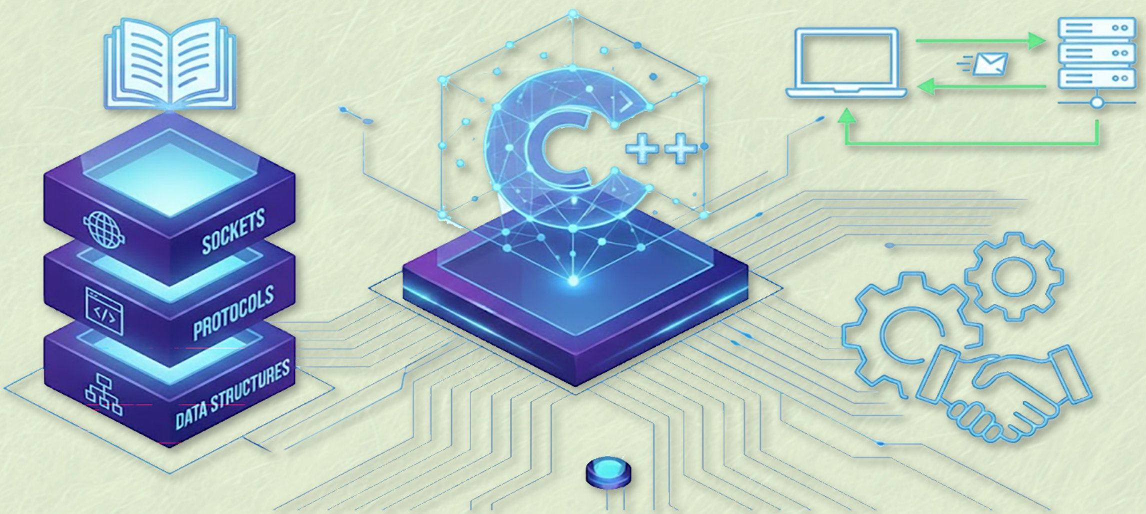
https://simplifycpp.org

Network Programming in C++

Fundamentals and a Simple TCP Example



Prepared by Ayman alheraki

Network Programming in C++

Fundamentals and a Simple TCP Example

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Author's Introduction

This mini-booklet is written for C++ developers who seek a deep and accurate understanding of network programming from a *systems-oriented* perspective, rather than through high-level frameworks, convenience libraries, or language-specific abstractions that obscure the underlying mechanics.

Network programming is often introduced as a sequence of API calls: create a socket, connect, send, receive, and close. While this procedural view may be sufficient for small demonstrations, it fails to explain what truly happens when data moves from one machine to another. In reality, network programming is a direct and continuous interaction with the operating system's networking stack, its resource management policies, and its I/O model.

Every networked application is, at its core, a systems program. It consumes kernel resources, participates in scheduling decisions, relies on buffering strategies, and must operate correctly in the presence of partial failures, latency, and unpredictable external conditions. Understanding these aspects is essential for writing software that is not only functional, but reliable, debuggable, and maintainable over time.

C++ occupies a unique position in this domain. Unlike managed languages, it does not hide the cost of abstraction or the consequences of design decisions. Unlike scripting languages, it offers deterministic performance, explicit lifetime control, and the ability to express low-level concepts without sacrificing structure. For these reasons, C++ continues to be a dominant language for building networked systems such as servers, databases, communication middleware, financial trading platforms, and infrastructure software.

This booklet deliberately avoids teaching networking as a set of recipes. Instead, it emphasizes understanding:

- how the operating system exposes networking facilities,
- how TCP communication is modeled and enforced,
- how C++ code maps to system calls and kernel behavior,
- and how design choices at a low level influence correctness, performance, and scalability.

The material presented here is intentionally conservative and foundational. It does not depend on trends, frameworks, or transient technologies. The principles discussed remain valid across different operating systems, hardware architectures, and future evolutions of the C++ language.

This work is meant to serve as a stable reference and a conceptual anchor. Readers who master these fundamentals will be well prepared to move on to advanced topics such as asynchronous I/O, event-driven architectures, high-performance servers, and distributed systems—without losing sight of the underlying mechanics that ultimately govern all networked software.

`Ayman Alheraki`

Preface

Network programming is often introduced to C++ developers through minimal examples: a few lines of code that open a socket, send a message, and exit. While such examples are useful as a first encounter, they rarely explain what actually happens once a program begins communicating with the outside world.

As a result, many developers learn networking by imitation rather than understanding. Code is copied, adapted, and reused without a clear mental model of how data flows through the operating system, how resources are allocated and released, or how failures propagate across network boundaries. This approach frequently leads to fragile programs that work under ideal conditions but fail unpredictably when exposed to real networks, real workloads, and real users.

Real-world networked software must operate in an environment defined by latency, partial reads, dropped connections, resource limits, and concurrent activity. Ignoring these realities does not simplify a program; it merely delays the point at which complexity and failure become unavoidable.

