Network Programming

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

A brief overview

In the context of network programming, "networking" refers to the practice and techniques involved in designing, implementing, and managing communication between computers and devices over a network. This can include a wide array of tasks and principles, including but not limited to:

- **Data Communication:** The fundamental aspect of networking, involving the exchange of data between two or more devices over a network. This can be achieved through various communication protocols and standards.
- **Protocols and Standards:** Networking relies on a set of rules and conventions (protocols) for communication between network devices. These protocols define how data is formatted, transmitted, and received. Examples include TCP/IP (Transmission Control Protocol/Internet Protocol), HTTP (HyperText Transfer Protocol), and FTP (File Transfer Protocol).
- **Network Architecture:** The design and layout of a network, including its components (e.g., routers, switches, gateways) and topology (e.g., star, mesh, ring). Network architecture decisions impact the network's performance, scalability, and security.
- **Socket Programming:** A means of connecting two nodes on a network to communicate with each other. One node listens on a particular port at an IP, while another node connects to it. Socket programming is used to facilitate communication between applications running on different computing devices.
- **APIs for Network Communication:** Programming interfaces such as Winsock for Windows, POSIX sockets for Unix/Linux, and various cross-platform networking libraries (e.g., Boost.Asio) that allow developers to implement networking functionalities.
- **Network Services Development:** Creating software that provides specific functionalities over a network, such as web servers, email servers, and file sharing systems.
- **Network Security:** Ensuring the confidentiality, integrity, and availability of data in the network. This includes implementing secure protocols (like HTTPS), encryption, firewalls, and intrusion detection systems.
- **Network Management:** Monitoring and maintaining network operations. This involves performance analysis, troubleshooting network problems, and ensuring that network resources are allocated efficiently.

Network Terminology

2.1 Nodes, links, and paths

node refers to any device that can send, receive, or forward information over a communications channel. Nodes can be computers, mobile devices, routers, switches, and other devices capable of processing or storing data.

A link, on the other hand, is the physical or logical connection between two or more nodes, enabling them to communicate. Links can be wired connections like Ethernet cables, optical fibers, or wireless connections such as Wi-Fi or Bluetooth.

Together, nodes and links form the basic components of a network, allowing for the transmission of data across diverse and complex systems.

A path is a sequence of nodes and links

In other words...

- **Node:** Host or intermediary
- **Link:** Point-to-point or broadcast to many other nodes at the same time
- **Link medium:** wired or wireless
- **Path:** Routed or switched (Elaborated in later section)

2.2 Networking protocol

Networking protocols are standardized sets of rules that determine how data is transmitted and received across a network. These protocols specify the formats for data packets, the procedures for signaling, error handling, and data encryption to ensure successful communication between devices.

More broadly, information is exchanged between nodes via **messages**, each message has an exact meaning intended to provoke a defined response of the reciever

Note:-

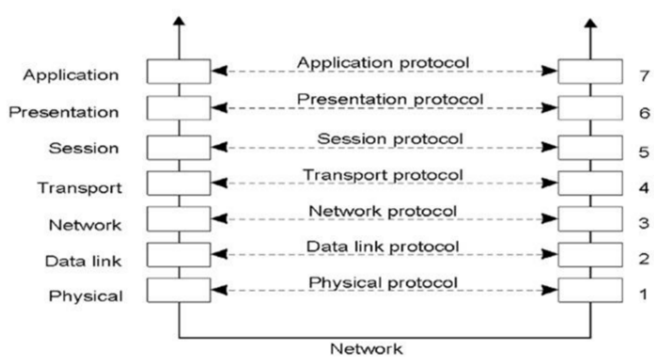
Messages used **well-defined format**

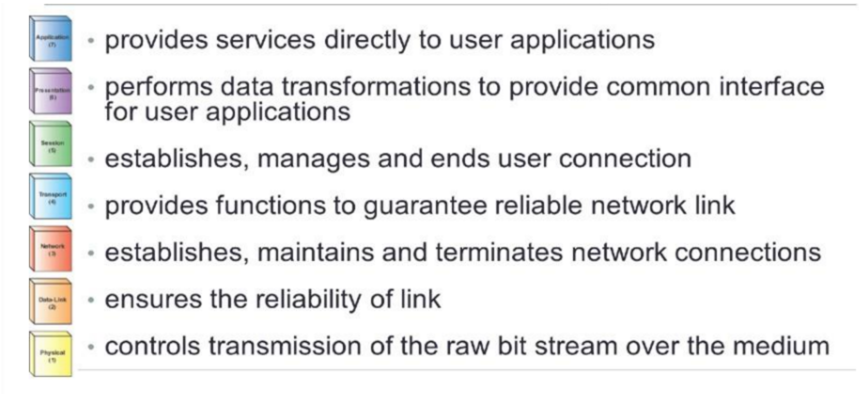
The OSI (Opens systems interconnection) Model

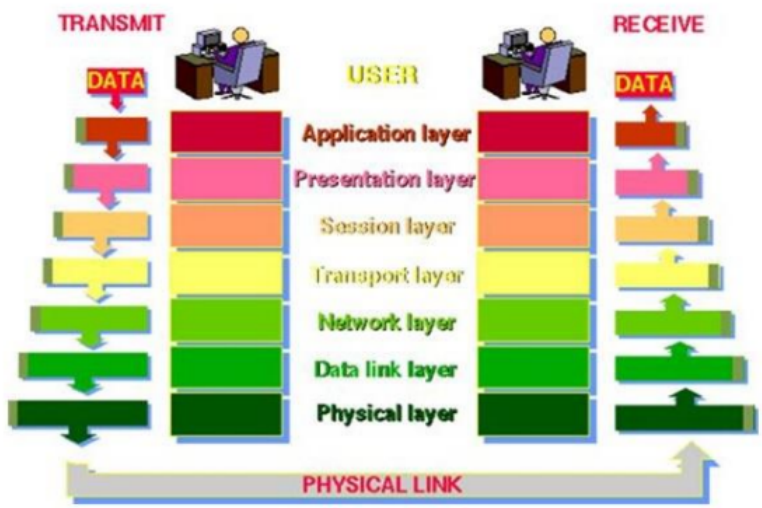
The OSI (Open Systems Interconnection) model is a conceptual framework used to understand and standardize the functions of a telecommunications or computing system without regard to its underlying internal structure and technology. Developed by the International Organization for Standardization (ISO), the OSI model divides the process of communication between two end-points in a network into seven layers. Each layer serves a specific function and communicates with the layers directly above and below it. From top to bottom, the layers are **APS TNDP**:

- **Application Layer (Layer 7):** The closest to the end user, this layer interacts with software applications that implement a communicating component. It provides protocols that allow software to send and receive information and present meaningful data to the user (e.g., HTTP for web browsing, SMTP for email).
- **Presentation Layer (Layer 6):** Translates data between the application layer and the network format. It ensures data is in a usable format and can encrypt or compress data if necessary.
- **Session Layer (Layer 5):** Manages sessions between applications, establishing, managing, and terminating connections between local and remote applications.
- **Transport Layer (Layer 4):** Responsible for data transfer between end systems and provides reliable data transfer services to the upper layers. This includes breaking down messages into smaller units if needed, and ensuring error-free data transfer (e.g., TCP and UDP).
- **Network Layer (Layer 3):** Manages device addressing, identifies the best paths for data transmission, and routes data packets between devices that are not locally attached. Routers operate at this layer.
- **Data Link Layer (Layer 2):** Provides data transfer across the physical link established by the physical layer. It deals with MAC addresses, error detection and correction, and frames data packets.
- **Physical Layer (Layer 1):** Concerns the physical equipment involved in data transfer, such as cables, switches, and the electrical signals that traverse these media.

In other words, these seven layers help to describe communications in a network







3.1 Main idea

The complexities of communication is organized into successive layers of protocols

- **Lower-level layers**: Specific to medium
- **Higher-level layers**: Specific to application

3.2 Physical layer: Wired media

- **Ethernet**
  - 10BASE-T, 100BASE-T, 1000BASE-T
  - 10GbE, 40GbE, 100GbE
- **Business/backbone**
  - DS1(T1): 1.54Mbs to DS5: 400Mbs
  - OC-1: 50Mbs to OC-768: 40Gbs
- **Last mile:**
  - Modem
  - DSL (Digital subscriber lines)
  - Cable: DOCSIS
  - FiOS (Fiber optic service)

3.3 Physical layer: Wireless media

- **Cellphone Data:**
  - EDGE, GPRS, HSPA+
  - 4G LTE up to 100Mbs
  - 5G over 100Mbs
- **Satellite**
  - Wildblue: 12Mbs
  - Hughesnet: 15Mbs
  - Starlink: 200Mbs
- **WiFi:** 802.11
  - Up to 150Mbs & MIMO
  - New: ”ac” up to 1Gbs
- **WiMax:** 802.16
  - up to 40Mbs
- **WPAN**
  - Bluetooth up to 2Mbs
  - NFC up to 423Kbs
  - ZigBee up to 256Kbs

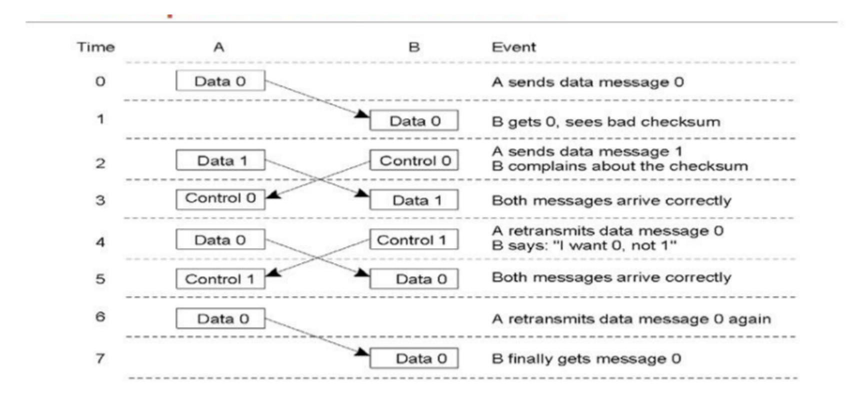
3.4 Data link layer: Functionalities

- **Medium access control**: Arbitrate who transmits
- **Addressing**: address of receiver, address of sender
- **Framing:** Delimited unit of transmission for data & control
- **Error control and reliability**
- **Flow control**

3.4.1 Example: Ethernet frame

Preamble	Destination MAC address	Source MAC address	Type/ Length	User Data	Frame Check Sequence (FCS)
8	6	6	2	46 - 1500	4

3.4.2 Example: Data link flow



3.5 Network layer (Internet protocol layer)

Provides host to host transmission service, where hosts are not necessarily adjacent

Layer provides services

- Addressing
  - Hosts have global addresses: IPv4, IPv6
  - Uses data link layer protocol to translate address
- **Routing and forwarding:** Find path from host to host

3.5.1 IPv4 Address

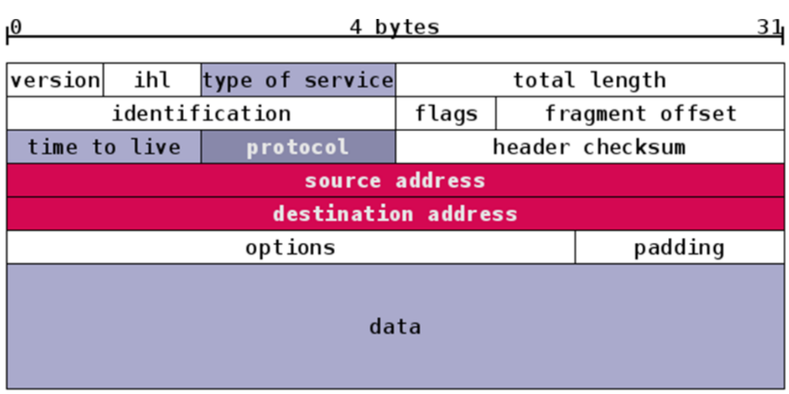
- IP address
  - 32bit unique identifier, written as quad
- network
  - first n bits of IP number, written as "/n"
  - 8 - class A, 16 - class B, 24 - class C
  - more than 24 - class D
- netmask
  - 32 bit number with first n bits all 1, rest 0
- broadcast
  - network number (first n bits), rest all 1
- gateway IP
- name server IP

