Lab 03 Go Network Programming - TCP & UDP

Duration: 2.5 hours | **Points:** 100

Lab Overview

Objective: Learn Go network programming basics through simple, hands-on TCP and UDP examples.

What You'll Build: Simple client-server programs that communicate over the network using TCP and UDP protocols.

Learning Style:

- 📖 **Read** sample code
- 🧠 **Understand** the concept
- ✍️ **Write** your own code based on examples
- 🚀 **Test** and see the results

Setup (5 minutes)

Create New Project

```
mkdir go-networking-lab
cd go-networking-lab
go mod init networking-lab
```

Create `main.go` file in your project directory.

Part 1: TCP Basics - Echo Server (In class practical)

📖 Sample Code: Understanding TCP Server & Client

TCP (Transmission Control Protocol) provides reliable, ordered, and error-checked delivery of data.

Create `tcp_server.go`:

```
package main

import (
```

```
    "bufio"
    "fmt"
    "net"
    "strings"
)

// TCP Echo Server - receives messages and sends them back
func startTCPServer() {
    // Listen on port 8080
    listener, err := net.Listen("tcp", ":8080")
    if err != nil {
        fmt.Println("Error starting server:", err)
        return
    }
    defer listener.Close()

    fmt.Println("TCP Server listening on :8080")

    for {
        // Accept incoming connection
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Error accepting connection:", err)
            continue
        }

        // Handle connection in goroutine
        go handleTCPConnection(conn)
    }
}

func handleTCPConnection(conn net.Conn) {
    defer conn.Close()

    fmt.Printf("Client connected from %s\n", conn.RemoteAddr())

    scanner := bufio.NewScanner(conn)
    for scanner.Scan() {
        message := scanner.Text()
        fmt.Printf("Received: %s\n", message)

        // Echo back the message in uppercase
        response := strings.ToUpper(message) + "\n"
        conn.Write([]byte(response))
    }

    fmt.Printf("Client %s disconnected\n", conn.RemoteAddr())
}
```

```go
func main() {
    startTCPServer()
}
```

Create `tcp_client.go`:

```go
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
)

func main() {
    // Connect to TCP server
    conn, err := net.Dial("tcp", "localhost:8080")
    if err != nil {
        fmt.Println("Error connecting:", err)
        return
    }
    defer conn.Close()

    fmt.Println("Connected to server!")
    fmt.Println("Type messages (type 'quit' to exit):")

    // Read from stdin and send to server
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        message := scanner.Text()

        if message == "quit" {
            break
        }

        // Send message to server
        fmt.Fprintf(conn, "%s\n", message)

        // Read response
        response, err := bufio.NewReader(conn).ReadString('\n')
        if err != nil {
            fmt.Println("Error reading response:", err)
            break
        }

        fmt.Printf("Server response: %s", response)
```

```
    }
}
```

Run:

```
# Terminal 1 - Start server
go run tcp_server.go

# Terminal 2 - Start client
go run tcp_client.go
```

Expected Output:

Server Terminal:

```
TCP Server listening on :8080
Client connected from 127.0.0.1:54321
Received: hello
Received: world
Client 127.0.0.1:54321 disconnected
```

Client Terminal:

```
Connected to server!
Type messages (type 'quit' to exit):
hello
Server response: HELLO
world
Server response: WORLD
quit
```

✍️ Your Task 1.1: TCP Chat Room Server

🎯 Real-World Application

Ever wondered how Slack, Discord, or Telegram work? They all use TCP to deliver messages reliably between users. In this task, you'll build your own chat server that can handle multiple users talking to each other in real-time - just like the apps you use every day! This is the foundation of any messaging application.

Challenge: Create a TCP chat room where multiple clients can send messages to each other.