This booklet is intentionally focused on fundamentals. Its purpose is to establish a clear and accurate foundation by addressing the following core areas:

- a conceptual understanding of the TCP/IP model and its layered design,

- how sockets are implemented and managed by the operating system,

- how C++ programs interact with system calls and kernel services,
- and how to write minimal yet correct TCP client and server programs.

Rather than presenting networking as a collection of patterns or framework-specific techniques, this work emphasizes the relationship between application code and the underlying system. Each concept is introduced with the goal of making behavior predictable and reasoning about correctness possible.

This booklet does not attempt to replace comprehensive references or encyclopedic works on networking. Instead, it is designed to serve as a stable entry point—a foundation upon which deeper study and more advanced techniques can be built with confidence. Readers who understand these fundamentals will be better equipped to evaluate libraries, adopt asynchronous models, and design scalable networked systems without relying on guesswork or trial-and-error.

Chapter 1

Introduction to Network Programming

Network programming enables independent processes to communicate with each other across a network using standardized communication protocols. These processes may run on the same machine, on different machines within a local network, or across geographically distributed systems connected through the internet.
At its core, network programming is about *communication between processes that do not share memory*. Unlike function calls or in-process messaging, network communication must account for latency, partial delivery, failures, and the absence of shared state.

1.1 Network Programming as Distributed IPC

From a conceptual standpoint, network programming is an extension of inter-process communication (IPC) beyond the boundaries of a single machine.
On a single system, IPC mechanisms such as pipes, shared memory, message queues, and Unix domain sockets rely on the operating system to manage communication between processes. Network programming uses a similar model, but the communication channel extends across machines connected by a network.

The fundamental difference is that once communication leaves a single machine:

- memory is no longer shared,
- timing becomes unpredictable,
- failures are common rather than exceptional,
- and communication costs are orders of magnitude higher.

These characteristics shape every design decision in networked software.

1.2 The Role of the Operating System

Applications do not implement networking protocols themselves. Instead, the operating system provides a *networking stack* that implements standardized protocols such as:

- Internet Protocol (IP)
- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

This networking stack resides largely within the kernel. It handles packet routing, congestion control, retransmission, segmentation, reassembly, and error detection.
Applications interact with this stack through system calls. In C and C++, this interface is exposed through the *socket API*.

1.3 Sockets as Communication Endpoints

A socket represents one endpoint of a communication channel. From the application's point of view, a socket behaves similarly to a file descriptor: it can be created, configured, read from, written to, and closed.

However, unlike regular files, sockets represent a connection to another process, potentially running on a remote machine.
Conceptually, a socket binds together:

- a protocol family (such as IPv4 or IPv6),
- a transport protocol (such as TCP or UDP),
- a local address and port,
- and optionally a remote address and port.

1.4 A First Look at Socket Creation

The first step in any network program is creating a socket. This is done using the `socket` system call.

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

int sock = socket(AF_INET, SOCK_STREAM, 0);
```

This single line already encodes several important decisions:

- `AF_INET` selects the IPv4 protocol family,
- `SOCK_STREAM` selects a stream-oriented socket (TCP),
- the final parameter selects the default protocol for this combination.

At this point, no network communication has occurred. The application has merely requested that the operating system allocate a socket resource.

1.5 Client and Server Roles

Network programs typically fall into two roles:

- servers, which wait for incoming connections,
- clients, which initiate connections.

Although both use sockets, their control flow differs significantly.
A server usually:

1. creates a socket,
2. binds it to a local address and port,
3. listens for incoming connections,
4. accepts clients,
5. exchanges data,
6. and closes the connection.

A client usually:

1. creates a socket,
2. connects to a server,
3. sends and receives data,
4. and closes the socket.

These roles reflect the asymmetry inherent in TCP communication.

1.6 Illustrative Minimal Server Example

The following example demonstrates the structural flow of a minimal TCP server. At this stage, the example is intentionally simple and omits robustness features that will be introduced later.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int main() {
    int server = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in address{};
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);

    bind(server, (sockaddr*)&address, sizeof(address));
    listen(server, 5);

    int client = accept(server, nullptr, nullptr);

    const char* msg = "Hello\n";
    write(client, msg, 6);

    close(client);
    close(server);
}
```

While short, this program already relies on a substantial amount of kernel functionality: address resolution, connection establishment, buffering, and data delivery.

