System Programming - Linux
Week 13 - Sockets

Lecturer: Dr. Aggrey Obbo (PhD)

June 16, 2025

Contents

1 Introduction

Interprocess Communication (IPC) is essential for allowing distinct processes to share data and synchronize their actions within an operating system. This is important for sharing resources, facilitating smooth data transfer, and developing efficient, modular applications. Linux provides several classic IPC techniques, such as pipes, message queues, shared memory (often used in conjunction with semaphores), and signals. Although these methods are effective, they can sometimes lack the flexibility needed for more intricate interactions. This lecture will present sockets as a robust and network-transparent IPC solution, emphasizing their distinct benefits for both local and distributed communication.

Objectives

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

- Discuss the need for Interprocess Communication (IPC) and illustrate how Linux sockets act as a versatile and network-agnostic IPC solution, providing a brief comparison with traditional IPC methods.

- Explain the Unix Domain Sockets (AF_UNIX) for both stream (SOCK_STREAM) and datagram (SOCK_DGRAM) communication showing proper handling of socket file creation and cleanup processes.
- Employ loopback interface sockets (AF_INET/AF_INET6) for local IPC, detailing their configuration and outlining their pros and cons in relation to Unix Domain Sockets.
- Evaluate and determine the suitable socket type (Unix Domain vs. Loopback, Stream vs. Datagram) for particular IPC scenarios based on performance, requirements for network transparency, and considerations for data integrity.
- Recognize and manage common error situations and security issues when creating socket-based IPC applications, including properly handling file permissions for AF_UNIX sockets and adhering to best practices for general socket options.

2 IPC and Sockets[1][2]

In Linux, a socket represents a communication endpoint in an abstract manner. It serves as a two-way communication channel that enables programs to transmit and receive data. Essentially, it's a software handle provided by the operating system to an application, allowing it to engage in interprocess communication (IPC) or network communication. While sockets are widely acknowledged as fundamental to network programming, enabling communication over the internet and local area networks, their functionality also significantly impacts Interprocess Communication (IPC) within a single system. Rather than being limited to network interfaces, sockets deliver a reliable and versatile method for separate processes on the same operating system to exchange information and synchronize tasks.

This intrinsic versatility arises from their architecture. Sockets provide both connection-oriented (stream) and connectionless (datagram) models for IPC, reflecting their counterparts in networking. This flexibility empowers developers to select the most appropriate communication model, whether it entails dependable, ordered byte streams between a client and a server process, or swift, independent message transmission for event notifications.

One major benefit of utilizing sockets for IPC is their network transparency. This essential characteristic indicates that the same application programming interface (API) used for developing distributed network applications can also be employed for communication between processes on the same machine. For developers, this results in notable advantages: code crafted for local socket communication can frequently be expanded or modified for remote interaction with minimal adjustments, typically just by changing the target address. This consistency streamlines development, maintenance, and planning for future scalability, making it a desirable option for intricate, modular software architectures.

In the realm of Linux IPC, we will concentrate on two primary socket families: AF_UNIX (Unix Domain Sockets) and AF_INET (loopback interface). AF_UNIX sockets offer an exceptionally efficient, low-overhead approach for local IPC, using file system paths for addressing and avoiding the standard network stack. In contrast, AF_INET sockets, when utilized with the loopback address (e.g., 127.0.0.1), mimic network communication locally, permitting the use of standard network protocols for inter-process communication without the data ever departing the machine.

3 Unix Domain Sockets (AF_UNIX / AF_LOCAL)[2][3]

A Unix domain socket (UDS) serves as a communication endpoint that facilitates interprocess communication (IPC) within the same operating system kernel. Unlike network sockets, UDS operates without using network protocols (such as TCP or UDP) or IP addresses; they function directly within the file system, appearing as special file types. This is illustrated in Figure 1 and 2 below.

In terms of interprocess communication, Unix domain sockets provide a very efficient approach due to their ability to bypass the network stack entirely. When two processes communicate through a UDS, data is transferred directly between their respective kernel buffers, removing the overhead that comes with network protocols, checksum calculations, and routing. This form of direct communication at the kernel level leads to considerably reduced latency and improved throughput compared to using TCP/IP sockets for processes running on the same machine.

