Decentralized public chat room

Team 6: Ayush Mittal, Marawan Eldeib, Rico Haas, and Bharat Puri

1 Introduction

This project focuses on developing a public chat room where all participants have equal status within the system. It demonstrates key principles of distributed systems, including:

- **Dynamic discovery of host:** New participants can join the group without prior knowledge of the host's IP address and port.
- **Voting:** The Bully Algorithm is used to elect a leader among the participants.
- **Fault tolerance:** A heartbeat mechanism detects node failures, including the leader's failure, triggering a new election when necessary.
- **Causal message ordering:** Vector clocks ensure that chat messages are delivered in causal order.
- **Reliable communication:** Both multicast and unicast communication are implemented with an acknowledgment mechanism to ensure reliability of the communication.

2 Requirements Analysis

2.1 Node Description and Communication

Each instance of our application is called a node, also referred to as a participant in this report. Each node represents a single user in the chat room. Each node generates a unique version 4 UUID, as specified by RFC 4122, Section 4.4, and selects a free UDP port from a predefined range.

All nodes connect to a single Wi-Fi switch. Wireless networks inherently suffer from a higher packet loss rate compared to wired networks, which our application mitigates through reliable unicast and multicast communication channels. Additionally, each node listens to a hardcoded multicast address and port, which all nodes share for multicast communication.

Among all nodes participating in the chat room, one is designated as the leader. The leader is responsible for managing new participants and allowing new nodes to join by maintaining and distributing the group view.

2.2 Dynamic Discovery of Hosts

When a new node is instantiated, it generates a unique ID and initializes its UDP unicast and multicast socket. Unicast socket is used only during election. Then, it attempts to discover the chat room's leader node by multicasting a WANT_TO_JOIN message. Non-leaders ignore the message, but the leader responds with a WANT_TO_JOIN RESPONSE, containing a list of all currently active participants, and other information necessary to send chat messages on the group.

Should the WANT_TO_JOIN packet get no response, the new node sends it again, up to n random times. If it times out for the nth time, the node assumes to be alone in the chat room and takes up the role of the leader after 10 seconds, sending a coordinator message (following to the Bully algorithm).

2.3 Voting

Whenever it becomes apparent that the leader is no longer alive, a new leader election is started by using the Bully algorithm. Nodes that try to join the chat room while an election is taking place, wait for it to finish. During the election process, the ELECTION and RESPONSE messages are sent on unicast channel to replicate message complexity of the Bully algorithm, however the COORDINATOR message is multicasted on the group.

2.4 Fault Tolerance

The application employs a heartbeat mechanism to detect node failures, including that of the leader. Each node continuously monitors the presence of other nodes in the chat room. To ensure reliable detection, every node multicast a heartbeat message at intervals of two seconds. If no heartbeat is received within five seconds, the node assumes the corresponding participant has failed and removes it from the group.

Additionally, the application is designed to handle failures that occur during the election process. It also accommodates dynamic scenarios, such as the addition of a new node while an election is in progress. Furthermore, the system supports the simultaneous initialization of multiple nodes without causing significant degradation in user experience within the chat environment.

2.5 Causal message ordering

Chat messages are delivered in causal order to maintain causality across all nodes. Each message includes its vector clock before being multicasted to the group.

One of the primary challenges in implementing causal message ordering is handling changes in the group view when nodes join or leave. These changes alter the dimensions of the vector clock, complicating synchronization. To simplify implementation, the application only adds entries to the vector clock when new nodes join but does not remove entries when nodes leave. This approach reduces complexity and ensures that:

- Messages in the hold-back queue that depend on a removed entry remain valid.
- No concurrent thread encounters issues due to the deletion of a vector clock entry.

Since the software architecture makes vector clock entry deletion a significant synchronization challenge, retaining entries avoids potential race conditions and inconsistencies.

2.6 Reliable communication

Some of the algorithms we use require reliable communication channels. For this we use a simple acknowledgment system where the recipient(s) send acknowledgements of messages back to the sender. The sender resends unacknowledged messages after a timeout. Multicast messages are sent a minimum of 8 times to ensure that even when not all intended recipients are known, they will likely receive the message regardless, even through a bad connection.

3 Architecture Design

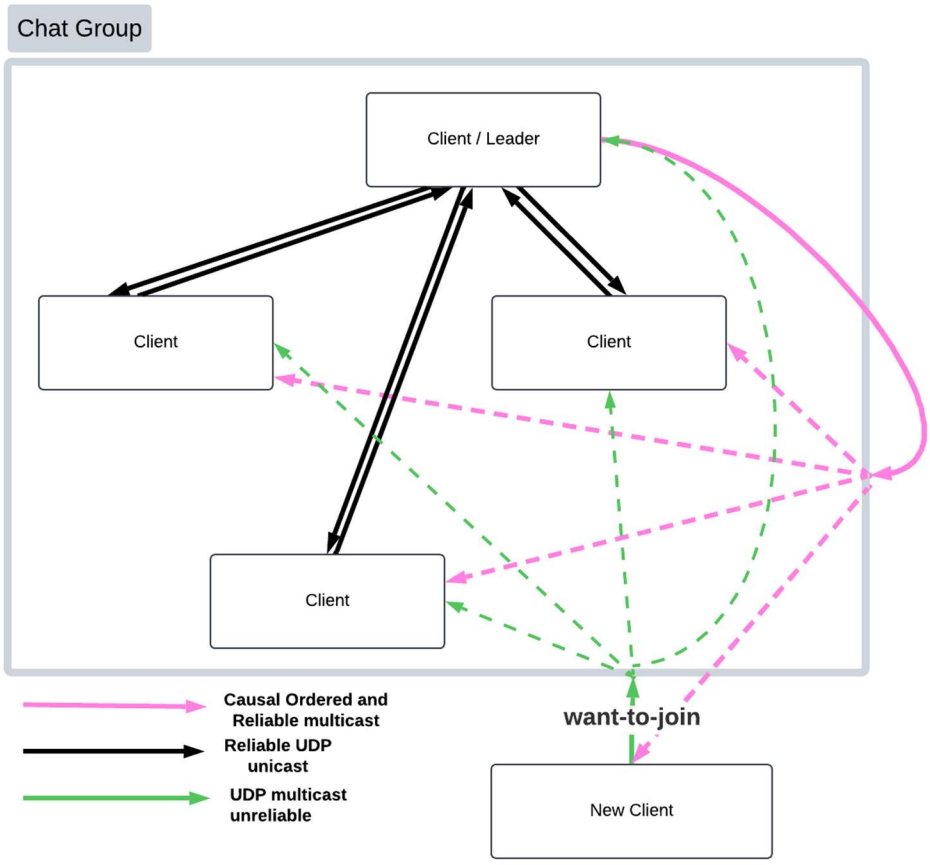

Figure 1 Communication flows of our application.

