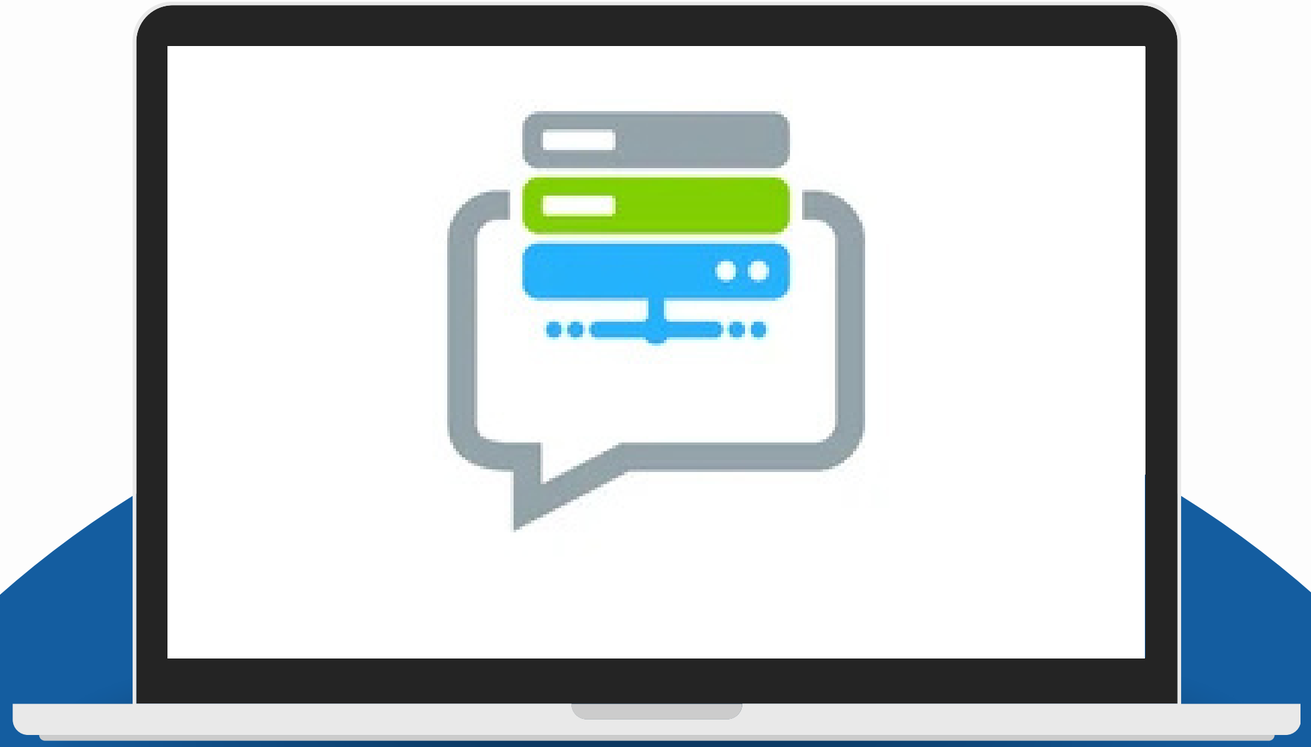
Systems Programming &
Computer Organization

TCP vs UDP Chat App in Go



Final Project Presentation By:
Kyler Brown & Mikaylie Jonch

Overview

- Goals and Purpose 01
- Architecture & Protocol Design 02
- Code Highlights & Design Choices 03
- Performance Comparison 04
- Challengess & Insights 05
- Summary & Recommendations 06

Goals and Purpose

Objective: Build a real-time, concurrent chat system using UDP and TCP in Go.

Functional Goals:

- Clients can connect and exchange messages
- Messages are broadcast to all active clients
- Graceful disconnection of clients
- Concurrent message handling using goroutines

