Distributed Chat System Technical Report

Author: Yujun Ma, Yining Sui

Note:

- **Description of the contents in the docs folder**

- Architecture.png: The architecture of this project in Diagram
- report.pdf: Describe the application and the distributed system capabilities

- **Due to time issues, I did not design the client's room selection, so please use the default room and do not modify it when using the client GUI. Thanks!**

- **To find out how to run it, check out README.md**

===

1. Introduction

This report provides a detailed description of a distributed chat system designed to operate on LAN networks. The system is built using a robust distributed architecture that employs the RAFT consensus algorithm, service discovery, consistent hashing, and fault tolerance mechanisms. The primary goal of this project is to demonstrate the implementation of key distributed systems concepts while providing a reliable real-time communication platform.

===

2. System Overview

The Multiplayer Chat System is a distributed application that enables real-time text communication between multiple users over a network. The system consists of multiple server instances that work together to provide high availability and fault tolerance. The architecture follows a leader-follower pattern managed by the RAFT consensus algorithm, which ensures consistency across the distributed system.

Key features of the system include:

- Distributed server architecture with automatic leader election
- Real-time messaging with persistent storage
- Automatic service discovery
- Fault tolerance with seamless recovery
- User-friendly graphical interface
- Command line operation for advanced users
- Data replication for system resilience

===

3. Architectural Components

3.1 Client Components

The system offers two client interfaces:

3.1.1 GUI Client (`client_gui.py`)

The GUI client provides a modern, user-friendly interface for end users. Built using Tkinter, it offers:

- Server connection management with automatic reconnection
- Room-based chat interface
- Real-time message display
- Server status monitoring (get status)
- User presence notifications

3.1.2 Command-Line Client (`client.py`)

The command-line client offers the same core functionality as the GUI client but through a text-based interface. This client is suitable for environments where graphical interfaces are not available or for users who prefer terminal-based applications.

3.2 Server Components

3.2.1 Chat Server (`server.py`)

The chat server is the primary component that handles client connections, message routing, and storage. Multiple server instances work together to form a distributed system. Key responsibilities include:

- Client connection management
- Message broadcasting
- Room management
- Integration with RAFT for consensus
- Data storage coordination

3.2.2 RAFT Implementation (`raft.py`)

The RAFT module implements the RAFT consensus algorithm, providing:

- Leader election
- Log replication
- Safety guarantees
- Term-based versioning
- Heartbeat mechanism

3.2.3 DNS Service (`dns_service.py`)

The DNS service provides service discovery capabilities, allowing servers and clients to find each other on the network. This optional component offers:

- Server registration
- Leader lookup
- Service health monitoring
- Address resolution

3.2.4 Data Store (`data_store.py`)

The data store component handles the persistent storage of messages and other system data. It utilizes consistent hashing for distributed data management, offering:

- Message persistence
- Distributed storage
- Data replication
- Consistent hashing for load distribution

3.2.5 MapReduce Framework (`map_reduce.py`)

The MapReduce component provides analytical capabilities for processing chat data. Though primarily designed for future extensions, it demonstrates:

- Distributed data processing
- Word counting analysis
- User activity analysis
- Parallel task execution

3.2.6 Consistent Hashing (`consistent_hash.py`)

The consistent hashing module implements the consistent hashing algorithm, which ensures efficient data distribution across server nodes while minimizing redistribution when nodes join or leave the system.

==

4. Distributed System Capabilities

4.1 Consensus with RAFT

The system implements the RAFT consensus algorithm to ensure all server nodes maintain a consistent state. This includes:

4.1.1 Leader Election

When the system starts or the current leader fails, the RAFT algorithm automatically triggers a leader election process. Servers transition through different states (follower, candidate, leader) until a new leader is elected by majority vote.

The implementation includes:
- Randomized election timeouts to prevent election conflicts
- Term-based voting system
- Majority-based decision making

4.1.2 Log Replication

Once a leader is elected, all client requests are routed to the leader. The leader:
- Appends new entries to its log
- Replicates the log to follower nodes

- Commits entries once they are safely replicated to a majority of servers

4.1.3 Safety Guarantees

The RAFT implementation provides several safety guarantees:
- Election Safety: At most one leader can be elected in a given term
- Leader Append-Only: Leaders never overwrite or delete entries in their logs
- Log Matching: If two logs contain an entry with the same index and term, all entries up to that index are identical
- Leader Completeness: If an entry is committed, it will be present in the logs of all future leaders

4.2 Fault Tolerance

The system is designed to be fault-tolerant, allowing it to continue operating even when one or more server nodes fail.

4.2.1 Server Failure Handling

In a cluster of N servers, according to the current 3-server test, as long as one of the servers is working it is possible to keep the clients running stably. When a server fails:

- If the failed server is a follower, the system continues to operate normally
- If the failed server is the leader, a new leader is automatically elected
- Data is replicated across multiple servers, ensuring no data loss

4.2.2 Client Reconnection

When a server failure is detected, clients automatically:

1. Detect the disconnection
2. Query the DNS service if available
3. Try alternative server addresses
4. Reconnect to the new leader
5. Rejoin the previous chat room

This process is transparent to the end user, with minimal interruption to the chat experience.