- **Causal Ordered and Reliable multicast messages:** Chat messages, WANT_TO_JOIN_RESPONSE (only reliable multicasted, not causally ordered).
- **Reliable UDP unicast:** "ELECTION" and "RESPONSE" messages.
- **UDP multicast unreliable:** Heartbeat, "WANT_TO_JOIN", "TRY_JOIN_AGAIN".

4 Implementation

4.1 Software layered architecture

Our program is designed following a layer architecture, similar to the OSI model. Chat messages are created at the very top (UI) layer and travel down the layers one by one. Each layer encases the message with more information, which is useful for the same layer on the receiving side. On the receiving side, the message is received at the very bottom (identity) layer, traveling upward one by one, each layer unpacking the encasings from the sending layers and taking the necessary actions. Like in the OSI model, each layer works together with other instances of the same layer held in other nodes, providing assurances like causal ordering and fault tolerance. From top to bottom:

- **UI layer:** Renders the terminal UI and handles user interactions.
- **Application layer:** Handles high-level application logic, composing initial messages and unpacking the final encasing of received messages with the actual message content.
- **Ordering layer:** Ensures causal ordering of chat messages.
- **Community layer:** Implements Bully Algorithm, group view and ensures fault tolerance through heartbeats.
- **Reliability layer:** Ensures reliable channels for both unicast and multicast sending.
- **Identity layer:** Creates the node's UUID and also opens the actual network sockets and maintains them.

Figure 2 and Figure 3 shows the flow of data between the layers when a packet is sent or received. However, the figure does not show all the functions implemented in the application.

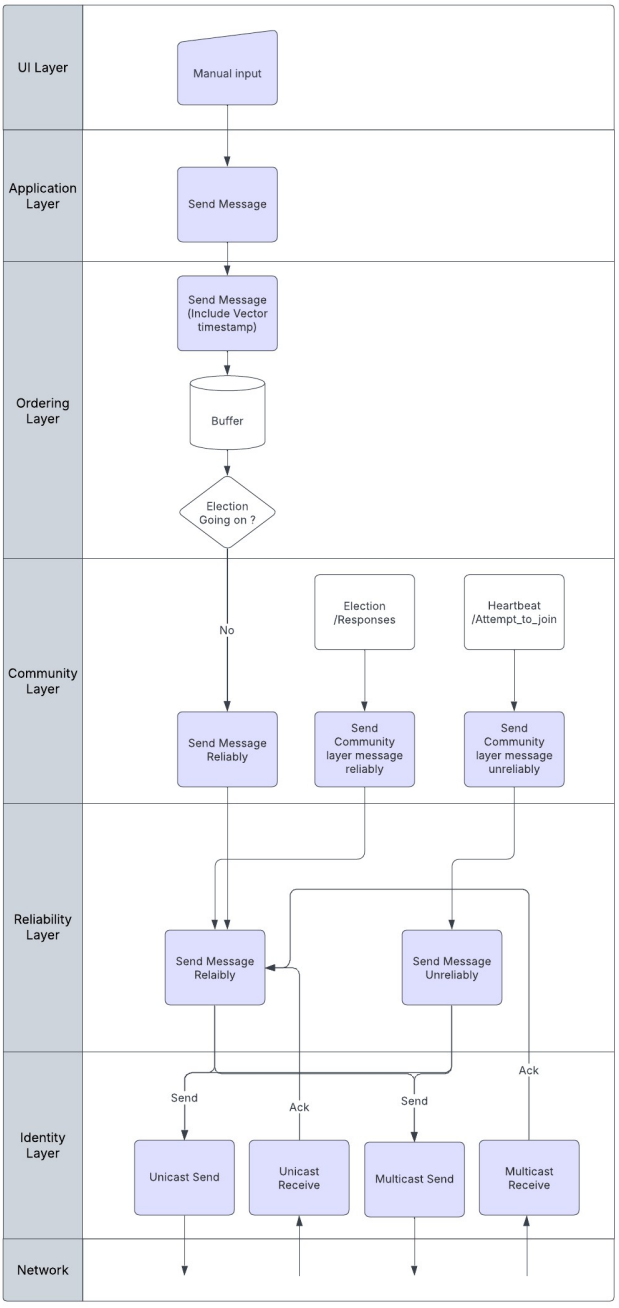

Figure 2 Software architecture (top to bottom flow). Note: This does not include all the implementation details