1.7 Illustrative Minimal Client Example

The corresponding client program initiates a connection to the server:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in server{};
    server.sin_family = AF_INET;
    server.sin_port = htons(8080);
    inet_pton(AF_INET, "127.0.0.1", &server.sin_addr);

    connect(sock, (sockaddr*)&server, sizeof(server));

    char buffer[64];
    read(sock, buffer, sizeof(buffer));

    close(sock);
}
```

Even in this minimal example, the client and server do not share memory, assumptions, or execution timing. They coordinate entirely through the operating system and the network stack.

1.8 Why Fundamentals Matter

These early examples may appear trivial, but they demonstrate the core reality of network programming: every operation can fail, block, or behave differently depending on external conditions.

Without a solid understanding of how the operating system, protocols, and sockets interact, it becomes impossible to reason about correctness and performance in larger systems.

This chapter establishes the conceptual foundation upon which all subsequent topics in this booklet are built. Understanding these fundamentals is essential before moving on to error handling, scalability, concurrency, and high-performance network designs.

Chapter 2

Networking Fundamentals

This chapter establishes the core concepts that underpin all network programming. While APIs and libraries may change, these fundamentals remain stable and must be clearly understood before writing correct and scalable networked software.

2.1 IP Addresses

An IP address uniquely identifies a host on a network. It serves the same conceptual role as a physical address in the postal system: it specifies where data should be delivered.
There are two major versions of the Internet Protocol in common use today.

2.1.1 IPv4

IPv4 addresses are 32-bit values, typically represented in dotted decimal notation:

```
192.168.1.10
```

Each component represents one byte of the address. Because IPv4 provides approximately 4.3 billion unique addresses, it has largely exhausted its available address space.

In practice, IPv4 networks rely heavily on techniques such as Network Address Translation (NAT), which further complicate network behavior from the application's point of view.

2.1.2 IPv6

IPv6 addresses are 128-bit values, represented in hexadecimal notation:

`2001:0db8:85a3::8a2e:0370:7334`

IPv6 dramatically increases the available address space and removes many of the architectural limitations of IPv4. From an application perspective, IPv6 introduces larger address structures and different formatting, but the fundamental networking model remains unchanged.

2.1.3 Address Representation in Code

In C++, IP addresses are not manipulated directly as integers. Instead, they are stored in protocol-specific structures managed by the operating system.
For IPv4, the `sockaddr_in` structure is commonly used:

```
sockaddr_in addr{};
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = INADDR_ANY;
```

The use of `htons` reflects an important concept: network byte order. All multi-byte values transmitted over the network are represented in big-endian format, regardless of the host architecture.

2.2 Ports

While an IP address identifies a host, a port identifies a specific service or application running on that host.

A single machine may run many networked applications simultaneously: web servers, databases, monitoring agents, and custom services. Ports allow the operating system to route incoming data to the correct process.

2.2.1 Port Ranges

Ports are 16-bit values, allowing a range from 0 to 65535. They are conventionally divided into ranges:

- Well-known ports (0–1023)
- Registered ports (1024–49151)
- Dynamic or ephemeral ports (49152–65535)

Servers typically bind to a fixed, well-known port. Clients are usually assigned ephemeral ports automatically by the operating system.

2.2.2 Binding to a Port

When a server starts, it must explicitly bind its socket to a port:

```
sockaddr_in addr{};
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(8080);

bind(server_fd, (sockaddr*)&addr, sizeof(addr));
```

Binding associates the socket with a specific local address and port. If the port is already in use, the operation fails, and the server must handle this condition explicitly.

2.3 The TCP Connection Tuple

A TCP connection is uniquely identified by a four-tuple:

- Source IP address
- Source port
- Destination IP address
- Destination port

This means that multiple clients can connect to the same server port simultaneously, as long as their source addresses or source ports differ.
From the server's perspective, each accepted connection represents a distinct communication channel, even though all connections arrive on the same listening port.

2.4 TCP vs UDP

TCP and UDP represent two fundamentally different approaches to network communication.

2.4.1 Transmission Control Protocol (TCP)

TCP is a connection-oriented protocol. Before any data is exchanged, a connection is established through a handshake process.
TCP provides:

- reliable delivery,
- in-order data transmission,
- retransmission of lost packets,

- flow control,
- and congestion control.

From the application's perspective, TCP presents a continuous byte stream. Message boundaries are not preserved, and the application must define its own framing protocol.

2.4.2 User Datagram Protocol (UDP)

UDP is connectionless. Each message is sent independently as a datagram.
UDP provides:

- minimal overhead,
- no delivery guarantees,
- no ordering guarantees,
- no congestion control.

While UDP is useful for latency-sensitive applications, it shifts significant complexity to the application layer.

2.5 Why This Booklet Focuses on TCP

This booklet focuses exclusively on TCP because it represents the foundation of most reliable networked systems.
Web servers, databases, file transfer systems, and distributed services are overwhelmingly built on TCP. Understanding TCP thoroughly is a prerequisite for mastering higher- level protocols and abstractions.
Once the TCP model is clearly understood, learning UDP or more advanced transport mechanisms becomes significantly easier.

This chapter establishes the addressing and protocol concepts that will be used throughout the remainder of this booklet. All subsequent examples and designs build directly upon these fundamentals.

Chapter 3

The Socket API in C++

The socket API is the fundamental interface through which applications interact with the operating system's networking stack. Despite its age, this API remains the foundation of virtually all network programming on modern operating systems.

Sockets are often described as *file-descriptor-like* objects. This description is accurate but incomplete. Like file descriptors, sockets are represented by small integers, managed by the kernel, and accessed through system calls. Unlike files, sockets represent communication channels whose behavior depends on protocols, network state, and remote peers.

Understanding the socket API is essential for writing correct and efficient networked software in C++.

3.1 Sockets as Kernel Resources

When a socket is created, the operating system allocates kernel-side data structures to manage:

- protocol state,
- send and receive buffers,

- connection metadata,
- and error conditions.

From the application's point of view, a socket is identified by an integer handle. From the kernel's point of view, that handle refers to a complex object whose behavior evolves over time.

Because sockets are kernel resources, they must be managed carefully. Failure to close sockets leads to resource leaks that cannot be reclaimed automatically.

3.2 Creating a Socket

All network communication begins with the creation of a socket using the `socket` system call.

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

int fd = socket(AF_INET, SOCK_STREAM, 0);
```

