EECS 22L: Poker Game v2.0

Prepared by: Team rand()

Affiliation: UCI Henry Samueli School of Engineering

Developers:

Yongye Li

Mary Campbell

Arjun Sivakumar

Joseph Principe

Cristian Pina Bravo

Zijie Huang

Date:

May 29, 2023

Contents

Glossary

- Texas Hold'em Poker: A version of poker where players try to get the best poker hand from the two they were dealt and the five community cards.
- Ante: A small forced bet that everyone at the table is required to pay before each hand.
- Dealer: The player who deals the cards. In online poker, the dealer is a computer.
- Suit: A group of thirteen cards that share the same symbol. Decks have two red (diamonds and hearts) and two black (spades and clubs) suits.
- Big Blind: The larger of the two forced bets in a game with blinds. The big blind is paid by the player two seats to the left of the dealer button.
- Small Blind: The smaller of the two forced bets in a game with blinds. The small blind is paid by the player one seat to the left of the dealer button.

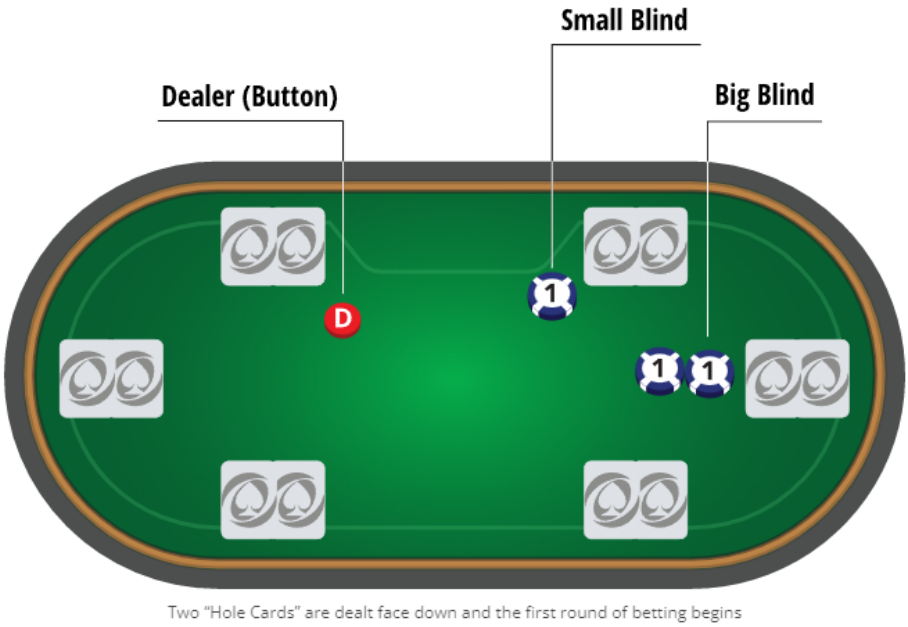

Figure 1: Dealer and blinds

- Check: When a player passes a bet, effectively betting zero. Players can only check if nobody has bet before them. If all players check, the betting round is over and another card is put into the community cards (if it is not already at five cards). If a player after a check does bet, then the betting round will continue and the player who checked will now have to either fold, call, or raise.
- Fold: When a player discards their hand for the current round. They do not play for the remainder of the round and cannot re-enter.
- Call: Matching the current highest bet in the round.
- Raise: Increasing the current highest bet.
- All-in: Betting all of one's chips.
- Community Cards: The cards dealt face-up in the middle of the table that can be used by all players to form their best hand.

- Flop: The first three community cards dealt.
- Turn: The fourth community card dealt.
- River: The fifth and final community card dealt.

Figure 2: Flop, turn, and river

- Showdown: The final phase of the game, where players reveal their hands and the best hand wins the pot.
- Pot: The total amount of chips bet in a hand.
- Poker Hand: A combination of five cards. A hand's ranking determines the player's score.
- Best Hand: The highest scoring poker hand at the end of the betting round.
- Royal Flush: A type of flush with the Ace, King, Queen, Jack, and 10 all of the same suit. This is the highest hand poker.