- 127.0.0.1
- 131.156.145.90
- 131.156.0.0/16
- 131.156.145.0/24
- 255.255.255.0
- 131.156.145.255
- 131.156.145.1
- 131.156.145.2

3.5.2 IPv6 Address

- IP address: 128-bit unique identifier
- 8 groups of 16-bit values,
  each group in 4 hex digits, separated by ":"
  - ex.: `2001:0db8:0000:0000:0000:ff00:0042:8329`
- can be abbreviated:
  - remove leading zeroes: `42` instead of `0042`
  - omit consecutive sections of zeroes: `2001:db8::ff00:42:8329`

3.5.3 IP Packet



3.6 IP Layer: Routing and Forwarding

Done by hosts on path from sending to reciever

- **Forwarding:** Host has 2 network interfaces, transfers packet from incoming to outgoing interface
- **Routing:**
  - Finds path from sender to receiver
  - **Simple routing:** know receiver or send to gateway
  - **Advanced routing:** determine which gateway to send to (typically with multiple outgoing network interfaces)

3.7 Transport layer

Provides end-to-end communication services for applications

Btye format as abstraction on underlying system format. Raises reliability

Also enables multiplexing, which provides multiple endpoints on a single node: **port**.

Refines connection address via port number

3.8 Ports

- 0 to 1023: well-known ports
  - 20 & 21: File Transfer Protocol (FTP)
  - 22: Secure Shell (SSH)
  - 23: Telnet remote login service
  - 25: Simple Mail Transfer Protocol (SMTP)
  - 53: Domain Name System (DNS) service
  - 80: Hypertext Transfer Protocol (HTTP) used in the World Wide Web
  - 110: Post Office Protocol (POP3)
  - 119: Network News Transfer Protocol (NNTP)
  - 143: Internet Message Access Protocol (IMAP)
  - 161: Simple Network Management Protocol (SNMP)
  - 443: HTTP Secure (HTTPS)
- 1024 to 49151: IANA registered ports
- 49152 to 65535: dynamic or private port

3.8.1 Transport layer programming

- **Common abstraction:** Socket
- Socket is end-point of communication link
  - Identified as Ip address + port number
- operates as client and server

3.8.2 Transport layer protocols

- TCP: transmission control protocol
  - connection oriented, guaranteed delivery
  - stream oriented: basis for: http, ftp, smtp, ssh
- UDP: user datagram protocol
  - best effort
  - datagram oriented: basis for: dns, rtp
- DCCP: datagram congestion control protocol
- SCTP: stream control transmission protocol

Domain Name Service (DNS)

4.1 Domain Names

- hierarchical distributed naming system
- Uses **FQDN**: Fully qualified domain name
- **DNS:** Domain name service
  - Resolves query for FQDN into ip address

4.2 C library function: getaddrinfo

4.2.1 Signature

```c
int getaddrinfo(const char* node, const char* service, const
↪  struct addrinfo* hints, struct addrinfo** res)
```

- Translates FQDN **node** into IP address
- **res** is pointer to a list of address info structures

Note:-

service and **hints** can be NULL

4.2.2 Address info structure

```c
struct addrinfo {
    int                 ai_flags;
    int                 ai_family;
    int                 ai_socktype;
    int                 ai_protocol;
    size_t              ai_addrlen;
    struct sockaddr     *ai_addr; // Socket address
    char                *ai_cannonname;
    struct addrinfo     *ai_next;
};
```

4.2.3 Socket address info structure

```
struct sockaddr_in {
    short               sin_family  // e.g. AF_INET
    unsigned_short      sin_port    // Port
    struct in_addr      sin_addr    // IP Address
    char                sin_zero[8] // Padding
};

struct in_addr {
    unsigned long       s_addr;
};
```

Note:-

sin__addr can be printed via **inet__ntoa** fn

4.2.4 Example: Get host by name

```cpp
/*
* getHostName.cxx
*
*         do DNS lookup
*
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char*argv[]) {
    struct addrinfo *res;
    int error;
    const char *hostname = "faculty.cs.niu.edu";

    if (argc > 1) {
        hostname = argv[1];
    }

    error = getaddrinfo(hostname, NULL, NULL, &res);
    if (error) {
        cerr << hostname << ": " << gai_strerror(error) << endl;
        exit(EXIT_FAILURE);
    }

    // convert generic sockaddr to Internet sockaddr_in
    struct sockaddr_in *addr = (struct sockaddr_in *) res->ai_addr;
    // convert network representation into printable presentation
    cout << hostname << " is: " << inet_ntoa(addr->sin_addr) << endl;
}
```

4.3 gai_strerror

Concept 1: The **gai_strerror()** function shall return a text string describing an error value for the **getaddrinfo()** and **getnameinfo()** functions listed in the <netdb.h> header.

4.3.1 Signature

```
const char* gai_strerror(int ecode);
```

4.3.2 ecodes

When the ecode argument is one of the following values listed in the <netdb.h> header:

[EAI_AGAIN]
[EAI_BADFLAGS]
[EAI_FAIL]
[EAI_FAMILY]
[EAI_MEMORY]
[EAI_NONAME]
[EAI_OVERFLOW]
[EAI_SERVICE]
[EAI_SOCKTYPE]
[EAI_SYSTEM]

the function return value shall point to a string describing the error. If the argument is not one of those values, the function shall return a pointer to a string whose contents indicate an unknown error.

