Technical Design: Server Component

# Program Flow

## Program Flow: Registration

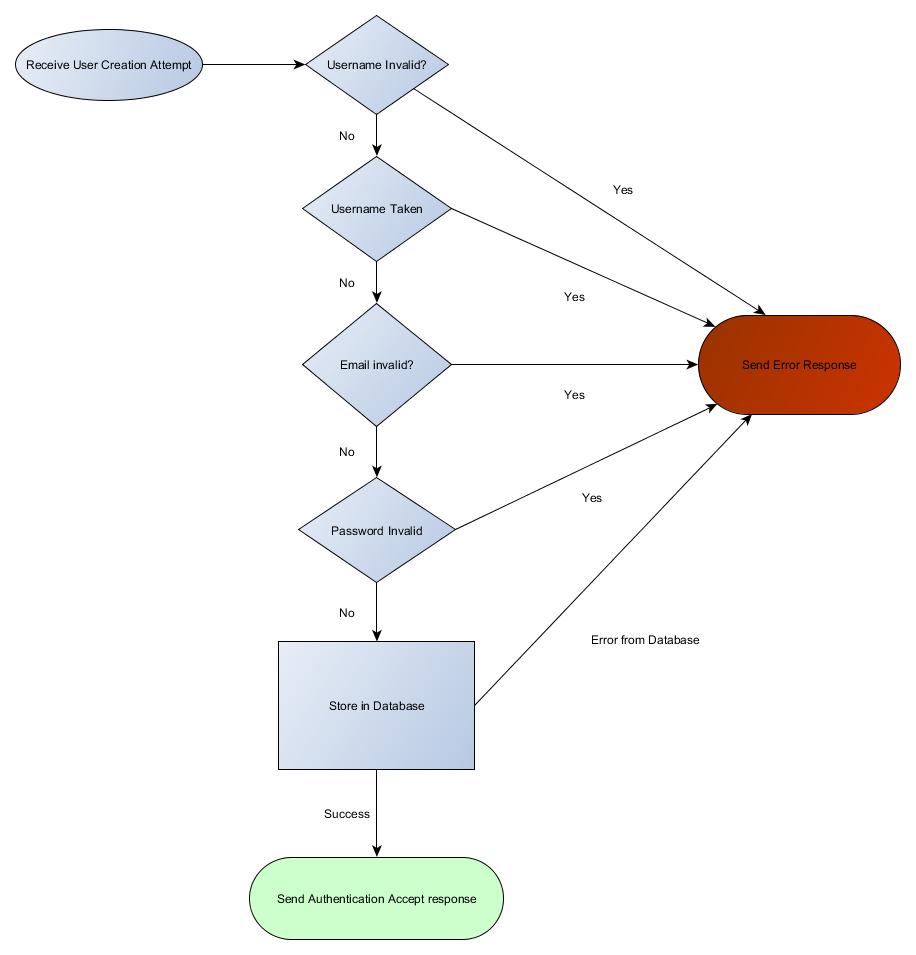


Figure : User Creation Attempt

During the registration process, a success message shall only be sent if the username is valid and not already in use, the email is valid and not already in use, and the password is valid. At that point, the new user shall be stored in the database, and a success message returned. If any of the preconditions fail, or if the database cannot be written to for some reason, an error shall be returned.

This shall be handled by the DAO object instantiated when the server was started.