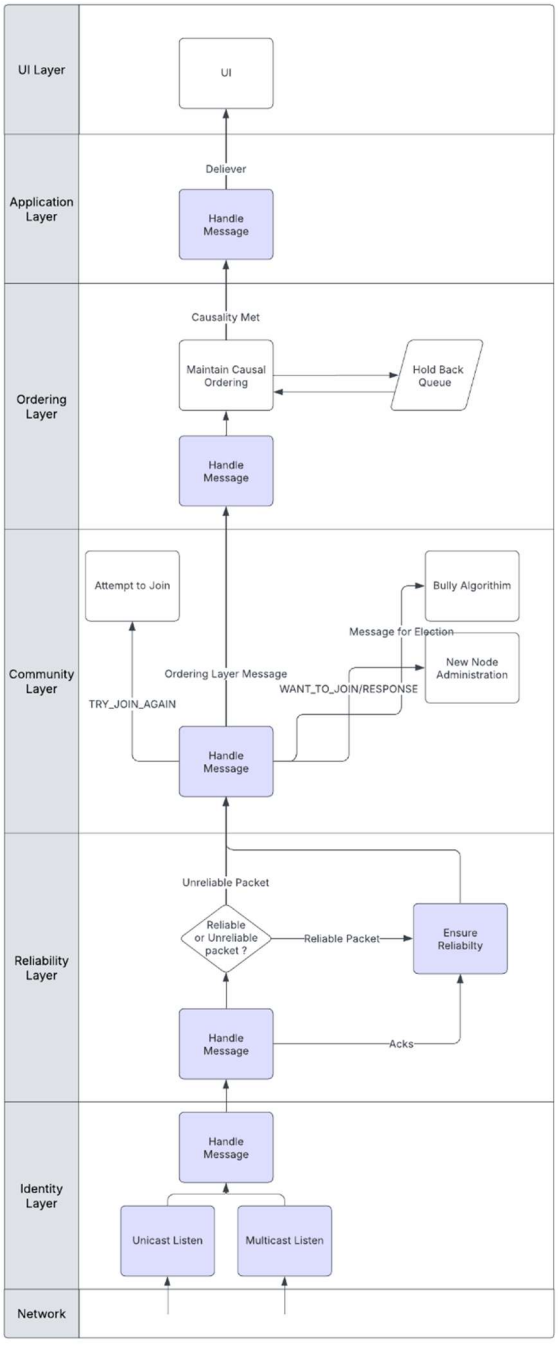

Figure 3 Software architecture (bottom to top flow). Note: This does not include all the implementation details

4.2 Dynamic Discovery of Hosts

When a new node starts, it generates a unique UUID and initializes both its unicast and multicast sockets. To discover the chat room's leader, it multicasts a WANT _TO JOIN message to the multicast group. Only the current leader responds with a WANT_TO_JOIN RESPONSE, which contains the following critical information for the new node:

- **Leader UUID:** The unique identifier of the current leader.
- **Updated group view:** A list of active participants, including the new node's UUID.
- **Most recent vector clock:** The leader's latest vector clock state.
- **Recipient ID:** The UUID of the node for which the response is intended.

Upon receiving this response, the new node initializes itself using the received information, while existing nodes update their group view and integrate the new member into their vector clock.

At this point, the new node is officially part of the chat room. But there are additional design considerations:

Consideration 1 (Resetting the Sequence Number): Upon receiving the WANT TO JOIN _RESPONSE, the new node not only initializes its vector clock using the leader's vector clock but also resets its sequence number to 0. The sequence number tracks the number of messages sent by the node within the group. This reset is necessary to handle an edge case depicted in Figure 4

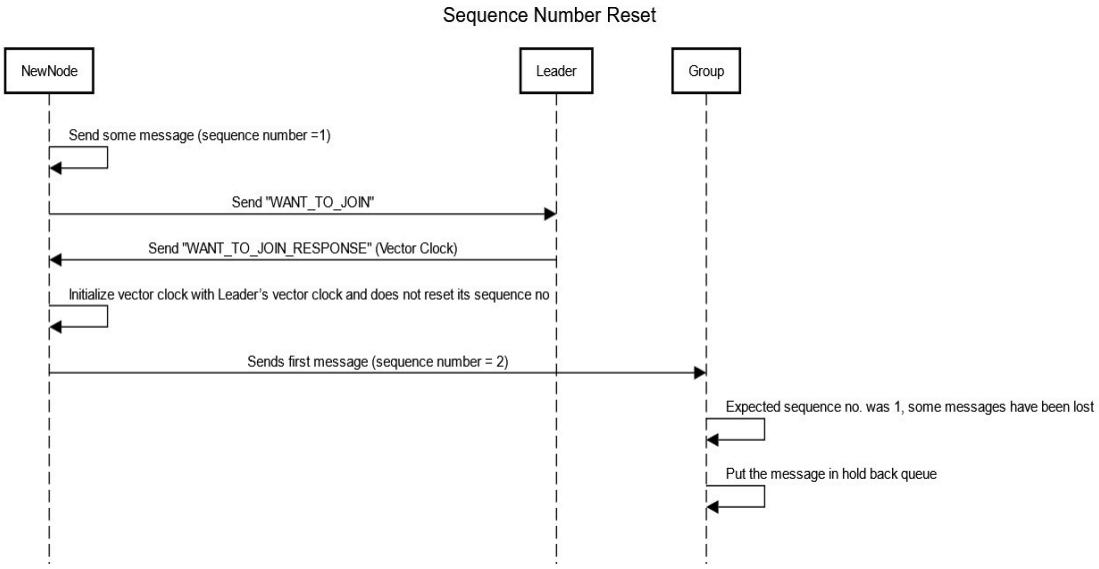

Figure 4 Sequence diagram for the edge case in consideration-1

Consideration 2 (Copying hold back queue to ensure proper communication): the new node must do more than replicate the leader's vector clock. It must also replicate the hold-back queue, which stores messages awaiting delivery. To facilitate this, once the leader sends the WANT_TO_JOIN_RESPONSE, it also multicasts the messages in its hold-back queue. The new node captures these messages and inserts them into its own hold-back queue.

Consideration 3 (No leader in the multicast group): in certain cases, a new node may not receive a response from the leader despite multiple attempts. This can occur in two scenarios:

- The node is the first participant in the group.
- An election is in progress, preventing an immediate response.

To handle this situation, the node takes the following actions:

- **Start multicasting its heartbeat**: If an election is ongoing, after the election, the newly elected leader will detect an unknown heartbeat and prompt the node to join the group.
- **Initiate an election after 10 seconds** (if no response is received):
  If the node is the first participant, it will start the election process to designate itself as the leader.
  For an overview of the implementation, refer Figure 5

