System Programming - Linux
Week 14 - Implementing Network Communication

Lecturer: Dr. Aggrey Obbo (PhD)

June 16, 2025

Contents

1 Introduction **1**

2 Network Communication in Linux **1**

3 Why Network Programming **1**

4 Explaining The Socket Abstraction **2**

5 Key Socket Protocols **2**

6 Socket Address Structure **2**

7 Socket Implemementation **3**
7.1 Client Socket Implementation 3
7.2 Server Socket Implementation - Echo Server 5
7.3 Compairing server and client implementations 8

8 Conclusion **8**

1 Introduction

This presentation emphasizes the vital importance of network configurations for facilitating smooth communication on Linux systems. In addition to programming, we will explore the hands-on elements of setting up interfaces, handling routing, and guaranteeing proper connectivity for your applications. Our goals consist of acquiring proficiency in key command-line tools for adjusting network settings, efficiently monitoring network traffic and system connections, and building fundamental skills for diagnosing common network problems that disrupt communication.

Objectives

By the end of this lecture, students will be able to:

- Understand the fundamental concept of sockets as a network programming interface.
- Differentiate between TCP (Stream) and UDP (Datagram) socket types.
- Be able to write a basic TCP client to connect and exchange data with a server.
- Identify core system calls used in basic socket programming.

2 Network Communication in Linux

Network programming involves facilitating communication between programs (processes) over network interfaces. Network programming expands inter-process communication from a single computer, to distributed applications such as web servers. It leverages Sockets APIs, which offer a standardized interface for processes to transmit and receive data using protocols such as TCP and UDP

3 Why Network Programming

At its essence, network programming tackles a fundamental limitation of computing on a single machine: the necessity for processes to connect and communicate beyond a single computer's boundaries. While inter-process communication (IPC) methods like pipes, shared memory, or message queues enable programs to exchange information locally, they do not suffice when processes are distributed across different physical machines, potentially located thousands of miles apart. This is where network communication becomes essential. It makes possible the development of distributed applications—systems made up of several independent components that operate on various network-connected computers to work toward a shared goal. Consider your everyday online interactions: when you access a website, your web browser (acting as a client application) interacts with a web server application hosted on a remote machine. When you send a message through a chat application, your client connects to a chat server, which then forwards the message to another client. Similarly, databases are frequently housed on dedicated servers that are reachable by numerous applications across a network. Without the ability to programmatically enable communication between different machines, most of today's internet-based services would simply not be feasible.

4 Explaining The Socket Abstraction[1]

To connect application logic with the intricacies of network hardware and protocols, operating systems offer an abstraction layer known as sockets. A socket serves as a communication endpoint. It is a software element that an application utilizes to transmit or receive data over a network. Think of it like a telephone connecting you to a vast global telephone network or a mailbox for sending and receiving letters. You engage with the telephone or mailbox (the socket) to start or receive communication, without needing to delve into the complex wiring or routing systems of the broader infrastructure. On Linux and other Unix-like operating systems, the standard suite of functions and data structures for network communication is referred to as the Berkeley Sockets API. This API delivers a coherent and robust way for developers to interact with the network stack.

5 Key Socket Protocols[2][3]

Beneath the socket abstraction exists a variety of network protocols, each tailored for specific functions. The two most fundamental ones are TCP and UDP:

TCP (Transmission Control Protocol): Often described as "connection-oriented," TCP creates a dedicated, reliable, and orderly stream of data between two applications. Before data exchange begins, a "handshake" is performed to establish the connection. TCP ensures that all transmitted data reaches its destination intact, in the proper sequence, and without duplication. If packets fail to arrive, TCP automatically retransmits them. This reliability makes it ideal for applications where data integrity is essential, such as HTTP (web browsing), SSH (secure remote access), and FTP (file transfer).

UDP (User Datagram Protocol): In contrast, UDP operates in a "connectionless" and "datagram-oriented" manner. It transmits data packets (datagrams) without first establishing a connection or assuring delivery, order, or uniqueness. Consider it akin to sending a postcard; you dispatch it without confirming its arrival or sequence. UDP's simplicity yields reduced overhead and swifter transmission, making it appropriate for applications where speed is prioritized over absolute reliability, and some data loss is tolerable. Typical applications include DNS (Domain Name System queries), VoIP (Voice over IP, where the occasional loss of packets is less harmful than delays), and online gaming, where immediate responsiveness is crucial.