Unix domain sockets are recognized by their corresponding path in the file system, akin to regular files or directories (for instance, /tmp/my_socket). One process creates the socket file (performing the role of the server), while other processes (acting as clients) connect to this file path. The operating system takes care of the communication channels linked to this file. As they are integrated into the file system, UDS can utilize standard file system permissions for controlling access, offering a familiar and effective security mechanism. This capability allows administrators to limit which users or groups can access a specific communication channel.

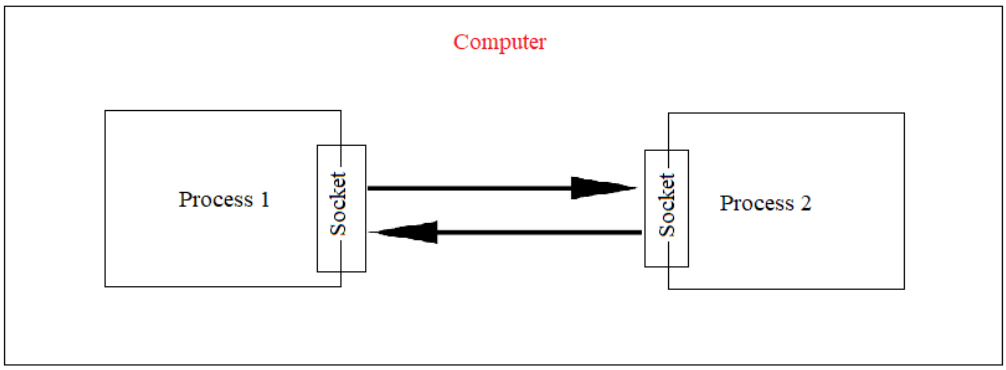

Figure 1: Interprocess communication within an operaring system / Computer

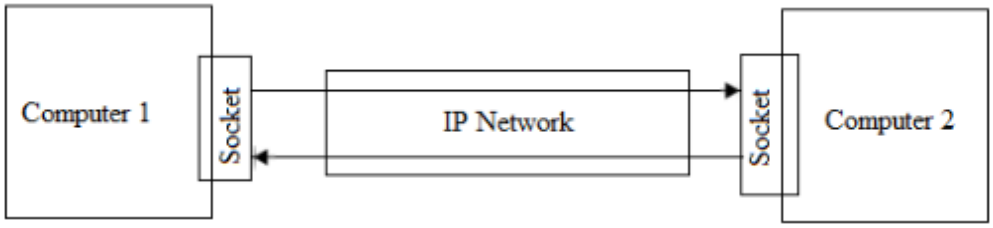

Figure 2: Interprocess communication across networks

Common scenarios that utilize Unix domain sockets include communication between a web server and a FastCGI or WSGI application server, database servers interacting with their clients on the same host, or any situation where two locally situated processes require high-performance and secure IPC without the overhead related to network protocols. They are especially beneficial in microservices architectures where services on the same host must communicate efficiently.

Unlike sockaddr_in, which is meant for internet protocols, the sockaddr_un structure is utilized in Unix-domain sockets to represent an address endpoint. It is specifically designed for communication between processes on the same machine. This structure generally includes two primary fields: sun_family, which is always assigned the value AF_UNIX or AF_LOCAL, and sun_path, a null-terminated string that denotes the filesystem path (for example, "/tmp/mysocket") or an abstract name for the socket. This path serves as the address that other processes use to establish a connection to the socket. It is essential for facilitating local communication without the overhead associated with network processes.

3.1 Server Side Domain Sockets

The workflow on the server side for stream sockets (SOCK_STREAM), consists of several essential steps for establishing and managing client connections or requests.

socket(): The first action is to create a socket by using the socket() system call. In the example given: int sockfd = socket(AF_UNIX, SOCK_STREAM, 0);, AF_UNIX indicates the address family meant for Unix domain sockets, implying that communication will occur locally within the same operating system, utilizing the file system as the address space. SOCK_STREAM represents a stream socket, which ensures a reliable, connection-oriented, and sequenced data flow (similar to TCP over IP networks, but locally). The 0 generally indicates the default protocol linked with the specific socket type and family. The sockfd is an integer descriptor that uniquely identifies this socket created within the server process.

