GO SOCKET PROGRAMMING

FROM FUNDAMENTALS TO PRODUCTION MASTERY

- TCP & UDP
- UNIX DOMAIN SOCKETS
- CONCURRENT SERVERS
- PROTOCOL DESIGN
- TLS ENCRYPTION
- WEBSOCKETS
- CONNECTION MANAGEMENT
- PERFORMANCE TUNING
- OBSERVABILITY
- PROXIES, LOAD BALANCERS & DISTRIBUTED SYSTEMS

QUINTES RYAN

Go Socket Programming

From Fundamentals to Production Mastery

Steve Publications

This book is available at https://leanpub.com/gosocketprogramming

This version was published on 2026-07-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

From Fundamentals to Production Mastery

This book teaches you how to build reliable, high-performance networked systems in Go, from understanding fundamental networking concepts through to advanced production-grade socket programming. You will learn TCP and UDP programming, Unix domain sockets, concurrent server architectures, protocol design, TLS encryption, WebSockets, connection management, performance tuning, observability, and real-world patterns such as proxies, load balancers, and distributed system components. Every concept is accompanied by complete, working Go code examples you can compile and run immediately. The book assumes basic familiarity with Go syntax and focuses on idiomatic use of the standard library, modern best practices, and production-ready techniques that scale from a single server to millions of connections.

Introduction: Why Sockets Matter in Go

A production service written in Go is failing intermittently. Requests time out for no apparent reason. The server has plenty of CPU and memory. Logs show connections accepted but then nothing happens. After hours of debugging, the root cause emerges: a client closed its connection without sending a FIN packet, and the server's goroutine handling that connection is now blocked forever on a Read call, waiting for data that will never arrive. The server keeps accumulating such zombie connections until it exhausts all available file descriptors and can no longer accept new connections at all.

This scenario is not hypothetical. It happens regularly in production environments worldwide. And the fix is straightforward if you understand how sockets work and how Go's networking primitives behave: set a read deadline on the connection. One line of code, but only if you know to add it and why.

That is what this book is about. It is not just about calling net.Listen and net.Dial. It is about understanding what happens when you call them, what can go wrong, and how to build networked systems that are correct, efficient, and resilient under real-world conditions.

The Promise of This Book

After reading this book, you will be able to:

- Explain the networking concepts that underpin socket programming, including the TCP/IP stack, DNS resolution, and the tradeoffs between connection-oriented and connectionless protocols.
- Build TCP servers and clients in Go using idiomatic patterns that handle many concurrent connections efficiently.
- Choose between TCP, UDP, and Unix domain sockets based on your application's requirements for reliability, latency, and topology.

- Design and implement application-layer protocols on top of raw sockets, including text protocols, length-prefixed binary framing, and custom RPC-style communication.
- Secure your connections with TLS encryption and mutual authentication.
- Manage connection lifecycles properly with timeouts, deadlines, keepalives, and graceful shutdown.
- Scale your network services to handle tens of thousands of concurrent connections without running out of resources.
- Implement production patterns such as proxies, load balancers, circuit breakers, and connection pools.
- Profile, benchmark, and test your network code to identify and fix performance problems.
- Add observability through logging, metrics, and distributed tracing to understand how your services behave in production.

How to Use This Book

This book progresses from fundamentals to advanced topics. Chapters 1 through 3 establish the foundation: networking concepts, Go's net package, and basic TCP server and client patterns. Chapters 4 through 8 cover the core mechanics of reliable network programming: TCP tuning, UDP, Unix domain sockets, concurrency patterns, and timeouts. Chapters 9 through 12 address application-layer concerns: protocol design, serialization, TLS encryption, and WebSockets. The final chapters focus on production-grade systems: high-performance networking, proxies and load balancers, distributed system components, and observability.

Each chapter contains complete, self-contained code examples that you can compile and run without modification. The examples are designed to be instructive rather than minimal; they include error handling, logging, and other production considerations from the start. Where appropriate, the book references relevant RFCs and official Go documentation for deeper study.

The book does not contain exercises or quizzes. Instead, it focuses on thorough explanations and working examples that you can adapt to your own projects. If you want to practice, modify the examples, break them, and fix them. That is the best way to learn network programming.

Prerequisites

This book assumes you understand basic Go syntax: variables, functions, structs, interfaces, pointers, slices, maps, and packages. You should know what goroutines and channels are and have written at least a small program using them. If you need a refresher on Go fundamentals, consider reading "The Go Programming Language" by Donovan and Kernighan or the official Go tour at tour.golang.org before starting this book.

No prior networking experience is required. Chapter 1 introduces the essential networking concepts from scratch, and later chapters explain networking details as they become relevant. If you already understand TCP/IP, you can skim Chapter 1 and use it as a reference.

Setting Up Your Environment

You will need Go installed on your system. The examples in this book are written for Go 1.22 or later, which includes several improvements to the net and crypto/tls packages. You can check your Go version with go version and download the latest version from go.dev/dl/.

Most examples run on Linux, macOS, and Windows without modification. A few topics, such as Unix domain sockets and certain kernel tuning options, are platform-specific and are clearly noted where applicable.

