Reflection on Network Programming: Crypt Battle Arena Implementation

Zeynep Çağıl
Student Number: 210316020
Course: Network Programming
Manisa Celal Bayar University, Manisa, Türkiye

ABSTRACT

This paper reflects on the development of Crypt Battle Arena, a multiplayer networked game. A hybrid TCP/UDP approach was implemented to balance reliability and responsiveness. Key challenges included real-time synchronization and client disconnection handling. The system employs server-authoritative design with timeout-based mechanisms. Testing confirmed robustness under various failure conditions.

Index Terms—Socket programming, TCP/UDP protocols, multiplayer systems, real-time networking

---

I. INTRODUCTION

This reflection examines the development of Crypt Battle Arena, demonstrating practical applications of network programming in real-time multiplayer environments. The project addresses synchronization, protocol selection, and fault tolerance challenges.

II. CHALLENGES AND SOLUTIONS

The main challenge was managing real-time synchronization between multiple clients. Using TCP alone caused delays during movement and combat, while UDP exclusively resulted in unreliable delivery for critical events. A hybrid approach was implemented: TCP for reliable operations (lobby management, player status, game commands) and UDP for time-sensitive gameplay (movement, projectiles, items, timers).

Client disconnection handling was challenging since UDP provides no explicit disconnect notifications. A server-side timeout mechanism was implemented, automatically removing inactive players after five seconds and reassigning host authority when necessary.

III. DIFFICULT ASPECTS OF SOCKET PROGRAMMING

Handling concurrency and shared state safely was most difficult. The server manages multiple TCP clients, UDP endpoints, and shared player data simultaneously, requiring careful synchronization to prevent race conditions. Understanding UDP behavior under packet loss was challenging—unlike TCP, UDP doesn't indicate disconnections, requiring additional logic for client availability tracking through timeout mechanisms.

IV. USE OF AI ASSISTANCE

AI-assisted tools were used as supplementary learning and debugging resources throughout the project.

ChatGPT was used to clarify conceptual differences between TCP and UDP protocols. For example, when deciding which protocol to use for player movement updates, ChatGPT helped explain the trade-offs between reliability and latency. It also reviewed multithreading logic, specifically helping to understand how to properly implement locks when multiple threads access shared player state data.

Google Gemini explored alternative architectural approaches for handling disconnections. For instance, it suggested comparing timeout-based detection versus heartbeat mechanisms, helping validate the choice of timeout approach. It also provided insights on client-server responsibility separation, clarifying which game logic should run on the server versus client.

Claude improved code readability by suggesting better variable naming conventions and helped refine documentation structure, particularly for explaining the hybrid TCP/UDP architecture in comments.

All AI-generated suggestions were carefully reviewed, manually implemented, and tested. No complete solutions were generated automatically—AI tools enhanced understanding rather than replacing independent problem-solving.

V. TCP VS UDP LEARNINGS

TCP provides reliable, ordered delivery suitable for critical game logic (lobby management, state synchronization). UDP offers low-latency communication essential for real-time movement and combat but requires additional logic like heartbeat messages and timeout detection. Choosing the correct protocol based on data characteristics is crucial for performance.

VI. FUTURE IMPROVEMENTS

Future enhancements include: advanced lag compensation (client-side prediction, server reconciliation), improved server-side validation to prevent cheating, persistent player data storage, dynamic console UI adaptation, and better load handling for concurrent clients to increase scalability.

VII. TESTING STRATEGY

Multiple simultaneous clients (minimum three) were tested on the same machine to verify real-time synchronization and lobby functionality. Testing scenarios included client disconnections during gameplay and lobby phases, server restarts, UDP packet loss tolerance, host disconnection and reassignment, and invalid connections. Tests confirmed correct system behavior under failure conditions.

VIII. CONCLUSION

This project provided hands-on experience with socket programming, multithreading, and real-time networking. Key learnings included protocol selection importance, server-authoritative design, and fault tolerance in multiplayer systems. The experience of building a complete multiplayer application strengthened understanding of network programming concepts and practical system design beyond theoretical study.