4.3.3 Return value

Upon successful completion, **gai_strerror()** shall return a pointer to an implementation-defined string.

User Datagram Protocol (UDP)

- Simple message-based connection-less protocol
  - Transmits information in one direction from source to destination without verifying the readiness or state of the reciever
- Uses **datagram** as message
- Stateless and fast

5.1 UDP Packet Format

bits	0 – 7	8 – 15	16 – 23	24 – 31
0	Source IP address			
32	Destination IP address			
64	Zeros	Protocol	UDP length	
96	Source Port		Destination Port	
128	Length		Checksum	
160+	Data			

5.2 UDP Progamming

- **Common abstraction:** Socket
- Socket is end-point of communication link
  - Identified as IP address + port number
  - can recieve data, can send data
- **Typical logic:** Server vs client
  - Server ready to recieve datagram from any client
  - client sends datagram to specific server
  - server responds with datagram to client

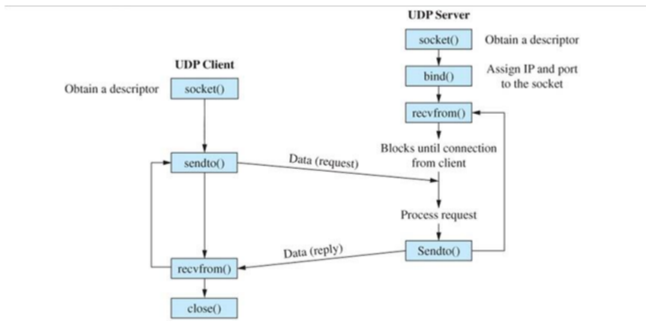
5.2.1 Socket system calls

server

client

System call	Meaning
socket	Create a new communication endpoint
bind	Attach a local address to a socket
recvfrom	Receive(read) some data over the connection
sendto	Send(write) some data over the connection
close	Release the connection

5.2.2 UDP communications pattern



5.3 System call: Socket

5.3.1 Signature

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

- Creates a new socket, as end point to a communications link
- **Domain** is set to **AF_INET**
- **Type** is set to **SOCK_DGRAM** for datagrams
- **Protocol** is set to 0, i.e. default UDP
- Returns socket descriptor
  - Used in **bind, sendto, recvfrom, close** (System calls)

5.4 System call: Bind

5.4.1 Signature

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

- **Assigns address to socket:** IP number and port
- **struct sockaddr** holds address information
  - Will accept **struct sockaddr_in** pointer
- **addrlen** specifies length of **addr** structure
- **Returns** 0 on success, -1 otherwise

5.4.2 Structure sockaddr: 16 bytes

```c
struct sockaddr {
    short       sa_family   // Address family
    char        sa_data[14] // Address data
};

struct sockaddr_in {
    short               sin_family   // address family
    unsigned short      sin_port     // port number: 2 bytes
    struct in_addr      sin_addr     // IP address: 4 bytes
    char                sin_zero[8]
};

struct in_addr {
    unsigned long       s_addr;
};
```

5.4.3 sockaddr_in members we care about

- **sin_family**: Always AF_INET in ipv4 communications
- **sin_port**: We use **htons(n)** fn to convert a c++ integer to a 16bit unsigned integer. e.g. **htons(4444)**
- **sin_addr**: Either
  - **INADDR_ANY**: For server, to specify the server IP
  - **inet_addr("numeric ip (quad) as string")**: For client.

5.5 Byte order

In C programming, especially when dealing with network applications, it's crucial to manage byte order correctly due to differences in how data is stored in memory across different computer architectures. Byte order refers to the sequence in which bytes are arranged into larger data types (like integers) when stored in memory. The two most common byte orders are:

1. **Big-endian:** The most significant byte (MSB) is stored at the smallest address, the next significant byte in the next address, down to the least significant byte (LSB) at the highest address.
2. **Little-endian:** The LSB is at the smallest address, and the MSB is at the highest address.

5.6 Helper functions: htons, htonl, ntohs, ntohl

To ensure data consistency across different systems, certain helper functions are used in network programming to convert between host and network byte orders. These are part of the POSIX (Portable Operating System Interface) standards, available in <arpa/inet.h> in C.

- **htonl() (Host TO Network Long):** Converts a 32-bit integer from host byte order to network byte order.
- **htons() (Host TO Network Short):** Converts a 16-bit integer from host byte order to network byte order.
- **ntohl() (Network TO Host Long):** Converts a 32-bit integer from network byte order to host byte order.
- **ntohs() (Network TO Host Short):** Converts a 16-bit integer from network byte order to host byte order.

When sending data across a network, you would typically use these functions to convert multi-byte integers from the host's native byte order to network byte order before transmission. Similarly, upon receipt, you would convert the data from network byte order back to the host's native byte order. This handling ensures that applications developed on different platforms can communicate effectively over a network.

For example, if you're developing a network application that sends a 32-bit integer from a little-endian machine (like most Intel and AMD processors), you would use htonl() to convert the integer to big-endian format before sending it. On the receiving side, if the machine is big-endian, it would use ntohl() to convert the received integer back to its native byte order.

5.6.1 Signatures

```c
uint16_t htons(uint16_t hostshort);
uint32_t htonl(uint32_t hostlong);
uint16_t ntohs(uint16_t netshort);
uint32_t ntohl(uint32_t netlong);
```

5.7 Helper function: inet_addr

The **inetb_addr** function is a commonly used helper function in C network programming, provided by the <arpa/inet.h> header file. Its primary purpose is to convert an IPv4 address in its standard text representation (a string in dotted-decimal format) into a numeric binary format, which is used in various network-related system calls and structures.

5.7.1 Signature

```
in_addr_t inet_addr(const char* cp)
```

Where **cp** is a constant character pointer to a null-terminated string representing the IPv4 address in dotted-decimal notation (e.g., ”192.168.0.1”).