To run the examples, create a directory for each one, initialize a Go module with go mod init example.com/hello, paste the code into main.go, and run it with go run main.go. Some examples require running both server and client components in separate terminal windows or processes; these are clearly indicated.

The Philosophy Behind Go's Networking Approach

Go takes a distinctive approach to network programming that reflects the language's broader design philosophy: simplicity, explicitness, and pragmatic concurrency.

Unlike languages that rely on callback-based asynchronous I/O (Node.js, JavaScript) or complex event loops (libevent, Netty), Go lets you write blocking-style code that scales. When a goroutine calls conn.Read(), it appears to block

until data arrives, but under the hood the Go runtime parks the goroutine and uses the operating system's efficient I/O multiplexing facilities (epoll on Linux, kqueue on macOS and BSD, IOCP on Windows) to wake it up when data is ready. You get the simplicity of blocking code with the scalability of asynchronous I/O.

This model is made possible by Go's integrated network poller, a runtime subsystem that manages the relationship between goroutines and file descriptors. When a goroutine performs a blocking network operation, the runtime registers the file descriptor with the OS poller and suspends the goroutine. When the poller detects that the file descriptor is ready, it resumes the appropriate goroutine. This happens entirely within the runtime, so your code never needs to call epoll_create, kevent, or any other platform-specific API directly.

The net package exposes this model through a small set of clean interfaces. The Conn interface defines Read, Write, Close, and a few metadata methods. The Listener interface defines Accept and Close. These interfaces are implemented by TCPConn, UDPConn, UnixConn, and TLS connections alike, so you can write code that works with any network type without knowing the details.

This design choice has profound implications for how you should think about network programming in Go. You do not need to build an event loop or manage callbacks. Instead, you spawn a goroutine per connection (or use a worker pool), and each goroutine handles its connection with straightforward, sequential code. The runtime takes care of the concurrency and I/O multiplexing for you.

But simplicity does not mean there are no pitfalls. Deadlines must be set explicitly or connections can hang forever. Buffers must be managed carefully or performance will suffer under load. Errors must be classified and handled appropriately or transient failures become permanent outages. This book covers all of these concerns and more, with the goal of helping you build network services that are not just correct in the happy path but robust under the messy conditions of production.

The journey ahead starts with understanding how networks work at a fundamental level. Without that foundation, socket APIs are magic boxes that produce mysterious failures when things go wrong. With it, you can reason about your code, debug problems systematically, and build systems that earn the trust of production environments.

Chapter 1: Networking Fundamentals: The Foundation You Cannot Skip

When a Go program calls net.Dial("tcp", "example.com:443"), a cascade of events unfolds across multiple layers of abstraction. The operating system resolves the hostname to an IP address, allocates a socket, initiates a three-way handshake with the remote server, negotiates TLS encryption, and finally establishes a reliable byte-stream connection that your program can read from and write to.

If you treat this process as magic, you will be blindsided when it fails. Networks are unreliable by nature. Packets get lost, routers drop connections, firewalls block traffic, and servers become overloaded. Understanding what happens under the hood is not academic exercise; it is essential for building services that survive these failures gracefully.

This chapter introduces the networking concepts you need to understand before diving into Go's socket programming APIs. If you already have a strong networking background, feel free to skim this chapter and return to it as a reference.

How the Internet Works at a Glance

The internet is a network of networks. Your computer connects to a local network, which connects to your ISP, which connects to other ISPs, and eventually to the server you are trying to reach. Data travels across this interconnected infrastructure in small units called packets.

Each packet contains two parts: the payload (your actual data) and headers (metadata about where the packet is going, how to handle it, and how to reassemble it with other packets). As a packet travels from source to destination, it passes through routers that examine the headers and decide which path to send the packet on next. Different packets from the same message may take different paths and arrive out of order or at different times.

This design is intentionally decentralized and fault-tolerant. There is no central authority routing your traffic. Each router makes independent decisions based on routing tables that describe the topology of the network. If one path fails, routers can redirect traffic through alternative routes. This is why the internet can survive partial failures and continue functioning even when parts of it are down.

The downside of this design is that the network provides no guarantees about delivery. Packets can be lost, duplicated, delayed, or corrupted. It is up to higher-level protocols to handle these failures and provide the reliability that applications need.

The TCP/IP Stack Explained Simply

The protocols that make networking work are organized into layers, each building on the one below it. This layered architecture, known as the protocol stack, allows different parts of the system to evolve independently. You can change how data is encrypted at the transport layer without affecting how it is routed at the network layer.

The most common model for describing these layers is the TCP/IP stack, which consists of four layers:

Application Layer: This is where your program lives. Protocols like HTTP, FTP, SMTP, and DNS operate at this layer. Application-layer protocols define how programs format and interpret data. When you write a Go program that sends an HTTP request, your code is implementing the application layer.

Transport Layer: The transport layer provides communication between processes on different machines. The two main transport-layer protocols are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP provides reliable, ordered delivery of data. UDP provides best-effort delivery with no guarantees. Your choice between TCP and UDP is one of the most important decisions in network programming.