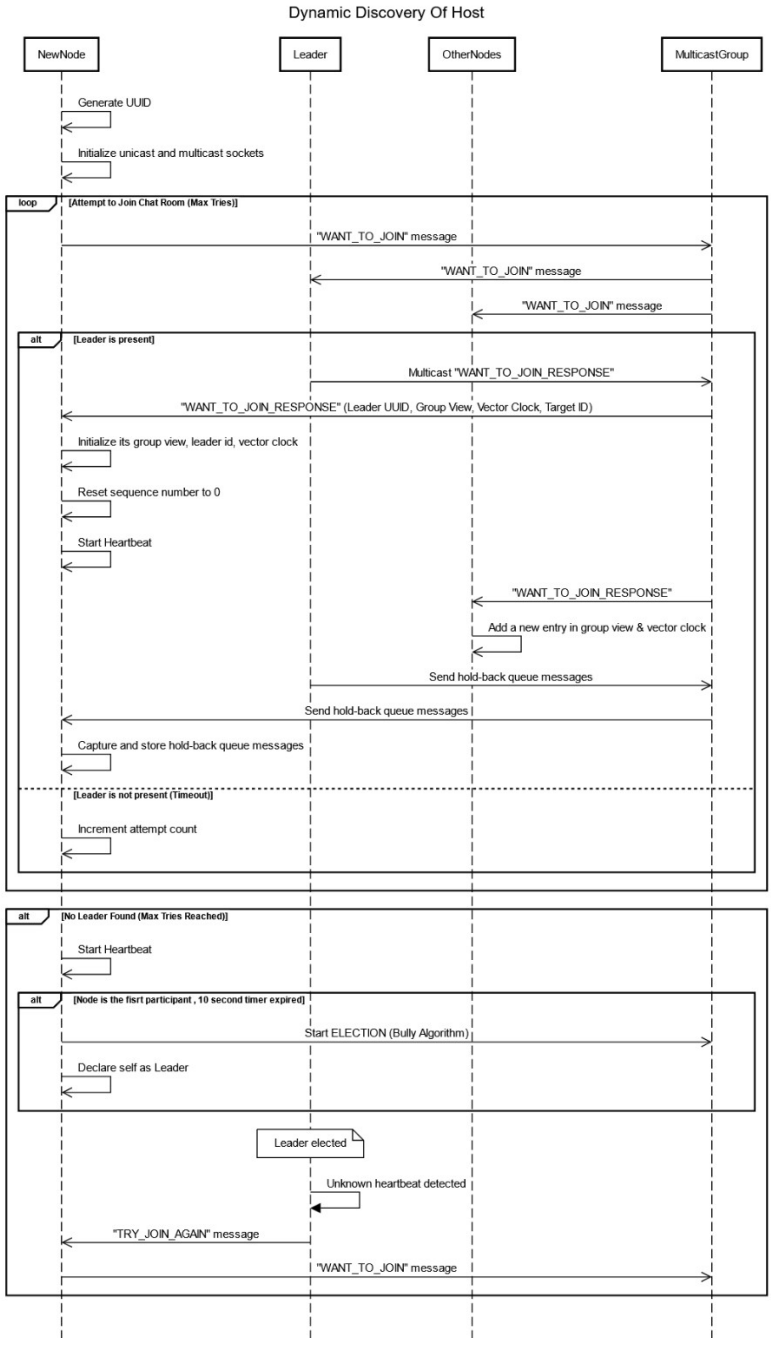

Figure 5 Sequence diagram for dynamic host discovery

4.3 Voting

The Bully Algorithm was chosen as the leader election mechanism for the group. It was chosen because algorithm by design is capable of handling node failures. Our implementation follows a state machine approach, as illustrated in Figure 6

The algorithm operates using both UDP unicast and multicast communication channels:

– **UDP unicast** is used to send ELECTION messages to nodes with higher UUIDs and RESPONSE messages to nodes with lower UUIDs.

– **UDP multicast** is used to send COORDINATOR messages to everyone.

The node with the highest UUID in group view is ultimately elected as the leader. A point to note here is that the election only happens between the members of group view, not with all the nodes present on the multicast group.

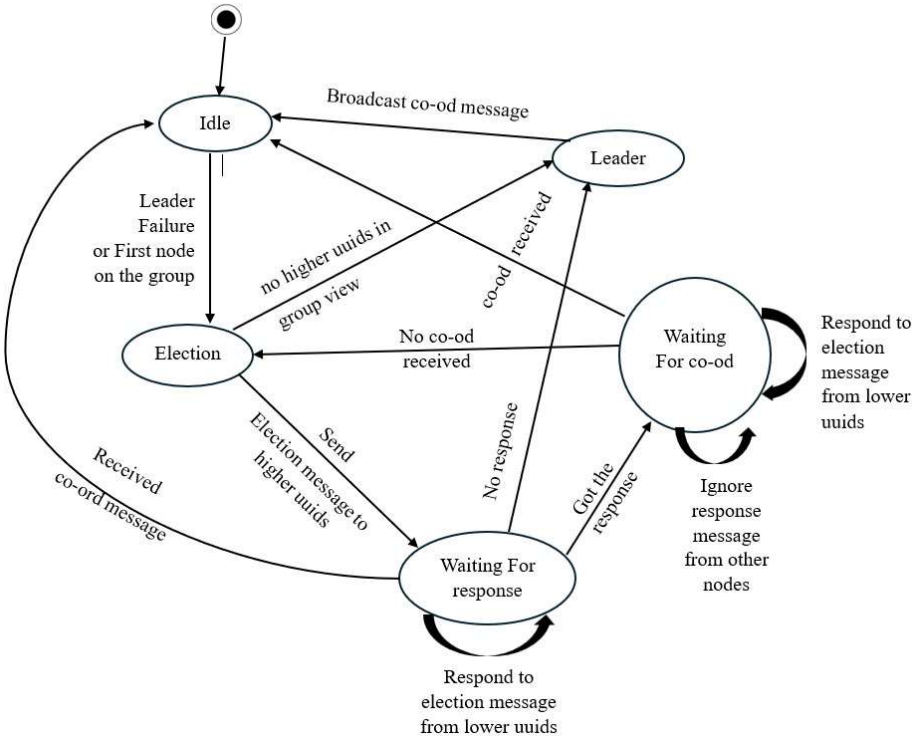

Figure 6. State diagram of Bully Algorithm

Election initiation conditions

- **Leader Failure:** The current leader stops sending heartbeats, triggering an election.
- **No Response to "WANT_TO_JOIN_RESPONSE":** If a new node fails to receive a response after the maximum number of retries, it assumes it is the first node in the multicast group and starts an election after 10 seconds.
- **Receiving an invalid COORDINATOR message:** If a node is in the IDLE state receives a COORDINATOR message from a node with a lower UUID, it initiates an election to ensure the correct leader is selected.

Preconditions A key pre-condition for implementing the **Bully Algorithm** is that every node must maintain a **group view**—a list of all active participants. This is essential for determining which UUIDs are higher or lower than its own when sending ELECTION messages.

- The leader is responsible for updating the group view and ensuring that all nodes have consistent view.
- Each individual node is responsible for removing participants from its own group view by monitoring heartbeat messages.
- The leader does not manage participant removal, as it can fail at any time. This design choice prevents bottlenecks and ensures system resilience.