Instructions:

1. **Server Function** `chatRoomServer()`:
    - Listen on port 9000

- Accept multiple clients
- Maintain a list of connected clients
- Broadcast messages from one client to all others
- Handle client disconnections gracefully

2. **Client Function** `chatRoomClient(username string)`:
   - Connect to server on port 9000
   - Send username on connection
   - Read messages from server in a goroutine
   - Send user input to server
   - Exit on typing "exit"
3. **Message Format:**
   - Server broadcasts: `[Username]: Message`
   - Join notification: `*** Username joined the chat ***`
   - Leave notification: `*** Username left the chat ***`

Starter Code:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "sync"
)

type ChatRoom struct {
    clients map[net.Conn]string
    mu      sync.Mutex
}

func NewChatRoom() *ChatRoom {
    return &ChatRoom{
        clients: make(map[net.Conn]string),
    }
}

func (cr *ChatRoom) broadcast(message string, sender net.Conn) {
    // TODO: Send message to all clients except sender
```

```
}

func (cr *ChatRoom) addClient(conn net.Conn, username string) {
    // TODO: Add client to chat room
}

func (cr *ChatRoom) removeClient(conn net.Conn) {
    // TODO: Remove client from chat room
}

func handleChatClient(conn net.Conn, cr *ChatRoom) {
    // TODO: Handle individual chat client
    // - Read username
    // - Announce join
    // - Read and broadcast messages
    // - Handle disconnection
}

func startChatServer() {
    // TODO: Start TCP server on port 9000
    // TODO: Accept connections and handle in goroutines
}

func main() {
    startChatServer()
}
```

Hints: - Use a `map[net.Conn]string` to track clients and their usernames

- Use `sync.Mutex` to protect concurrent access to the clients map
- Broadcast format: `fmt.Sprintf("[%s]: %s\n", username, message)`
- Use `bufio.NewScanner()` to read lines from client

Expected Output:

Server:

```
Chat server listening on :9000
Alice joined the chat
Bob joined the chat
Broadcasting from Alice: Hello everyone!
Broadcasting from Bob: Hi Alice!
Alice left the chat
```

Client 1 (Alice):

```
Enter username: Alice
*** Alice joined the chat ***
```

```
Hello everyone!
[Bob]: Hi Alice!
exit
```

Client 2 (Bob):

```
Enter username: Bob
*** Bob joined the chat ***
[Alice]: Hello everyone!
Hi Alice!
```

✍️ Your Task 1.2: TCP File Transfer

🎯 Real-World Application

Every time you send a photo on WhatsApp, upload a document to Google Drive, or download a file from the internet, TCP file transfer is working behind the scenes. Companies like Dropbox and WeTransfer built billion-dollar businesses on this technology. In this task, you'll create your own file transfer system that can reliably send files across the network - the same principle used by professional file-sharing services!

Challenge: Create a simple file transfer system using TCP.

Instructions:

1. **Server Function** `fileTransferServer()`:
    - Listen on port 8081
    - Accept file transfer requests
    - Receive filename and file size
    - Receive file data in chunks
    - Save file to `./received/` directory
    - Send confirmation message
2. **Client Function** `sendFile(filename string)`:
    - Connect to server
    - Send filename and file size
    - Read file and send in 1KB chunks
    - Wait for confirmation
    - Display progress

Starter Code:

```
package main

import (
    "fmt"
    "io"
    "net"
    "os"
    "path/filepath"
)

func fileTransferServer() {
    // TODO: Start server on port 8081
    // TODO: Accept connections
    // TODO: Receive file metadata (name, size)
    // TODO: Receive file data in chunks
    // TODO: Save file
    // TODO: Send confirmation
}

func sendFile(filename string, serverAddr string) {
    // TODO: Connect to server
    // TODO: Open file
    // TODO: Send filename and size
    // TODO: Send file in chunks (1024 bytes)
    // TODO: Display progress
    // TODO: Wait for confirmation
}

func main() {
    // TODO: Implement server mode or client mode
}
```

Hints: - File metadata format: `"FILENAME:filesize\n"`

- Use `io.Copy()` or read in chunks with `file.Read(buffer)`
- Progress: `fmt.Printf("\rSent: %d/%d bytes (%.2f%%)", sent, total, percentage)`
- Create directory: `os.MkdirAll("./received", 0755)`

Expected Output:

Server:

```
File Transfer Server listening on :8081
Client connected
```

```
Receiving file: document.txt (1024 bytes)
File saved successfully to ./received/document.txt
```

Client:

```
Sending file: document.txt
Connected to server
Sent: 1024/1024 bytes (100.00%)
✓ File transferred successfully!
```

Part 2: UDP Basics - Message Broadcasting (In class practical)

📖 Sample Code: Understanding UDP

UDP (User Datagram Protocol) is connectionless and doesn't guarantee delivery or order.

Create `udp_server.go`:

```go
package main

import (
    "fmt"
    "net"
)

func main() {
    // Listen on UDP port 8082
    addr, err := net.ResolveUDPAddr("udp", ":8082")
    if err != nil {
        fmt.Println("Error resolving address:", err)
        return
    }

    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        fmt.Println("Error listening:", err)
        return
    }
    defer conn.Close()

    fmt.Println("UDP Server listening on :8082")

    buffer := make([]byte, 1024)

    for {
        // Read from UDP
        n, clientAddr, err := conn.ReadFromUDP(buffer)
        if err != nil {
```

```
            fmt.Println("Error reading:", err)
            continue
        }

        message := string(buffer[:n])
        fmt.Printf("Received from %s: %s\n", clientAddr, message)

        // Echo back
        response := fmt.Sprintf("ACK: %s", message)
        conn.WriteToUDP([]byte(response), clientAddr)
    }
}
```

Create `udp_client.go`:

```
package main

import (
    "fmt"
    "net"
    "time"
)

func main() {
    // Resolve server address
    serverAddr, err := net.ResolveUDPAddr("udp", "localhost:8082")
    if err != nil {
        fmt.Println("Error resolving address:", err)
        return
    }

    // Create UDP connection
    conn, err := net.DialUDP("udp", nil, serverAddr)
    if err != nil {
        fmt.Println("Error connecting:", err)
        return
    }
    defer conn.Close()

    // Send messages
    messages := []string{"Hello", "UDP", "World"}

    for _, msg := range messages {
        // Send message
        _, err := conn.Write([]byte(msg))
        if err != nil {
            fmt.Println("Error sending:", err)
            continue
        }
```

```
        fmt.Printf("Sent: %s\n", msg)

        // Receive response
        buffer := make([]byte, 1024)
        conn.SetReadDeadline(time.Now().Add(2 * time.Second))

        n, err := conn.Read(buffer)
        if err != nil {
            fmt.Println("Error receiving:", err)
            continue
        }

        fmt.Printf("Received: %s\n\n", string(buffer[:n]))
        time.Sleep(500 * time.Millisecond)
    }
}
```

Expected Output:

Server:

```
UDP Server listening on :8082
Received from 127.0.0.1:54321: Hello
Received from 127.0.0.1:54321: UDP
Received from 127.0.0.1:54321: World
```

Client:

```
Sent: Hello
Received: ACK: Hello

Sent: UDP
Received: ACK: UDP

Sent: World
Received: ACK: World
```

✍️ Your Task 2.1: UDP Ping Monitor

🎯 Real-World Application

Network monitoring tools like Pingdom, Datadog, and New Relic help companies track server uptime and performance 24/7. When a website goes down, these tools alert the team immediately. In this task, you'll build your own monitoring system using UDP - perfect for checking server status quickly without the overhead of TCP. This is how DevOps engineers keep services running smoothly!

Challenge: Create a network monitoring tool that pings multiple servers using UDP.

Instructions:

1. **Server Function** `pingServer(port int)`:
    - Listen on specified UDP port
    - Respond to ping requests with timestamp
    - Format: `PONG <server_id> <timestamp>`
2. **Client Function** `pingMonitor(servers []string)`:
    - Send ping to multiple servers concurrently
    - Measure round-trip time (RTT)
    - Display results with status (online/timeout)
    - Calculate average RTT
    - Run continuously every 2 seconds

Starter Code:

```go
package main

import (
    "fmt"
    "net"
    "time"
)

type PingResult struct {
    ServerAddr string
    RTT        time.Duration
    Success    bool
}

func pingServer(port int, serverID string) {
    // TODO: Create UDP listener
    // TODO: Wait for ping requests
    // TODO: Respond with PONG message
}

func pingOnce(serverAddr string, timeout time.Duration) PingResult {
    // TODO: Send PING to server
    // TODO: Measure time
    // TODO: Wait for PONG response
    // TODO: Calculate RTT
    // TODO: Return result
```

```
}

func pingMonitor(servers []string) {
    // TODO: Ping all servers concurrently
    // TODO: Collect results
    // TODO: Display status table
    // TODO: Calculate statistics
}

func main() {
    // TODO: Start servers in goroutines
    // TODO: Run monitor
}
```

Hints: - Use `time.Now()` to measure RTT

- Set read deadline: `conn.SetReadDeadline(time.Now().Add(timeout))`
- Ping message: `"PING"`
- Pong message: `fmt.Sprintf("PONG %s %d", serverID, time.Now().Unix())`

Expected Output:

```
=== UDP Ping Monitor ===
Starting ping servers on ports: 9001, 9002, 9003

Pinging servers...

Server              Status    RTT
-----------------------------------------
localhost:9001      ✓ Online  1.2ms
localhost:9002      ✓ Online  0.8ms
localhost:9003      ✗ Timeout -

Average RTT: 1.0ms
Success Rate: 66.67% (2/3)

Pinging servers...
...
```

✍️ Your Task 2.2: UDP Discovery Service

🎯 Real-World Application

Ever noticed how your phone automatically finds printers, smart TVs, or Chromecast devices without you entering any IP addresses? That's service discovery in action!

Technologies like Apple's Bonjour, Spotify Connect, and smart home devices use UDP broadcasting to announce their presence on the network. In this task, you'll build your own service discovery system - the same technology that makes "just works" magic happen!

Challenge: Create a service discovery system using UDP broadcast.

Instructions:

1. **Server Function** `discoveryServer(serviceName string, port int)`:
    - Listen for discovery requests on port 8083
    - Respond with service information (name, address, port)
    - Format: `SERVICE:<name>:<address>:<port>`
2. **Client Function** `discoverServices()`:
    - Broadcast discovery request to network
    - Collect responses from all available services
    - Display found services in a table
    - Timeout after 3 seconds

Starter Code:

```go
package main

import (
    "fmt"
    "net"
    "strings"
    "time"
)

type ServiceInfo struct {
    Name    string
    Address string
    Port    int
}

func discoveryServer(serviceName string, servicePort int) {
    // TODO: Listen on discovery port (8083)
    // TODO: Wait for "DISCOVER" messages
    // TODO: Respond with service information
}

func discoverServices() []ServiceInfo {
    // TODO: Create UDP connection
    // TODO: Send broadcast message "DISCOVER"
```

```
    // TODO: Collect responses for 3 seconds
    // TODO: Parse service information
    // TODO: Return list of services
}

func main() {
    // TODO: Start multiple service servers
    // TODO: Wait a bit for servers to start
    // TODO: Run discovery
    // TODO: Display results
}
```

Hints: - Broadcast address: `255.255.255.255:8083`

- Enable broadcast: `conn.(*net.UDPConn).SetWriteBuffer(1024)`
- Discovery message: `"DISCOVER"`
- Parse response: `strings.Split(response, ":")`

Expected Output:

```
=== Service Discovery ===

Starting services:
- Database Service on port 5432

- Web Service on port 8080

- API Service on port 3000

Discovering services...