Network Layer: The network layer handles routing packets across networks. IP (Internet Protocol) operates at this layer. IP addresses identify machines on the network, and routers use these addresses to decide where to send packets. IP itself provides no reliability guarantees; it simply delivers packets from source to destination if it can.

Link Layer: The link layer handles communication between directly connected devices on the same physical network. Ethernet, Wi-Fi, and other local networking technologies operate at this layer. The link layer deals with physical addresses (MAC addresses), frame formatting, and error detection at the hardware level.

When your Go program sends data over a TCP connection, the data passes through these layers in order. At each layer, a header is added that contains metadata for that layer's protocol. This process is called encapsulation. On the receiving side, the headers are removed in reverse order as the data moves up the stack.

Understanding this layered model helps you reason about where problems occur. A DNS resolution failure happens at the application layer. A packet being dropped by a firewall might happen at the network layer. A corrupted frame detected by a NIC happens at the link layer. Different layers require different debugging tools and approaches.

IP Addresses and Ports

Every machine on the internet has an IP address, which serves as its unique identifier for routing purposes. There are two versions of IP in use today: IPv4 and IPv6.

IPv4 addresses are 32-bit numbers, typically written in dotted-decimal notation such as 192.168.1.1 or 8.8.8.8. The total address space of IPv4 is approximately 4.3 billion addresses, which has proven insufficient for the growing number of connected devices. IPv6 addresses are 128-bit numbers, written in hexadecimal notation such as 2001:0db8:85a3:0000:0000:8a2e:0370:7334. The IPv6 address space is effectively unlimited for practical purposes.

In Go, you work with IP addresses through the net.IP type, which represents both IPv4 and IPv6 addresses as byte slices. The net.ParseIP function converts a string to a net.IP, and the String method converts it back. You rarely need to manipulate IP addresses directly in application code, but understanding them helps when configuring servers to listen on specific interfaces or debugging connectivity issues.

An IP address identifies a machine, but not a specific process running on that machine. Ports solve this problem. A port is a 16-bit number (0 through 65535) that identifies a specific service or process on a machine. When you

connect to example.com:80, the IP address of example.com determines which machine to send the packet to, and port 80 determines which process on that machine should receive it.

Port numbers are divided into three ranges:

- Well-known ports (0-1023): Assigned by IANA for standard services. Port 80 is HTTP, port 443 is HTTPS, port 22 is SSH, port 53 is DNS. On Unix systems, only processes running as root can bind to these ports.
- Registered ports (1024-49151): Assigned by IANA for specific applications but can be used by any process. Many popular services use ports in this range, such as MySQL on 3306 and PostgreSQL on 5432.
- Dynamic or private ports (49152-65535): Available for temporary use. When a client initiates a connection, the operating system typically assigns it a random port from this range as its source port.

In Go, you specify the network and address together when creating a connection. The net.Listen("tcp", ":8080") call tells the server to listen on all available network interfaces on port 8080. The net.Dial("tcp", "example.com:443") call connects to example.com on port 443. Using net.JoinHostPort(host, port) is the recommended way to construct addresses safely, especially when dealing with IPv6 addresses that contain colons.

DNS Basics for Network Programmers

Domain names like example.com are human-readable aliases for IP addresses. The Domain Name System (DNS) translates these names into IP addresses that computers can use. When your Go program calls net.Dial("tcp", "example.com:443"), the first step is typically a DNS lookup to find the IP address of example.com.

The DNS lookup process works as follows:

1. Your program asks the operating system to resolve the hostname.
2. The OS checks its local cache for a recent answer.
3. If not cached, the OS queries a configured DNS resolver, typically your ISP's DNS server or a public resolver like 8.8.8.8 (Google) or 1.1.1.1 (Cloudflare).

4. The resolver may need to query multiple authoritative name servers to find the answer.
5. The IP address is returned to your program, which uses it to establish the connection.

This process can take anywhere from a fraction of a millisecond (if cached) to several hundred milliseconds or more (if there are network issues or the resolver needs to traverse the DNS hierarchy). For this reason, DNS failures and slow lookups are common sources of latency and errors in networked applications.

Go provides several functions for DNS resolution:

- net.LookupHost(name) returns the IP addresses associated with a hostname.
- net.LookupIP(name) is similar but allows specifying the IP version.
- net.LookupTXT(name), net.LookupMX(name), and others resolve specific record types.
- The resolver behavior can be controlled via the GODEBUG environment variable (GODEBUG=netdns=go forces the pure Go resolver; GODEBUG=netdns=cgo forces the system resolver).

In production applications, DNS issues are more common than many developers expect. A DNS server might return stale records after you have changed your infrastructure. A DNS query might time out due to network congestion. Some resolvers cache aggressively, which can cause problems when you need changes to take effect quickly. For critical services, consider implementing your own DNS caching layer or using service discovery mechanisms that reduce reliance on external DNS.

Connection-Oriented vs Connectionless Communication

The choice between connection-oriented and connectionless communication is fundamental to network programming. TCP is connection-oriented; UDP is connectionless. Understanding the tradeoffs helps you choose the right tool for your application.

