket communication requires addressing mechanisms to identify both local and remote endpoints.
- An address specification typically includes:
 - Address Family: Indicates the type of address being used (e.g., IPv4, IPv6). Represented by constants like `AF_INET` (for IPv4) or `AF_INET6` (for IPv6).
 - Protocol-Specific Addressing Data: Such as an IP address and a port number.
- The Sockets API uses a generic socket address structure, `struct sockaddr`, to pass address information to system calls. However, this structure is not meant to be used directly by programmers for filling in address details.

Listing 1: Generic Socket Address Structure

```
1 struct sockaddr {
2     unsigned short    sa_family; /* address family, AF_*** */
3     char              sa_data[14]; /* 14 bytes of protocol address */
4 };
```

The `sa_data` field is problematic because the actual address format and size vary depending on the address family.

5.2 IPv4 Address Structure

- Because the generic `struct sockaddr` is difficult to use directly, protocol-specific structures are defined. For IPv4, the `struct sockaddr_in` (where "in" stands for Internet) is used:

Listing 2: IPv4 Socket Address Structure

```
1 struct sockaddr_in {
2     short              sin_family; /* Address family (AF_INET) */
3     unsigned short     sin_port; /* Port number (Network Byte
4     Order) */
5     struct in_addr     sin_addr; /* IP address (Network Byte Order) */
6     unsigned char      sin_zero[8]; /* Padding to make structure the
7                                     same size as sockaddr.
8                                     Must be set to all zeros. */
9 };
```

```

9  /* Structure for storing an IPv4 address */
10 struct in_addr {
11     uint32_t      s_addr;          /* 32-bit IPv4 address (Network
        Byte Order) */
12 };

```

- For IPv6 addresses, a similar structure named `struct sockaddr_in6` is used.
- When passing an address to a socket function that expects a `struct sockaddr*`, you typically cast a pointer to your protocol-specific structure (e.g., `(struct sockaddr*)&my_ipv4_addr`).

5.3 Byte Ordering in Socket Programming

Computers can store multi-byte numbers in memory in two primary ways (endianness):

- **Big-Endian:** The most significant byte (MSB) of the number is stored at the lowest memory address. Example: The number 0x0A0B0C0D would be stored in memory as bytes 0A, 0B, 0C, 0D.
- **Little-Endian:** The least significant byte (LSB) of the number is stored at the lowest memory address. Example: The number 0x0A0B0C0D would be stored as 0D, 0C, 0B, 0A.

Different computer architectures use different endianness (e.g., Intel x86 is little-endian). To ensure interoperability in network communication:

- Network protocols standardize on **Big-Endian** as the "Network Byte Order" for multi-byte fields in packet headers, including IP addresses and port numbers.
- Your host machine might use a different byte order ("Host Byte Order").
- The Sockets API provides functions to convert values between host byte order and network byte order:
 - `htons()`: "Host to Network Short" - Converts a 16-bit unsigned integer (e.g., port number) from host byte order to network byte order.
 - `htonl()`: "Host to Network Long" - Converts a 32-bit unsigned integer (e.g., IPv4 address) from host byte order to network byte order.
 - `ntohs()`: "Network to Host Short" - Converts a 16-bit unsigned integer from network byte order to host byte order.
 - `ntohl()`: "Network to Host Long" - Converts a 32-bit unsigned integer from network byte order to host byte order.

It is crucial to use these functions for port numbers and IP addresses when populating socket address structures or interpreting them from received packets.

5.4 Converting IP Addresses (Text to Binary)

- IPv4 addresses are often written in a human-readable "dotted-decimal" notation (e.g., "192.168.1.100"). For use in socket address structures, this must be converted to its 32-bit binary representation (in network byte order).
- The function `inet_aton()` ("ASCII to Network") serves this purpose:

Listing 3: Converting Dotted-Decimal IP to Binary

```
1 /* Example: Converting IP address from string to binary network
   format */
2 // Assuming my_addr is a struct sockaddr_in
3 inet_aton("10.12.110.57", &(my_addr.sin_addr));
```

- **Warning:** Unlike many C functions, `inet_aton()` returns 0 on failure and a non-zero value on success.
- Other functions like `inet_addr()` (older, less safe) and `inet_pton()` ("presentation to network", preferred for IPv4/IPv6) also exist.
- Required header files for these functions typically include:

```
1 #include <sys/socket.h> /* For socket structures */
2 #include <netinet/in.h> /* For sockaddr_in, in_addr, htons, etc.
   */
3 #include <arpa/inet.h> /* For inet_aton, inet_pton, etc. */
```