Found 3 services:

Service Name       Address          Port
-------------------------------------------
Database Service   192.168.1.100    5432
Web Service        192.168.1.100    8080
API Service        192.168.1.100    3000

Discovery complete!
```

Part 3: TCP vs UDP Comparison (Homework)

📖 Concept: When to Use TCP vs UDP

TCP (Transmission Control Protocol): - ✅ Reliable delivery

- ✅ Ordered packets
- ✅ Error checking
- ✅ Connection-oriented
- ❌ Slower
- ❌ More overhead

Use cases: File transfer, web browsing, email, chat applications

UDP (User Datagram Protocol): - ✅ Fast

- ✅ Low overhead
- ✅ Connectionless
- ❌ No delivery guarantee
- ❌ No ordering
- ❌ No error checking

Use cases: Video streaming, gaming, DNS, voice calls

✍️ Your Task 3.1: Performance Comparison Tool

🎯 Real-World Application

When Netflix streams video or Zoom handles video calls, engineers must decide: TCP or UDP? This choice affects millions of users' experience daily. Gaming companies like Riot Games (League of Legends) use UDP for fast action updates, while banking apps use TCP for reliable transactions. In this task, you'll build a benchmarking tool to measure and compare both protocols - giving you the data to make informed architectural decisions just like senior engineers at tech companies!

Challenge: Create a tool that compares TCP and UDP performance for data transfer.

Instructions:

1. **Function** `tcpTransferTest(dataSize int) time.Duration`:
    - Transfer data of specified size via TCP
    - Measure total time taken
    - Return duration
2. **Function** `udpTransferTest(dataSize int) time.Duration`:
    - Transfer data of specified size via UDP
    - Measure total time taken
    - Return duration
3. **Function** `runPerformanceTest()`:
    - Test with different data sizes: 1KB, 10KB, 100KB, 1MB
    - Run multiple iterations (5 times each)
    - Calculate average times
    - Display comparison table
    - Show speed difference percentage

Starter Code:

```
package main

import (
    "fmt"
    "net"
    "time"
)

func tcpServer(port string, dataSize int) {
    // TODO: Start TCP server
    // TODO: Accept connection
    // TODO: Receive data
    // TODO: Send acknowledgment
}

func tcpTransferTest(serverAddr string, dataSize int) time.Duration {
    // TODO: Connect to TCP server
    // TODO: Send data
    // TODO: Measure time
    // TODO: Return duration
}

func udpServer(port string, dataSize int) {
```

```
    // TODO: Start UDP server
    // TODO: Receive packets
    // TODO: Send acknowledgment
}

func udpTransferTest(serverAddr string, dataSize int) time.Duration {
    // TODO: Connect to UDP server
    // TODO: Send data
    // TODO: Measure time
    // TODO: Return duration
}

func runPerformanceTest() {
    // TODO: Test different data sizes
    // TODO: Run multiple iterations
    // TODO: Calculate averages
    // TODO: Display results
}

func main() {
    runPerformanceTest()
}
```

Expected Output:

```
=== TCP vs UDP Performance Comparison ===

Testing data transfer speeds...

Data Size | TCP Avg Time | UDP Avg Time | UDP Faster By
---------|--------------|--------------|---------------
1 KB      | 2.5 ms       | 0.8 ms       | 68%
10 KB     | 5.2 ms       | 1.5 ms       | 71%
100 KB    | 45 ms        | 12 ms        | 73%
1 MB      | 420 ms       | 110 ms       | 74%