6 Socket Address Structure

Let's explore the important topic of Socket Address Structures, which are essential for how network applications pinpoint and connect with specific endpoints (other programs) across a network.

At the core of identifying a network endpoint is the struct sockaddr. This is a generic, abstract structure utilized by the kernel to represent different types of socket addresses (for instance, IPv4, IPv6, Unix domain). However, as developers, we often do not directly interact with sockaddr. Instead, we utilize protocol-specific structures that are "cast" to sockaddr when sent to socket functions. For IPv4 communication, the main structure we engage with is struct sockaddr_in.

The struct sockaddr_in is specifically crafted to accurately denote an IPv4 network address along with a port number. It usually comprises three essential members:

sa_family_t sin_family;: This field defines the address family, indicating the type of address in use. For IPv4, this is always set to AF_INET. This informs the kernel that the following fields in the structure should be understood as an IPv4 address and port. in_port_t sin_port;: This contains the 16-bit TCP or UDP port number. The port number identifies a particular application or service operating on a host. An important point here is that sin_port must be provided in network byte order. struct in_addr sin_addr: This nested structure holds the 32-bit IPv4 address of the host. Similar to sin_port, the IP address within sin_addr must also be in network byte order. The concept of Endianness (Host vs. Network Byte Order) is crucial in this context. Various computer architectures store multi-byte values (such as 16-bit ports or 32-bit IP addresses) in memory differently: either with the most significant byte first (big-endian, which corresponds to network byte order) or the least significant byte first. To guarantee that all machines connected via a network interpret multi-byte data uniformly, a standardized network byte order (big-endian) is mandated for all information exchanged over the network, including IP addresses and port numbers contained within socket address structures. If your host system operates using a different byte order, you will need to convert values before inserting them into sockaddr_in fields and convert them back when reading. This is accomplished with a set of crucial functions:

- htons() (host to network short): Converts a 16-bit value (such as a port number) from host to network byte order.
- ntohs() (network to host short): Converts a 16-bit value from network to host byte order.
- htonl() (host to network long): Converts a 32-bit value (like an IP address) from host to network byte order.
- ntohl() (network to host long): Converts a 32-bit value from network to host byte order.

While sin_addr contains the IP address in a binary format, users and configuration files typically work with human-readable "dotted-decimal" strings (such as "192.168.1.1"). This requires IP Address Conversion functions:

inet_pton() (presentation to network): This function transforms an IP address string (e.g., "192.168.1.1") into its numeric binary representation, suitable for sin_addr, and supports both IPv4 and IPv6. This is the modern, preferred function.

inet_ntop() (network to presentation): This function does the reverse, converting a numeric binary IP address from sin_addr back into a human-readable string. This is also the modern, preferred function for both IPv4 and IPv6.

7 Socket Implemementation[2][3]

Clients proactively initiate connections to specified server addresses, while servers passively await and accept these incoming requests. An understanding of their individual sequences of system calls from socket creation and address binding to establishing connections and exchanging data is essential for developing reliable distributed applications.

7.1 Client Socket Implementation

To illustrate the fundamental client-side TCP connection flow socket operations in Linux, we use the standard C functions.

```
#include <stdio.h> // For printf and perror
#include <stdlib.h> // For exit
#include <string.h>
#include <unistd.h> // To close
#include <arpa/inet.h> // For inet_pton
#include <sys/socket.h> // For socket functions
#define PORT 12345
#define SERVER_IP "127.0.0.1" // Localhost
```

```
#define BUFFER_SIZE 1024
int main()

    int sock_fd;
struct sockaddr_in server_addr;
char buffer[BUFFER_SIZE] = 0;
const char *message = "Hello from client!";
// 1. Create a socket: AF_INET for IPv4, SOCK_STREAM for TCP
sock_fd = socket(AF_INET, SOCK_STREAM, 0);
if (sock_fd ¡ 0)

    perror("Socket creation failed");
exit(EXIT_FAILURE);


    printf("Socket created successfully.\n");
// Prepare the server address structure
memset(&server_addr, 0, sizeof(server_addr)); // Clear structure
server_addr.sin_family = AF_INET; // Specifies the IPv4 address family
server_addr.sin_port = htons(PORT); // Port number is converted to network byte order

    // Converting the server IP from text to binary form and store in sin_addr
if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) ¡= 0)
perror("Invalid address/ Address not supported");
close(sock_fd);
exit(EXIT_FAILURE);


    printf("Server address prepared.\n");
// 2. Connecting to the server
if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) ¡ 0)
perror("Connection failed");
close(sock_fd);
exit(EXIT_FAILURE);


    printf("Connected to server at %s:%d.\n", SERVER_IP, PORT);
// 3. Sending data to the server after connection
if (send(sock_fd, message, strlen(message), 0) ¡ 0)
perror("Send failed");
close(sock_fd);
exit(EXIT_FAILURE);

printf("Message sent: '%s'\n", message);

    // 4. Receive data from the server
ssize_t bytes_received = recv(sock_fd, buffer, BUFFER_SIZE - 1, 0);
if (bytes_received ¡ 0)

    perror("Receive failed");
else if (bytes_received == 0)
printf("Server closed connection.\n");
else
buffer[bytes_received] = '\0'; // Null-terminate received data
printf("Received echo: '%s'\n", buffer);

// 5. Closing the socket
```

```
    close(sock_fd);
printf("Socket closed.\n");
return 0;
```