Adding a Participant When a new node joins, the leader multicasts an updated group view along with the WANT_TO_JOIN_RESPONSE. Upon receiving this message, all nodes:

- Update their group view to include the new participant.
- Initialize the new participant's vector clock to 0.

For more details refer to Figure 5, Dynamic Discovery Of Host

Removing a Participant: Node removal is handled by the group_view function within the community_layer.py. This function runs at regular intervals based on the following parameters:

- HEARTBEAT_INT (2 seconds): Each node sends a heartbeat message every 2 seconds.
- HEARTBEAT_TIMEOUT (5 seconds): The group view function executes every 5 seconds, checking for missing heartbeats.

Since heartbeats should be received at least twice within each HEARTBEAT_TIMEOUT, a failure to update the timestamp within this period results in the removal of the participant from the group view. Each node independently

manages this removal process to ensure fault tolerance. For more details refer to Figure 7

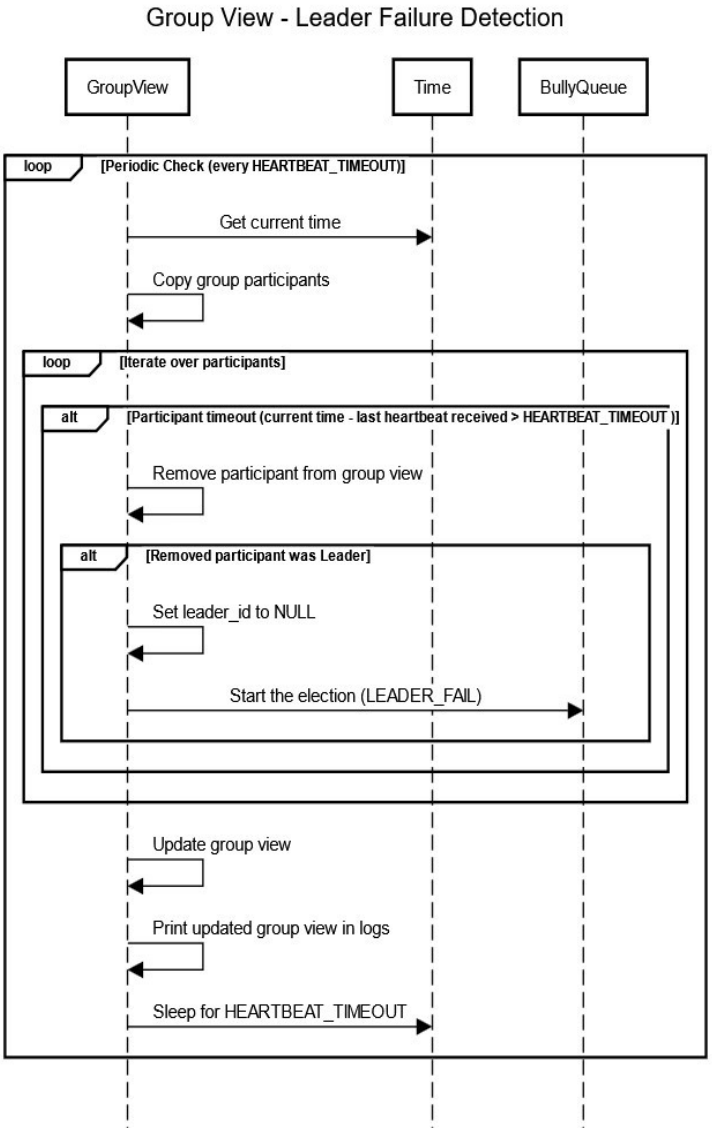

Figure 7 Sequence diagram of group view function

4.4 Fault Tolerance

Our application is designed to detect and handle the following types of failures, ensuring appropriate actions are taken when necessary:

- **Node and Leader Failure:** The application can detect failures in any node, including the leader, and initiates corrective actions accordingly. In the case of a leader failure, the Bully Algorithm is triggered to elect a new leader. No action is taken when any other node fails.
- **Node Failure During Election:** If a node fails during an ongoing election, the algorithm can adapt and continue the election process, ensuring that a leader is eventually selected.
- **Network Partition:** A network partition can result in multiple leaders being within the multicast group. The system is capable of handling this scenario and ensures that only one leader is operational at any given time.

Node and leader failure

The application employs a heartbeat mechanism to detect node failures, including leader failure. Each node monitors the heartbeat of all other nodes, and if a node fails to send a heartbeat within the HEARTBEAT TIMEOUT (5 seconds), it is removed from the group view.

- **Node Failure:** If a node fails, no special action is required by the leader or any other participant. The failed node is simply removed from the group view based on the heartbeat monitoring.
- **Leader Failure:** When the leader fails, the system initiates an election. Multiple nodes may start the election process simultaneously, but the node with the highest UUID will always win the election and become the new leader.