5.5 Complete Socket Address Initialization (IPv4 Example)

Here's how you would typically initialize a `struct sockaddr_in` for an IPv4 address:

Listing 4: Initializing an IPv4 Socket Address Structure

```
1 /* Create and initialize IPv4 socket address */
2 struct sockaddr_in my_addr;
3
4 my_addr.sin_family = AF_INET; /* Set address family
   to IPv4 */
5 my_addr.sin_port = htons(3490); /* Set port number (e
   .g., 3490),
6                                     converted to
                                     network byte
                                     order */
7 /* Set IP address from dotted-decimal string, converted to network
   byte order binary */
8 if (inet_aton("10.12.110.57", &(my_addr.sin_addr)) == 0) {
9     // Handle error: invalid IP address string
10    perror("inet_aton failed");
11    // exit or return error
```



```
12 }  
13 memset(&(my_addr.sin_zero), 0, sizeof(my_addr.sin_zero)); /* Zero  
    out the padding */
```

5.6 Implementation Details and Best Practices

- **Type Casting:** When calling socket functions like `bind()` or `connect()` that expect a `const struct sockaddr*`, you must cast the address of your protocol-specific structure (e.g., `(struct sockaddr*)&my_addr`).
- **Padding:** The `sin_zero` field in `struct sockaddr_in` must be entirely filled with zeros using `memset()`. This is for compatibility with the generic `struct sockaddr`.
- **Byte Ordering:** Always ensure that `sin_port` and `sin_addr.s_addr` are in Network Byte Order. Use `htons()` for the port and ensure functions like `inet_aton()` or `htonl()` (if converting a manually constructed IP) are used for the address.
- **Error Checking:** Always check the return values of socket functions and address conversion functions for errors.

6 Sockets Programming (TCP)

TCP socket programming involves creating endpoints for reliable, connection-oriented communication between processes, often across a network. This section details the key steps and system calls for establishing and using a TCP connection.

6.1 Creating a Socket (`socket()`)

The first step for both client and server is to create a socket, which is an endpoint for communication. This is done using the `socket()` system call:

Listing 5: `socket()` System Call

```
1 int socket(int domain, int type, int protocol);
```

- **domain:** Specifies the communication protocol family (address family).
 - `AF_INET`: For IPv4 Internet protocols.
 - `AF_INET6`: For IPv6 Internet protocols.
- **type:** Determines the socket type, indicating the semantics of communication.
 - `SOCK_STREAM`: For reliable, two-way, connection-based byte streams (used by TCP).
 - `SOCK_DGRAM`: For connectionless, unreliable datagrams (used by UDP).

- **protocol**: Specifies the actual protocol to use. Usually set to 0, which causes the system to select the default protocol for the given **domain** and **type** (e.g., TCP for **AF_INET** and **SOCK_STREAM**).

On success, `socket()` returns a non-negative integer called a **file descriptor** (or socket descriptor), which is used to refer to this socket in subsequent system calls. On failure, it returns -1 and sets **errno** to indicate the error.

6.2 Binding to an Address (`bind()`) - Server Side

After creating a socket, a server needs to associate it with a specific local IP address and port number on the server machine. This is done using the `bind()` system call:

Listing 6: `bind()` System Call

```
1 int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
    ;
```

- **sockfd**: The file descriptor of the socket returned by `socket()`.
- **addr**: A pointer to a socket address structure (e.g., `struct sockaddr_in` for IPv4, cast to `struct sockaddr*`) containing the local IP address and port to bind to. The IP address can be a specific interface's IP or **INADDR_ANY** (to listen on all available network interfaces). The port number must be in network byte order.
- **addrlen**: The size of the address structure pointed to by **addr**.

`bind()` returns 0 on success and -1 on error. Clients usually do not call `bind()`, as the OS can assign them an arbitrary local port number.