This call specifies:

- the protocol family (`AF_INET` for IPv4),
- the socket type (`SOCK_STREAM` for TCP),
- and the protocol (usually zero to select the default).

At this stage, no network traffic has occurred. The socket exists only as a local kernel object.

3.3 Binding a Socket

Binding associates a socket with a local address and port. This step is typically required for servers.

```
sockaddr_in addr{};
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(8080);

bind(fd, (sockaddr*)&addr, sizeof(addr));
```

Binding does not make the socket active. It simply tells the operating system where incoming connections or datagrams should be delivered.

3.4 Listening for Connections

For TCP servers, the next step is to mark the socket as passive using `listen`.

```
listen(fd, 5);
```

The argument specifies the backlog: the maximum number of pending connections the kernel may queue before refusing new ones.
At this point, the socket represents a listening endpoint, not an active connection.

3.5 Accepting Connections

When a client attempts to connect, the server accepts the connection using `accept`.

```
int client_fd = accept(fd, nullptr, nullptr);
```

This call returns a new socket descriptor. The original socket remains in the listening state, while the new socket represents a specific client connection.
This distinction is critical:

- the listening socket accepts new clients,
- the connected socket exchanges data with one client.

3.6 Connecting to a Server

Clients initiate communication using `connect`.

```
sockaddr_in server{};
server.sin_family = AF_INET;
server.sin_port = htons(8080);
inet_pton(AF_INET, "127.0.0.1", &server.sin_addr);

connect(fd, (sockaddr*)&server, sizeof(server));
```

The `connect` call triggers the TCP handshake. Only after it completes successfully can data be sent or received.

3.7 Sending and Receiving Data

Data exchange occurs through `send` and `recv`, or through generic I/O calls such as `read` and `write`.

```
send(client_fd, buffer, length, 0);
recv(client_fd, buffer, sizeof(buffer), 0);
```

A critical concept is that TCP provides a byte stream. A single call to `send` does not correspond to a single call to `recv`. Applications must handle partial sends and partial receives explicitly.

3.8 Closing a Socket

Sockets must be closed explicitly when no longer needed.

```
close(fd);
```

Closing a socket releases kernel resources and signals the remote peer that the connection has ended.
Failure to close sockets leads to resource exhaustion and unpredictable behavior.

3.9 Error Handling in the Socket API

Nearly every socket-related system call can fail. Errors may arise from invalid parameters, resource limits, network conditions, or remote behavior.

```
if (fd < 0) {
    perror("socket");
}
```

Robust network programs treat error handling as a first-class concern, not an afterthought.

3.10 The Role of C++

C++ does not replace the socket API. Instead, it provides tools that allow developers to use it safely and expressively.
C++ enables:

- RAII-based management of socket lifetimes,
- strong type abstractions around addresses and protocols,

- structured error handling strategies,
- and clear separation of responsibilities.

Later chapters will demonstrate how modern C++ techniques can be used to wrap low-level socket operations without hiding their behavior or cost.

This chapter establishes the raw interface upon which all higher-level designs are built. A clear understanding of the socket API is essential before attempting to introduce abstractions or frameworks.

Chapter 4

TCP Communication Model

The Transmission Control Protocol (TCP) defines a reliable, connection-oriented communication model built on top of the Internet Protocol. Understanding this model is essential for reasoning about correctness, performance, and failure behavior in networked applications.

Unlike message-oriented systems, TCP provides a continuous, ordered stream of bytes between two endpoints. The protocol guarantees that bytes are delivered in order, without duplication, or the connection fails explicitly.

4.1 Connection-Oriented Communication

TCP communication is always associated with a connection. Before any data can be exchanged, both endpoints must participate in a connection establishment phase.

This contrasts with connectionless protocols, where data may be sent without prior coordination. In TCP, the connection represents shared state maintained by both the operating system and the protocol implementation.

Once established, the connection defines:

- the communicating endpoints,
- sequence numbers for transmitted data,
- flow-control windows,
- and congestion-control state.