For more details refer Figure 8

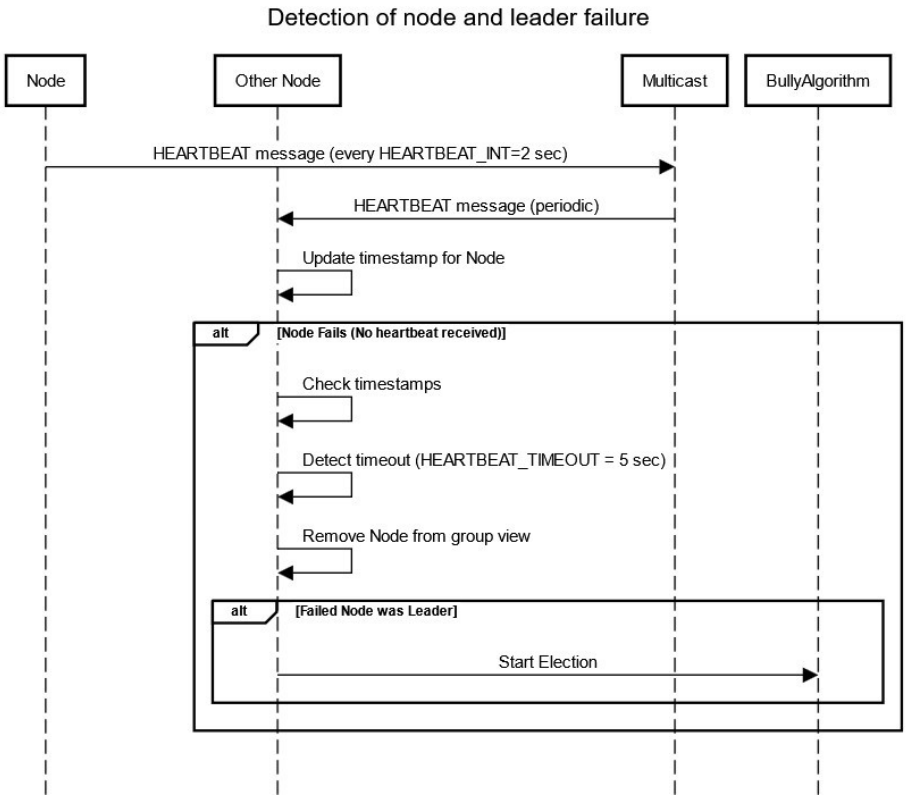

Figure 8 Sequence diagram of node and leader failure detection

Detection of node failure during election

The election algorithm is designed to handle various failure scenarios, including node failures that occur during the election process. The following situations are accounted for:

- **Node Fails After Sending RESPONSE Message:** If a node responds with a RESPONSE message and then crashes, the nodes with lower UUIDs will eventually reach the ELECTION_COD_TIMEOUT (5 seconds) while waiting for the CO-ORDINATOR message. As a result, they will initiate a new election.
- **Node Fails After Sending Election Message:** If a node fails after sending an election message, it does not affect the final outcome of the election. The process continues as expected.
- **Node Fails Before Sending RESPONSE Message:** If a node fails before sending a RESPONSE message, it is treated as though the node failed before the election began, and the process continues accordingly.

- **Node Fails After Becoming the Leader:** If a node fails after being elected as the leader, a new election will be triggered to select a replacement leader.

Network partition which can lead to multiple leaders on the multicast group

A network partition can result in multiple leaders within the same multicast group. This situation may occur, for example, when a leader switches networks between two LANs (a scenario we tested) or when a temporary router failure causes a network partition.

To address this, the leader responds to heartbeat messages from nodes that are not part of the group view. Upon detecting such a node, the leader multicasts a message inviting the node to join the group. This message includes the leader's UUID.

If the recipient of this message is also a leader, it compares its UUID with the current leader's. If the recipient's UUID is smaller, it steps down from its leadership role and becomes a participant, joining the group. If the recipient's UUID is greater, it forces the current leader to step down by making a request to join its group. In this way, the node with the higher UUID compels the node with the smaller UUID to relinquish its leadership.

Limitations and Inconsistencies

While this solution helps address the issue, it is not flawless. In the short term, it may cause some inconsistencies in the group view. However, over time, the system will converge to a consistent group view with a single, unified leader.

The situation can become even more challenging if two different groups suddenly merge into a single network. In such cases, it may take some time for the group view to synchronize across all participants, leading to potential delays in achieving consistency. In such cases election might lead to the selection of a leader whose UUID is not the greatest among the actual group.

For more details refer to Figure 9

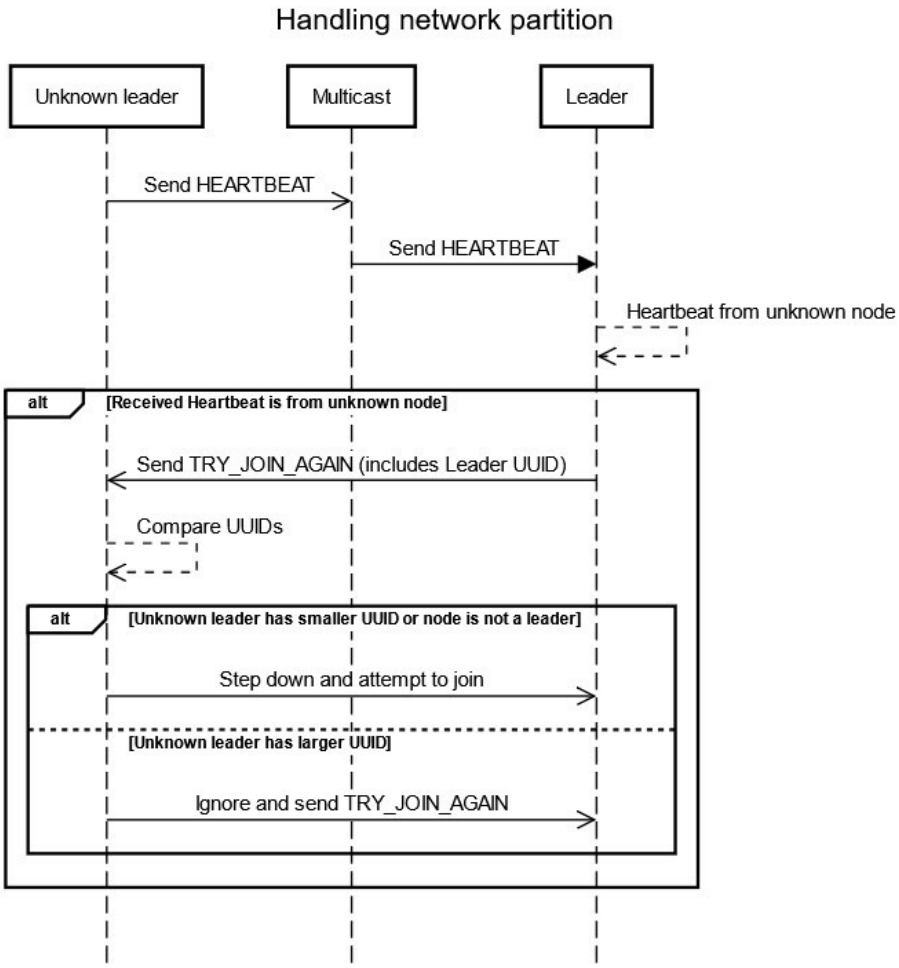

Figure 9 Sequence diagram of network partition handling

4.5 Causal Ordering of Chat messages

The application implements causal ordering for chat messages, assuming the networking layer supports reliable multicast and unicast.