5.7.2 Return values

- On success, **inet_addr** returns the IPv4 address as a uint32_t in **network byte order (big-endian)**.
- If the string in cp does not contain a valid IP address, the function returns **INADDR_NONE** (usually defined as (uint32_t) -1).

5.8 System call: recvfrom

5.8.1 Signature

```
ssize_t recvfrom(int sockfd, void* buf, size_t len, int flags,
↪  struct sockaddr* src_addr, socklen_t* addrlen)
```

- Recieves a datagram **buf** of size **len** from socket **sockfd**
  - will wait until a datagram is available
  - **flags** specify wait behavior, e.g: 0 for default
- **src_addr** will hold address information of sender
  - **struct sockaddr** defines address structure
  - **addrlen** specifies length of **src_addr** structure
- Returns the number of bytes recieved, i.e size of datagram

5.9 System call: sendto

5.9.1 Signature

```
ssize_t sendto(int sockfd, const void* buf, size_t len, int
↪  flags, const struct sockaddr* dest_addr, socklen_t addrlen)
```

- Sends datagram **buf** of size **len** to socket **sockfd**
  - Will wait if there is no ready reciever
  - Flags specifies wait behavior, e.g: 0 for default
- **dest_addr** holds address information of reciever
  - **struct sockaddr** defines address structure
  - **addrlen** specifies length of **dest_addr** structure
- Returns the number of bytes sent, i.e size of datagram

5.10 System call: close

5.10.1 signature

```
int close(int fd)
```

- Closes socket specified by **fd** socket descriptor
- returs zero on success

5.11 UDP Programming example: simple server - echo

```
/*
* echoServer.cxx
*
* UPD echo server
*
*         loops/waits for message received from client
*               send message back to client
*
*         command line arguments:
*                 argv[1] port number to receive from
*
*/
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {

    if (argc != 2) {
        cerr << "USAGE: echoServer port\n";
        exit(EXIT_FAILURE);
    }

    char buffer[256];
    int received = 0;

    int sock;
    struct sockaddr_in server_address;  // structure for address of server
    struct sockaddr_in client_address;  // structure for address of client
    unsigned int addrlen = sizeof(client_address);

    // Create the UDP socket
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

```

```
    // Construct the server sockaddr_in structure
    memset(&server_address, 0, sizeof(server_address));     /* Clear struct */
    server_address.sin_family = AF_INET;                     /* Internet/IP */
    server_address.sin_addr.s_addr = INADDR_ANY;             /* Any IP address */
    server_address.sin_port = htons(atoi(argv[1]));          /* server port */

    // Bind the socket
    if (bind(sock, (struct sockaddr *) &server_address, sizeof(server_address)) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    cout << "echoServer listening on port: " << argv[1] << endl;

    // Run until cancelled
    while (true) {
        // Receive a message from the client
        if ((received = recvfrom(sock, buffer, 256, 0, (struct sockaddr *) &client_address, &addrlen)) < 0) {
            perror("recvfrom");
            exit(EXIT_FAILURE);
        }
        cout << "Client (" << inet_ntoa(client_address.sin_addr) << ") sent " << received << " bytes: " << buffer << endl;
        // Send the message back to client
        if (sendto(sock, buffer, received, 0, (struct sockaddr *) &client_address, addrlen) < 0) {
            perror("sendto");
            exit(EXIT_FAILURE);
        }
    }

    close(sock);
    return 0;
}
```

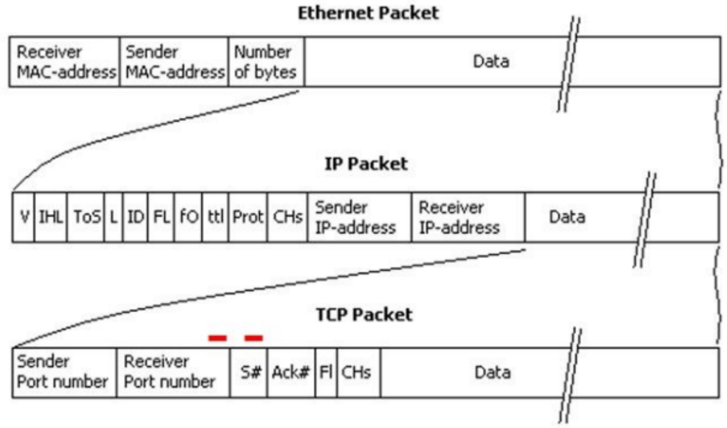
5.12 UDP programming example: Simple client - echo

```cpp
 /*
 * echoClient.cxx
 *
 * UPD echo client
 *
 *         sends message to echo server
 *         waits for message received from server
 *
 *         command line arguments:
 *                 argv[1] IP number of server
 *                 argv[2] port number to send to
 *                 argv[3] message to send
 *
 */
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {

        if (argc != 4) {
                cerr << "USAGE: echoClient server_ip port message\n";
                exit(EXIT_FAILURE);
        }

        char buffer[256];
        int echolen, received = 0;

        int sock;
        struct sockaddr_in server_address;  // structure for address of server
        unsigned int addrlen = sizeof(server_address);

        // Create the UDP socket
        if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                perror("Failed to create socket");
                exit(EXIT_FAILURE);
        }
```