Connection-oriented (TCP): Before any data is exchanged, a connection is established through a three-way handshake. The client sends a SYN packet,

the server responds with SYN-ACK, and the client confirms with ACK. This handshake ensures both sides are ready to communicate and synchronizes sequence numbers for reliable delivery. Once connected, TCP provides:

- Guaranteed delivery: Every byte sent is either delivered to the receiver or reported as an error.
- Ordering: Bytes arrive in the same order they were sent.
- Flow control: The receiver can signal when it cannot handle data as fast as it is being sent.
- Congestion control: TCP reduces its sending rate when it detects network congestion.

These guarantees come at a cost. TCP adds overhead through headers, acknowledgments, and retransmissions. It can introduce latency, especially for small messages. And the connection state must be maintained on both sides, consuming resources.

TCP is the right choice when reliability matters more than speed: web traffic, file transfers, database connections, and most application-layer protocols.

Connectionless (UDP): With UDP, there is no handshake and no connection state. The sender simply packages data into datagrams and sends them toward the receiver. Each datagram is independent; lost datagrams are not retransmitted, out-of-order datagrams are not reordered, and there is no flow control or congestion control.

UDP provides:

- Lower latency: No handshake, no acknowledgments, no retransmissions.
- Lower overhead: Smaller headers and less processing.
- Broadcast and multicast support: A single datagram can be delivered to multiple receivers.
- Simplicity: No connection state to manage.

These advantages come with significant risks. Data can be lost, duplicated, or arrive out of order without any notification. There is no built-in mechanism for detecting failures or recovering from them.

UDP is the right choice when speed matters more than reliability, or when the application can handle its own reliability: real-time video streaming, online gaming, DNS queries, and VoIP.

In Go, you choose between TCP and UDP by specifying the network type in your calls. net.Listen("tcp", ":8080") creates a TCP server; net.ListenUDP("udp", addr) creates a UDP listener. The APIs are different because the semantics are different, but both are straightforward to use once you understand what each protocol provides.

Chapter Summary

This chapter has introduced the networking fundamentals that underpin all socket programming:

- The internet routes data in packets across a decentralized network of routers, with no guarantees of delivery.
- The TCP/IP stack organizes protocols into layers: application, transport, network, and link.
- IP addresses identify machines; ports identify processes on those machines.
- DNS translates human-readable hostnames into IP addresses, and DNS issues are a common source of production problems.
- TCP provides reliable, ordered delivery with overhead; UDP provides fast, unreliable delivery with simplicity.

These concepts are not just background information. They directly affect how you design and implement your network services in Go. When you understand why packets are lost, you write code that handles transient errors. When you understand how DNS works, you add timeouts to your lookups and cache results appropriately. When you understand the tradeoffs between TCP and UDP, you choose the right protocol for your application's requirements.

The next chapter dives into Go's net package, the primary API you will use for all socket programming in this book.

Chapter 2: Go's net Package: Your Socket Programming Toolbox

Go's net package is the foundation of all network programming in the language. It provides a portable interface for TCP/IP, UDP, Unix domain sockets, and DNS resolution. The package is designed around a small set of clean interfaces that abstract away platform-specific details while still giving you access to low-level socket options when you need them.

Most of the code you write for network programming in Go will use functions and types from this package: Dial, Listen, Accept, the Conn interface, TCPConn, UDPConn, and address resolution helpers. Understanding this package thoroughly is essential before moving on to building real servers and clients.

The net.Dial and net.Listen Functions

The two most important functions in the net package are Dial and Listen. Dial connects to a remote server; Listen creates a server that accepts incoming connections. Both take the same basic arguments: a network type and an address.

The network type is a string that specifies the protocol and address family. Common values include:

- "tcp" for TCP (both IPv4 and IPv6)
- "tcp4" for TCP over IPv4 only
- "tcp6" for TCP over IPv6 only
- "udp" for UDP (both IPv4 and IPv6)
- "udp4" for UDP over IPv4 only
- "udp6" for UDP over IPv6 only
- "unix" for Unix domain sockets (stream-oriented, like TCP)
- "unixgram" for Unix domain sockets (datagram-oriented, like UDP)

The address is a string in the format "host:port" for TCP and UDP, or a file path for Unix domain sockets. Using net.JoinHostPort(host, port) is the recommended way to construct addresses because it handles IPv6 addresses correctly (which contain colons and would break naive concatenation).

Here is the simplest possible TCP client that connects to a server and sends an HTTP request:

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "net"
)

func main() {
    // Connect to the server at golang.org on port 80.
    // net.Dial blocks until the connection is established or an error occurs.
    conn, err := net.Dial("tcp", "golang.org:80")
    if err != nil {
        log.Fatalf("Failed to connect: %v", err)
    }
    // Always close the connection when done.
    defer conn.Close()

    // Send a minimal HTTP/1.0 GET request.
    // The request ends with \r\n\r\n (two CRLF sequences).
    _, err = fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
    if err != nil {
        log.Fatalf("Failed to send request: %v", err)
    }