Each chat message is multicasted to the group contains a vector timestamp as metadata, which helps other nodes order the messages causally. Before delivering any message to the application layer, the program checks if the FIFO ordering for a particular node is maintained. This is achieved as follows:

Let the sender's ID be denoted as S, and the vector clock present in the message as **sender_vect_clock** (a Python dictionary). Let the receiver's vector clock be denoted as **receiver_vect_clock** (another Python dictionary). The following condition represents the FIFO ordering for the sender:

receiver_vect_clock [S] + 1 == sender_vect_clock [S]

Here, **dependency_vector, receiver_vect_clock, and sender_vect_clock** are Python dictionaries where the keys are node IDs, and the values represent the vector timestamps. A causal dependency is satisfied if, for every node in the dependency vector, its value is less than or equal to the corresponding value in the **receiver_vect_clock**. The dependency vector is essentially the **sender_vect_clock** without the sender ID's entry:

dependency_vector = del sender_vect_clock[S]

For each node ID in the dependency vector:

```
for node_id in list(dependency_vector.keys()):
    if dependency_vector[node_id] <= receiver_vect_clock[node_id]:
        pass
    else:
        return False
```

If the message fails to meet the above two criteria's (FIFO and Causal ordering) the message is put to hold back queue. Whenever a message is received by a node, the hold back queue is also checked if any of the message can be delivered to the application layer. For more details refer to Figure 10

Additional consideration:

During testing, it was observed that if chat messages are exchanged between group members while an election is in progress, the vector clocks at a newly joined node may not be properly initialized. In some cases, the newly elected leader itself fails to deliver messages. Pinpointing the exact cause of this issue is challenging, but synchronization problems are the most likely explanation.

To address this issue, we modified the system so that messages from the ordering layer (where vector clocks are implemented) are now buffered in the community layer (where the Bully election algorithm is implemented). Messages are only multicast once the election process is complete, ensuring proper vector clock initialization and preventing inconsistencies.

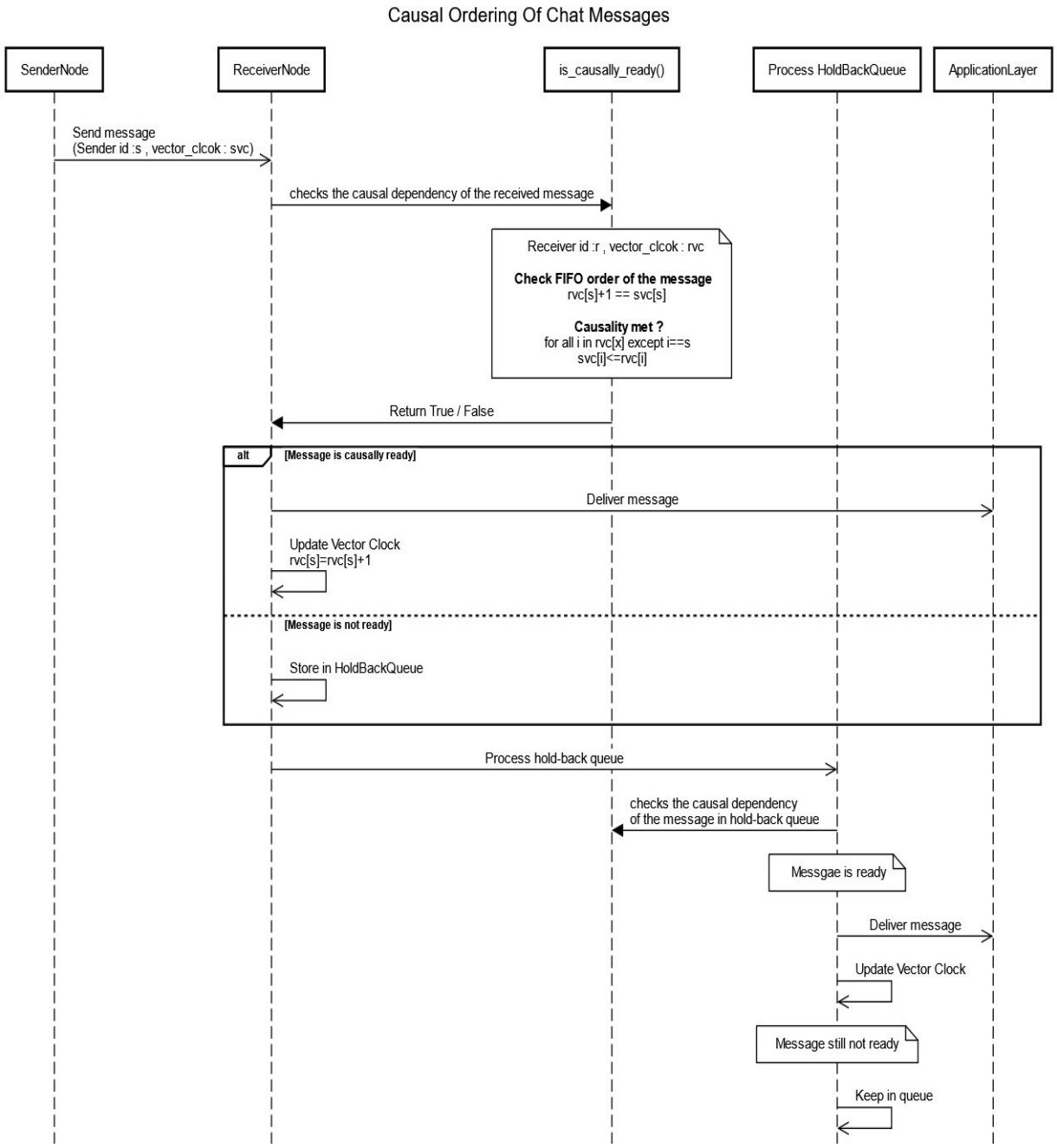

Figure 10 Sequence diagram of causal message ordering

4.6 Ordered Reliable Multicast

While causal ordering of messages is ensured through the vector clocks, reliable sending is ensured by a routine that resends messages that haven't been acknowledged by all intended recipients up to 15 times, waiting 1.5 seconds between each attempt. Multicast messages are sent at least 8 times, since we don't always know everyone who is supposed to receive it.