Figure 3: Royal flush

- Straight Flush: A combination of a flush and a straight where all cards are of the same suit and are in increasing order (eg, 4, 5, 6, 7, 8). This is the second highest hand.

Figure 4: Straight flush

- Four of a Kind: Four of the same card. The fifth card is the highest card in the player's hand or on the table. This is the fourth highest hand.

Figure 5: Four of a kind

- Full House: A three of a kind with a pair. This is the fifth highest hand

Figure 6: Full house

- Flush: Five cards all of the same suit in no particular order. This is the sixth highest hand. If two players both have a flush, the player with the highest card in the flush wins.

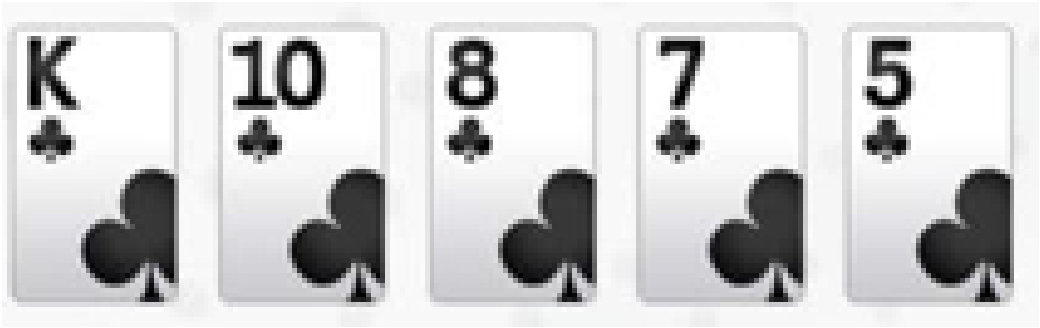

Figure 7: Flush

- Straight: Five cards that are in increasing order but are not of the same suit. Aces can either follow a king (10, J, Q, K, A) or be followed by a two (A, 2, 3, 4, 5) to create a straight. This is the seventh highest hand.

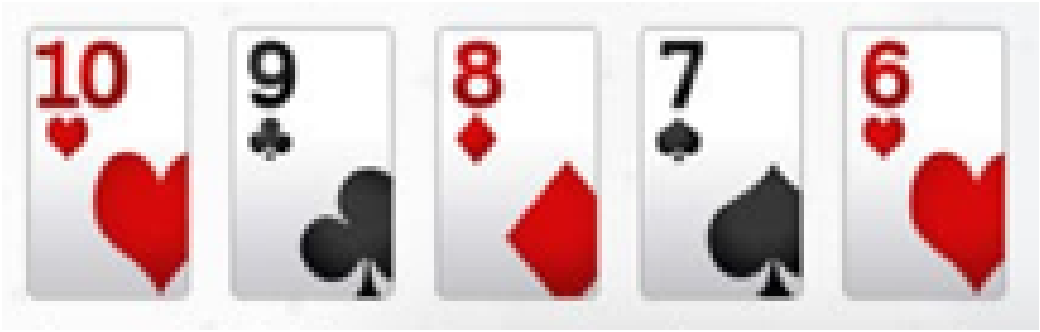

Figure 8: Straight

- Three of a Kind: Three of the same card. The highest two cards complete the hand.