4.2 The TCP Connection Lifecycle

Every TCP connection follows a well-defined lifecycle that is enforced by the operating system.
From the application's perspective, this lifecycle is expressed through a series of socket operations.

4.2.1 Server-Side Lifecycle

A typical TCP server progresses through the following stages:

1. **Socket Creation**

   The server creates a socket to represent a potential communication endpoint.

   ```
   int server_fd = socket(AF_INET, SOCK_STREAM, 0);
   ```

   At this point, the socket exists only locally and is not yet associated with any address.

2. **Binding to an Address and Port**

   The server binds the socket to a specific local address and port.

   ```
   bind(server_fd, (sockaddr*)&addr, sizeof(addr));
   ```

   Binding determines where incoming connection requests will be delivered.

3. **Listening for Connections**

   The server marks the socket as a passive listening socket.

   ```
   listen(server_fd, 5);
   ```

   The operating system now begins queueing incoming connection attempts.

4. **Accepting a Client**

   When a client connects, the server accepts the connection.

   ```
   int client_fd = accept(server_fd, nullptr, nullptr);
   ```

   This operation creates a new socket that represents a single, active connection to a client.

5. **Exchanging Data**

   Once the connection is established, both sides may send and receive data.

   ```
   send(client_fd, buffer, length, 0);
   recv(client_fd, buffer, sizeof(buffer), 0);
   ```

   Data is transmitted as a stream of bytes with no inherent message boundaries.

6. **Closing the Connection**

   When communication is complete, the server closes the socket.

   ```
   close(client_fd);
   ```

   Closing the socket initiates connection teardown and resource cleanup.

4.2.2 Client-Side Lifecycle

A TCP client follows a simpler but equally important sequence of steps:

1. **Socket Creation**

   ```
   int sock = socket(AF_INET, SOCK_STREAM, 0);
   ```

2. **Connecting to the Server**

   ```
   connect(sock, (sockaddr*)&server, sizeof(server));
   ```

   The `connect` call triggers the TCP handshake and blocks until the connection is established or fails.

3. **Sending and Receiving Data**

   ```
   send(sock, buffer, length, 0);
   recv(sock, buffer, sizeof(buffer), 0);
   ```

   Both operations may block or return partial results.

4. **Closing the Connection**

   ```
   close(sock);
   ```

   Closing signals the end of communication and releases kernel resources.

4.3 The TCP Handshake

Connection establishment in TCP involves a handshake process that synchronizes both endpoints.
Although the handshake is handled entirely by the operating system, it has important implications for application behavior:

- connection attempts may fail or time out,
- servers may be overwhelmed by connection requests,
- and latency is introduced before data exchange begins.

Understanding this phase is critical for designing responsive and resilient systems.

4.4 Byte Streams and Message Framing

TCP does not preserve message boundaries. Applications that send structured data must define their own framing mechanisms.
For example, an application may prefix each message with its length or use delimiters. Failure to implement proper framing leads to subtle and difficult bugs.

4.5 Blocking Behavior

By default, TCP sockets operate in blocking mode. Calls to `connect`, `accept`, `send`, and `recv` may block until the operation completes.
Blocking simplifies program structure but limits scalability. Later chapters will discuss alternative models built on the same TCP foundation.

4.6 Why the TCP Model Matters

The TCP communication model defines the contract between applications and the network. Ignoring its properties leads to incorrect assumptions about timing, ordering, and reliability.
A clear understanding of TCP is essential before introducing concurrency, asynchronous I/O, or higher-level abstractions.

This chapter provides the conceptual framework that will guide the practical examples and design decisions throughout the remainder of this booklet.

Chapter 5

A Simple TCP Client and Server

This chapter presents a complete, minimal TCP server and client written in C++. The goal is not to build a production-ready system, but to demonstrate a correct end-to-end TCP interaction while exposing every important step in the communication process.
Each example is intentionally simple, yet fully functional. Every line corresponds directly to concepts introduced in the previous chapters.

5.1 Design Goals of the Example

The examples in this chapter are designed with the following goals:

- demonstrate the full TCP lifecycle,
- keep control flow explicit and readable,
- avoid hidden abstractions,
- and expose error handling at each step.

The server accepts a single client, sends a short message, and exits. The client connects, receives the message, and terminates.

5.2 TCP Server Example

The server program performs the following tasks:

1. create a listening socket,
2. bind it to a local port,
3. listen for incoming connections,
4. accept a single client,
5. send data to the client,
6. and close all sockets.