unlink() (Crucial for cleanup): Before implementing any binding, unlink() is executed to unlink("/tmp/my_uds_socket");. The goal is to delete the socket file from the file system if it is already present. It serves as a necessary cleanup procedure especially following a server crash or an abnormal termination. If the socket file remains, the subsequent bind() call will fail, as it tries to create a file that already exists, resulting in an "Address already in use" error. By unlinking, we ensure that the new server instance starts with a clean slate.

bind(): The bind() system call,
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

connects the created socket (sockfd) to a specific file system path for example /tmp/my_uds_socket). This action effectively "names" the socket within the file system. Upon successful execution of bind(), a special socket file (not a regular file) is generated at the indicated path. This file acts as the endpoint for clients attempting to connect to the server. The permissions set on this socket file dictate which users or processes are allowed to access and connect to the server.

listen(): Following binding, the listen() call, int listen(int sockfd, int backlog);, changes the socket into a passive listening socket. This indicates that the socket is now prepared to accept incoming connection requests from clients. The backlog parameter defines the maximum number of connection requests that can be queued by the operating system before any new attempts are denied. A larger backlog permits the server to manage a greater number of simultaneous connection requests.

accept(): The accept() system call, int new_sockfd = accept(sockfd, NULL, NULL);, is a blocking function that pauses execution until a client connection is received. When a client tries to connect to the listening socket (sockfd), accept() retrieves the first pending connection request from the queue and establishes a new socket descriptor (new_sockfd). This new_sockfd becomes a dedicated "connected" socket for interaction with that particular client. The original sockfd remains open and continues to listen for additional incoming connections.

read() / write(): After a new connection is established through accept(), the server and client can communicate by using standard file I/O operations: read() and write(). These functions work on the new_sockfd to transmit and receive data streams, just like reading from or writing to a typical file. read() is employed to obtain data from the client, while write() is used to send data to the client. This ongoing read/write cycle constitutes the fundamental data exchange process between the connected client and server.

3.2 Client Side Domain Sockets

To initiate a connection to a Unix domain stream socket on the client-side, there are two main steps:

socket(): Like on the server side, the client begins by creating a socket using the socket() system call: int sockfd = socket(AF_UNIX, SOCK_STREAM, 0);. AF_UNIX indicates that this will be a Unix domain socket intended for local inter-process communication within the same machine. SOCK_STREAM signifies a stream socket, which provides a reliable, ordered, and connection-oriented data flow, similar to TCP. The returned sockfd acts as a file descriptor representing this new, unconnected client socket.

connect(): This step for the client involves connect(): int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);. This function's role is to actively create a connection from the client's sockfd to the server's listening socket. The addr parameter points to a sockaddr_un structure (for Unix domain sockets), which stores the file system path of the server's socket (e.g., /tmp/my_uds_socket). When connect() is called, the client tries to connect to the server at the specified path. If the server is listening and accepts the connection, a full-duplex communication channel is formed. The connect() call typically blocks until the connection is made successfully or an error arises (e.g., server not listening, file not found). After the connection is established, the client can use read() and write() on sockfd to interchange data with the server.

4 Datagram Sockets (SOCK_DGRAM) with AF_UNIX

Datagram Sockets (SOCK_DGRAM) using AF_UNIX facilitate a connectionless method for local inter-process communication. In contrast to stream sockets, they offer an unreliable, message-oriented service. Information is transmitted as separate packets, and there is no assurance regarding delivery, sequence, or duplication. They are well-suited for fast, lightweight communications where some data loss can be tolerated.

The workflow proceeds as ollows:

4.1 Server-Side Datagram Workflow (SOCK_DGRAM)

socket(AF_UNIX, SOCK_DGRAM, 0): The server initiates the process by creating a datagram socket. AF_UNIX denotes the Unix domain, indicating that the communication occurs locally. SOCK_DGRAM specifies a connectionless datagram socket, capable of sending and receiving separate packets of data without guarantees regarding delivery, sequence, or duplicate protection.

bind() to a path: For a SOCK_DGRAM server, binding is essential. The server binds its socket to a designated file system path (for example, /tmp/my_dgram_socket). This path serves as the server's recognized address, which clients will use to transmit their messages. Unlike stream sockets, this binding does not establish a lasting connection; it simply registers the server's "listening" point for incoming datagrams.

recvfrom(), sendto(): The server employs recvfrom() to handle incoming datagrams. This function not only enables the server to receive data but also reveals the sender's address (assuming the client has bound its socket to a path or if the sender is anonymous). To respond to a client, the server utilizes sendto(), identifying the sender's address (obtained from recvfrom()) as the recipient.