1. To initiate network communication in Linux, the first step is to create a socket, which serves as an endpoint for communication. This is done using the socket() system call, which generally follows the prototype int socket(int domain, int type, int protocol);.

   The domain parameter specifies the communication domain, such as AF_INET for IPv4 internet protocols or AF_UNIX for local inter-process communication.

   The type parameter defines the kind of socket; for TCP communication, SOCK_STREAM is used (a stream socket that ensures reliable, connection-oriented service), while UDP uses SOCK_DGRAM (a datagram socket).

   The protocol is typically set to 0, indicating to the system to select the default protocol for the specified domain and type (for instance, TCP for AF_INET and SOCK_STREAM). When the socket() function is successfully executed, it returns a non-negative integer referred to as a socket descriptor or file descriptor, which uniquely identifies the newly created socket within the process. This descriptor is utilized in further socket operations.

2. Before a client can establish a connection to a server, it must be aware of the server's network address and port. This information is organized within a socket address structure. For IPv4, the struct sockaddr_in is employed. This structure includes fields for the address family (sin_family, which is set to AF_INET), the port number (sin_port), and the IP address (sin_addr). It is critical that both sine_port and the IP address in sin_addr are in network byte order (big-endian). If the local machine uses a little-endian byte order (which is common in x86 CPUs), it is necessary to utilize functions like htons() (host to network short) for port numbers and htonl() (host to network long) or inet_pton() (presentation to network) for IP addresses to perform the required conversions. For instance, inet_pton(AF_INET, "192.168.1.10", &server_addr.sin_addr); converts an IP address in dotted-decimal format to the binary network byte order format compatible with sin_addr.

3. After the socket has been created and the server's address structure is set up, the client can establish a connection to the server using the connect() system call. The typical prototype is int connect(int sockfd, const struct sockaddr addr, socklen_t addrlen);.

   The sockfd parameter is the socket descriptor that was returned by socket().

   The addr parameter is a pointer to the server's socket address structure (e.g., (struct sockaddr *)&server_addr), detailing the IP address and port number for the connection.

   The addrlen parameter indicates the size of the address structure. For TCP sockets, the connect() function carries out the three-way handshake (SYN, SYN-ACK, ACK) to create a reliable connection with the designated server. If the connection establishes successfully, connect() returns 0; otherwise, it returns -1, and the errno variable is set to reflect the error (for example, ECONNREFUSED if no server is available).

4. Once the connection is made, the client can commence data exchange with the server. In Linux, sockets are treated similarly to files, allowing the use of standard file I/O functions such as write() and read():
   ssize_t write(int fd, const void *buf, size_t count);
   ssize_t read(int fd, void *buf, size_t count);
   Alternatively, more specific socket I/O functions such as send() and recv() can be utilized, providing extra control through flags:
   ssize_t send(int sockfd, const void *buf, size_t len, int flags);
   ssize_t recv(int sockfd, void *buf, size_t len, int flags);
   Both sets of functions facilitate the transfer of bytes over the established TCP connection. They return the number of bytes that were successfully transmitted or received, 0 if the peer has gracefully closed its connection (for read/recv), or -1 in the event of an error.

