AY 2023/2024

Politecnico di Milano

Causally Ordered Group Chat

Rishabh Tiwari - rishabh.tiwari@mail.polimi.it - 10987397
Simone Errigo - simone.errigo@mail.polimi.it - 11007128
Andrea - andrea@mail.polimi.it - 1234567

Professor
Gianpaolo Saverio Cugola

Version 1.1
June 18, 2024

Contents

1 features

This section covers all the features implemented in our application,

1.1 List of features:

- **Definition of UDP Sockets for Broadcast and Multicast**:

  Broadcast socket

  - **Datagram Socket:** Represents a UDP socket for sending and receiving datagram packets in a specified port, used for broadcast communication.

  Multicast socket:

  - **Multicast socket:** Is a UDP DatagramSocket, with additional capabilities for joining "groups" of other multicast hosts on the internet. The multicast datagram socket class is useful for sending and receiving IP multicast packets.

  Ip addresses assignment

  - **Datagram socket:** The local IP address is assigned for the Datagram Socket. It is taken from the machine's network interface and assigned automatically.
  - **Multicast Socket:** A free multicast IP address is automatically assigned to rooms created by users in the range between 224.0.0.19 and 224.0.0.254 for simplicity

- **Automatic discovery of rooms and users over LAN**:

  Broadcast Messages

  - **Heartbeat Messages:** Regularly sent by each node to announce their presence and the rooms they are aware of. This updates the list of known users and rooms across the network.
  - **Handling Heartbeats:** Upon receiving a heartbeat message, nodes extract the sender's username, userid and their list of known nodes with the users inside from it and use it to update their own list

  Multicast Messages

  - **Room-Specific Messages:** Used for communication within a specific chat room. Each room has a unique multicast IP address for this purpose.
  - **Joining Rooms:** When a user joins a room, they join the associated multicast group to receive messages for that room.

  User Discovery

  - **Listening for Broadcasts:** Nodes listen for broadcast messages to discover other users on the network.
  - **Updating Known Users:** When a node receives a broadcast message, it extracts the user information and updates its list of known users.

Room Discovery

  - **Listening for Multicasts:** Nodes listen for heartbeat messages which contain the list of the rooms and the users inside of them
  - **Room Creation:** When a new room is created is added to the list of known rooms of its creator and is then sent through the heartbeatmessage

Handling Disconnections

  - **High Availability:** Nodes can continue to function (send messages, create/delete rooms) and synchronize messages once they reconnect to the network.

How It's Done in Code

  - **Connection.java:** Handles the network communication, sending, and receiving multicast and broadcast messages.
  - **Node.java:** Manages user operations such as creating, joining, and leaving rooms, and sending messages.
  - **RoomRegistry.java:** Keeps track of the active and deleted rooms.
  - **VectorClock.java and MessageQueue.java:** Ensure causal ordering of messages.

- **Creation and deletion of rooms that spreads and persists over the nodes of the network**

  Room Creation

  - **Initiation:**
    - A user creates a room by specifying a room ID and a list of participants.
    - The 'Node' class handles room creation, generating a unique multicast IP for the room.
  - **Broadcasting:**
    - A multicast message is sent to inform all nodes about the new room.
    - The 'registryHeartbeatMessage' is used to broadcast the creation.
  - **Joining:**
    - Participants join the multicast group to start receiving messages.

  Room Deletion

  - **Initiation:**
    - A user requests to delete a room they created.
    - The 'Node' class handles room deletion.
  - **Broadcasting:**
    - A 'deleteMessage' is broadcasted to inform all nodes about the room deletion.
  - **Updating Nodes:**
    - Upon receiving the deletion message, nodes remove the room from their registry.

    - The ‘RoomRegistry‘ class is updated to reflect the deletion.

Persistence Across Nodes

- **Reliable Communication:**
    - Use of multicast ensures that all participants receive updates.
    - Heartbeat messages help in maintaining updated information.
- **RoomRegistry Class:**
    - Manages the list of active and deleted rooms.
    - Ensures that room information is consistent across all nodes.
- **Handling Disconnections:**
    - Nodes can synchronize their state when they reconnect to the network.
    - Log requests and responses help in recovering missed updates.

How It’s Done in Code

- **Connection.java:** Handles network communication, including sending and receiving multicast and broadcast messages.
- **Node.java:** Manages user operations such as creating, joining, and leaving rooms, and sending messages.
- **RoomRegistry.java:** Keeps track of the active and deleted rooms.
- **VectorClock.java and MessageQueue.java:** Ensure causal ordering of messages.

- **Causal ordering of all messages in a group chat:**

Causal Ordering

- **Definition:** Ensuring that messages are delivered in the order of their causal dependencies.
- **Importance:** Prevents confusion by maintaining the logical sequence of messages.

Vector Clocks

- **Initialization:** Each participant in a room has a vector clock.
- **Message Timestamps:** Each message is tagged with the sender’s vector clock.
- **Clock Updates:**
    - Local Increment: The sender increments its clock before sending a message.
    - Remote Update: Upon receiving a message, the recipient updates its clock.

Message Queues

- **Purpose:** Temporarily hold messages until they can be delivered in causal order.
- **Implementation:**
    - Each node maintains a message queue.
    - Messages are delivered only when their causal dependencies are met.

Handling Messages

- **Sending Messages:**
  - Increment the local vector clock.
  - Attach the vector clock to the message.
- **Receiving Messages:**
  - Check the vector clock of the message.
  - Update local vector clock if all causal dependencies are met.
  - Otherwise, hold the message in the queue.