4.2 Client-Side Datagram Workflow

socket(AF_UNIX, SOCK_DGRAM, 0): The client also begins by creating a SOCK_DGRAM Unix domain socket.

sendto(): The client is able to send messages to the server immediately using sendto(), specifying the server's recognized bound path as the target. The client does not need to bind its own socket to send messages.

bind() to a local path (Optional): If the client anticipates receiving responses from the server at a specific local address (for instance, if multiple clients may be operational and the server needs to differentiate them by their reply address), the client may optionally bind() its socket to a local file system path before sending messages. This action ensures the server knows the client's address when it calls recvfrom().

connect()(Optional): Although SOCK_DGRAM is fundamentally connectionless, a client has the option to call connect() to the server's path. This does not create an actual connection but effectively "locks in" the server's address for future send() and recv() operations. This streamlines the syntax (avoiding the need to specify the address with every send()), while communication remains connectionless.

Note that sendto() / recvfrom() enables sending and receiving data to and from a specified destination address respectively.

Closing and Cleanup close(): This function is used to terminate the socket descriptor, freeing resources. unlink(): This step is vital for server-side AF_UNIX sockets (both SOCK _STREAM and SOCK_DGRAM). It eliminates the relevant file system entry (the socket file). When to unlink(): It is advisable to unlink() the socket path before executing the server's bind() call to ensure a clean start, particularly following a server crash. Moreover, the server should unlink() the path upon a graceful shutdown to clear the file from the system.

5 Loopback Interfaces Sockets for Local Interprocess Communication

Local Inter-Process Communication (IPC) is an effective software architecture that enables separate processes on the same machine to share data. Unix Domain Sockets (UDS) have been preferred for local IPC due to their performance, however, loopback interface sockets using AF_INET or AF_INET6) also provide an effective alternative. The loopback interface sockets utilize the TCP/IP stack to restrict communication solely to the local machine. This is made possible through binding sockets to the loopback address: 127.0.0.1 for IPv4 (AF_INET) or ::1 for IPv6 (AF_INET6). A server process listens on a designated port on this loopback address, and client processes connect to that specific address and port.

5.1 Configuring Loopback Interface Sockets

The process is similar to the standard network socket programming (To be covered in the next lesson). The server side and client sides have to be set up as follows:

Server Side

1. Create a socket: socket(AF_INET, SOCK_STREAM, 0) (for TCP) or socket(AF_INET6, SOCK_STREAM, 0).
2. Set SO_REUSEADDR to allow for immediate re-binding following a crash.
3. Bind the socket to 127.0.0.1:port or [::1]:port using bind().
4. Await incoming connections with listen().
5. Accept client connections using accept().
6. Communicate via send() and recv().

Client Side

1. Create a socket: socket(AF_INET, SOCK_STREAM, 0) or socket(AF_INET6, SOCK_STREAM, 0).
2. Connect to the server's loopback address and port using connect().
3. Communicate via send() and recv().

5.2 Advantages of Loopback Interface Sockets

- Adopts the Security Model: Loopback sockets adopt the network security model, enabling detailed access control via firewalls, despite the fact that the communication is local.
- Language Independent: Almost all programming languages offer strong TCP/IP socket libraries, facilitating integration regardless of the programming language used for various processes.
- Portability: Developers with experience in network programming can easily apply their skills. The TCP/IP stack is prevalent across operating systems, rendering loopback socket code highly portable.
- Network Tools Compatibility: Common network diagnostic tools such as netstat and others can be utilized to monitor and debug loopback communication, making troubleshooting easier.

5.3 Disadvantages of Loopback Interface Sockets Compared to Unix Domain Sockets

Despite their strengths, loopback interface sockets do have a number of limitations too. These include:

- Overhead: Loopback sockets are subject to the complete overhead of the TCP/IP stack that includes checksumming and header processing, which may not be appropriate for local communication. When UDS, bypasses this stack, it tends to be faster and more efficient.
- Configuration Complexity: Setting up and administering loopback sockets can be slightly more complicated than UDS which uses file-system paths.
- Security Issues: Although the communication is local, a misconfigured loopback service might be exploited if an attacker gains access to the local machine, whereas UDS benefit from file system permissions for access control.
- Resource Consumption: Loopback sockets utilize ports, which are limited resources, although this typically does not pose a significant problem for local IPC.
- Lack of Ancillary Data: UDS can transmit file descriptors and other ancillary data, a capability not supported by standard loopback sockets.