5. After completing the data exchange, the client should properly close its end of the connection using the close() system call: int close(int fd);. This function frees the socket descriptor, releases kernel resources, and initiates the TCP connection termination handshake. Neglecting to close sockets can result in resource leaks and hinder appropriate port reuse.

7.2 Server Socket Implementation - Echo Server

Implementing a TCP server, such as an echo server, on Linux involves a distinct sequence of socket system calls designed to handle incoming client connections and subsequent data exchange. Unlike a client that actively connects, a server passively waits for connections. A sample code written in C programming language is as follows below:

```
#include <stdio.h> // For printf and perror
#include <stdlib.h> // For exit
#include <string.h> // #include <unistd.h> // To close
#include <arpa/inet.h> // For inet_ntop
#include <sys/socket.h> // For socket functions

    #define PORT 12345
#define BUFFER_SIZE 1024
#define MAX_PENDING_CONNECTIONS 5 // Max clients waiting in queue

    int main()
int listen_fd, client_fd;
struct sockaddr_in server_addr, client_addr;
socklen_t client_addr_len = sizeof(client_addr);
char buffer[BUFFER_SIZE] = 0;
char client_ip[INET_ADDRSTRLEN];

    // 1. Create a socket: AF_INET for IPv4, SOCK_STREAM for TCP
listen_fd = socket(AF_INET, SOCK_STREAM, 0);
if (listen_fd ¡ 0)
perror("Listen socket creation failed");
exit(EXIT_FAILURE);
printf("Listen socket created successfully.\n");

    // Prepare the server address structure
memset(&server_addr, 0, sizeof(server_addr)); // Clear structure
server_addr.sin_family = AF_INET; // Specifies the IPv4 address family
server_addr.sin_addr.s_addr = INADDR_ANY; // Listen on all available interfaces
server_addr.sin_port = htons(PORT); // Port number, converted to network byte order

    // 2. Binding the socket to a specified IP and Port
if (bind(listen_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) ¡ 0)
perror("Bind failed");
close(listen_fd);
exit(EXIT_FAILURE);

printf("Socket bound to port %d.\n", PORT);

    // 3. Setting server to passively listen for incoming connections
if (listen(listen_fd, MAX_PENDING_CONNECTIONS) ¡ 0)
perror("Listen failed");
close(listen_fd);
exit(EXIT_FAILURE);

printf("Server listening on port %d...\n", PORT);

    // Main server loop (iterative - handles one client at a time)
while (1)
// 4. Accept a client connection
client_fd = accept(listen_fd, (struct sockaddr *)&client_addr, &client_addr_len);
if (client_fd ¡ 0)
perror("Accept failed");
```

```
continue; // Try to accept next connection


    // Get client IP address for logging
inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, sizeof(client_ip));
printf("Accepted connection from %s:%d\n", client_ip, ntohs(client_addr.sin_port));

    // 5. Receive and echo data
ssize_t bytes_received;
while ((bytes_received = recv(client_fd, buffer, BUFFER_SIZE - 1, 0)) ¿ 0)
buffer[bytes_received] = '\0'; // Null-terminate received data
printf("Received from client %s: '%s'\n", client_ip, buffer);
send(client_fd, buffer, bytes_received, 0); // Echo back


    if (bytes_received == 0)
printf("Client %s disconnected.\n", client_ip);
else
perror("Receive failed");


    // 6. Closes the client socket
close(client_fd);
printf("Closed connection with client %s.\n", client_ip);


    // For good practise, cleanup close(listen_fd);
return 0;
```

Here's a breakdown of the typical TCP server implementation flow:

1. To Create a Socket (socket()): Like the client, the server begins by utilizing socket() to establish a listener socket. For a TCP server, the usual parameters are socket(AF_INET, SOCK_STREAM, 0). This function generates a stream-oriented socket within the IPv4 domain, prepared to receive TCP connections.

2. Binding an Address to the Socket (bind()): A server requires a distinct network address and port number for client connectivity. The bind() system call allocates this local address to the newly created socket: int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);.

   sockfd: The socket descriptor returned from socket(). addr: A pointer to a struct sockaddr_in (for IPv4) outlining the server's IP address and port. For the IP address, servers frequently utilize INADDR_ANY (or 0.0.0.0) to bind to all accessible network interfaces on the host. The port number must be in network byte order using htons(). addrlen: The size of the address structure. This step effectively informs the operating system: "This socket is designated to listen for connections on this specific IP address and port."