    // Read the response line by line using a buffered reader.
    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        fmt.Println(scanner.Text())
    }
    if err := scanner.Err(); err != nil {
        log.Fatalf("Failed to read response: %v", err)
    }
}
```

Line by line breakdown:

- net.Dial("tcp", "golang.org:80") initiates a TCP connection. It first resolves the hostname golang.org to an IP address via DNS, then performs the

three-way handshake with the server. This call blocks until either the connection succeeds or fails.

- The returned conn is of type net.Conn, which is an interface that provides Read, Write, Close, and metadata methods. We do not need to know the concrete type (it will be *net.TCPConn) because we program against the interface.
- defer conn.Close() ensures the connection is closed when main returns, even if an error occurs. This is critical for avoiding resource leaks.
- fmt.Fprintf(conn, ...) writes the HTTP request to the connection. The conn implements io.Writer, so any function that writes to an io.Writer works with it.
- bufio.NewScanner(conn) creates a scanner that reads the response line by line. This is convenient for text-based protocols but not suitable for binary data or large responses where you want more control over buffering.

Here is the corresponding server side:

```
1  package main
2
3  import (
4      "log"
5      "net"
6  )
7
8  func main() {
9      // Listen on all interfaces (0.0.0.0) on port 8080 for TCP connections.
10     // The colon before the port means "all available interfaces".
11     listener, err := net.Listen("tcp", ":8080")
12     if err != nil {
13         log.Fatalf("Failed to listen: %v", err)
14     }
15     // Close the listener when the program exits.
16     defer listener.Close()
17
18     log.Println("Server listening on :8080")
19
20     // Accept blocks until a client connects.
21     conn, err := listener.Accept()
22     if err != nil {
23         log.Fatalf("Failed to accept connection: %v", err)
24     }
25     defer conn.Close()
26
```

```
    log.Printf("Accepted connection from %s", conn.RemoteAddr())

    // Echo whatever the client sends back to it.
    buffer := make([]byte, 1024)
    for {
        n, err := conn.Read(buffer)
        if err != nil {
            log.Printf("Read error: %v", err)
            break
        }
        _, writeErr := conn.Write(buffer[:n])
        if writeErr != nil {
            log.Printf("Write error: %v", writeErr)
            break
        }
    }
}
```

Line by line breakdown:

- net.Listen("tcp", ":8080") creates a TCP listener bound to port 8080 on all network interfaces. The returned listener is of type net.Listener, which provides Accept and Close methods.
- listener.Accept() blocks until a client connects. When a connection arrives, it returns a new net.Conn representing that specific connection, while the listener continues waiting for more connections.
- conn.RemoteAddr() returns the address of the connected client as a net.Addr interface.
- The Read/Write loop reads data from the client into a buffer and writes it back. This is a simple echo server. In a real application, you would parse the incoming data according to your protocol and generate appropriate responses.

Both Dial and Listen can fail for many reasons: the address might be invalid, the port might be in use, a firewall might block the connection, or the remote server might refuse the connection. Always handle errors explicitly; never ignore them with blank identifiers in production code.

For more control over connection parameters such as timeouts, local address binding, and dual-stack behavior, use net.Dialer instead of net.Dial directly:

```
dialer := net.Dialer{
    Timeout:   5 * time.Second,
    LocalAddr: &net.TCPAddr{IP: net.IPv4(127, 0, 0, 1)},
}
conn, err := dialer.Dial("tcp", "example.com:443")
```

The Conn Interface Explained

The net.Conn interface is the central abstraction for network connections in Go. Every type of connection (TCP, UDP over a connected socket, Unix domain socket, TLS-wrapped connection) implements this interface. Programming against the Conn interface rather than concrete types makes your code flexible and easier to test.

Here is the complete definition of the Conn interface:

```
type Conn interface {
    // Read reads data from the connection.
    // Read can be made to time out and return an error with Timeout() == true
    // after a fixed time limit; see SetDeadline and SetReadDeadline.
    Read(b []byte) (n int, err error)

    // Write writes data to the connection.
    Write(b []byte) (n int, err error)

    // Close closes the connection.
    // Any blocked Read or Write operations will be unblocked and return errors.
    Close() error

    // LocalAddr returns the local network address.
    LocalAddr() Addr

    // RemoteAddr returns the remote network address.
    RemoteAddr() Addr

    // SetDeadline sets the read and write deadlines associated
    // with the connection. It is equivalent to calling both
    // SetReadDeadline and SetWriteDeadline.
    // A deadline is an absolute time after which I/O operations
    // fail with a timeout (see type Error) instead of
    // blocking. The deadline applies to all future and pending
    // I/O, not just the immediately following call to Read or
    // Write. After Deadline is exceeded, the connection can
    // still be used, but subsequent I/O will time out.
    // A zero value for t means I/O operations will not time out.
```