## Program Flow: Login

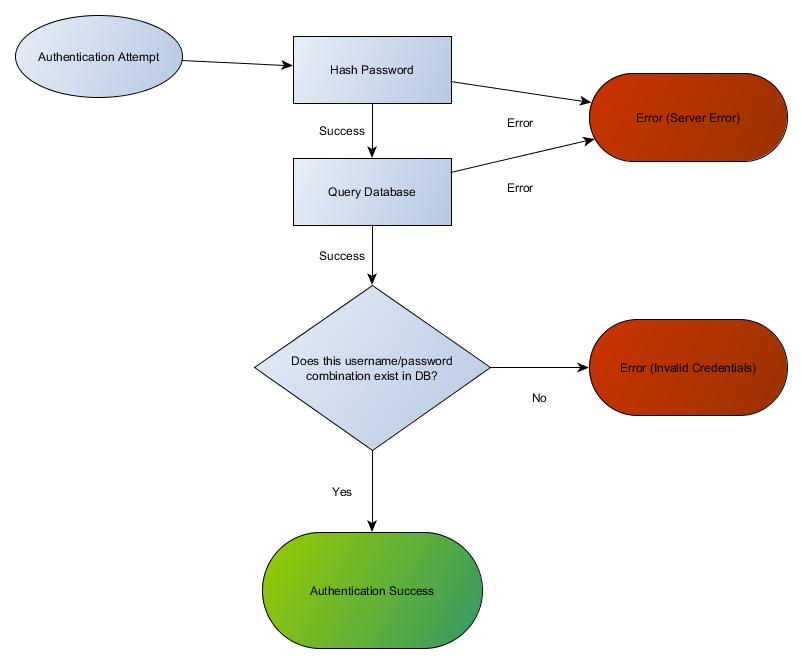


Figure : Authentication Attempt

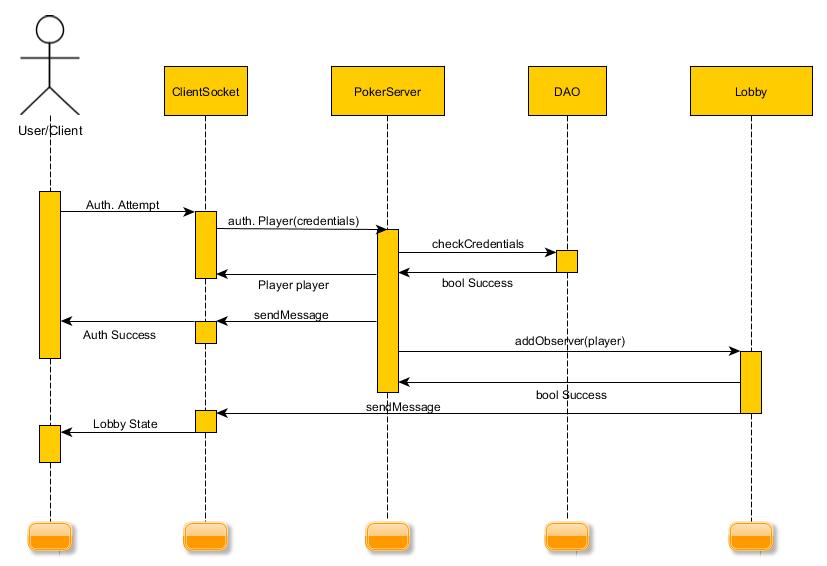


Figure 3: Login Sequence

During the login process, the database shall be queried with the combination of username and hashed password. If a record does not exist with that combination in it, an invalid credentials error shall be sent to the client. If there is an issue accessing the database, an error shall be sent to the client. Otherwise, a success message shall be sent back to the client.