Summary:
- UDP is consistently faster than TCP
- Average speed improvement: 71.5%
- TCP provides reliability, UDP provides speed
```

Part 4: Real-World Application - Multi-Protocol Chat (Homework)

📖 Case Study: Hybrid Chat System

Scenario: - Support both TCP and UDP clients

- TCP for reliable message delivery
- UDP for presence/heartbeat updates
- Server manages both protocols simultaneously

Features: - TCP: Message delivery with history

- UDP: Real-time status updates ("User is typing...")
- Message persistence
- Connection status monitoring

✍️ Your Task 4.1: Hybrid Chat Server (30 min)

🎯 Real-World Application

Modern messaging apps don't just use one protocol - they use the best tool for each job! WhatsApp uses TCP for sending messages reliably, but uses UDP for real-time "typing..." indicators and voice calls. LinkedIn uses TCP for posts but UDP for "online now" status updates. In this task, you'll build a hybrid system that combines both protocols - exactly how production messaging systems are architected at companies like Facebook, Microsoft Teams, and Signal!

Challenge: Build a chat server that uses both TCP for messages and UDP for status updates.

Instructions:

1. **Server Components:**
    - TCP server on port 9000 (messages)
    - UDP server on port 9001 (status updates)
    - Message history storage
    - Client status tracking
2. **Message Types:**
    - TCP Messages: `MSG:<username>:<message>`
    - UDP Status: `STATUS:<username>:<typing|online|away>`
    - History Request: `HISTORY:<count>`
3. **Features:**

- Store last 100 messages
- Broadcast messages to all TCP clients
- Update status for all clients on UDP
- Handle client disconnections

Starter Code:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "sync"
    "time"
)

type Message struct {
    Username  string
    Content   string
    Timestamp time.Time
}

type UserStatus struct {
    Username string
    Status   string // "online", "typing", "away"
    LastSeen time.Time
}

type HybridChatServer struct {
    tcpClients    map[net.Conn]string
    messageHistory []Message
    userStatuses  map[string]*UserStatus
    mu            sync.RWMutex
}

func NewHybridChatServer() *HybridChatServer {
    return &HybridChatServer{
        tcpClients:     make(map[net.Conn]string),
        messageHistory: make([]Message, 0, 100),
        userStatuses:   make(map[string]*UserStatus),
    }
}

func (s *HybridChatServer) addMessage(username, content string) {
    // TODO: Add message to history (max 100)
```

```go
}

func (s *HybridChatServer) updateStatus(username, status string) {
    // TODO: Update user status
}

func (s *HybridChatServer) broadcastMessage(msg Message) {
    // TODO: Send message to all TCP clients
}

func (s *HybridChatServer) startTCPServer() {
    // TODO: Start TCP server for messages
    // TODO: Handle connections
    // TODO: Process messages
}

func (s *HybridChatServer) startUDPServer() {
    // TODO: Start UDP server for status updates
    // TODO: Receive status updates
    // TODO: Broadcast status to all clients
}

func main() {
    server := NewHybridChatServer()

    // Start both servers
    go server.startTCPServer()
    go server.startUDPServer()

    // Keep running
    select {}
}
```

Expected Output:

Server:

```
=== Hybrid Chat Server ===
TCP Server listening on :9000
UDP Server listening on :9001

Alice connected via TCP
Bob connected via TCP

[TCP] Alice: Hello everyone!
[UDP] Bob: typing
[TCP] Bob: Hi Alice!
[UDP] Bob: online
[UDP] Alice: away
```

```
Alice disconnected
```

Client Output:

```
Connected to chat server (TCP: 9000, UDP: 9001)
Enter username: Alice
*** Joined chat room ***

[10:30:15] Alice: Hello everyone!
[10:30:18] Bob: Hi Alice!

Status Update: Bob is typing...
Status Update: Bob is online
```

Hints: - Run TCP and UDP servers in separate goroutines

- Use channels to coordinate between servers
- Store messages in a circular buffer
- Heartbeat: Send status every 5 seconds
- Format timestamp: `time.Now().Format("15:04:05")`

Part 5: Network Protocol Design (Optional Bonus)

✍️ Your Task 5.1: Custom Protocol Implementation

🎯 Real-World Application

Every major online game - Fortnite, PUBG, League of Legends, Valorant - uses custom network protocols optimized for their specific needs. These aren't just TCP or UDP, but carefully designed application-layer protocols that minimize latency and bandwidth while maximizing performance. Game companies pay top dollar for network engineers who can design efficient protocols. In this task, you'll design your own game protocol from scratch - giving you a glimpse into the work of specialized network engineers at gaming companies and high-frequency trading firms!

Challenge: Design and implement a custom application-layer protocol for a multiplayer game server.