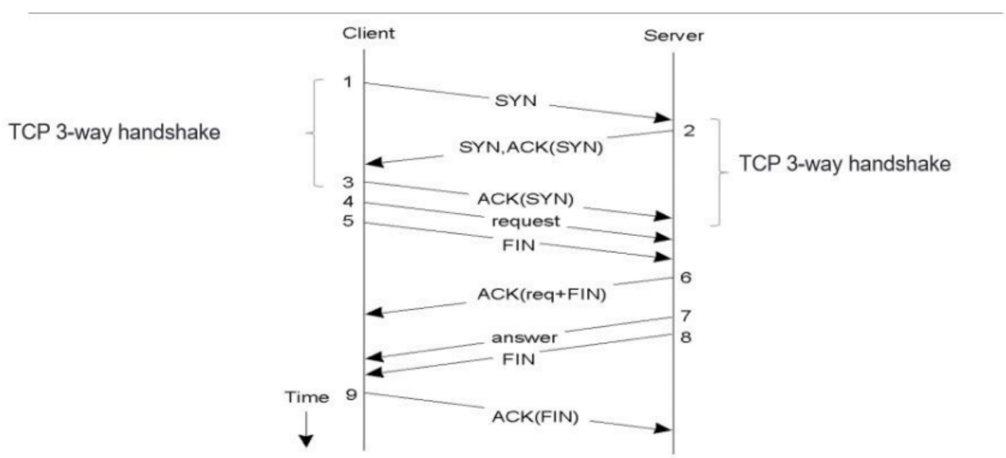
```
  // Construct the server sockaddr_in structure
    memset(&server_address, 0, sizeof(server_address)); /* Clear struct */
    server_address.sin_family = AF_INET; /* Internet/IP */
    server_address.sin_addr.s_addr = inet_addr(argv[1]); /* IP address */
    server_address.sin_port = htons(atoi(argv[2])); /* server port */

    // Send the message to the server (don't forget to count the terminating null)
    echolen = strlen(argv[3]) + 1;
    if (sendto(sock, argv[3], echolen, 0, (struct sockaddr *) &server_address, sizeof(server_address)) != echolen) {
        perror("sendto");
        exit(EXIT_FAILURE);
    }

    // Receive the message back from the server
    if ((received = recvfrom(sock, buffer, 256, 0, (struct sockaddr *) &server_address, &addrlen)) != echolen) {
        perror("recvfrom");
        exit(EXIT_FAILURE);
    }

    cout << "Server (" << inet_ntoa(server_address.sin_addr) << ") echoed: " << received << " bytes: " << buffer << endl;

    close(sock);
    return 0;
}
```

Transmission Control Protocol (TCP)

6.1 TCP / IP Protocol packet



6.2 TCP Communication



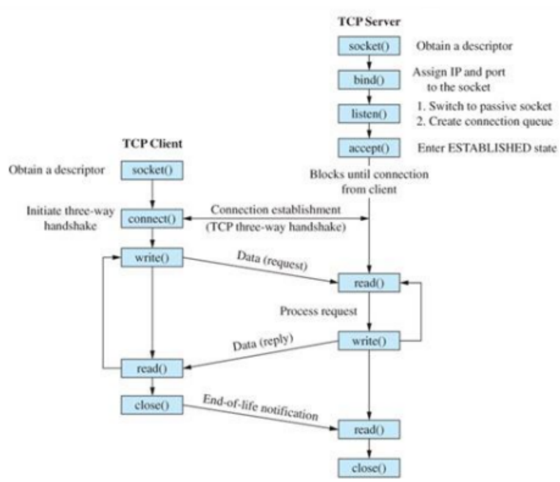
6.3 Socket System Calls

server

client

Primitive	Meaning
socket	Create a new communication endpoint
bind	Attach a local address to a socket
listen	Announce willingness to accept connections
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
write	Send(write) some data over the connection
read	Receive(read) some data over the connection
close	Release the connection

6.4 TCP Communications Pattern



6.5 System call: Socket

6.5.1 Signature

```c
int socket(int domain, int type, int protocol)
```

- Creates a new socket, as end point to a communications link
- **Domain** is set to **AF_INET**

- **Type** is set to **SOCK_STREAM** for stream communications
- **Protocol** is set to 0, i.e 0 default TCP
- Returns socket descriptor
  - used in bind, listen, accept, connect, write, read, close

6.6 Client System call: connect

6.6.1 Signature

```
int connect(int sockfd, const struct sockaddr* addr, socklen_t
↪  addrlen)
```

- connects socket to remote IP number and port
- **struct sockaddr** holds address information
  - will accept **struct sockaddr_in** pointer
- **addrlen** specifies length of **addr** structure
- returns 0 on success, -1 otherwise

6.7 TCP Client illustration

```
/*
 * echoTCPClient.cxx
 *
 * TCP echo client
 *
 *        sends message to echo server
 *        waits for message received from server
 *
 *        command line arguments:
 *                argv[1] IP number of server
 *                argv[2] port number to send to
 *                argv[3] message to send
 *
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netinet/in.h>

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {

        if (argc != 4) {
                cerr << "USAGE: echoTCPClient server_ip port message\n";
                exit(EXIT_FAILURE);
        }

        char buffer[256];
        int echolen, received = 0;

        int sock;
        struct sockaddr_in server_address;  // structure for address of server

        // Create the TCP socket
        if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("socket");
                exit(EXIT_FAILURE);
        }

```

```
    // Construct the server sockaddr_in structure
        memset(&server_address, 0, sizeof(server_address)); /* Clear struct */
        server_address.sin_family = AF_INET; /* Internet/IP */
        server_address.sin_addr.s_addr = inet_addr(argv[1]); /* IP address */
        server_address.sin_port = htons(atoi(argv[2])); /* server port */

        // connect to server
        if (connect(sock, (struct sockaddr *) &server_address, sizeof(server_address)) < 0) {
                perror("connect");
                exit(EXIT_FAILURE);
        }

        // Send the message to the server (don't forget to count the terminating null)
        echolen = strlen(argv[3]) + 1;
        if (write(sock, argv[3], echolen) < 0) {
                perror("write");
                exit(EXIT_FAILURE);
        }

        // Receive the message back from the server
        if ((received = read(sock, buffer, 256)) < 0) {
                perror("read");
                exit(EXIT_FAILURE);
        }

        cout << "Server (" << inet_ntoa(server_address.sin_addr) << ") echoed: " << received << " bytes: " << buffer << endl;

        close(sock);
        return 0;
}
```