## Program Flow: Joining a Game

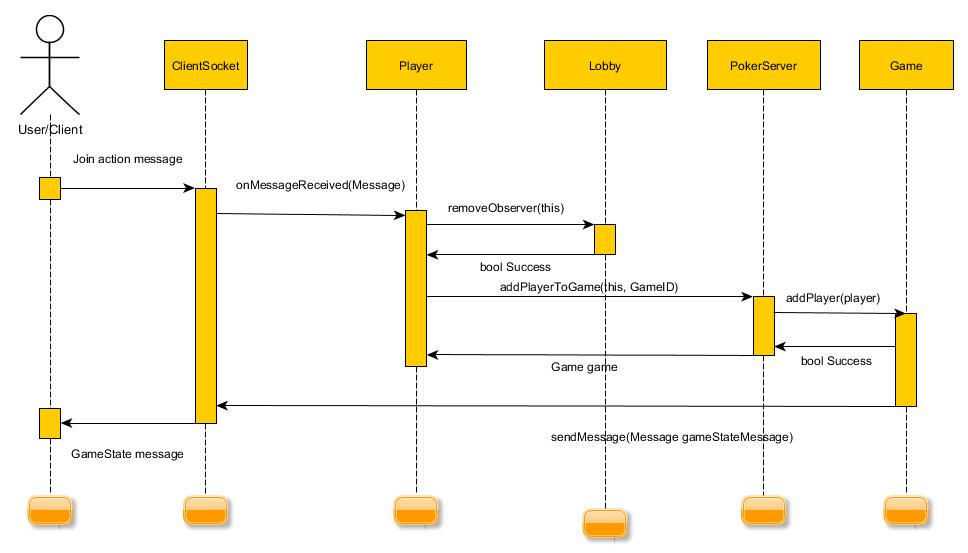


Figure 4: Joining a Game

Shortly after joining the server, the client will receive a LobbyState message indicating the games that are in progress. The client may attempt to join any such game. If the game is full, the join will be rejected with an error message. Otherwise, the client will be added to the game and removed from the Lobby. They will also receive a GameState message to indicate the current state of the game. Note that they will not have a turn until the next deal.

## Program Flow: Game Actions

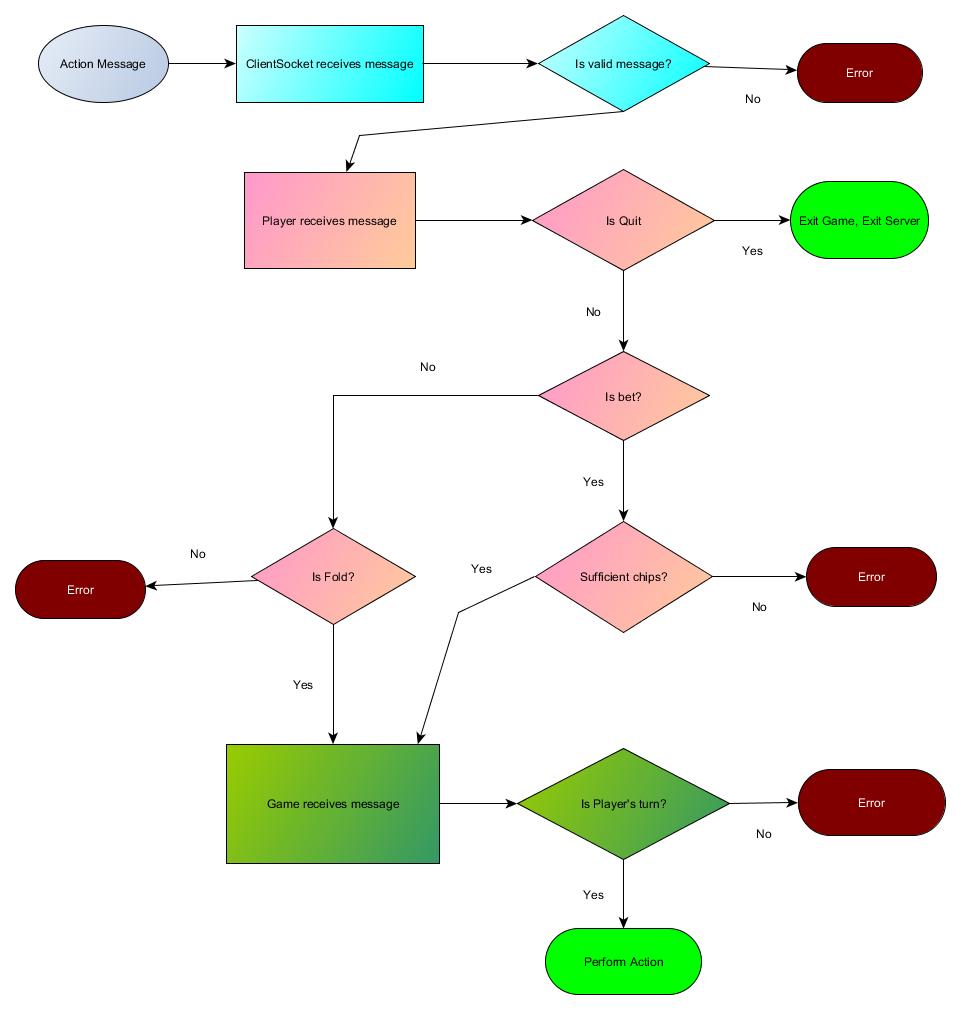


Figure 5: Performing an Action

The client may attempt to submit an action. First, the Client passes the message to the Player.