5.2.1 Complete Server Implementation

```
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <iostream>

int main() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("socket");
        return 1;
    }
```

```
    sockaddr_in addr{};
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(8080);

    if (bind(server_fd, (sockaddr*)&addr, sizeof(addr)) < 0) {
        perror("bind");
        close(server_fd);
        return 1;
    }

    if (listen(server_fd, 5) < 0) {
        perror("listen");
        close(server_fd);
        return 1;
    }

    std::cout << "Server listening on port 8080\n";

    int client_fd = accept(server_fd, nullptr, nullptr);
    if (client_fd < 0) {
        perror("accept");
        close(server_fd);
        return 1;
    }

    const char* msg = "Hello from server\n";
    send(client_fd, msg, std::strlen(msg), 0);

    close(client_fd);
    close(server_fd);
}
```

5.2.2 Server Walkthrough

The server begins by creating a TCP socket. At this point, the operating system allocates internal data structures to track protocol state and buffers.

The socket is then bound to port 8080 on all available network interfaces. Binding determines where connection requests will be delivered.

Calling `listen` transitions the socket into a passive state. The server is now ready to accept incoming connection requests.

The `accept` call blocks until a client connects. When it returns, the server receives a new socket descriptor that represents a single client connection. The original socket remains available to accept additional clients.

Once the connection is established, the server sends a short message to the client and then closes the connection. Closing the socket signals the end of communication and releases kernel resources.

5.3 TCP Client Example

The client program initiates communication with the server. It performs the following steps:

1. create a socket,
2. specify the server address,
3. establish a TCP connection,
4. receive data,
5. and close the socket.

5.3.1 Complete Client Implementation

```
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <iostream>

int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("socket");
        return 1;
    }

    sockaddr_in server{};
    server.sin_family = AF_INET;
    server.sin_port = htons(8080);
    inet_pton(AF_INET, "127.0.0.1", &server.sin_addr);

    if (connect(sock, (sockaddr*)&server, sizeof(server)) < 0) {
        perror("connect");
        close(sock);
        return 1;
    }

    char buffer[128]{};
    read(sock, buffer, sizeof(buffer));

    std::cout << "Received: " << buffer;

    close(sock);
}
```

5.3.2 Client Walkthrough

The client begins by creating a TCP socket. Unlike the server, the client does not explicitly bind the socket to a local port. Instead, the operating system automatically assigns an ephemeral port.

The client specifies the server's address using `sockaddr_in` and converts the human-readable IP address into binary form.

The `connect` call initiates the TCP handshake. Only after the handshake completes successfully does the connection become usable.

The client then reads data from the socket. Because TCP provides a byte stream, the `read` call may return less data than requested. In this simple example, the message is short enough to be read in a single operation.

After receiving the message, the client closes the socket, signaling the end of communication.

5.4 Important Observations

Several important concepts are illustrated by these examples:

- server and client roles are asymmetric,
- sockets are kernel-managed resources,
- data is exchanged as a stream, not messages,
- and every operation can potentially fail.

Although minimal, these programs represent a complete and correct TCP interaction. They form a foundation upon which more complex and robust systems can be built.

Subsequent chapters will extend these examples with improved error handling, resource management, and scalability considerations.

Chapter 6

Error Handling and Robustness

Error handling is not an auxiliary concern in network programming; it is a central design requirement. Unlike local computation, network communication is inherently unreliable. Connections may be interrupted, data may arrive partially, and system resources may become unavailable at any time.

Robust network programs are built on the assumption that *every* network operation can fail, block, or produce unexpected results. This chapter examines the most common failure modes and demonstrates how to handle them correctly in C++.

6.1 The Nature of Network Failures

In a local program, failure often indicates a programming error. In a networked program, failure is part of normal operation.

Network failures arise from many sources:

- physical network issues,
- remote process termination,

- congestion and timeouts,
- operating system resource limits,
- and malformed or unexpected peer behavior.

Programs that do not explicitly account for these conditions are inherently fragile.

6.2 Partial Reads and Writes

One of the most common sources of bugs in TCP programs is the assumption that a single `send` corresponds to a single `recv`. This assumption is incorrect.
TCP provides a byte stream. Data may be fragmented or coalesced arbitrarily by the network stack.

6.2.1 Partial Writes

The `send` function may write fewer bytes than requested.

```
ssize_t bytes_sent = send(sock, buffer, length, 0);
```

If `bytes_sent` is less than `length`, the application must send the remaining data explicitly.

```
ssize_t total_sent = 0;
while (total_sent < length) {
    ssize_t n = send(sock,
                     buffer + total_sent,
                     length - total_sent,
                     0);
    if (n <= 0) {
        perror("send");
        break;
```

```
    }
    total_sent += n;
}
```

6.2.2 Partial Reads

Similarly, `recv` may return fewer bytes than requested, even if more data is expected.

```
ssize_t n = recv(sock, buffer, sizeof(buffer), 0);
```

Applications must be prepared to accumulate data until a complete message has been received, according to the application's framing protocol.

6.3 Interrupted System Calls

System calls may be interrupted by signals, causing them to fail with `EINTR`.

```
ssize_t n = recv(sock, buffer, sizeof(buffer), 0);
if (n < 0 && errno == EINTR) {
    // retry operation
}
```

Robust programs either retry interrupted calls or structure their logic to tolerate interruptions.