To facilitate this, all messages are assigned a unique ID. Each recipient will only deliver an ID + sender UUID combination once, regarding all future messages with same combinations as duplicates. Though the recipient acknowledges all messages it receives, even duplicates. All this ensures that higher level routines like the Bully algorithm can assume reliable communication channels. For more details refer to Figure 11

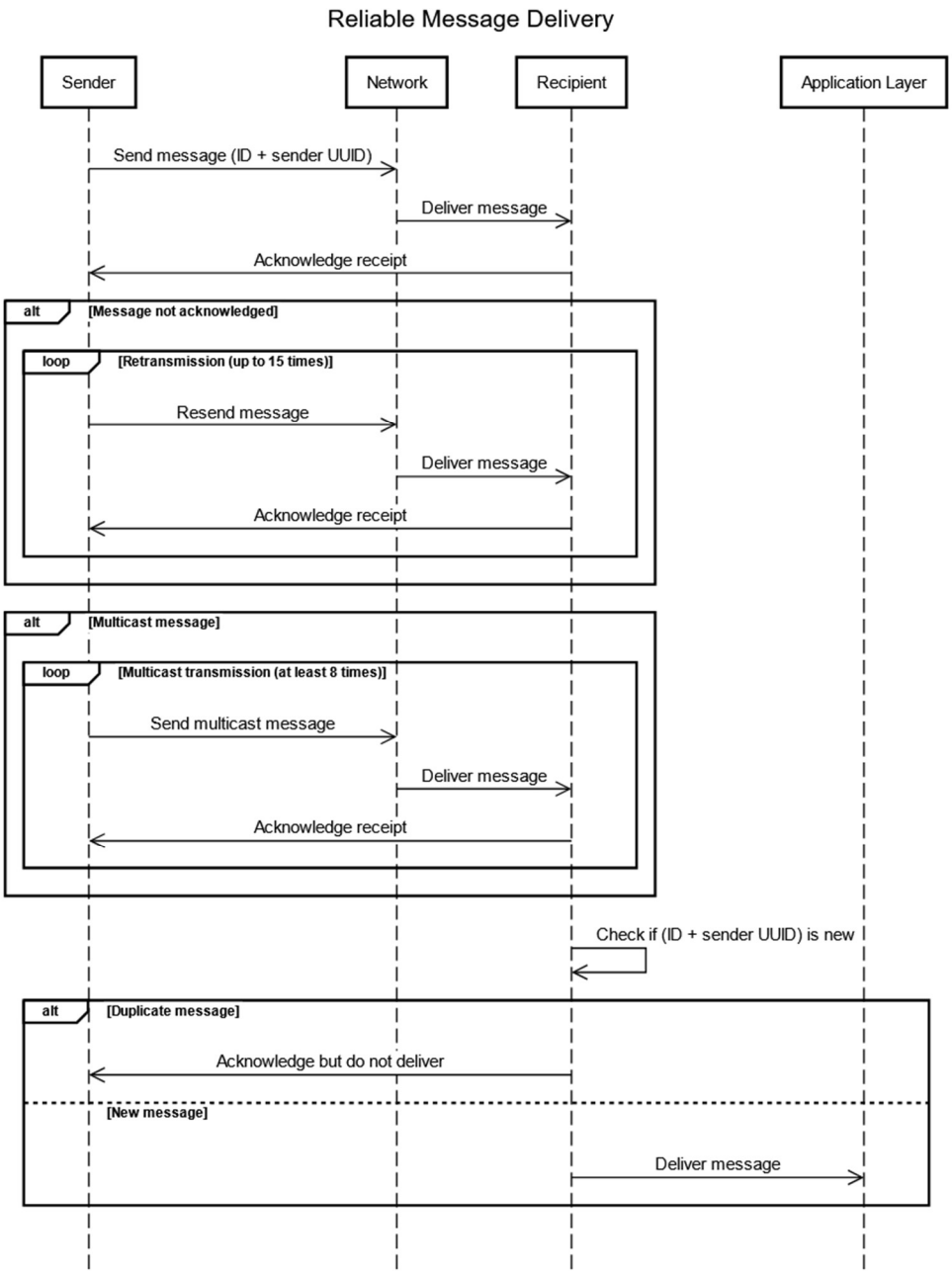

Figure 11 Sequence diagram for reliable communication

5 Discussion and conclusion

5.1 Discussion

The development of a distributed chat system operating over a local area network (LAN) presents several challenges, particularly in combination of dynamic situations like leader failure, election and node joining / leaving. One critical issue is the **lack of deletion of elements from the vector clock** when the node leaves, this can lead to excessive growth in message sent over the group and memory usage as the number of participants increases. This issue is particularly concerning in situations where participant churn is frequent. Future work may consider where obsolete entries from the vector clocks are removed.

Another major challenge is two groups joining after **network partitioning**, which can lead to multiple leaders within the multicast group. This scenario occurs when a group switches networks between two LANs or when a temporary router failure splits the network into disjoint segments. In such cases, different partitions may elect their own leaders, causing inconsistencies in the global group view when they join back.

To address this issue, a mechanism has been implemented as discussed in Section 4.4 Fault Tolerance. However, this solution is not without limitations. While the process generally leads to a stable system over time, there may be temporary inconsistencies in the group view. This is especially problematic when two previously separate groups merge into a single network, requiring time to reconcile differences in leadership and participant membership. In some cases, the election process might select a leader whose UUID is not the highest among all actual participants because of incomplete group view.

Additionally, the system does not enforce global synchronization of the group view. Each node modifies its local view of the network asynchronously, leading to short-term inconsistencies in how different nodes perceive the group. Since every node monitors the heartbeat of all other participants, this also adds to the load on every participant. This can become an issue if the number of participants grow significantly.

5.2 Conclusion

The implemented system successfully demonstrates **fault tolerance, leader election through the Bully algorithm, and causal ordering using vector clocks**. However, challenges such as **network partitioning, asynchronous group view updates, and temporary inconsistencies** highlight the complexities inherent in distributed systems. Testing has played a crucial role in validating the robustness of the system. The **Bully algorithm was rigorously tested** to ensure correct leader election under various failure scenarios, while **causal ordering was verified** to maintain message sequence integrity.

These tests confirmed that, despite transient inconsistencies, the system eventually stabilizes and maintains a single, consistent group view with reliable message ordering.

Future improvements could focus on better garbage collection mechanisms for vector clocks, and implementing a more sophisticated synchronization protocol for maintaining a globally consistent group view. Overall, the system provides a solid foundation for reliable distributed communication while highlighting areas for further refinement and optimization.

GitHub Project Link: https://github.com/ayushmittalde/chat_app_DS

No. of words: 3748