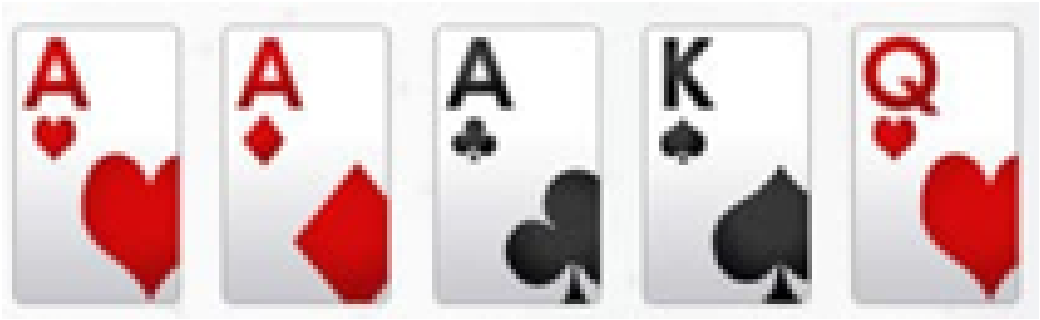

Figure 9: Three of a kind

- Two Pair: Two sets of pairs. The remaining card is the highest card in the player's hand or on the table.

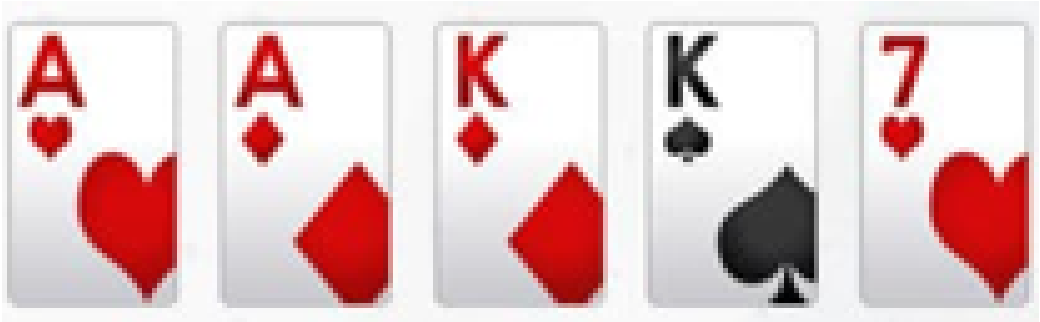

Figure 10: Two pair

- Pair: Two of the same card. The remaining three cards are the three highest available.

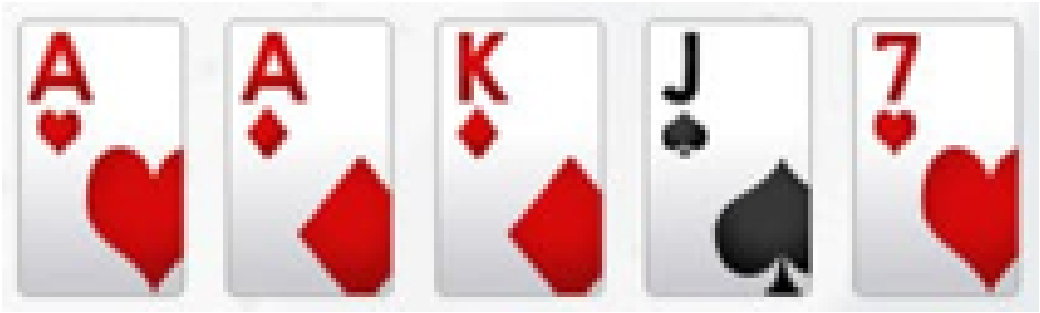

Figure 11: Pair

- High Card: A single card that is high in value (eg. an Ace). This hand occurs when none of the higher hands are available.

Figure 12: High card

1 Poker Client Software Architecture Overview

1.1 Main data types and structures

- **Player**: A struct that contains information about the player, including their name, seat number, points, hand, and whether they are all-in or folded.

```
typedef struct {
    char name[NAME_LENGTH];
    int seat;
    int points;
    Card hand[2];
    int isAllIn;
    int isFolded;
} Player;
```

1.2 Major software components

- **Client Module**: Handles the connection and communication with the server.
- **Game Logic Module**: Implements the game rules and game state management.
- **UI Module**: Manages the user interface and user interaction.