3. Listening for Incoming Connections (listen()): Once bound, the socket remains merely an endpoint; it's not actively awaiting connections yet. The listen() system call transitions the socket into a state ready to accept connections: int listen(int sockfd, int backlog);.

   sockfd: The bound socket descriptor.
   backlog: Indicates the maximum number of pending connections that the kernel should queue for this socket. If a connection request is received when the queue is full, the client's request may be denied. A typical value might range from 5 to 10.
   listen() does not itself accept connections; it simply marks the socket as prepared to do so.

4. Accepting a Client Connection (accept()): This is central to a server's functioning. When a client tries to connect to the listening socket, the kernel completes the three-way TCP handshake. The accept() system call then retrieves the first connection request from the queue of pending connections and establishes a new socket descriptor for that specific connection: int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);.

sockfd: The descriptor for the listening socket. addr: An optional pointer to a struct sockaddr that will be populated with the client's address information. addrlen: A pointer to a socklen_t that specifies the size of the addr buffer. accept() blocks until a client connection is available. Upon successful return, it offers a new socket descriptor (referred to as client_sockfd) for communication with that particular client. The original listening socket (sockfd) stays open and continues to listen for additional connections. Sending/Receiving Data (send()/recv() or write()/read()): Once accept() returns client_sockfd, the server can leverage this new descriptor to communicate directly with the connected client. For an echo server, this entails a loop:

recv(client_sockfd, buffer, buffer_size, 0): Retrieves data sent by the client into a buffer. send(client_sockfd, buffer, bytes_received, 0): Sends the received data back to the client. This loop persists until the client closes its end of the connection (indicated by recv() returning 0) or an error occurs.

5. Closing the Client Socket (close()): Once communication with a specific client concludes (e.g., the client has disconnected or the server logic necessitates termination), client_sockfd should be closed through close(). This releases resources associated with that distinct client connection. The listening socket remains operational to accept new clients.

An iterative echo server executes steps 5 and 6 in a sequential manner, addressing one client at a time. In contrast, for concurrent servers capable of managing multiple clients at once, step 4 (post-accept()) would generally involve forking a new process or generating a new thread to handle client_sockfd, allowing the main process to immediately accept() the next incoming connection.

7.3 Compairing server and client implementations

When building networked applications on Linux, both client and server socket implementations utilize the Berkeley Sockets API, but still their functions and flows remain different. This section compares and contrasts these.

For the client side socket Implementation, the main goal is to establish a connection with a specific server and exchange information. The other steps of the work flow: socket(), connect(), send()/recv() and close(), remain as described earlier.

Where as for the server-side socket Implementation, the server operates in a passive capacity: it waits for incoming connection requests from clients and then manages them. Its lifecycle remains as described above and in the previous lecture. The listening socket remains open to accommodate future clients.

Figure **??** is a brief comparison of the two sides

Feature	**Client-Side**	**Server-Side**
Role	Active	Passive
Binding	Not required	Binds to a specific, well-known IP/Port
Listen/Accept	Not used	Required to handle incoming requests
connect()	Used to establish connection	Not used (receives connections via accept())
Socket Descriptors	One primary socket per connection	One listening socket, multiple client-specific sockets
Complexity	Generally simpler	More complex eg. in concurrency

Figure 1: A comparison of the Servier and Client sides workflows

8 Conclusion

This lecture offers a fundamental insight into the implementation of network communication within the Linux operating system. We started by exploring the crucial function of network interfaces as the main channel through which a system engages with external networks. Following that, the lecture carefully distinguished between TCP (Stream) and UDP (Datagram) socket types, clarifying their unique communication methods and appropriateness for different application needs. A significant focus was placed on outlining the procedural steps and necessary system calls for building a basic TCP client, which facilitates connection setup and data transfer with a server. This thorough coverage provides students with the vital knowledge and initial skills necessary to navigate the complexities of Linux network programming.

List of References

1. “Operating System Concepts” by Silberschatz, Galvin, Gagne
2. Stevens, W. R., Fenner, B., & Rudoff, A. M. (2007). The sockets networking API. (No Title).
3. Donahoo, M. J., & Calvert, K. L. (2009). TCP/IP sockets in C: practical guide for programmers. Morgan Kaufmann.