6.4 Connection Drops

A TCP connection may be closed by the remote peer at any time. This condition is detected when `recv` returns zero.

```
ssize_t n = recv(sock, buffer, sizeof(buffer), 0);
if (n == 0) {
    // peer has closed the connection
}
```

Programs must treat this as a normal termination event, not an error.

6.5 Resource Exhaustion

Network programs rely on limited system resources:

- file descriptors,
- memory buffers,
- and kernel networking structures.

Failures such as `EMFILE` or `ENOBUFS` indicate that system limits have been reached. Robust programs detect these conditions and respond gracefully, rather than crashing or leaking resources.

6.6 Defensive Programming Strategies

Robust network software employs several defensive strategies:

- checking the return value of every system call,
- validating all external input,
- enforcing timeouts and limits,
- and cleaning up resources deterministically.

6.7 The Role of C++ in Robustness

C++ provides tools that greatly assist in writing robust networked programs:

- RAII for automatic resource management,
- strong typing for protocol structures,
- scoped lifetime control,
- and structured error handling.

When used correctly, these tools reduce the likelihood of leaks and undefined behavior without hiding failure modes.

6.8 Failure as a First-Class Concept

In network programming, failure is not exceptional. It is expected.
Designing with failure in mind leads to systems that degrade gracefully, recover predictably, and remain maintainable under real-world conditions.
This chapter lays the groundwork for writing network programs that remain correct and reliable long after the initial implementation has been completed.

Chapter 7

Performance and Scalability Basics

Performance and scalability are often introduced late in the development of networked software, yet many architectural decisions that affect them are made at the earliest stages. Understanding the performance characteristics of basic network I/O models is therefore essential before attempting to design systems that must handle high concurrency or large workloads.

This chapter introduces the fundamental performance considerations of network programming and explains why certain designs scale while others do not.

7.1 Blocking I/O as the Baseline Model

By default, sockets operate in blocking mode. In this model, calls such as `accept`, `recv`, and `send` block the calling thread until the operation completes.

Blocking I/O has several advantages:

- the control flow is simple and intuitive,
- the code closely reflects the logical communication sequence,

- error handling is localized and straightforward.

For simple applications and low connection counts, blocking I/O is often the most appropriate choice.

7.2 The Scalability Limits of Blocking I/O

The primary limitation of blocking I/O is that each blocking operation occupies a thread while waiting for network activity.
In a typical blocking server, a common pattern is:

- one thread per connection,
- or one thread per request.

While this model is easy to implement, it does not scale well:

- threads consume memory and kernel resources,
- context switching introduces overhead,
- large numbers of idle threads waste CPU time.

As the number of concurrent connections grows, performance degrades and resource exhaustion becomes likely.

7.3 Non-Blocking Sockets

Non-blocking sockets allow I/O operations to return immediately, even if no data is available.
A socket can be placed into non-blocking mode using a file control operation.

```c
#include <fcntl.h>

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

In non-blocking mode, operations such as `recv` and `send` return immediately with an indication that the operation would block.
This shifts control over waiting and scheduling from the kernel to the application.

7.4 Event-Driven Models

Event-driven architectures are built on non-blocking sockets combined with an event notification mechanism.
Instead of blocking on a single socket, the application:

- registers interest in events,
- waits for notifications,
- and processes only sockets that are ready.

Common event notification mechanisms include:

- `select`
- `poll`
- platform-specific scalable mechanisms

Event-driven models allow a small number of threads to manage thousands of concurrent connections.

7.5 Thread Pools

Another approach to scalability is the use of thread pools.
In this model:

- a fixed number of worker threads is created,
- incoming work is queued,
- threads process tasks as they become available.

Thread pools limit resource usage and reduce the overhead associated with creating and destroying threads.
They are often combined with blocking or non-blocking I/O, depending on the design.

7.6 Asynchronous I/O

Asynchronous I/O allows the operating system to perform network operations in the background and notify the application upon completion.
From the application's perspective:

- operations are initiated without blocking,
- completion is handled through callbacks or events.

Asynchronous I/O can offer excellent scalability but introduces additional complexity in control flow and error handling.

7.7 Choosing the Right Model

There is no universally optimal I/O model. The correct choice depends on:

- expected concurrency levels,
- latency requirements,
- complexity constraints,
- and available system resources.

Understanding the strengths and limitations of each approach allows developers to make informed architectural decisions.

7.8 Foundations First

All high-performance network designs are built on the same fundamental TCP and socket concepts presented in earlier chapters.
Without a clear understanding of blocking behavior, byte streams, and failure modes, advanced techniques become difficult to reason about and debug.
This chapter provides a conceptual bridge between basic socket programming and more advanced scalable architectures, which can be explored confidently once the fundamentals are firmly understood.

Conclusion

Network programming in C++ is not defined by the number of APIs a developer can recall, nor by familiarity with a particular framework or library. At its core, it is a systems discipline that requires a clear understanding of how applications interact with the operating system, the network stack, and remote peers.

Throughout this booklet, the emphasis has been placed on fundamentals: how TCP connections are established and managed, how data flows as a byte stream, how sockets represent kernel-managed resources, and how failures are an expected and normal part of networked software. These concepts form the foundation upon which all reliable and scalable network systems are built.

Once these fundamentals are mastered, developers are no longer forced to rely on trial-and-error or opaque abstractions. Instead, they gain the ability to reason about behavior, diagnose failures, and make informed design decisions. This understanding enables a smooth transition toward more advanced architectures, including asynchronous I/O models, event-driven servers, thread pools, and fully distributed systems.

It is important to recognize that advanced frameworks do not eliminate complexity; they reorganize it. Without a solid grasp of the underlying TCP and socket model, such frameworks can become sources of confusion rather than productivity.

Strong fundamentals are therefore not a preliminary step to be discarded once “real” development begins. They remain the most reliable optimization across the entire lifecycle of a networked system, guiding design, implementation, debugging, and long-term maintenance.

A developer who understands the system beneath the code is equipped not only to use existing tools effectively, but also to evaluate them critically and, when necessary, build the right abstractions with confidence and precision.

Appendices

The appendices provide reference material that complements the main chapters. They are intended to be consulted as needed, rather than read linearly, and focus on practical details that frequently arise during real-world network programming.

Appendix A: Common Socket Structures

Network programming in C and C++ relies on a small set of fundamental data structures that represent addresses and protocol information. Understanding these structures is essential for interpreting socket code and diagnosing errors.

`sockaddr`

The `sockaddr` structure is a generic container used by the socket API. It does not represent a concrete address format by itself. Instead, it serves as a common interface that allows different address types to be passed to system calls.

In practice, applications rarely fill a `sockaddr` directly. They typically use a protocol-specific structure and cast it when calling socket functions.

`sockaddr_in`

The `sockaddr_in` structure represents an IPv4 address.

It contains:

- the address family,
- the port number,
- and the IPv4 address.