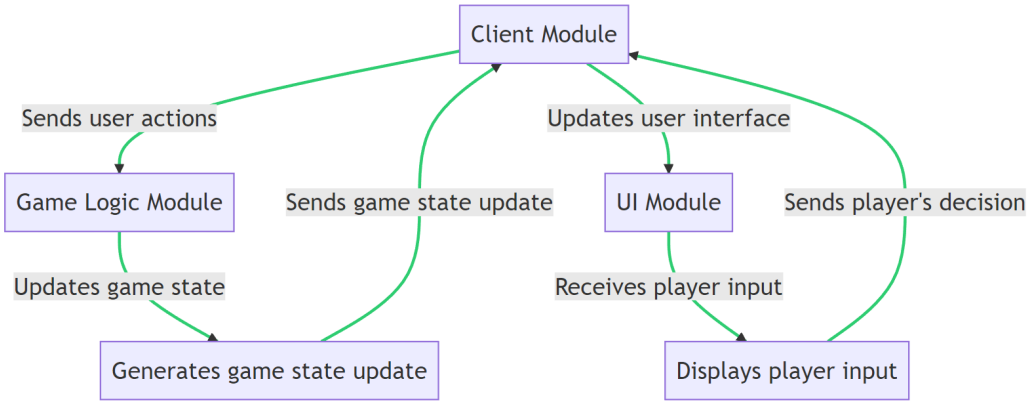

Figure 13: Client modules and their interactions

1.3 Module interfaces

1.3.1 Game Logic Module

- **initPlayer**: Initializes a player with the given name.

```
void initPlayer(Player* player,
    char* name);
```

- **makeBet**: Make a bet of the given amount.

```
void makeBet(Game* game,
    Player* player,
    int betAmount);
```

- **callBet**: Call the current highest bet.

```
 void callBet(Game* game,
    Player* player);
```

- **raiseBet**: Raise the current highest bet by the given amount.

```
void raiseBet(Game* game,
    Player* player,
    int raiseAmount);
```

- **allIn**: Bet all of the player's points.

```
void allIn(Game* game,
    Player* player);
```

- **fold**: Fold the player's hand.

```
void fold(Game* game,
    Player* player);
```

1.3.2 UI Module (still in progress)

- **displayGameState**: Displays the current game state to the user.

```
void displayGameState(const Game* game);
```

- **getUserAction**: Gets the user's action.

```
char* getUserAction();
```

1.3.3 Client Module (still in progress)

- **connectToServer**: Connects to the game server at the given address and port.

```
void connectToServer(const char* address, int port);
```

- **sendAction**: Sends the user's action to the server.

```
void sendAction(const char* action);
```

- **receiveUpdate**: Receives an update on the game state from the server.

```
1 Game* receiveUpdate();
```

1.4 Overall program control flow

The overall control flow of the client program is as follows:

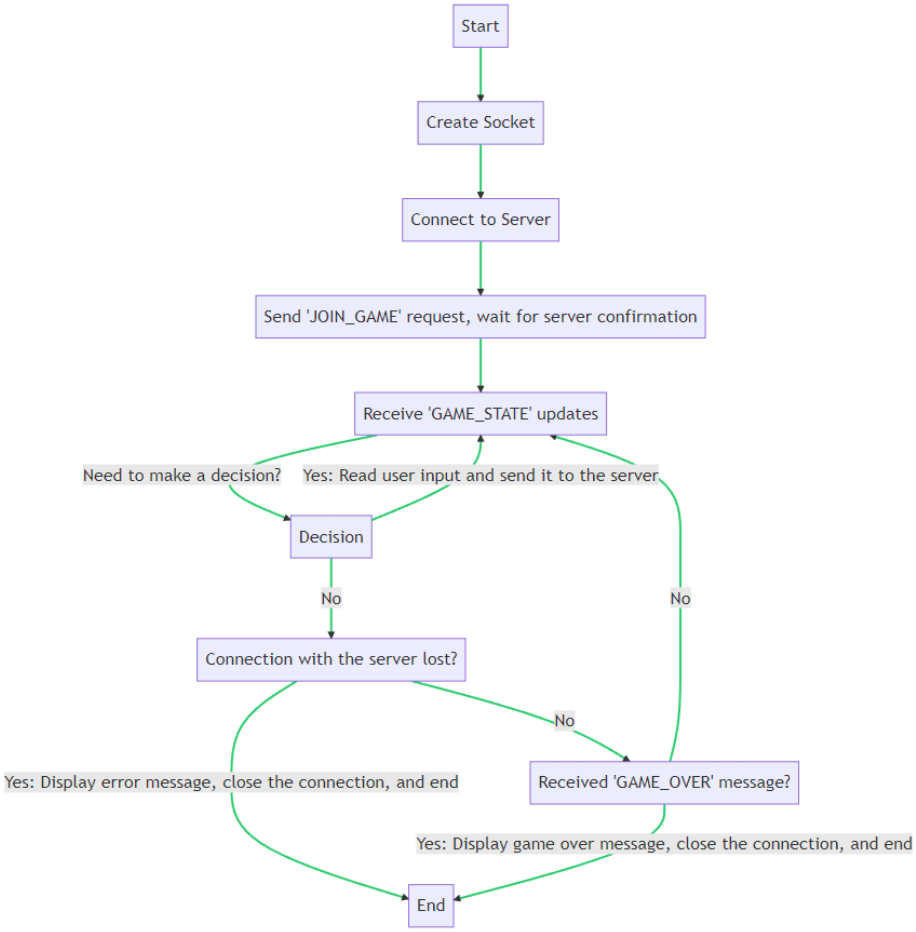

Figure 14: Client control flow

1.5 Automated Client: Poker Bot (still in progress)

The Poker Bot is an automated client that plays the game according to a predefined strategy. Instead of getting user input, it decides on actions based on the current game state and its strategy. Its structure and control flow are similar to the normal client, but with the following differences:

- **getBotAction**: Determines the bot's action based on the current game state and its strategy.

```
1 char* getBotAction(Game* game);
```

2 Poker Server Software Architecture Overview

2.1 Main data types and structures

- **Card**: A struct that contains information about a card, including its rank and suit.

```
typedef struct {
    int rank;
    int suit;
} Card;
```

- **Suit**: An enum that represents the four suits of a deck of cards.

```
typedef enum {
    HEARTS,
    DIAMONDS,
    CLUBS,
    SPADES
} Suit;
```

- **Rank**: An enum that represents the thirteen ranks of a deck of cards.

```
typedef enum {
    RANK_2 = 2,
    RANK_3,
    RANK_4,
    RANK_5,
    RANK_6,
    RANK_7,
    RANK_8,
    RANK_9,
    RANK_10,
    RANK_JACK,
    RANK_QUEEN,
    RANK_KING,
    RANK_ACE
} Rank;
```

- **Deck**: A struct that contains information about a deck of cards, including the cards and the index of the top card.

```
typedef struct {
    Card cards[NUM_CARDS];
    int topCardIndex;
} Deck;
```

- **Game**: A struct that contains information about the game, including the players, community cards, deck, game stage, pot, current bet, and current player.

```
typedef struct {
    Player players[NUM_PLAYERS];
    int numPlayers;
    int totalPlayers;
    int totalRobots;
    int totalRound;
    int initialChips;
    Card communityCards[5];
    Deck deck;
    GameStage stage;
    GameState state;
    int pot;
    int currentBet;
    int currentPlayerIndex;
    int minBet;
    int smallBlindIndex;
} Game;
```

- **GameStage**: An enum that represents the four stages of a game of poker.

```
typedef enum {
    PRE_FLOP,
    FLOP,
    TURN,
```

```
    RIVER,
    POST_RIVER,
} GameStage;
```

- **GameState**: An enum that represents the three states of a game of poker.

```
typedef enum {
    GAME_WAITING_FOR_PLAYERS,
    GAME_STARTED,
    GAME_ENDED,
} GameState;
```

