Group 4: Peer-to-Peer Chat System (P2P)

Kozheen Taher Esa
Joonas Ahovalli
Roni Tuohino
Group 4

December 2024

Contents

1 Project Goals and Core Functionality

The P2P-Chat system is a decentralized and fault-tolerant chat application. Users can create group chats (networks), join them, and exchange messages within a group in a fully distributed manner.

As a core functionalities:

- **Distributed shared state** for group messages
- **Dynamic leader election** for managing group memebership and fault recovery.
- **Fault tolerance** through liveness monitoring and constitent state replication.
- **Minimal reliance** on the Node Discovery Service (NDS) to allow for scalability and decentralization.

As a key features:

- **Local message history:** each node maintain local copies of chat history, making messages available.
- **Bully Algorithm:** Leader election is conducted using the Bully algorithm, which makes sure that system is consistent during node failures.
- **Synchronization and State Consistency:** Achieved through logical clocks and message broadcasting.

2 Design Principles

2.1 System Architecture

Our system uses a **hybrid peer-to-peer (P2P) model** with following components

1. **Node (Clients):**
    - Each node can act as a participant or a leader in a group.
    - Maintains a local history of chat messages.
    - Participates in leader election.
2. **Node Discovery Service (NDS):**
    - A centralized service for initial group discovery and leader identification.
    - Does not participate in intra-group communication, serving only as a registry for groups (networks).

2.2 Shared state

Each node stores a synchronized local copy of the group chat history. Messages are uniquely identified using UUIDs to prevent duplication.

3 Communication Protocol

The system uses **JSON-RPC over HTTP** as its communication protocol. The following are the key messaging functions and their components:

- `create_network`: Used to request the NDS to create a new group.
- `join_network`: Sends a request to the group leader to join an existing group.
- `send_message`: For message broadcasting to all group members or forwards the message to the leader, depending on the situation.
- `heartbeat`: For liveness by monitoring the activity of nodes and verifying the leader's availability.

3.1 Consensus Mechanism

Leader election in the system is performed using a modified version of the **Bully Algorithm**. The key difference from the traditional Bully Algorithm is that the node with the **lowest ID** becomes the leader. This ensures that the leader is always the node that has been active in the network for the longest time.

In the event of a leader failure, the remaining nodes initiate the election process to select a new leader, ensuring **continuity and fault tolerance** within the network.

3.2 Synchronization and Consistency

Synchronization in the system is achieved using **logical clocks** to maintain the order of messages. All messages are routed through the leader, ensuring that the **logical clock values** clearly indicate the order of messages across the network.

In cases of missing messages, the system uses **leader-driven synchronization**, where the leader coordinates with nodes to resolve inconsistencies and restore a consistent state across the group.

3.3 Fault Tolerance

Nodes periodically send **heartbeats** to monitor the liveness of other nodes and detect potential failures.

In the event of a **leader failure**, the system initiates a new **leader election**. Once a new leader is elected, the **Node Discovery Service (NDS)** is updated to reflect the change, maintaining an accurate record of the group structure.

4 System Functionalities

4.1 Shared Distributed State

Each node maintains a **local copy** of the group's chat history, therefore the message log is accessible even in the event of network issues.

Additionally, nodes store **metadata** about group members, including their unique **IDs** and **IP addresses**.

4.2 Synchronization and Consistency

Messages are **propagated to all nodes** using a **broadcast mechanism** initiated by the leader. Thus all participants in the group receive updates consistently.

Logical clocks are used to achieve **causal ordering of messages**, maintaining a clear and coherent sequence of events within the system.

4.3 Consensus and Leader Election

The **leader** is responsible for **coordinating group membership** and **broadcasting messages** to all nodes in the group.

In the event of a leader failure, the system uses the **Bully Algorithm** to **elect a new leader**.

4.4 Node Discovery

The **Node Discovery Service (NDS)** is responsible for:

- Maintaining a **mapping of group names to leader IPs**.
- Being contacted by nodes **only during group creation** and **refreshing available groups**.

4.5 Fault Tolerance

- **Heartbeats** are sent periodically to **monitor liveness**.
- Nodes that fail to respond are **removed from the group**.
- Leaders are responsible for **synchronizing state** with peers to ensure **consistency**.

5 Scalability and Performance

Adding new nodes to the system introduces only a minimal increase in communication load due to the leader-based coordination mechanism. However, broadcasting messages through a leader might cause issues in very large systems. As the system scales, the leader's role in broadcasting messages could become a bottleneck.

The Node Discovery Service (NDS) operates independently since it is only used for group discovery. It is possible to deploy multiple NDS servers to improve scalability and reliability, as the NDS does not participate in intra-group communication.

Performance metrics include latency, which can be measured during message broadcasting, and throughput, which is the number of messages processed per second. Throughput depends on the number of nodes in the system and the frequency of messages sent by users. Although these metrics were not formally measured, there was a noticeable lag in latency because the leader could not process multiple requests concurrently.

To address this, we implemented a monkey patch to enable asynchronous processing with `gevent`, optimizing response latency. Additional optimizations included using randomized heartbeat intervals to reduce election conflicts and logical clock synchronization to minimize redundant messages. These adjustments ensured better system performance while maintaining simplicity.

6 Demonstration

The demonstration was conducted live during the demo session using three virtual machines to simulate independent nodes. Each node was configured to communicate with at least two other nodes.

- **Group Creation and Membership:**
  - Successfully demonstrated the creation of a new group.
  - Nodes were added as members to the group through leader coordination.
- **Message Broadcasting:**
  - Demonstrated the ability to send and broadcast messages across the group.
  - Verified message propagation and causal ordering using logical clocks.
- **Handling Node Failures:**
  - Simulated node failures and showed leader election using the Bully Algorithm.
  - Demonstrated state recovery after leader election.
- **Node Removal and Group State Updates:**
  - Removed a node from the group.
  - Updated group state was visually observable in the user interface.

7 Lessons Learned

In distributed systems, early-stage design and planning become even more important due to the intricate nature of many small, moving parts. Debugging and refactoring can quickly become overwhelming as the system grows in complexity. Each functionality must be thoroughly tested early to prevent cascading issues later in the development cycle. Logging and modularity are key principles that should be prioritized from the outset to enable easier debugging and testing.

One significant challenge we faced was debugging errors in a setup with multiple nodes, especially without a centralized logging system. Tracking down issues in such an environment is inherently complex, and implementing a centralized logging mechanism could greatly improve the debugging experience. Leader election introduced its own set of challenges. While theoretically straightforward, edge cases in real-world scenarios required significant attention to detail. The Bully algorithm, as implemented in this project, worked but required careful handling of node failures and unexpected conditions.

Another major limitation came from the choice of Python as the programming language. While Python is excellent for rapid development, it lacks a support for gRPC implementation that returns Pythonic objects directly. Sending and receiving requests often became inconvenience, adding unnecessary complexity to the development process. Despite these challenges, working with gRPC provided us with a deeper understanding of its inner workings and its role in distributed systems.

There was also challenges wit concurrency and synchronization presented. While threading and asynchronous calls were utilized, locks were not implemented due to time constraints. This, while manageable in the prototype, left us with the need to implement synchronization mechanisms to for thread safety and race condition prevention.

Team coordination also proved to be a challenge. Communicating requirements and aligning everyone's vision of the system consumed time. A significant portion of this time was spent on GUI development and learning the `Textualize` framework. While this was a valuable learning experience, it diverted focus from the primary objectives of the course.

Overall, these show importance of proper planning, efficient debugging tools, and clear communication within the team. Despite these hurdles, the project served as a valuable learning experience.