6.3 Listening for Connections (`listen()`) - Server Side

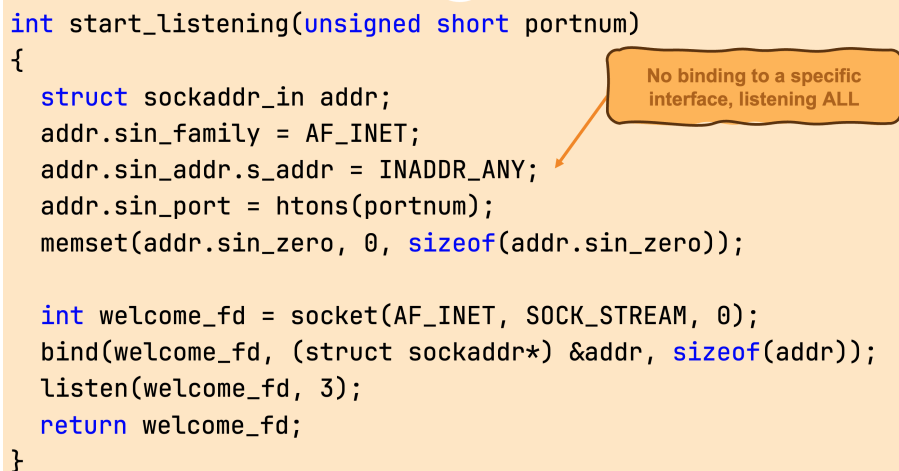
Once a server socket is bound to an address, the server must indicate its willingness to accept incoming connection requests. This is done by calling `listen()`, which marks the socket as a passive (listening) socket:

Listing 7: `listen()` System Call

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

- **sockfd**: The file descriptor of the bound socket.
- **backlog**: An integer specifying the maximum length of the queue for pending connections. If the queue is full and a new connection request arrives, the client might receive an error (e.g., **ECONNREFUSED**).

`listen()` returns 0 on success and -1 on error. This call prepares the socket to accept incoming client connection requests; the TCP stack will queue these requests up to the backlog limit.



```
int start_listening(unsigned short portnum)
{
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(portnum);
    memset(addr.sin_zero, 0, sizeof(addr.sin_zero));

    int welcome_fd = socket(AF_INET, SOCK_STREAM, 0);
    bind(welcome_fd, (struct sockaddr*) &addr, sizeof(addr));
    listen(welcome_fd, 3);
    return welcome_fd;
}
```

No binding to a specific interface, listening ALL

Figure 7: Server Listening - Code example showing binding to all interfaces (`INADDR_ANY`) and listening.

6.4 Accepting Connections (`accept()`) - Server Side

After calling `listen()`, the server waits for and accepts incoming connection requests using the `accept()` system call:

Listing 8: `accept()` System Call

```
1 int accept(int sockfd, struct sockaddr *cli_addr, socklen_t *
    cli_addrlen);
```

- **sockfd**: The file descriptor of the listening socket.
- **cli_addr**: (Optional) A pointer to a buffer that, on return, will be filled with the address information of the connecting client. If not needed, can be `NULL`.
- **cli_addrlen**: (Optional) A pointer to a `socklen_t` variable. Before the call, it should be set to the size of the buffer pointed to by `cli_addr`. On return, it will contain the actual size of the client's address structure. If `cli_addr` is `NULL`, this should also be `NULL`.

The `accept()` call typically blocks (waits) until a client attempts to connect to the listening socket. When a connection is established, `accept()` creates a **new socket** (a connected socket) for this specific client-server communication and returns its file descriptor. The original listening socket (`sockfd`) remains open and continues to listen for more connection requests from other clients. The new socket descriptor is used for all subsequent communication (`read()`, `write()`) with this particular client. `accept()` returns -1 on error.

6.5 Client Connection Process (`connect()`) - Client Side

The client side initiates a connection to a server:

1. Create a socket using `socket()` (similar to the server).
2. Establish a connection to the server using the `connect()` system call:

Listing 9: `connect()` System Call

```
1 int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t  
   addrlen);
```

- **sockfd**: The file descriptor of the client's socket.
- **serv_addr**: A pointer to a socket address structure containing the server's IP address and port number.
- **addrlen**: The size of the server's address structure.

When the client calls `connect()`, the OS attempts to establish a TCP connection with the server (performing the three-way handshake). If the server has called `listen()` on the specified port and `accept()` is pending or called subsequently, the connection is established. `connect()` returns 0 on success and -1 on error.

6.6 Data Communication (`read()/write()`, `recv()/send()`)

Once a TCP connection is established (after `connect()` succeeds on the client and `accept()` returns a new socket on the server), both server and client can exchange data using:

- Standard file I/O system calls: `read()` to receive data and `write()` to send data.
- Socket-specific system calls: `recv()` to receive data and `send()` to send data (these offer more control via a `flags` parameter).

These functions use the connected socket descriptors: on the server, the descriptor returned by `accept()`; on the client, the descriptor returned by its `socket()` call.