Although UDS generally provides better performance for exclusively local IPC, loopback interface sockets represent a solid, familiar, and highly portable option, especially when utilizing existing network programming skills or needing compatibility with network diagnostic tools.

6 Choice of Socket For Implemention

When developing network applications, it is vital to choose the right socket type to guarantee the best performance, reliability, and functionality. This choice depends on several important factors. Below are some of the factors normally considered in making the appropriate solution. Grasping these aspects is crucial for creating strong and efficient network solutions.

- *Performance*
  UDS typically deliver better performance for local IPC. By circumventing the complete TCP/IP stack and functioning solely within the kernel, they result in reduced overhead, fewer context switches, and increased throughput. In contrast, loopback sockets, while still local, are subject to the overhead associated with network protocols (like checksums and headers). Various benchmarks often reveal UDS outperforming others by considerable margins (e.g., 2-3 times faster).
- *Network Transparency*
  Loopback sockets provide network transparency. This allows developers to utilize the same code for both local and remote communications, easing development efforts, especially if there's potential for future distribution of the application. On the other hand, UDS are limited to local communication; they cannot interact with processes on different machines. If there's a notable chance of deploying in a distributed manner, loopback sockets might be the better alternative.

- *Data Integrity*
  Stream Sockets (TCP): Guarantee reliable, ordered, and error-checked data delivery. Data is managed as a continuous stream, and by default, message boundaries are not preserved (framing must be implemented separately). This is suitable for applications where it's crucial to avoid data loss or corruption (such as file transfers or database interactions).

  Datagram Sockets (UDP): Provide connectionless, unreliable communication. Messages are dispatched as separate packets, with no assurances regarding delivery, ordering, or duplication prevention. While they are faster due to reduced overhead, applications must handle reliability if required.

- *Performance*
  Datagram sockets usually present lower latency and greater throughput for single messages, thanks to their connectionless design and minimal overhead. Stream sockets, however, require connection setup and management, contributing additional overhead.

Despite the short falls and strengths, both Stream (TCP) and Datagram (UDP) sockets are inherently built for network communications and ensure network transparency. But to achieve optimal local IPC performance and ease of use, one should opt for Unix Domain Stream Sockets if the communication is limited to the same machine and requires orderly data. And if If network transparency is a significant factor, facilitating an effortless transition to distributed architectures, Loopback Stream Sockets are an advisable option. In addition, for applications where speed is prioritized over assured delivery, and wherein occasional message loss or disarrangement can be tolerated Datagram sockets may work well.

7 Example - Implementing Client Server Sockets[1][2][3]

The following example has both the server and client C programs utilizing the standard POSIX socket functions. Figure 3 illustrates steps followed to establish client server connections.(*Find the server and client programs from reference materials*).
Also take note of the differences in the implemention of TCP and UDP sockets.