- **Request**: A struct that contains information about a client request, including the player's name, the type of request, and the request parameters.

```
typedef struct {
    char playerName[NAME_LENGTH];
    RequestType type;
    int betAmount;
    int raiseAmount;
    int totalPlayers;
    int robotCount;
    int initialChips;
    int roundNumber;
    int smallBlind;
    int seatNumber;
} Request;
```

- **RequestType**: An enum that represents the ten types of requests that a client can make.

```
typedef enum {
    REQUEST_CREATE_ROOM = 1,
    REQUEST_JOIN,
    REQUEST_GET,
```

```
5       REQUEST_BET,
6       REQUEST_CALL,
7       REQUEST_RAISE,
8       REQUEST_CHECK,
9       REQUEST_ALL_IN,
10      REQUEST_FOLD,
11  } RequestType;
```

2.2 Major software components

- **Game Logic Module**: This module is responsible for managing the game state, including initializing the game, dealing cards, advancing the game to the next stage, and evaluating the cards to determine the winner of the round.
- **Server Module**: This module is responsible for handling client requests and broadcasting updates to all clients. This includes accepting new players, receiving actions from the current player, and sending game updates to all clients.

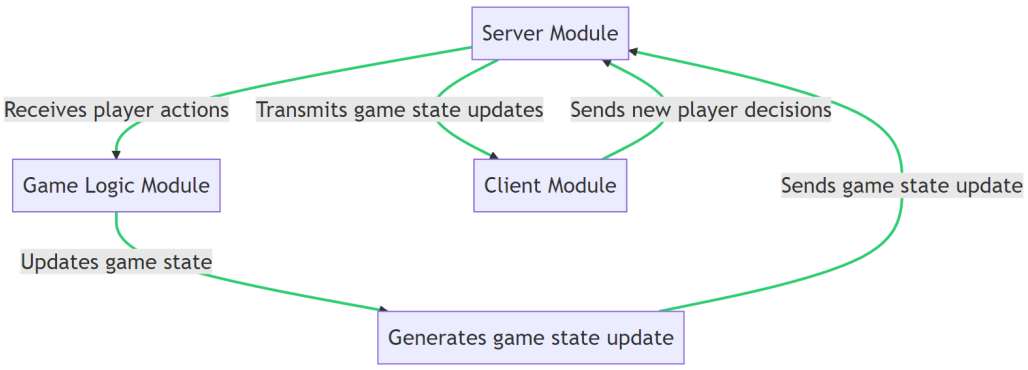

Figure 15: Server modules and their interactions

-

2.3 Module interfaces

2.3.1 Game Logic Module

- **createCard**: Creates a card with the given rank and suit.

```
Card createCard(int rank,
    int suit);
```

- **initGame**: Initializes the game with the given players.

```
void initGame(Game *game);
```

- **dealCards**: Deals cards to the players.

```
void dealCards(Game *game);
```

- **goToNextStage**: Advances the game to the next stage.

```
void goToNextStage(Game *game);
```

- **evaluateCard**: Evaluates the given cards and returns the score of the best hand.

```
int evaluateCard(Card *hand, Card *communityCards);
```

- **createDeck**: Creates a deck of cards.

```
Deck createDeck();
```

- **shuffleDeck**: Shuffles the given deck of cards.

```
void shuffleDeck(Deck* deck);
```

- **drawCard**: Draws a card from the given deck.

```
Card drawCard(Deck* deck);
```

- **isRoyalFlush**: Check if the given cards contain a royal flush.

```
int isRoyalFlush(Card *cards, int numCards);
```

- **isStraightFlush**: Check if the given cards contain a straight flush.

```
int isStraightFlush(Card *cards, int numCards);
```

- **isFourOfAKind**: Check if the given cards contain a four of a kind.

```
int isFourOfAKind(Card *cards, int numCards);
```

- **isFullHouse**: Check if the given cards contain a full house.

```
int isFullHouse(Card *cards, int numCards);
```

