Lab 12: Implementing Transport Services

Assessments	@October 27, 2025 → October 28, 2025
Dates	@October 27, 2025 → October 28, 2025
Topics	Implementing Transport Services
Instructor	Kanthanet (Nam)

Go Programming Lab: Implementing Transport Services (2 Hours)

This lab focuses on implementing and observing the core transport layer services (**Flow Control, Congestion Control, and Error Control**) by comparing a custom, simplified **Reliable TCP-like** implementation in Go with the native **Unreliable UDP** protocol.

Part 1: Setup and Basic UDP Communication (30 Minutes)

Objective

Establish a simple client-server UDP connection to understand its non-existent transport services.

Instructions

1. **Project Setup:** Create a new Go module (`go mod init transport-lab`).
2. **UDP Server (`server.go`):**Go
   - Create a UDP listener on a specific port (e.g., `:8080`).
   - Use a loop to continuously read incoming messages (`conn.ReadFromUDP`).
   - Print the received message and its source address.

```
// server.go
package main
import (
    "fmt"
    "net"
    "time"
)
func main() {
    // Listen on UDP port 8080
    addr, _ := net.ResolveUDPAddr("udp", ":8080")
    conn, _ := net.ListenUDP("udp", addr)
    defer conn.Close()
    fmt.Println("UDP Server listening on :8080")

    buffer := make([]byte, 1024)
    for {
```

```
        n, remoteAddr, _ := conn.ReadFromUDP(buffer)
        fmt.Printf("Received: %s from %s\n", string(buffer[:n]), remoteAddr.String())
    }
}
```

3. **UDP Client (`client.go`):**Go
    - Connect to the server.
    - Send 10 short, sequential messages in a fast loop (e.g., "Message 1", "Message 2", etc.).
    - **Observe:** Run the client and server simultaneously. Note how quickly the messages arrive and whether any are lost if you artificially flood the network (e.g., by sending 100 messages quickly).

```
// client.go (Snippet - use a loop to send 10 sequential messages)
// ...
conn, _ := net.Dial("udp", "127.0.0.1:8080")
for i := 1; i <= 10; i++ {
    message := fmt.Sprintf("UDP Segment #%d", i)
    conn.Write([]byte(message))
    time.Sleep(10 * time.Millisecond) // Slow down slightly to observe
}
// ...
```

Part 2: Implementing Basic Error Control (30 Minutes)

Objective

Add a simplified **Error Control** mechanism (Acknowledgment and Retransmission) to the client-server logic, moving closer to TCP.

Instructions

1. **Server Modification (Error Control):**
    - After receiving a message, the server must **immediately send an acknowledgment (ACK)** back to the client.

```
// server.go (Modification)
// ... inside the loop after receiving data ...
n, remoteAddr, _ := conn.ReadFromUDP(buffer)
// Simulate processing delay or potential loss
if rand.Intn(10) < 2 { // 20% chance to 'lose' the ACK
    fmt.Printf("ERROR: Dropping ACK for %s\n", string(buffer[:n]))
} else {
```

```
    ack := []byte("ACK:" + string(buffer[:n]))
    conn.WriteToUDP(ack, remoteAddr) // Send ACK back
}
// ...
```

2. **Client Modification (Error Control):**
    - After sending a message, the client must **wait for the ACK** (set a small timeout).
    - If the timeout is reached, the client must **retransmit** the message.

```
// client.go (Modification - inside the loop)
// ...
conn.Write([]byte(message))

// Set a read deadline (Timeout)
conn.SetReadDeadline(time.Now().Add(50 * time.Millisecond))

ackBuffer := make([]byte, 1024)
_, _, err := conn.ReadFromUDP(ackBuffer)

if err != nil {
    fmt.Printf("TIMEOUT: Retransmitting %s\n", message)
    conn.Write([]byte(message)) // Simple retransmission
} else {
    fmt.Printf("ACK received for %s\n", message)
}
// ...
```

3. **Observation:** Run the modified client and server. Notice the increased time for delivery due to waiting for ACKs, but also observe the automatic retransmission when the server "drops" an ACK. This demonstrates **reliability** via **Error Control**.

Part 3: Implementing Flow and Congestion Control Concepts (40 Minutes)

Objective

Introduce simplified **Flow Control** (Receiver Window) and **Congestion Control** (Simulated Slow Start) into the client.

Instructions

1. **Flow Control (Client-side implementation):**
    - The client maintains a variable, `receiverWindow` (start it at 4). This simulates the server's available buffer space.
    - The client can only send a maximum of `receiverWindow` unacknowledged segments.
    - After sending, the client must wait for ACKs before its available window increases.

2. **Congestion Control (Simulated Slow Start):**
    - The client maintains a `congestionWindow` variable (start it at 1).
    - The client can only send `min(receiverWindow, congestionWindow)` segments.
    - **Slow Start:** For every successful ACK received, the `congestionWindow` increases by 1 (or a small amount), simulating an exponential increase until a drop/timeout occurs.

```
// client.go (Advanced Modification)
// ... initialization ...
receiverWindow := 4 // Simulating the server's buffer capacity
congestionWindow := 1 // Start of Slow Start
// ...

// Inside the sending loop (simplified logic)
for segmentsToSend := 1; segmentsToSend <= congestionWindow && segmentsToSend <=
receiverWindow; segmentsToSend++ {
    // 1. Send segment
    // 2. Wait for ACK (as implemented in Part 2)

    if ackReceived {
        congestionWindow++ // Slow Start: Increase window exponentially on success
        // If the congestion window is very large, you could cap it
        // to simulate Congestion Avoidance (linear increase).
    } else {
        // Error/Timeout occurred
        congestionWindow = 1 // Simulating Fast Retransmit/Recovery: Drop window to 1
        // Retransmit logic here
    }
}
```

3. **Observation:** Observe how the client's sending rate (`congestionWindow`) starts slowly and accelerates until an error occurs (the simulated ACK drop), after which it resets to 1. This demonstrates a core principle of **TCP's Congestion Control**.

Part 4: Analysis and Comparison (20 Minutes)

Discussion & Documentation

1. **Protocol Comparison:**
    - In your implementation, what specific line of code or logic provided **Error Control**?
    - How did your simplified **Flow Control** prevent the client from overwhelming the receiver's buffer?
    - What was the core difference in behavior (speed vs. reliability) between the native UDP implementation (Part 1) and your custom TCP-like implementation (Parts 2 & 3)?

2. **TSAPs (Ports):** Briefly explain where the concept of the **Transport Service Access Point (TSAP)** or **Port** is still essential in your Go programs, even though you used a simpler `Dial` function. (Hint: The address `127.0.0.1:8080` contains the port.)