6.7 Connection Lifecycle Example (TCP)

A typical TCP server follows this sequence of operations:

1. Create listening socket: `socket()`
2. Bind socket to local address/port: `bind()`
3. Listen for incoming connections: `listen()`
4. Accept a client connection: `accept()` (returns a new socket descriptor for the client)

5. Exchange data with the client: `read()/recv()`, `write()/send()` (using the new socket descriptor)
6. Close client connection: `close()` (the new socket descriptor)
7. (Repeat step 4-6 for other clients, or close listening socket if done)

A typical TCP client follows this sequence:

1. Create socket: `socket()`
2. Connect to server: `connect()`
3. Exchange data with server: `write()/send()`, `read()/recv()`
4. Close connection: `close()`

To handle multiple clients concurrently, a server can use various models, such as creating a new thread or process for each connection returned by `accept()`, or using non-blocking I/O with I/O multiplexing functions like `select()`, `poll()`, or `epoll()`.

This socket programming model provides the foundation for network communication in most client-server applications, from web servers to email clients and beyond.

6.8 Understanding Network Interfaces and Binding

A single machine can be connected to multiple networks simultaneously through different physical or virtual network interfaces (e.g., Ethernet card, Wi-Fi adapter, loopback interface). Each active interface typically has its own IP address.

- Network interfaces include the "loopback" interface (used for communication within the same host, often with IP address 127.0.0.1), Ethernet interfaces, WLAN (Wi-Fi) interfaces, and others.
- A machine can have multiple IP addresses, often one per active network interface.
- Generally, the same IP address cannot be assigned to different active interfaces on the same machine within the same network.

When a server binds a socket using `bind()`:

- It supplies an IP address and a port number.
- The IP address determines which network interface(s) the server will listen on.
 - Specifying a particular IP address restricts listening to that specific interface.
 - Specifying `INADDR_ANY` (for IPv4) or `in6addr_any` (for IPv6, often by initializing `sin6_addr` to zeros) tells the OS to listen for incoming connections on all available network interfaces of the machine.
- The port number specifies the service endpoint on that IP address.

The file descriptor returned by system calls like `socket()` is an integer ID that the user process uses to refer to the socket when making further system calls to the OS (e.g., `bind`, `listen`, `read`, `write`, `close`). This is analogous to how file descriptors work for regular file operations.

6.9 Advanced Data Communication Details

Once a connection is established, data communication occurs through the connected socket using read and write operations.

6.9.1 Basic Read and Write Operations (`read()`/`write()`)

For `read()`:

- Attempts to read AT MOST `bytes_to_read` bytes from the socket associated with file descriptor `fd` into the buffer pointed to by `buff`.
- Returns the number of bytes actually read. This can be less than `bytes_to_read` if fewer bytes are currently available or if a signal interrupts the call.
- Returns 0 if the peer has performed an orderly shutdown (end-of-file on the stream).
- Returns -1 on error, and `errno` is set to indicate the specific error.
- By default, `read()` on a blocking socket will block until at least one byte of data is available or an error/EOF occurs.
- It does not necessarily read all `bytes_to_read` in a single call, even if more data is available in the socket's receive buffer. TCP is a stream protocol, not a message protocol.
- Therefore, you must typically loop to ensure you read the desired amount of data or handle partial reads appropriately.

An example of a loop for reading data:

Listing 10: Read Example

```
1 int read_data(int s, char *buf, int n)
2 {
3     int bcount;           /* counts bytes read */
4     int br;               /* bytes read this pass */
5     bcount = 0; br = 0;
6
7     while(bcount < n) { /* loop until full buffer */
8         br = read(s, buf, n-bcount);
9         if((br > 0) ) {
10             bcount += br;
11             buf += br;
12         }
13         if(br < 1) {
14             return -1;
```

```
15     }
16 }
17 return bcount;
18 }
```

6.9.2 Advanced Socket I/O (`send()`/`recv()`)

For more fine-grained control over socket I/O, the `send()` and `recv()` system calls are often preferred:

Listing 11: `send()` and `recv()` System Calls

```
1 ssize_t send(int sockfd, const void *buff, size_t len, int flags);
2 ssize_t recv(int sockfd, void *buff, size_t len, int flags);
```

- These functions are similar to `write()` and `read()` respectively, but they include an additional `flags` parameter.
- The `flags` parameter allows for various options, such as:
 - `MSG_DONTWAIT` (for `recv`): Enables non-blocking operation for this call only.
 - `MSG_PEEK` (for `recv`): Allows peeking at the incoming data in the receive buffer without actually removing it.
 - `MSG_OOB` (for `send/recv`): For sending/receiving out-of-band data.
 - And others, depending on the OS and protocol.
- If `flags` is set to 0, `send()` behaves like `write()`, and `recv()` behaves like `read()`.