```
sockaddr_in addr{};
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = INADDR_ANY;
```

This structure is used for both binding local addresses and specifying remote endpoints.

Other Address Structures

Other protocol families define their own structures, such as those for IPv6. While the fields differ, the conceptual role remains the same: they describe how to reach a communication endpoint.

Appendix B: Common Errors

Network programming exposes applications to a wide range of error conditions. Some of the most frequently encountered errors include the following.

`ECONNREFUSED`

This error indicates that a connection attempt was actively refused by the remote host. Common causes include:

- the server is not running,
- the server is not listening on the specified port,
- a firewall is blocking the connection.

`EADDRINUSE`

This error occurs when attempting to bind a socket to a port that is already in use. It often arises when:

- a previous instance of the server has not terminated cleanly,
- the operating system is holding the port in a transient state.

Proper socket options and graceful shutdown procedures help mitigate this issue.

`ETIMEDOUT`

This error indicates that a connection attempt or operation has exceeded the allowed time. Timeouts are common on unreliable networks and must be handled gracefully. Applications should treat timeouts as expected events rather than exceptional failures.

Appendix C: Platform Notes

While the socket API is conceptually consistent across platforms, there are important practical differences between POSIX systems and Windows.

On POSIX systems, sockets are treated as file descriptors and integrate naturally with the standard I/O model. Initialization is implicit, and cleanup is performed by closing the descriptor.

On Windows, the Winsock API requires explicit initialization and cleanup. Despite these differences, the underlying concepts—connections, streams, and endpoints—remain identical. Understanding these platform distinctions allows developers to write portable code while respecting the requirements of each environment.

References

The material presented in this booklet is based on long-established standards, authoritative technical literature, and practical experience with real-world systems. The following references provide deeper coverage of the topics discussed and are widely regarded as reliable sources within the systems and network programming community.

- **UNIX Network Programming, Volume 1** — W. Richard Stevens A foundational work that provides an in-depth and rigorous treatment of the socket API, TCP/IP protocols, and practical network programming patterns on UNIX systems.

- **TCP/IP Illustrated** — W. Richard Stevens A detailed exploration of the TCP/IP protocol suite, focusing on how protocols behave in real implementations and how theoretical concepts translate into actual network traffic.

- **POSIX.1 Specification** The official specification defining the behavior of system calls and interfaces used by POSIX-compliant operating systems, including socket operations and related I/O mechanisms.

- **Linux Man Pages Project** A comprehensive reference documenting Linux system calls, library functions, and kernel interfaces, offering precise descriptions of socket-related APIs and error conditions.

- **ISO C++ Standard Documentation** The authoritative reference for the C++ language,

describing core language rules, standard library facilities, and the guarantees provided by modern C++ across platforms and implementations.

These sources are recommended for readers who wish to deepen their understanding, verify behavior across systems, or explore advanced topics beyond the scope of this booklet.