6.8 Server system call: bind

6.8.1 Signature

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

- assigns address to socket: IP number and port
- **struct sockaddr** holds address information
  - will accept **struct sockaddr_in** pointer
- **addrlen** specifies length of **addr** structure
- returns 0 on success, -1 otherwise

6.9 Server system call: Listen

6.9.1 Signature

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

- marks socket as passive socket
  - it will be used to accept incoming requests via accept
  - Term: ”server socket”
- **backlog** specifies length of incoming connection queue
- returns 0 on success, -1 otherwise

6.10 Server system call: Accept

6.10.1 Signature

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

- extracts connection request from incoming queue
- creates a new connected socket
  - returns a new file descriptor for that socket, returns -1 on failure
- **struct sockaddr** holds address information
  - will accept struct **sockaddr_in** pointer
- **addrlen** specifies length of **addr** structure

6.11 TCP Server illustration

```
/*
 * echoTCPServer.cxx
 *
 * TCP echo server
 *
 *        loops/waits for message received from client
 *              send message back to client
 *
 *        command line arguments:
 *                argv[1] port number to receive from
 *
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netinet/in.h>

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
        // check arguments
        if (argc != 2) {
                cerr << "USAGE: echoTCPServer port\n";
                exit(EXIT_FAILURE);
        }

        // Create the TCP socket
        int sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("socket");
                exit(EXIT_FAILURE);
        }
        // create address structures
        struct sockaddr_in server_address;  // structure for address of server
        struct sockaddr_in client_address;  // structure for address of client
        unsigned int addrlen = sizeof(client_address);
```

```
    // Construct the server sockaddr_in structure
        memset(&server_address, 0, sizeof(server_address));   /* Clear struct */
        server_address.sin_family = AF_INET;                  /* Internet/IP */
        server_address.sin_addr.s_addr = INADDR_ANY;          /* Any IP address */
        server_address.sin_port = htons(atoi(argv[1]));       /* server port */

        // Bind the socket
        if (bind(sock, (struct sockaddr *) &server_address, sizeof(server_address)) < 0) {
                perror("bind");
                exit(EXIT_FAILURE);
        }

        // listen: make socket passive and set length of queue
        if (listen(sock, 64) < 0) {
                perror("listen");
                exit(EXIT_FAILURE);
        }

        cout << "echoServer listening on port: " << argv[1] << endl;

        // Run until cancelled
        while (true) {
                int connSock=accept(sock, (struct sockaddr *) &client_address, &addrlen);
                if (connSock < 0) {
                        perror("accept");
                        exit(EXIT_FAILURE);
                }
                // read a message from the client
                char buffer[1024];
                int received = read(connSock, buffer, sizeof(buffer));
                if (received < 0) {
                        perror("read");
                        exit(EXIT_FAILURE);
                }
                cout << "Client (" << inet_ntoa(client_address.sin_addr) << ") sent " << received << " bytes: " << buffer << endl;
                // write the message back to client
                if (write(connSock, buffer, received) < 0) {
                        perror("write");
                        exit(EXIT_FAILURE);
                }
                close(connSock);
        }

        close(sock);
        return 0;
}
```

TCP Server & Shell Job Control

7.1 Improve TCP Client

- Become Useful as generic client to any TCP server
- **Improvements:**
  - Accept FQDN as serve address
  - read and process complete server response

7.1.1 Accept FQDN as server address

```
// Lookup FQDN
struct addrinfo* res, hints;
memset(&hints,0,sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
int error = getaddrinfo(argv[1], argv[2], &hints, &res);

if (error) { ... }

// Create the tcp socket as normal

// Connect to server
if (connect(sock, res->ai_addr, res->ai_addrlen) < 0 ) { ... }
```

7.1.2 Process complete server response

```
// Recieve the message back from the server
do {
    recieved = read(sock, buf, sizeof(buf))
    if (recieved < 0) { ... }
    cout.write(buf, received);
} while (recieved > 0);
```

7.1.3 Full example

```
/*
 * TCPClient.cxx
 *
 * TCP client
 *
 *         sends message to TCP server
 *         waits for message received from server
 *
 *         command line arguments:
 *                 argv[1] FQDN of server
 *                 argv[2] port number to send to
 *                 argv[3] request to send
 *
 */
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <cstdlib>
#include <cstring>
#include <cstdio>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
        if (argc != 4) {
                cerr << "USAGE: TCPClient server_name port request\n";
                exit(EXIT_FAILURE);
        }
        // lookup FQDN
        struct addrinfo *res, hints;

        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_INET;
        hints.ai_socktype = SOCK_STREAM;

        int error = getaddrinfo(argv[1], argv[2], &hints, &res);
        if (error) {
                cerr << argv[1] << ": " << gai_strerror(error) << endl;
                exit(EXIT_FAILURE);
        }
        char buffer[1024];
        int sent, received;

```

```
        // Create the TCP socket
        int sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("Failed to create socket");
                exit(EXIT_FAILURE);
        }

        // connect to server
        if (connect(sock, res->ai_addr, res->ai_addrlen) < 0) {
                perror("connect");
                exit(EXIT_FAILURE);
        }

        // Send the message string to the server
        sent = write(sock, argv[3], strlen(argv[3])+1);
        if (sent < 0) {
                perror("write");
                exit(EXIT_FAILURE);
        }

        // Receive the message back from the server
        do {
                received = read(sock, buffer, sizeof(buffer));
                if (received < 0) {
                        perror("read");
                        exit(EXIT_FAILURE);
                }
                cout.write(buffer, received);
        } while (received > 0);
        cout << endl;

        close(sock);
}
```

7.2 Improve TCP server

7.2.1 TCP Server Fork

- server starts loop
  - blocks on accept for connection from client
  - after accept:
    - accept returns dedicated connection socket
    - server forks into parent and child process
- parent process
  - closes dedicated connection socket
  - continues to block for next accept
- child process
  - serves client request
  - communicates with client via dedicated connection socket