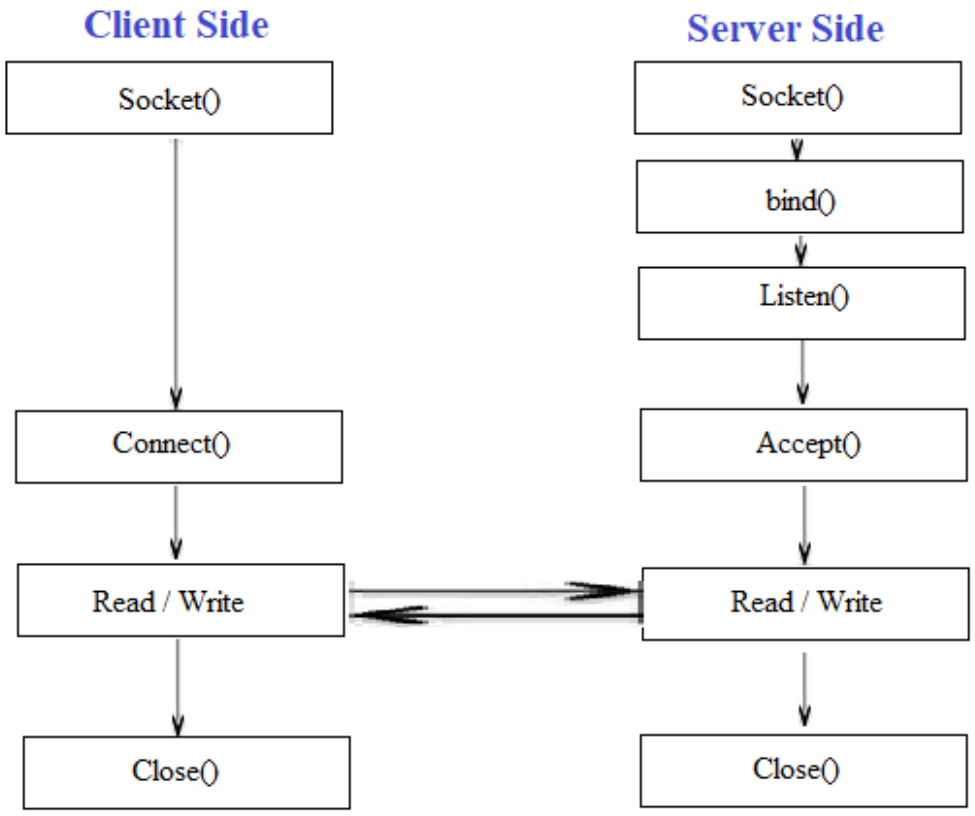

Figure 3: Establishing Client Server connection

The Server Program (server.c)

1. Includes the necessary headers like stdio.h (for I/O), stdlib.h (for exit), string.h (for memset), unistd.h (for close), arpa/inet.h (for inet_pton and network byte order conversion), and sys/socket.h (for socket functions).
2. Socket Creation: socket(AF_INET, SOCK_STREAM, 0) creates a TCP socket for IPv4. setsockopt: Configures the socket to reuse the address immediately after termination, preventing "Address already in use" errors during quick restarts.
3. Binding: bind() assigns the socket to INADDR_ANY (which resolves to 127.0.0.1 for loopback in this context) and the specified PORT. htons() converts the port number to network byte order.
4. Listening: listen() puts the server socket into a listening state, ready to accept incoming connections. The second argument defines the maximum backlog of pending connections.
5. Accepting: accept() blocks until a client connects. It then creates a new socket (new_socket) for this specific client connection and returns it.
6. Data Exchange: A while loop continuously calls recv() to get data from the connected client. If data is received, it's printed and then echoed back to the client using send(). The loop breaks if the client disconnects (valread == 0) or an error occurs.
7. Cleanup: close() is called for both the client connection socket and the main server socket to release resources.

The Client Program (client.c):

1. Includes: Similar headers as the server.
2. Socket Creation: socket(AF_INET, SOCK_STREAM, 0) creates an IPv4 TCP socket.
3. Address Conversion: inet_pton(AF_INET, SERVER_IP, &serv_addr.sin_addr) converts the human-readable 127.0.0.1 IP address into a binary format suitable for the socket.

4. Connecting: connect() attempts to establish a connection to the server at the specified IP address and port.
5. Sending/Receiving: Once connected, the client send()s its message and then recv()s the server's echoed response.
6. Cleanup: close() is called to close the client socket.

Save the server code as server.c and the client code as client.c. and compile using a C compiler such as gcc
Run the server first in one terminal as ./server and then then run the client in another terminal: using ./client. The result will display the messages exchanged between the two processes.

8 Conclusion

Our investigation into Linux sockets has revealed their crucial significance in Interprocess Communication (IPC). We have observed how they surpass the constraints of conventional IPC methods, providing a flexible, network-independent solution that is appropriate for a range of communication requirements. Through the development of client-server applications with Unix Domain Sockets (AF_UNIX) for both stream and datagram communication, we have acquired hands-on experience in handling socket file lifecycles and ensuring thorough cleanup.

Moreover, our in-depth examination of loopback interface sockets (AF_INET/AF_INET6) provided an alternative for local IPC, enabling us to thoughtfully assess the advantages and disadvantages of Unix Domain versus loopback sockets in terms of performance, network transparency, and data integrity. This positions us to wisely choose the right socket type for any IPC situation. Lastly, by tackling common error scenarios and addressing security considerations, such as file permissions for AF_UNIX and overall best practices for sockets, we are now more equipped to create secure, dependable, and efficient socket-based IPC applications.

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.