Protocol Specification:

Message Format:

```
[TYPE][LENGTH][PAYLOAD]
- TYPE: 1 byte (message type)
```

- LENGTH: 2 bytes (payload length)

- PAYLOAD: variable length

Message Types: - 0x01: JOIN (username)

- 0x02: MOVE (x, y coordinates)
- 0x03: CHAT (message)
- 0x04: LEAVE
- 0x05: PLAYER_LIST
- 0x06: GAME_STATE

Instructions:

1. **Server** gameServer():
    - TCP server on port 9999
    - Track player positions
    - Broadcast game state every 100ms
    - Handle player movements
    - Process chat messages
2. **Client** gameClient(username string):
    - Connect to game server
    - Send JOIN message
    - Send random MOVE commands
    - Display game state
    - Handle user chat input
3. **Protocol Functions:**
    - encodeMessage(msgType byte, payload []byte) []byte
    - decodeMessage(data []byte) (msgType byte, payload []byte, error)

Starter Code:

```
package main

import (
```

```
    "encoding/binary"
    "fmt"
    "net"
)

const (
    MSG_JOIN        byte = 0x01
    MSG_MOVE        byte = 0x02
    MSG_CHAT        byte = 0x03
    MSG_LEAVE       byte = 0x04
    MSG_PLAYER_LIST byte = 0x05
    MSG_GAME_STATE  byte = 0x06
)

type Player struct {
    Username string
    X, Y     int
    Conn     net.Conn
}

type GameServer struct {
    players map[string]*Player
    // TODO: Add necessary fields
}

func encodeMessage(msgType byte, payload []byte) []byte {
    // TODO: Encode message according to protocol
    // [TYPE][LENGTH][PAYLOAD]
}

func decodeMessage(data []byte) (byte, []byte, error) {
    // TODO: Decode message
    // Return msgType, payload, error
}

func (gs *GameServer) handlePlayer(conn net.Conn) {
    // TODO: Handle player connection
    // TODO: Process messages
    // TODO: Broadcast updates
}

func (gs *GameServer) broadcastGameState() {
    // TODO: Send game state to all players
}

func main() {
    // TODO: Start game server
}
```

Expected Output:

```
=== Game Server ===
Server listening on :9999

Player joined: Alice at (0, 0)
Player joined: Bob at (0, 0)

Game State Update:
- Alice: (5, 3)
- Bob: (2, 7)

[Chat] Alice: Hi everyone!
[Chat] Bob: Hello!

Player moved: Alice -> (8, 3)

Player left: Bob
```

Submission Requirements (100 points)

Tasks (90 points)

- ☐ Task 1.1: TCP Chat Room Server (15 points)
- ☐ Task 1.2: TCP File Transfer (15 points)
- ☐ Task 2.1: UDP Ping Monitor (15 points)
- ☐ Task 2.2: UDP Discovery Service (15 points)
- ☐ Task 3.1: Performance Comparison Tool (15 points)
- ☐ Task 4.1: Hybrid Chat Server (15 points) - **REQUIRED**

Code Quality (10 points)

- ☐ Code compiles and runs (4 points)
- ☐ Uses TCP/UDP correctly (2 points)
- ☐ Proper error handling (2 points)
- ☐ Clean, readable code (2 points)

Bonus (+10 points)

- ☐ Task 5.1: Custom Protocol Implementation (10 points)

Testing Checklist

Before submitting, make sure:

1. ✓ Server starts without errors
2. ✓ Client can connect to server
3. ✓ Messages are sent and received correctly
4. ✓ Multiple clients can connect simultaneously
5. ✓ Disconnections are handled gracefully
6. ✓ No goroutine leaks (use proper cleanup)
7. ✓ Timeouts work correctly

Common Mistakes to Avoid

❌ Mistake 1: Not Closing Connections

```
// Wrong - connection leak
conn, _ := net.Dial("tcp", "localhost:8080")
// ... do something ...
// Forgot to close!
```