4.3 Distributed Data Management

The system employs consistent hashing for efficient data distribution and management.

4.3.1 Consistent Hashing

The consistent hashing algorithm:
- Maps both servers and data to a circular hash ring
- Assigns data to the next server node in the ring
- Minimizes data redistribution when servers join or leave
- Provides predictable data placement

4.3.2 Data Replication

To ensure data durability, each piece of data is replicated across multiple servers:
- The replication factor is configurable
- Data is stored on adjacent nodes in the consistent hash ring
- Read operations check multiple nodes for availability

4.4 Service Discovery

The system includes a DNS service for automatic service discovery.

4.4.1 Server Registration

When a server starts, it registers with the DNS service, providing:
- Server ID
- Host address and port
- Leader status

4.4.2 Leader Discovery

Clients and servers can query the DNS service to find the current leader:
- Clients connect directly to the leader for optimal performance
- Follower servers can locate the leader for redirection or replication

4.4.3 Health Monitoring

The DNS service monitors server health through periodic heartbeats:
- Servers send regular heartbeat messages
- Servers that fail to send heartbeats are removed from the registry
- Clients are directed to healthy servers

4.5 Communication Protocol

All communication between components uses gRPC, which provides:
- Efficient binary serialization via Protocol Buffers

- Streaming capabilities for real-time updates
- Well-defined service interfaces
- Language-agnostic communication
- Automatic code generation for client/server stubs

==

5. Protocol Specification and Implementation Details

5.1 Protocol Buffers Schema

The system defines three main services in the Protocol Buffers schema:

1. **ChatService**: For client-server communication
    - JoinChat (unary-stream): Clients join a chat room and receive a stream of messages
    - SendMessage (unary-unary): Clients send messages to the server
    - GetServerStatus (unary-unary): Clients query server status
    - Ping (unary-unary): Heartbeat check

2. **RaftService**: For server-server communication
    - RequestVote (unary-unary): Used during leader election
    - AppendEntries (unary-unary): Used for log replication and heartbeats
    - RedirectToLeader (unary-unary): Redirects clients to the current leader

3. **DNSService**: For service discovery
    - Lookup (unary-unary): Find a server by name
    - Register (unary-unary): Register a server in the DNS

5.2 Message Types

The protocol defines several message types:
- ChatMessage: Represents a chat message
- JoinRequest: Used to join a chat room
- SendResponse: Response to a send message operation
- ServerStatus: Contains server state information
- LogEntry: A single entry in the RAFT log
- AppendEntriesRequest/Response: Used for RAFT log replication
- VoteRequest/Response: Used for RAFT leader election

5.3 RAFT Implementation Details

The RAFT implementation follows the original paper's specifications:

1. **Server States**:
   - Follower: Receives and applies log entries from the leader
   - Candidate: Initiates leader election and requests votes
   - Leader: Handles client requests and replicates logs

2. **Term-based Versioning**:
   - Each election begins a new term
   - Terms are used to detect stale information

3. **Election Process**:
   - Randomized election timeout (150-300ms)
   - Vote request to all peers
   - Majority vote required to become leader

4. **Log Replication**:
   - Leader sends AppendEntries to followers
   - Entries are committed once replicated to a majority
   - Followers confirm successful replication

==

6. Scalability and Performance

6.1 Scalability Considerations

The system is designed to scale within a LAN network environment:

- The consistent hashing algorithm evenly distributes data as nodes are added
- The RAFT consensus algorithm supports clusters of arbitrary size
- The DNS service provides dynamic discovery of new servers

However, there are some inherent limitations:
- RAFT requires majority consensus, limiting geographical distribution

- All write operations must go through the leader, creating a potential bottleneck
- The current implementation is optimized for LAN environments

6.2 Performance Optimizations

Several optimizations improve system performance:

- Streaming gRPC connections minimize connection establishment overhead
- Message batching reduces network traffic
- Consistent hashing reduces data redistribution costs
- Local caching of messages reduces disk I/O
- Parallel processing of MapReduce tasks

==

7. Future Enhancements

The current system provides a solid foundation that could be extended with:

1. **End-to-End Encryption**: Adding encryption for message privacy
2. **User Authentication**: Implementing secure user authentication
3. **Web Client**: Developing a browser-based client interface
4. **Message Persistence**: Enhancing storage with database integration
5. **Multi-room Support**: Expanding room capabilities
6. **Media Sharing**: Supporting image and file sharing
7. **Enhanced Analytics**: Expanding MapReduce capabilities for advanced analytics
8. **Federation**: Enabling communication across separate chat clusters

==

8. Conclusion

This distributed chat system successfully demonstrates key distributed systems concepts including consensus algorithms, fault tolerance, service discovery, and consistent hashing. By implementing the RAFT consensus algorithm, the system provides strong consistency guarantees while maintaining availability in the face of node failures.

The separation of concerns between different components (chat server, RAFT, data store, etc.) creates a modular design that could be extended or modified to support additional features. The use of industry-standard technologies like gRPC and Protocol Buffers ensures efficient communication and interoperability.