Architecture and Protocol Design

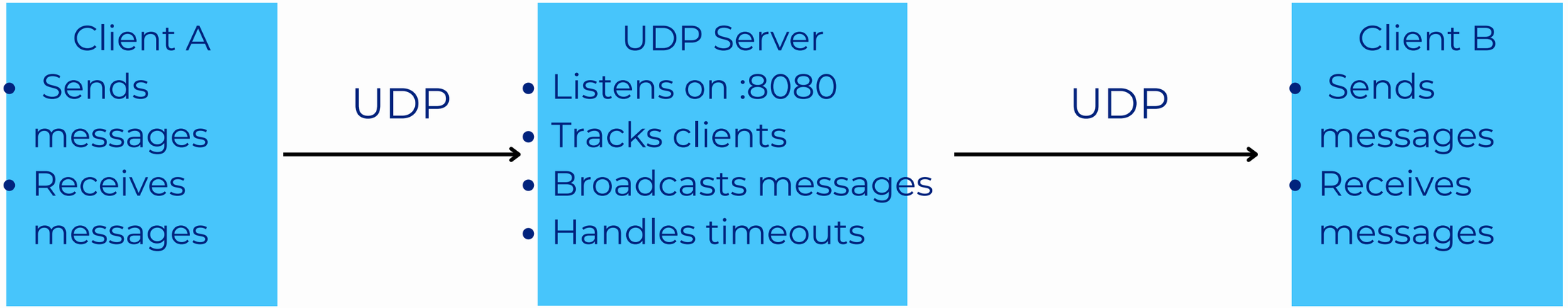
Server Responsibilities:

- Receive and broadcast messages.
- Track active clients by IP:Port.
- Monitor client activity for timeout/disconnect.

Client Responsibilities:

- Send user messages to server.
- Listen for and display broadcasted messages.

UDP Design Overview



Code Highlights and Design Choices

Concurrency:

- Server handles all I/O in a single thread (UDP is non-blocking by nature)
- Goroutines handle client timeouts and incoming messages

Modularity:

- handleMessage(), broadcast(), and monitorTimeouts() separate logic

Client Experience:

- Simple text interface
- ":ping" command for latency check

Performance Comparison

Test	**Average Latency (ms)**	**Message Loss**	**Throughput (msgs/sec)**
Low Load	1.2	0%	~800
High Load	4.5	1-2%	~2000
Simulated	45	5-10%	~500

Challenges & Insights

UDP Limitation:
No built-in delivery guarantee → requires simplicity over reliability

Client Tracking:
Required custom timeout logic and active ping monitoring

Testing:
Required manual load simulation and ping RTT analysis

Summary & Recommendations

- UDP chat server is lightweight, scalable, and easy to implement
- Best for local or trusted networks; not ideal for critical communication
- Golang’s net and goroutine model simplify concurrent network programming

Architecture and Protocol Design

TCP server

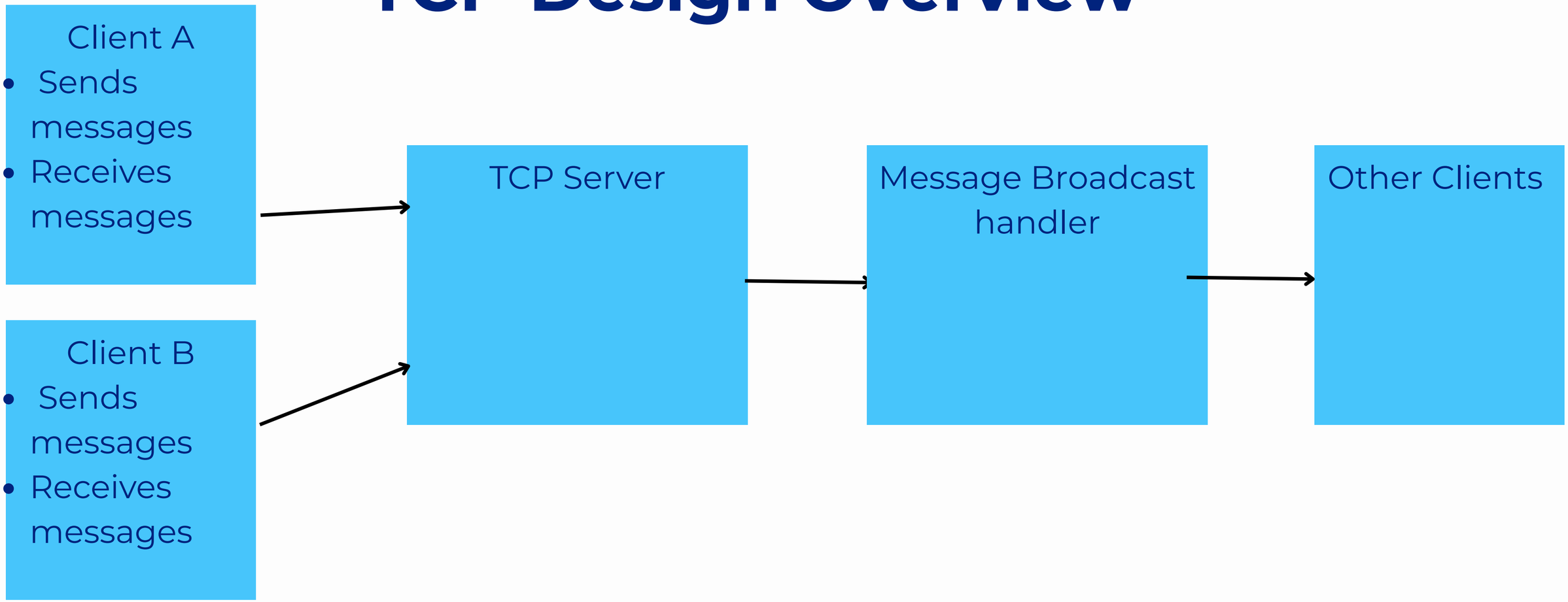
TCP Server Architecture:

- Clients: Multiple clients can connect to the server, sending and receiving messages.
- Server: The server handles connections, processes messages, and sends responses, managing multiple clients simultaneously.
- Logging: Each client’s activity is logged in a separate file for later analysis.
- Metrics: Latency, throughput, packet loss, and session duration are tracked for performance analysis.

Code Highlights and Design Choices

- Message Handling:
  - The server listens for incoming client messages and handles commands such as /time, /date, /ping, /clients, and /help.
  - Each command has a predefined response, and the server can also broadcast messages to all connected clients
- Metrics Logging:
  - Metrics such as latency, message throughput, packet loss, and session duration are tracked and logged for each client.
  - CSV Logging: Client interactions and metrics are written to CSV files for later analysis.
- Concurrency and Scalability:
  - The server uses goroutines to handle multiple clients concurrently, allowing the server to scale with increasing client numbers.
  - Timeouts: Inactivity timeouts disconnect idle clients after a specified period.

TCP Design Overview



Performance Comparison

Key Metrics to Compare:

- Latency: Time taken for a message to travel from the client to the server and back.
- Throughput: Number of messages processed per second by the server.
- Packet Loss: Percentage of sent messages that were lost, particularly important for UDP.
- Session Duration: How long clients remain connected to the server.

Testing With TC

Test Setup:

- Tool: tc on Linux
- Network interface: eth0 (or relevant interface in Codespace)
- Sample conditions applied:
  - Latency: 100ms delay
  - sudo tc qdisc add dev eth0 root netem delay 100ms
  - Packet Loss: 10% loss
  - sudo tc qdisc change dev eth0 root netem loss 10%
  - Bandwidth Limit: 1Mbps
  - sudo tc qdisc change dev eth0 root tbf rate 1mbit burst 32kbit latency 400ms

Testing With TC Observation

Condition	Average Latency	Packet Loss (%)	Throughput (msg/sec)	User Impact
No Network Rules	20ms	0%	12.5	smooth and fast respones
+100ms Latency	125ms	0%	9.2	slower interactions
+10% Loss	127ms	10%	8.1	missing respones became noticable

Comparison/ final Observations

Feature	UDP Chat Server	TCP Chat Server
Protocol Type	Connectionless, unreliable	Connection-oriented, reliable
Message Delivery	No guarantees (messages may be lost or reordered)	Guaranteed delivery and in order
Overhead	Minimal header/data overhead	Higher overhead (connection handshake, stream mgmt)
Client Management	Server identifies clients by IP:Port	Each client gets a persistent `net.Conn` object
Concurrency Model	One goroutine handles all UDP reads	One goroutine per TCP client connection
Scalability	More lightweight; better under high churn	Heavier with more active connections
Latency Measurement	Must be custom-built (e.g., PING/PONG)	Can measure using round-trip on connection directly
Error Handling	Needs manual logic for timeouts/retries	Built-in connection lifecycle and error reporting
Broadcasting Messages	Manual write to each client's IP:Port	Write directly to each client's connection
Best Use Case	Fast LAN chat, games, IoT broadcasts	Public chat apps, file transfers, critical data

THANK YOU!