JAVA MESSAGING APPLICATION

James Hoang, Abdulmalik Jinadu, Enrico Milli Dominguez, Armand Omar
COMP1549: Advanced Programming
University of Greenwich
Old Royal Naval College
United Kingdom

Key words: *Networked Systems, Java, Sockets, Client-Server, Observer Pattern, Fault Tolerance, JUnit, Group Chat*

Abstract - This report details the design and implementation of a networked group chat system developed for the COMP1549 Advanced Programming module. The system, implemented in Java, facilitates real-time communication between multiple clients via a central server, employing a client-server architecture over TCP sockets. Key features include group formation, broadcast and private messaging, dynamic coordinator selection, and fault tolerance mechanisms to handle client and coordinator disconnections. The implementation leverages the Observer design pattern for efficient message dissemination and JUnit for unit testing key components.

I. Introduction

The ability for distributed systems to communicate reliably and efficiently is fundamental in modern computing. This project addresses the challenge of creating a robust group-based communication system, a common requirement in applications ranging from instant messaging platforms to collaborative work tools and distributed control systems. The motivation was to explore and implement core concepts in networked and distributed systems, including client-server interaction, state management across multiple threads, dynamic role assignment (coordination), and resilience to network or component failures.

The rationale behind this specific implementation was to build a practical Java application using TCP sockets for reliable, connection-oriented communication. The focus was on creating a system where members could join a group, exchange messages publicly or privately, and where the system could gracefully handle members leaving, including the designated coordinator. Measuring the success by ensuring correct message delivery, proper state maintenance (like the active member list), and seamless coordinator transition.

II. Design/implementation

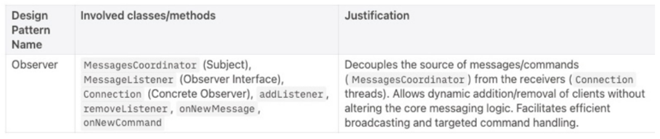
The system employs a client-server architecture. The central “Server” listens for incoming client connections using Java's “ServerSocket”. Upon connection, it spawns a dedicated “Connection” thread for each client, managing communication via “Socket”, “PrintWriter”, and “Scanner”. This thread-per-client model allows concurrent handling of multiple users. Communication relies on TCP, chosen for its inherent reliability and guaranteed message ordering, crucial for a chat application.

The “Server” manages group state through two essential components. The “UserListMap” is a “ConcurrentHashMap” that stores connected user information including ID, IP, Port, and Coordinator status. This structure ensures thread-safe access and offers methods for adding, removing, and querying user details while preventing duplicate logins by verifying username uniqueness. Complementing this, the “MessagesCoordinator” functions as a central hub for message broadcasting and command routing. It implements the Observer pattern by maintaining a collection of “MessageListener” objects, with each object being a “Connection” instance, enabling efficient communication throughout the system.

When a client connects (“Client.java”), it establishes a socket connection and provides configuration (IP, Port, ID) using a “FlagHandler” for command-line arguments. If no ID is provided, a UUID is generated. The client first checks username availability with the server. On success, a “MessagesHandler” thread starts on the client-side to listen for incoming messages from the server, whilst the main client thread handles user input. A “MessagePrinter” class manages the CLI display, ensuring incoming messages appear above the user's current input line.

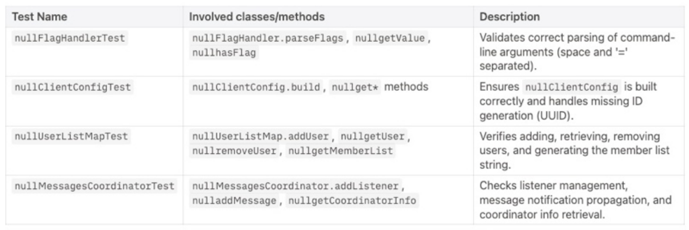
The system incorporates two fundamental features that handle the requirements. The Coordinator Role is automatically assigned to the first client connecting to the server through the "Connection.makeCoordinator" method. This coordinator, instantiated as a dedicated class by the coordinator's "Connection" thread, manages tasks including handling "/list" or "/members" requests by querying the "UserListMap", and routing private "/message" commands between users. Additionally, it conducts periodic checks every 20 seconds using a "TimerTask" to identify and manage unresponsive clients, with all users being informed about the current coordinator upon joining the network. The messaging functionality is implemented using prefixes. Messages without command prefixes are treated as broadcasts, with the client's "Connection" thread forwarding them to "MessagesCoordinator.addMessage", which then notifies all registered listeners via "onNewMessage". Private messages (formatted as "/message targetUser message") are sent to the "MessagesCoordinator" through "addCommand", which forwards them to the active "Coordinator". The coordinator then retrieves the target user's details from "UserListMap" and uses "MessagesCoordinator.sendMsgToIp" to deliver the message directly to the recipient's "Connection" thread. Throughout this process, various prefixes such as "MSG-FROM-CHAT", "MSG-FROM-COORDINATOR", and user/client identifiers are employed to clearly distinguish between different message types and origins in client-server communications.