```
30      SetDeadline(t time.Time) error
31
32      // SetReadDeadline sets the deadline for future Read calls
33      // and any currently-blocked Read call.
34      SetReadDeadline(t time.Time) error
35
36      // SetWriteDeadline sets the deadline for future Write calls
37      // and any currently-blocked Write call.
38      SetWriteDeadline(t time.Time) error
39  }
```

Each method has important semantics:

Read(b []byte): Reads up to len(b) bytes from the connection into the buffer b. Returns the number of bytes read (n) and an error. The key subtlety is that Read may return fewer than len(b) bytes even if no error occurred; this is normal and expected. You must loop until you have read all the data you need or handle partial reads correctly. A return of n == 0 with err == nil should be treated as "no data available yet" (though this is rare in practice). Common errors include io.EOF (connection closed by peer), context.DeadlineExceeded (read timed out), and various network errors.

Write(b []byte): Writes len(b) bytes from the buffer b to the connection. Returns the number of bytes written (n) and an error. Like Read, Write may write fewer than len(b) bytes without returning an error, so you must check n and handle partial writes. In practice, for TCP connections on modern systems, Write usually succeeds completely or fails entirely, but you should not rely on this.

Close(): Closes the connection, freeing associated resources. After Close is called, any blocked Read or Write operations will be unblocked and return errors. Calling Close multiple times is safe (subsequent calls return nil). Always close connections when you are done with them to avoid exhausting file descriptors.

LocalAddr() and RemoteAddr(): Return the local and remote addresses as net.Addr interfaces. These are useful for logging, access control, and debugging. You can type-assert the result to *net.TCPAddr, *net.UDPAddr, or *net.UnixAddr to get protocol-specific details like IP address and port number.

SetDeadline, SetReadDeadline, SetWriteDeadline: These methods set timeouts on I/O operations. A deadline is an absolute time (time.Time), not a duration. When the deadline passes, pending Read or Write calls return

an error with Timeout() == true. Setting the deadline to the zero value (time.Time{}) disables the timeout. Deadlines are critical for production code; without them, connections can hang indefinitely if the peer stops responding.

The Conn interface is deliberately minimal. It does not include methods for protocol-specific operations like setting TCP options or sending out-of-band data. Those are available on the concrete types (*net.TCPConn, etc.) through type assertions. This design keeps the core abstraction simple while allowing specialized functionality when needed.

TCP-Specific Types and Methods

When you call net.Listen("tcp", ...) or net.Dial("tcp", ...), the underlying implementation uses TCP-specific types from the net package. The most important is *net.TCPConn, which implements Conn but adds TCP-specific methods for tuning connection behavior.

To access these methods, you type-assert the conn to *net.TCPConn:

```
tcpConn, ok := conn.(*net.TCPConn)
if !ok {
    log.Fatal("Expected a TCP connection")
}
tcpConn.SetNoDelay(false)
```

Here are the key TCP-specific methods:

SetKeepAlive(keepalive bool): Enables or disables TCP keepalive probes on the connection. When enabled, the operating system periodically sends small packets to verify that the peer is still reachable. If the peer does not respond after several attempts, the connection is closed. This helps detect dead connections where the peer has disappeared without sending a FIN packet.

SetKeepAlivePeriod(d time.Duration): Sets the interval between keepalive probes. The default system value is typically very long (hours on many systems), so you usually want to set this to something shorter for production services that need to detect dead peers quickly. Note that the actual behavior depends on operating system configuration; some systems also have parameters for the number of retries before giving up.

SetNoDelay(noDelay bool): Controls Nagle's algorithm. When noDelay is true (TCP_NODELAY enabled), small packets are sent immediately without waiting to coalesce them with other data. This reduces latency but can increase network traffic. When noDelay is false, Nagle's algorithm buffers small writes until either an acknowledgment is received or enough data accumulates to fill a packet. In Go, TCP_NODELAY is enabled by default, so SetNoDelay(true) is typically redundant. You would set it to false only if you want to reduce packet count at the cost of latency.

SetLinger(sec int): Controls what happens when Close is called on a connection that has unsent data. If sec is 0, the connection is closed immediately and any unsent data is discarded. If sec is positive, the system blocks for up to sec seconds trying to send the remaining data. If sec is negative, the default system behavior applies (typically blocking until data is sent or a timeout occurs).

SetReadBuffer(bytes int) and SetWriteBuffer(bytes int): Set the size of the kernel receive and send buffers for this connection. Larger buffers can improve throughput for high-bandwidth connections but consume more memory. These settings are typically only useful for specialized high-performance applications; the default buffer sizes work well for most use cases.

Here is an example that configures a TCP connection with production-appropriate settings:

```
package main

import (
    "log"
    "net"
    "time"
)

func configureTCP(conn net.Conn) error {
    tcpConn, ok := conn.(*net.TCPConn)
    if !ok {
        return nil // Not a TCP connection; nothing to configure.
    }

    // Enable keepalive to detect dead connections.
    if err := tcpConn.SetKeepAlive(true); err != nil {
        return err
    }

    // Check every 30 seconds if the peer is still alive.
```

```
    if err := tcpConn.SetKeepAlivePeriod(30 * time.Second); err != nil {
        return err
    }

    // Ensure low latency by keeping TCP_NODELAY enabled (default in Go).
    if err := tcpConn.SetNoDelay(true); err != nil {
        return err
    }

    return nil
}

func main() {
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatalf("Listen failed: %v", err)
    }
    defer listener.Close()

    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Printf("Accept error: %v", err)
            continue
        }

