Client-Server Protocol

In our instant message application, client should be able to:

1. Send messages (to individual users or groups)
2. Receive messages (both online and offline)
3. View historical messages
4. Participate in multiple chats simultaneously
5. Manage chat groups (create, join, leave)
6. Handle user authentication (login, logout)

Commands and Actions

Define commands that the client can send to the server and the expected server responses. Communication between client and server is structured using JSON. Key message types include:

Registration:

Client -> Server:

```
{
  "type": "register",
  "username": "user1",
}
```

Server -> Client:

```
{
  "type": "registration_response",
  "status": " register_success"
}
```

Login:

Client -> Server:

{

"type": "login",

"username": "user1",

}

Server -> Client:

{

"type": "login_response",

"status": "login_success",

"pending_messages": [

"type": " fetch_history ",

{"message_id": "msgid123", "sender": "user2", "content": "Hello!"}

]

}

Send and Receive Message:

Private Chat:

Client:

{

"type": " SEND_MESSAGE ",

"recipient": "user1",

"content": "Hi user1"

}

Server:

{

"type": " MESSAGE_SENT ",

}

{

"type": " RECEIVE_MESSAGE ",

"sender": "user2",

"content": "Hi user1"

"timestamp": "2024-08-21T10:01:00Z"

}

Group Chat:

Client:

{

"type": " SEND_GROUP_MESSAGE ",

"groupname": "groupname1",

“sender”: “user1”,

"content": "Hi"

}

Server:

{

"type": " GROUP _MESSAGE_SENT ",

}

{

"type": " RECEIVE_ GROUP_ MESSAGE ",

"groupname": "groupname1",

"sender": "user1",

"content": "Hi"

"timestamp": "2024-08-21T10:02:00Z"

}

Protocol Grammar

Registration

registration_command = " register " (space) username

registrarion_response = "register_success" | "register_fail" (space) reason

Login

login_command = "login" (space) username

login_response = "login_success" | "login_fail" (space) reason

Logout

logout_command = "logout"

logout_response = "logout_success"

Messaging

send_command = "send" (space) recipient | group (space) message

send_response = "send_success" | "send_fail" (space) reason

receive_message = "receive" (space) sender | group (space) message (space) timestamp

Historical Message

pending_messages_command = "fetch_history" (space) recipient (space) start_time (space) end_time

pending_messages_response = "history" (space) messages

Create group

create_group_command = "create_group" (space) group_name

create_group_response = "create_group_success" (space) group_id | "create_group_fail" (space) reason

Join Group

join_group_command = "join_group" (space) group_id

join_group_response = "join_group_success" | "join_group_fail" (space) reason

Leave group

leave_group_command = "leave_group" (space) group_id

leave_group_response = "leave_group_success" | "leave_group_fail" (space) reason

State Management

Server-Side State

The server needs to maintain

- User Sessions: Which users are currently logged in.
- Message Queues: For delivering offline messages.
- Chat Histories: Messages sent in each chat (user-to-user or group).
- Group Memberships: Which users belong to which groups.

Client-Side State

The client should store:

- Session Information: The current logged-in user and possibly a session token.
- Message Cache: Recently received messages, especially for chats the user is actively participating in.
- Chat Memberships: List of groups the user is a part of.

Conversation Design

Overview

In this IM system, a **Conversation** represents a series of messages exchanged between two or more users. The Java classes will encapsulate the logic for managing conversations, handling messages, and managing the lifecycle of connections via WebSockets.

Java Classes

1. **Conversation**
    - **Description**: Represents a conversation between two or more users. It contains a list of participants and messages exchanged within the conversation.
    - **Public Methods**:
        - `void addParticipant(User user)`: Adds a user to the conversation.
        - `void removeParticipant(User user)`: Removes a user from the conversation.
        - `void addMessage(Message message)`: Adds a message to the conversation.
        - `List<Message> getMessages()`: Retrieves the list of all messages in the conversation.
        - `List<User> getParticipants()`: Retrieves the list of all participants.
        - `int getConversationId()`: Returns the unique identifier of the conversation.
    - **Interactions**: This class interacts with `User` and `Message` classes to manage the participants and messages within a conversation.
2. **Message**
    - **Description**: Represents a message within a conversation. Contains details like sender, timestamp, and content.
    - **Public Methods**:
        - `String getContent()`: Retrieves the content of the message.
        - `User getSender()`: Retrieves the sender of the message.
        - `LocalDateTime getTimestamp()`: Retrieves the timestamp of the message.
        - `int getMessageId()`: Returns the unique identifier of the message.
    - **Interactions**: This class interacts with the `User` class to associate a message with its sender and the `Conversation` class to be added to a conversation.
3. **User**
    - **Description**: Represents a user in the system. Contains user details like username, status, and connected WebSocket session.
    - **Public Methods**:

        - `String getUsername()`: Retrieves the username of the user.
        - `boolean isOnline()`: Checks if the user is currently online.
        - `WebSocketSession getSession()`: Retrieves the current WebSocket session of the user.
        - `void sendMessage(Message message)`: Sends a message to the user.
    - **Interactions**: This class interacts with the `Message` class when sending/receiving messages and with the `Conversation` class to participate in conversations.
4. **ConversationManager**
    - **Description**: Manages multiple conversations, creating new ones and managing existing ones.
    - **Public Methods**:
        - `Conversation createConversation(List<User> participants)`: Creates a new conversation with the specified participants.
        - `Conversation getConversation(int conversationId)`: Retrieves an existing conversation by its ID.
        - `void endConversation(int conversationId)`: Ends a conversation and removes it from the active list.
    - **Interactions**: This class manages instances of the `Conversation` class, creating and retrieving them as needed. It interacts with the `User` class to add participants to conversations.

Class Interactions

- **Message Sending**: When a user sends a message, the `User` class uses the `WebSocketSession` to send the message. The message is then added to the corresponding `Conversation` through the `addMessage()` method. The `Conversation` manages the list of messages and participants, ensuring that the message is broadcast to all participants.
- **Conversation Management**: The `ConversationManager` class handles the creation and retrieval of conversations. When a new conversation is initiated, it creates an instance of the `Conversation` class and adds the relevant users as participants. It is also responsible for ending conversations and ensuring they are removed from the active list.

Snapshot Diagram (Rough Draft):