Design Patterns



Design Pattern Name	Involved classes/methods	Justification
Observer	`MessagesCoordinator` (Subject), `MessageListener` (Observer Interface), `Connection` (Concrete Observer), `addListener`, `removeListener`, `onNewMessage`, `onNewCommand`	Decouples the source of messages/commands (`MessagesCoordinator`) from the receivers (`Connection` threads). Allows dynamic addition/removal of clients without altering the core messaging logic. Facilitates efficient broadcasting and targeted command handling.

Tests



Test Name	Involved classes/methods	Description
`nullFlagHandlerTest`	`nullFlagHandler.parseFlags`, `nullgetValue`, `nullhasFlag`	Validates correct parsing of command-line arguments (space and '=' separated).
`nullClientConfigTest`	`nullClientConfig.build`, `nullget*` methods	Ensures `nullClientConfig` is built correctly and handles missing ID generation (UUID).
`nullUserListMapTest`	`nullUserListMap.addUser`, `nullgetUser`, `nullremoveUser`, `nullgetMemberList`	Verifies adding, retrieving, removing users, and generating the member list string.
`nullMessagesCoordinatorTest`	`nullMessagesCoordinator.addListener`, `nulladdMessage`, `nullgetCoordinatorInfo`	Checks listener management, message notification propagation, and coordinator info retrieval.

III. Analysis and Critical Discussion

This implementation provides a platform for group communication with coordinator management and fault tolerance. The use of Java Sockets with TCP ensures reliable message delivery, which is critical for a chat application. The separation of concerns into distinct classes ("Server", "Connection", "Coordinator", "MessagesCoordinator", "UserListMap", "Client", "ClientConfig", "FlagHandler", "MessagesHandler", "MessagePrinter") promotes modularity and maintainability.

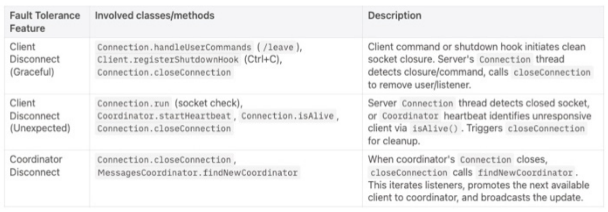
The Observer pattern, implemented via "MessagesCoordinator" and "MessageListener", proved highly effective in decoupling the message source from the numerous client connections. This makes the system extensible, as new types of listeners could potentially be added without modifying "MessagesCoordinator". The use of "ConcurrentHashMap" in "UserListMap" ensures thread safety when multiple "Connection" threads access the user list concurrently.

Testing using JUnit was integral to verifying the correctness of core components, particularly the argument parsing ("FlagHandler"), client configuration ("ClientConfig"), user management ("UserListMap"), and message coordination logic ("MessagesCoordinator"). These tests provide confidence in the reliability of these individual units.

However, certain challenges and limitations exist. The thread-per-client model, whilst simple to implement, does not scale well to a very large number of concurrent users due to the overhead associated with managing many threads. Approaches like using thread pools or Java's virtual threads could offer better scalability. Furthermore, storing the entire user list in the server's memory limits the system's capacity; a persistent storage mechanism (like a database or cache such as Redis) would be necessary for larger-scale deployments.

Lastly, fault tolerance is addressed for client departures (graceful and unexpected) and coordinator failure. However, Security is a mostly excluded as the current implementation lacks authentication beyond a simple username check and uses no encryption.

Fault Tolerance



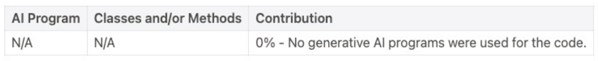
Fault Tolerance Feature	Involved classes/methods	Description
Client Disconnect (Graceful)	`Connection.handleUserCommands` (`/leave`), `Client.registerShutdownHook` (Ctrl+C), `Connection.closeConnection`	Client command or shutdown hook initiates clean socket closure. Server's `Connection` thread detects closure/command, calls `closeConnection` to remove user/listener.
Client Disconnect (Unexpected)	`Connection.run` (socket check), `Coordinator.startHeartbeat` , `Connection.isAlive` , `Connection.closeConnection`	Server `Connection` thread detects closed socket, or `Coordinator` heartbeat identifies unresponsive client via `isAlive()` . Triggers `closeConnection` for cleanup.
Coordinator Disconnect	`Connection.closeConnection` , `MessagesCoordinator.findNewCoordinator`	When coordinator's `Connection` closes, `closeConnection` calls `findNewCoordinator` . This iterates listeners, promotes the next available client to coordinator, and broadcasts the update.

IV. Conclusions

Our Java messaging application met all the required features including broadcast messaging, private messaging via a coordinator, and dynamic coordinator selection upon failure. Software engineering principles were applied, including the use of the Observer design pattern for decoupling and JUnit for unit testing, enhancing the system's modularity and reliability. Fault tolerance mechanisms were incorporated to handle both graceful and unexpected client departures, and the case of coordinator departures.

Whilst the system functions correctly according to the specification, areas for future enhancement include improving scalability beyond the thread-per-client model (e.g., using virtual threads), implementing robust security features (authentication and encryption), and developing a graphical user interface. Overall, the project provided valuable practical experience in network programming, concurrent systems, and distributed application design.

AI Usage



AI Program	Classes and/or Methods	Contribution
N/A	N/A	0% - No generative AI programs were used for the code.

References

No external references were used beyond standard Java documentation and course materials.