        // Configure TCP options immediately after accepting.
        if err := configureTCP(conn); err != nil {
            log.Printf("Configure TCP error: %v", err)
            conn.Close()
            continue
        }

        go handleConnection(conn)
    }
}

func handleConnection(conn net.Conn) {
    defer conn.Close()
    // Handle the connection here.
}
```

UDP-Specific Types and Methods

UDP programming in Go uses different types and patterns than TCP because UDP is connectionless. There is no Accept call; instead, you use net.ListenUDP to create a UDP listener, then read and write datagrams directly.

The key type is *net.UDPConn, which provides methods for sending and receiving UDP packets:

ReadFrom(b []byte): Reads a UDP packet from the connection into buffer b and returns the number of bytes read along with the address of the sender. This is the primary way to receive data on a UDP listener.

WriteTo(b []byte, addr Addr): Sends a UDP packet containing the bytes in b to the address specified by addr. Each call sends an independent datagram; there is no connection state maintained between calls.

ReadFromUDP(b []byte): A convenience method that returns the sender's address as *net.UDPAddr instead of net.Addr.

WriteToUDP(b []byte, addr net.UDPAddr): A convenience method that takes a *net.UDPAddr directly.

Here is a simple UDP echo server:

```
package main

import (
    "log"
    "net"
)

func main() {
    // Resolve the address to listen on.
    addr, err := net.ResolveUDPAddr("udp", ":8080")
    if err != nil {
        log.Fatalf("ResolveUDPAddr failed: %v", err)
    }

    // Create a UDP listener.
    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        log.Fatalf("ListenUDP failed: %v", err)
    }
    defer conn.Close()

    log.Println("UDP server listening on :8080")

    buffer := make([]byte, 1024)
    for {
        // ReadFromUDP blocks until a packet arrives.
        n, clientAddr, err := conn.ReadFromUDP(buffer)
        if err != nil {
            log.Printf("Read error: %v", err)
```

```
            continue
        }

        log.Printf("Received %d bytes from %s: %s", n, clientAddr,
        ↪  string(buffer[:n]))

        // Echo the data back to the sender.
        _, err = conn.WriteToUDP(buffer[:n], clientAddr)
        if err != nil {
            log.Printf("Write error: %v", err)
        }
    }
}
```

And a matching UDP client:

```
package main

import (
    "log"
    "net"
)

func main() {
    // Resolve the server address.
    addr, err := net.ResolveUDPAddr("udp", "localhost:8080")
    if err != nil {
        log.Fatalf("Resolve failed: %v", err)
    }

    // Create a UDP connection.
    conn, err := net.DialUDP("udp", nil, addr)
    if err != nil {
        log.Fatalf("DialUDP failed: %v", err)
    }
    defer conn.Close()

    // Send a message to the server.
    message := []byte("Hello, UDP server!")
    _, err = conn.Write(message)
    if err != nil {
        log.Fatalf("Write failed: %v", err)
    }

    // Read the response.
    buffer := make([]byte, 1024)
    n, err := conn.Read(buffer)
    if err != nil {
        log.Fatalf("Read failed: %v", err)
```

```
    }

    log.Printf("Received: %s", string(buffer[:n]))
}
```

Key differences from TCP:

- No connection handshake; you just start sending and receiving packets.
- ReadFromUDP returns the sender's address because there is no established connection.
- Each packet is independent; lost packets are not retransmitted.
- Packets can arrive out of order or be duplicated.
- The buffer size limits how much data you can receive in a single packet; oversized packets are silently dropped by the operating system.

Resolving Addresses with net.ResolveTCPAddr and Friends

Before creating a connection or listener, you often need to convert a string address into a structured address type. The net package provides Resolve* functions for this purpose:

net.ResolveTCPAddr(network, addr): Converts a "host:port" string into a *net.TCPAddr. Useful when you need TCP-specific address methods or want to validate the address format before dialing.

net.ResolveUDPAddr(network, addr): Converts a "host:port" string into a *net.UDPAddr. Required for ListenUDP and DialUDP calls that take a structured address.

net.ResolveIPAddr(network, addr): Resolves an IP address without a port. Used for low-level network operations.

net.ResolveUnixAddr(network, addr): Resolves a Unix domain socket path into a *net.UnixAddr.

These functions perform hostname resolution (DNS lookup) as part of their work. They can fail if the hostname cannot be resolved or if the address format is invalid. Here is how they are typically used:

```go
// Resolve a TCP address before dialing.
tcpAddr, err := net.ResolveTCPAddr("tcp", "example.com:443")
if err != nil {
    log.Fatalf("Failed to resolve address: %v", err)
}
conn, err := net.DialTCP("tcp", nil, tcpAddr)
```

In many cases, you do not need to call these functions explicitly because Dial and Listen accept string addresses and perform resolution internally. However, using Resolve* functions gives you more control: you can validate addresses before use, extract components like IP and port separately, and handle resolution errors independently from connection errors.

For constructing addresses programmatically, net.JoinHostPort is essential:

```go
// Correct way to join host and port (handles IPv6 properly).
addr := net.JoinHostPort("::1", "8080") // Returns "[::1]:8080"