If the action is a bet and the Player cannot bet that amount, the action is rejected with an error message. If the action is a quit, the Player immediately resigns from the Game and returns to the Lobby. Otherwise, the message is passed to the Game.

If it is not the Player’s turn, the action is rejected with an error message. Otherwise, the action is processed, the Game’s state is updated, and all players are sent a GameState message to indicate the change in state.

# Classes

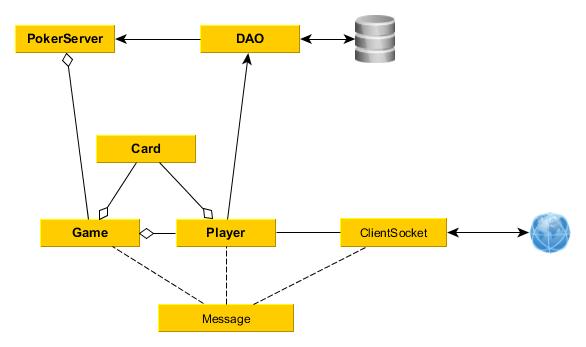


Figure : Class Diagram

## PokerServer

Instances of the class “PokerServer” shall represent a single concurrent server running. Each instance of the server shall listen on a specific port. It shall have the responsibility for managing connections and creating game objects as needed. This shall be a Singleton object

**Instance Members:**

* HashMap<Integer, Game> activeGames
* ArrayList<Player> allPlayers
* ArrayList<Player> lobbyPlayers
* DAO Database

**Instance Methods:**

* main
* createGame
* addClient
* addPlayerToGame
* authenticatePlayer

**Class Methods:**

* getInstance

## DAO

Instances of the class “DAO” shall represent individual databases in which data may be persisted. In practice, there should only be one instance per server.

**Instance Members**

**Instance Methods**

* updatePlayer
* checkCredentials
* addPlayer

## Lobby <iObservable>

Instances of the class “Lobby” represent a lobby where players reside when they are not in a Game.

**Instance Members**

* HashMap<Players, boolean> players

## Game <iObservable>

Instances of the class “Game” shall represent a single game currently in progress. It shall handle the turns of the game, including enforcing the business rules. There is the possibility of observing a game without being a player in it, though no client currently implements this behavior.

**Instance Members**

* Integer gameID
* HashMap<Players, boolean> players
  + The Boolean value indicates if they get a turn or not.
* ArrayList<iGameObserver> observers
* ArrayList<Card> tableCards
* ArrayList<Card> deck
* Integer chipsInPot

**Instance Methods:**

* parseMessage
* dealCard
* addPlayer
* removePlayer

## Card

Instances of the class “Card” shall represent individual cards in a deck. A deck consists of 52 Card objects, each different from the last.

**Instance Members**

* Enum suit
* String value

**Instance Methods**

**Class Methods**

* generateDeck

## Player <iStateObserver, iClientObserver>

Instances of the class “Player” shall represent a single player, either in a game or in the lobby.

**Instance Members**

* String username
* URL avatarURL
* Integer chipsRemaining
* ArrayList<Card> currentHand
* Game currentGame
* Lobby currentLobby
* iClient client

**Instance Methods:**

* addCardToHand
* turnStart
* placeBet
* fold
* getAvatar
* getClientForPlayer
* joinGame
* leaveGame

## ClientSocket <iClient, iObservable>

Instances of the class “ClientSocket” represent external clients connected to the system via TCP sockets. This class will send messages to the client and listen for messages from the client, automatically rejecting anything that is not signed appropriately. This implements the iClient interface.

**Instance Members**

* Socket socket
* String authenticationKey
* ArrayList<iClientObserver> observers

## Message

Instances of the class “Message” represent a message coming from or going to a client.

**Instance Members**

* String text
* int code
* iClient clientInvolved

# Interfaces

## iClient

This represents a client that can send and receive messages

**Instance Methods**

* receiveMessage
* sendMessage

## iClientObserver

This represents something that observes a client. It defines a common callback to be used.

**Instance Methods**

* onMessageReceived

## iStateObserver

This represents something that observes the state of a game or lobby. It defines a common callback to be used which takes as an argument the Message that represents the new state.

**Instance Methods**

* onStateChanged

## iObservable

This represents something that can be observed.

**Instance Methods**

* addObserver
* removeObserver