☑ Solution: Always use defer

```
conn, err := net.Dial("tcp", "localhost:8080")
if err != nil {
    return
}
defer conn.Close() // Always clean up
```

❌ Mistake 2: Blocking Read Without Timeout

```
// Wrong - can hang forever
buffer := make([]byte, 1024)
conn.Read(buffer) // Blocks indefinitely
```

☑ Solution: Set read deadline

```
conn.SetReadDeadline(time.Now().Add(5 * time.Second))
n, err := conn.Read(buffer)
if err != nil {
    // Handle timeout
}
```

❌ Mistake 3: Not Handling Partial Reads

```
// Wrong - might not read all data
buffer := make([]byte, 1024)
n, _ := conn.Read(buffer)
data := buffer[:n]
```

☑ Solution: Use io.ReadFull or loop

```
buffer := make([]byte, expectedSize)
_, err := io.ReadFull(conn, buffer)
if err != nil {
    // Handle error
}
```

✗ Mistake 4: UDP Message Size

```
// Wrong - UDP packet might be truncated
message := make([]byte, 100000) // Too large!
conn.WriteToUDP(message, addr)
```

☑ Solution: Keep UDP messages small

```
// Maximum safe UDP payload size
const MaxUDPSize = 508 // bytes
message := make([]byte, MaxUDPSize)
conn.WriteToUDP(message, addr)
```

Quick Reference

TCP Operations

```
// Server
listener, _ := net.Listen("tcp", ":8080")
conn, _ := listener.Accept()
conn.Write([]byte("hello"))
buffer := make([]byte, 1024)
n, _ := conn.Read(buffer)
conn.Close()

// Client
conn, _ := net.Dial("tcp", "localhost:8080")
conn.Write([]byte("hello"))
buffer := make([]byte, 1024)
n, _ := conn.Read(buffer)
conn.Close()
```

UDP Operations

```
// Server
addr, _ := net.ResolveUDPAddr("udp", ":8082")
conn, _ := net.ListenUDP("udp", addr)
buffer := make([]byte, 1024)
n, clientAddr, _ := conn.ReadFromUDP(buffer)
conn.WriteToUDP([]byte("response"), clientAddr)

// Client
```

```
serverAddr, _ := net.ResolveUDPAddr("udp", "localhost:8082")
conn, _ := net.DialUDP("udp", nil, serverAddr)
conn.Write([]byte("message"))
buffer := make([]byte, 1024)
n, _ := conn.Read(buffer)
```

Error Handling

```
conn, err := net.Dial("tcp", "localhost:8080")
if err != nil {
    fmt.Println("Error:", err)
    return
}
defer conn.Close()

// Check for timeout
if netErr, ok := err.(net.Error); ok && netErr.Timeout() {
    fmt.Println("Operation timed out")
}
```

Concurrent Server Pattern

```
listener, _ := net.Listen("tcp", ":8080")
for {
    conn, err := listener.Accept()
    if err != nil {
        continue
    }
    go handleConnection(conn) // Handle in goroutine
}
```

Getting Help

During Lab: - 🤖 Ask AI: “How do TCP connections work in Go?”

- 👥 Ask classmates
- 👨‍🏫 Ask instructor

Useful Prompts: - “Explain TCP vs UDP”

- “Go network programming example”
- “How to handle multiple TCP clients”
- “UDP broadcasting in Go”

Protocol Comparison Cheatsheet

Feature	TCP	UDP
Connection	Yes	No
Reliability	Guaranteed	Best effort
Ordering	Yes	No
Speed	Slower	Faster
Overhead	Higher	Lower
Use Case	Chat, Files	Gaming, Video

Congratulations!

You've learned Go network programming basics!

Remember: - Use **TCP** when you need reliability - Use **UDP** when you need speed - Always handle errors and clean up connections - Test with multiple clients to find race conditions

Next Steps: - Explore WebSockets - Learn about HTTP/2 - Study network security (TLS) - Build a real-time application

---

End of Lab 03