// Naive concatenation fails for IPv6.
addr := "::1" + ":8080" // Returns "::1:8080", which is invalid.
```

Chapter Summary

This chapter has covered the core API of Go's net package:

- net.Dial connects to a remote server; net.Listen creates a server that accepts connections.
- The network type string ("tcp", "udp", "unix") determines the protocol and address family.
- Use net.JoinHostPort to construct addresses safely, especially with IPv6.
- The Conn interface (Read, Write, Close, deadlines, addresses) is the central abstraction for all connection types.
- TCP connections (*net.TCPConn) support tuning via SetKeepAlive, SetNoDelay, SetLinger, and buffer size methods.
- UDP programming uses ListenUDP/DialUDP with ReadFromUDP/WriteToUDP for datagram-based communication.
- Address resolution functions (ResolveTCPAddr, etc.) convert strings to structured address types.

With this foundation, you are ready to build real TCP servers and clients that handle multiple connections concurrently, manage connection lifecycles properly, and implement application-layer protocols. The next chapter covers these patterns in detail.

Chapter 3: Building TCP Servers and Clients: The Connection Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Your First TCP Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Your First TCP Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

The Accept Loop and Connection Handling Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Graceful Shutdown with context.Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Managing Connection State and Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 4: TCP Reliability, Flow Control, and Performance Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

How TCP Guarantees Delivery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Nagle's Algorithm and TCP_NODELAY

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

TCP Keep-Alive and Detecting Dead Connections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Window Scaling and Flow Control Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Handling Partial Writes Correctly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Tuning TCP Parameters in Go

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 5: UDP Programming: When Connectionless Is Right

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Why Choose UDP Over TCP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Basic UDP Client and Server Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Handling Partial Reads and Packet Loss

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Implementing Reliability on Top of UDP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Multicast and Broadcast with UDP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 6: Unix Domain Sockets: Fast Local Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

What Are Unix Domain Sockets and Why Use Them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Creating Unix Domain Socket Servers and Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

File Permissions and Security with Unix Sockets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Abstract Namespace Sockets (Linux-specific)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Comparing Performance: Unix Sockets vs Loopback TCP

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 7: Concurrency Patterns for Network Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

One Goroutine Per Connection: The Standard Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Worker Pools for Connection Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Channels for Coordinating Network Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

sync.Pool for Reusing Buffers and Connections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Avoiding Common Concurrency Bugs in Network Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 8: Timeouts, Deadlines, and Context Cancellation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Why Timeouts Are Non-Negotiable in Production Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

SetDeadline vs SetReadDeadline vs SetWriteDeadline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Integrating context.Context with Network Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Handling Slowloris and Resource Exhaustion Attacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 9: Protocols: From Text to Binary to Custom Designs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Line-Based Text Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Length-Prefixed Message Framing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Binary Protocols with encoding/binary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Protocol Design Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Building a Custom RPC Protocol from Scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 10: Serialization: JSON, Protobuf, and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

JSON with encoding/json (When It Is Good Enough)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Protocol Buffers with Google's protobuf-go

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

MessagePack and Other Binary Formats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Choosing a Serialization Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Streaming Large Payloads Without Loading Everything Into Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 11: TLS/SSL: Encrypting Your Connections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

How TLS Works (Handshake, Certificates, Cipher Suites)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Wrapping TCP Connections with tls.TLSConn

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Creating Self-Signed Certificates for Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Mutual TLS (mTLS) for Client Authentication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Configuring TLS for Production (Cipher Suites, Versions, OCSP)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 12: WebSockets and Real-Time Bidirectional Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

How WebSockets Work (Handshake, Frames, Upgrade)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Using gorilla/websocket in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Building a Real-Time Chat Server with WebSockets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Handling Connection Heartbeats and Reconnection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Scaling WebSocket Servers Across Multiple Instances

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 13: High-Performance Networking: Scaling to Millions of Connections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Epoll, Kqueue, and Go's Network Poller (How Go Scales)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Connection Multiplexing Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Zero-Copy Techniques and Memory-Mapped I/O

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Tuning the Kernel for High-Performance Networking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Debugging Network Problems with External Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Profiling Network Performance with pprof

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 14: Production Patterns: Proxies, Load Balancers, and Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Building a TCP Proxy in Go

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Implementing a Round-Robin Load Balancer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Health Checking and Circuit Breaking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Connection Pooling for Client-Side Efficiency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Observability: Logging, Metrics, and Tracing for Network Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Service Discovery Patterns for Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

gRPC: The Industry Standard for RPC Over Sockets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 15: Testing Network Code — Unit Tests, Integration Tests, and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Why Testing Network Code Is Hard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Unit Testing with net.Pipe

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Mocking net.Conn for Controlled Test Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Integration Testing with Real Listeners

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Using golang.org/x/net/nettest

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Race Condition Testing Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Testing Protocol Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Stress and Load Testing Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter 16: Debugging and Diagnosing Network Problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

When Profiling Is Not Enough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Packet Capture with tcpdump

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Analyzing Packets with Wireshark

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

System Call Tracing with strace

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Connection State Inspection with ss

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Debugging Goroutine Hangs with delve

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Real-World Debugging Walkthroughs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Combining Tools for Complete Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Conclusion: The Art of Reliable Network Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Guiding Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

The Go Advantage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

What Comes Next

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

Final Thought

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gosocketprogramming.