7.2.2 TCP Server Fork basic logic

```
while (true) {
    connSock = accept(sock, ...);
    if (fork()) { // Parent
        close(connSock);
    } else { // Child
        // Process client's request via connSock
        ...
    }
}
```

7.2.3 TCP Server Fork example

```
 /*
 * TCPServerFork.cxx
 *
 * TCP echo server
 *
 *         loops/waits/forks for message received from client
 *               send message back to client
 *
 *         command line arguments:
 *                 argv[1] port number to receive from
 *
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netinet/in.h>

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

void processClientRequest( int connSock) {
        int received;
        char buffer[1024];

        // read a message from the client
        if ((received = read(connSock, buffer, sizeof(buffer))) <= 0) {
                perror("read");
                exit(EXIT_FAILURE);
        }

        cout << "Client sent " << received << " bytes: " << buffer << endl;

        // write the message back to client
        if (write(connSock, buffer, received) < 0) {
                perror("write");
                exit(EXIT_FAILURE);
        }
        close(connSock);
        exit(EXIT_SUCCESS);
}
```

```
int main(int argc, char *argv[]) {
        if (argc != 2) {
                cerr << "USAGE: TCPServerFork port\n";
                exit(EXIT_FAILURE);
        }

        // Create the TCP socket
        int sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("socket");
                exit(EXIT_FAILURE);
        }
        // create address structures
        struct sockaddr_in server_address;  // structure for address of server
        struct sockaddr_in client_address;  // structure for address of client
        unsigned int addrlen = sizeof(client_address);

        // Construct the server sockaddr_in structure
        memset(&server_address, 0, sizeof(server_address));   /* Clear struct */
        server_address.sin_family = AF_INET;                  /* Internet/IP */
        server_address.sin_addr.s_addr = INADDR_ANY;          /* Any IP address */
        server_address.sin_port = htons(atoi(argv[1]));       /* server port */

        // Bind the socket
        if (bind(sock, (struct sockaddr *) &server_address, sizeof(server_address)) < 0) {
                perror("bind");
                exit(EXIT_FAILURE);
        }

        // listen: make socket passive and set length of queue
        if (listen(sock, 64) < 0) {
                perror("listen");
                exit(EXIT_FAILURE);
        }

        cout << "TCPServer listening on port: " << argv[1] << endl;
```

```c
	// Run until cancelled
	while (true) {
		int connSock=accept(sock, (struct sockaddr *) &client_address, &addrlen);
		if (connSock < 0) {
			perror("accept");
			exit(EXIT_FAILURE);
		}
		// fork
		if (fork()) {		// parent process
			close(connSock);
		} else {				// child process
			processClientRequest(connSock);
		}
	}
	close(sock);
	return 0;
}
```

7.3 Improved TCP Server and Client example: List directory

- After accept, server forks to service client request
  - Parent process will loop to next accept
- Child process serves client request
  - Read directory path name from client
  - Open directory
  - Read directory entries, send file names to client
  - End process

7.3.1 Program

```
   /*
 * TCPServerReadDir.cxx
 *
 * TCP server
 *
 *          loops/forks to serve request from client
 *                opens directory, sends back lines of file names to client
 *
 *          command line arguments:
 *                  argv[1] port number to receive requests on
 *
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <dirent.h>
#include <unistd.h>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

void processClientRequest(int connSock) {
        int received;
        char path[1024], buffer[1024];

        // read a message from the client
        if ((received = read(connSock, path, sizeof(path))) < 0) {
                perror("receive");
                exit(EXIT_FAILURE);
        }
        cout << "Client request: " << path << endl;

        // open directory
        DIR *dirp = opendir(path);
        if (dirp == 0) {
                // tell client that an error occurred
                strcpy(buffer, path);
                strcat(buffer, ": could not open directory\n");
                if (write(connSock, buffer, strlen(buffer)) < 0) {
                        perror("write");
                         exit(EXIT_FAILURE);
                }
                exit(EXIT_SUCCESS);
        }

        // read directory entries
        struct dirent *dirEntry;
        while ((dirEntry = readdir(dirp)) != NULL) {
                strcpy(buffer, dirEntry->d_name);
                strcat(buffer, "\n");
                if (write(connSock, buffer, strlen(buffer)) < 0) {
                        perror("write");
                        exit(EXIT_FAILURE);
```

```
int main(int argc, char *argv[]) {

        if (argc != 2) {
                cerr << "USAGE: TCPServerReadDir port\n";
                exit(EXIT_FAILURE);
        }

        // Create the TCP socket
        int sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("socket");
                exit(EXIT_FAILURE);
        }
        // create address structures
        struct sockaddr_in server_address;  // structure for address of server
        struct sockaddr_in client_address;  // structure for address of client
        unsigned int addrlen = sizeof(client_address);

        // Construct the server sockaddr_in structure
        memset(&server_address, 0, sizeof(server_address));   /* Clear struct */
        server_address.sin_family = AF_INET;                  /* Internet/IP */
        server_address.sin_addr.s_addr = INADDR_ANY;          /* Any IP address */
        server_address.sin_port = htons(atoi(argv[1]));       /* server port */

        // Bind the socket
        if (bind(sock, (struct sockaddr *) &server_address, sizeof(server_address)) < 0) {
                perror("bind");
                exit(EXIT_FAILURE);
        }

        // listen: make socket passive and set length of queue
        if (listen(sock, 64) < 0) {
                perror("listen");
                exit(EXIT_FAILURE);
        }

        cout << "TCPServerReadDir listening on port: " << argv[1] << endl;

        // Run until cancelled
        while (true) {
                int connSock=accept(sock, (struct sockaddr *) &client_address, &addrlen);
                if (connSock < 0) {
                        perror("accept");
                        exit(EXIT_FAILURE);
                }
                // fork
                if (fork()) {           // parent process
                        close(connSock);
                } else {                        // child process
                        processClientRequest(connSock);
                }
```