VectorClock Class

- **Initialization:** Initializes the vector clock for all participants.
- **Incrementing Clock:** Method to increment the local clock.
- **Updating Clock:** Method to update the local clock based on received messages.
- **Clock Comparison:** Checks if the local clock is updated.

MessageQueue Class

- **Message Log:** Maintains logs of messages for each participant.
- **Adding Messages:** Adds messages to the log and updates based on received messages.
- **Causal Order Check:** Compares clocks for correct message order.

LoggedMessage Class

- **Representation:** Represents a logged message with content, user ID, and clock.
- **Comparison:** Compares messages based on vector clocks for ordering.

How It's Done in Code

- **Connection.java:** Handles network communication, including sending and receiving multicast and broadcast messages.
- **Node.java:** Manages user operations such as creating, joining, and leaving rooms, and sending messages.
- **RoomRegistry.java:** Keeps track of the active and deleted rooms.
- **VectorClock.java and MessageQueue.java:** Ensure causal ordering of messages.

- **Synchronization of missed messages through log requests and responses:**
  Log Requests

  - **Purpose:** Request missing messages from other nodes when a user reconnects to the network.
  - **Trigger:** Initiated when a node detects that its message log is outdated compared to others.
  - **Message Type:** 'logRequestMessage' is used to request the log.

Log Responses

- **Purpose:** Provide the requested log of messages to the requesting node.
- **Trigger:** Sent in response to a ‘logRequestMessage‘.
- **Message Type:** ‘logResponseMessage‘ is used to send the log.

Synchronization Process

- **Detection of Out-of-Sync State:**
  - Nodes periodically compare their vector clocks with received messages.
  - If discrepancies are found, a log request is sent.
- **Sending Log Requests:**
  - The node sends a ‘logRequestMessage‘ to the multicast group.
- **Receiving Log Responses:**
  - Other nodes respond with ‘logResponseMessage‘ containing the missing messages.

Vector Clocks and Message Queues

- **Vector Clocks:**
  - Ensure that messages are synchronized in causal order.
  - Updated based on the received logs.
- **Message Queues:**
  - Temporarily store received messages until they can be delivered in order.
  - Updated based on the logs received in the response.

Handling Disconnections

- **High Availability:**
  - Nodes can continue to function even if temporarily disconnected.
  - Synchronization occurs upon reconnection.
- **Resynchronization:**
  - Log requests and responses ensure that nodes catch up with missed messages.

How It’s Done in Code

- **Connection.java:** Handles sending and receiving log requests and responses.
- **Node.java:** Manages the detection of out-of-sync state and initiates log requests.
- **MessageQueue.java:** Updates message logs based on received responses.
- **VectorClock.java:** Ensures causal ordering of synchronized messages.

- **Resilience to disconnections, a user is still able to send messages and leave groups while disconnected those are added to the log**

  Handling Disconnections

- **High Availability:**
  - Users can continue to interact with the chat application even when disconnected from the network.
- **Local Storage:**
  - Actions such as sending messages and leaving groups are stored locally.

Sending Messages While Disconnected

- **Local Logging:**
  - Messages sent while disconnected are added to the local log.
- **Synchronization:**
  - Upon reconnection, the local log is synchronized with the network.
  - ‘logRequestMessage‘ and ‘logResponseMessage‘ are used to update the logs.

Leaving Groups While Disconnected

- **Local Update:**
  - The action of leaving a group is recorded locally.
- **Network Update:**
  - Upon reconnection, the leave action is broadcasted to other nodes.

Vector Clocks and Message Queues

- **Vector Clocks:**
  - Maintain causal order of messages, even during disconnections.
  - Updated based on the local and received logs.
- **Message Queues:**
  - Temporarily store messages and actions until they can be synchronized.

Code Implementation

- **Connection.java:**
  - Manages network communication, handling both online and offline scenarios.
- **Node.java:**
  - Manages user actions such as sending messages and leaving groups, both online and offline.
- **MessageQueue.java:**
  - Stores messages and logs locally, ensuring they are added to the network log upon reconnection.
- **VectorClock.java:**
  - Ensures causal ordering of messages and actions, synchronizing vector clocks upon reconnection.

- **Message log for each group letting you leave the room and comeback while still keeping the information of the old sessions**

  $\boxed{\text{Message Log}}$

  - **Purpose:** Maintain a persistent log of messages for each group.
  - **Components:**
    * Message content.
    * User ID of the sender.
    * Vector clock for causal ordering.

  $\boxed{\text{Leaving and Rejoining Rooms}}$

  - **Leaving a Room:**
    * User leaves the multicast group.
    * Local message log remains intact.
  - **Rejoining a Room:**
    * User rejoins the multicast group.
    * Local message log is synchronized with the latest messages from the group.

  $\boxed{\text{Synchronization Process}}$

  - **Log Requests:**
    * Sent to the multicast group to request missing messages.
  - **Log Responses:**
    * Other nodes respond with the required messages.
    * Local message log is updated.

  $\boxed{\text{Vector Clocks and Message Queues}}$

  - **Vector Clocks:**
    * Ensure messages are ordered causally.
    * Updated based on both sent and received messages.
  - **Message Queues:**
    * Temporarily store messages until they can be delivered in order.
    * Ensure consistency of the message log.

  $\boxed{\text{Code Implementation}}$

  - **Connection.java:**
    * Manages network communication, including joining and leaving multicast groups.
  - **Node.java:**
    * Manages user actions such as leaving and rejoining rooms.
    * Initiates log requests and handles log responses.
  - **MessageQueue.java:**
    * Stores and updates the message log.
  - **VectorClock.java:**
    * Ensures causal ordering of messages.