- **isFlush**: Check if the given cards contain a flush.

```
int isFlush(Card *cards, int numCards);
```

- **isStraight**: Check if the given cards contain a straight.

```
int isStraight(Card *cards, int numCards);
```

- **isThreeOfAKind**: Check if the given cards contain a three of a kind.

```
int isThreeOfAKind(Card *cards, int numCards);
```

- **isTwoPair**: Check if the given cards contain a two pair.

```
int isTwoPair(Card *cards, int numCards);
```

- **isThreeOfAKind**: Check if the given cards contain a three of a kind.

```
int isThreeOfAKind(Card *cards, int numCards);
```

- **isPair**: Check if the given cards contain a pair.

```
int isPair(Card *cards, int numCards);
```

2.3.2 Server Module (still in progress)

- **FatalError**: Prints the error message and exits the program.

```
void FatalError(const char *ErrorMsg);
```

- **MakeServerSocket**: Makes a server socket that listens on the given port.

```
int MakeServerSocket(uint16_t PortNo);
```

- **ProcessRequest**: Processes the request from the client.

```
void ProcessRequest(int DataSocketFD, Game *game, fd_set *Act
```

- **ServerMainLoop**: The main loop of the server.

```
void ServerMainLoop(int ServSocketFD, Game *game, int Timeout
```

- **parseRequest**: Parses the request from the client and map it to the corresponding structure.

```
Request* parseRequest(const char *msg);
```

- **findPlayerBySocket**: Finds the player name with the given socket.

```
Player* findPlayerBySocket(Game *game, int socket);
```

- **ProcessGameEvent**: Processes the game event after processing the request from the client.

```
void ProcessGameEvent(Game *game, int DataSocketFD);
```

2.4 Overall program control flow

The overall control flow of the server program is as follows:

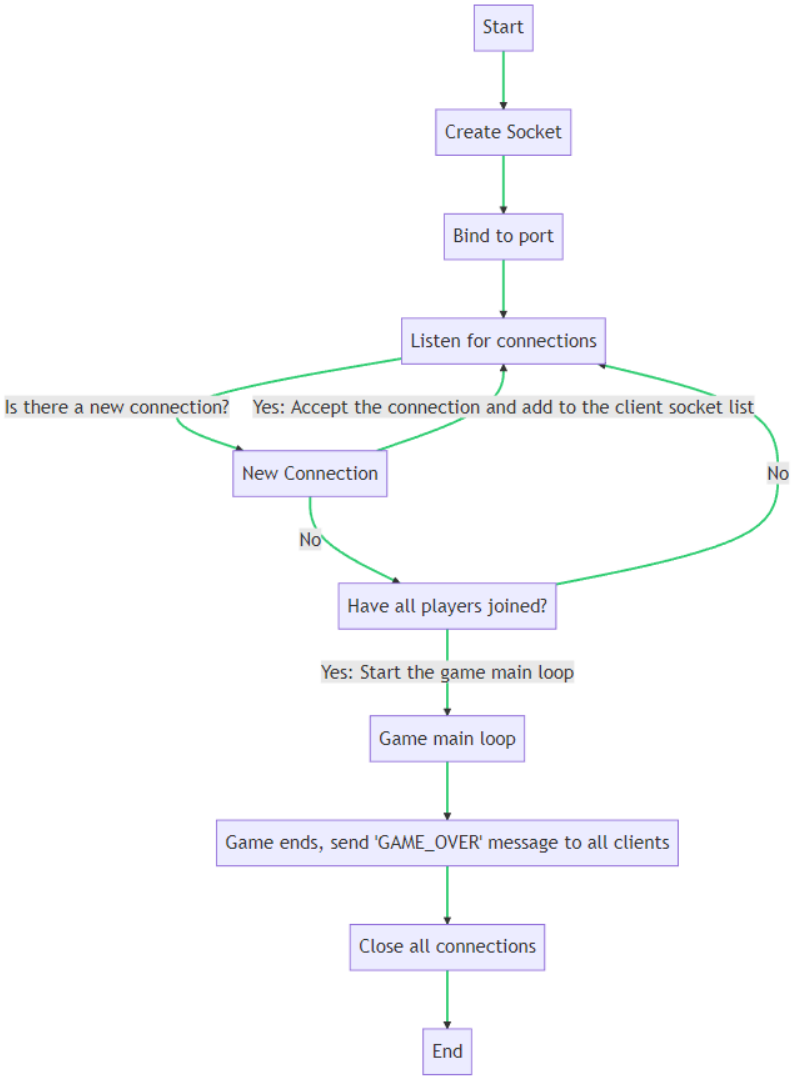

Figure 16: Server control flow

The overall control flow of the game loop is as follows:

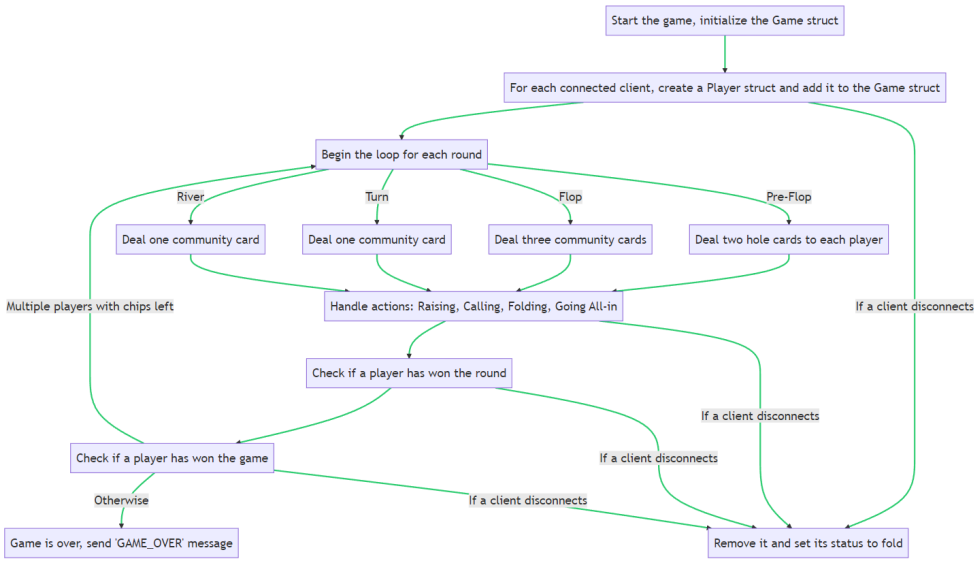

Figure 17: Game control flow

3 Installation

3.1 System requirements

- Processor: 1 GHz or faster
- RAM: 2 GB
- Disk Space: 200 MB
- Operating System: Linux OS (CentOS-7-x86_64)
- Network: Internet connection for multiplayer games

3.2 Setup and Configuration

To install the online poker game, follow these steps:

- Extract files to your desired location.
- Open a terminal and navigate to the program directory.

- Run the "make" command to build the program. This will create an executable file in the program directory.
- Before running the program, ensure that your system meets the network requirements for multiplayer games.
- To start the program, run the executable file from the terminal.
- The program will open in a new window, ready for you to play.

3.3 Uninstalling

To uninstall the program, follow these steps:

- Open a terminal and navigate to the program directory.
- Run the "make clean" command. This will remove the executable file and any object files created during the build process.
- Delete the program directory and any associated files.

4 Documentation of packages, modules, interfaces

4.1 Detailed description of data structures

4.1.1 Card

This struct contains information about a card, including its rank and suit. It consists of two fields:

- **rank**: An integer that represents the rank of the card. This value is in the range 2-14, where 11-14 represents Jack, Queen, King, and Ace, respectively.
- **suit**: An integer that represents the suit of the card. The values 0-3 represent Hearts, Diamonds, Clubs, and Spades, respectively.

4.1.2 Deck

The Deck struct represents a deck of cards. It includes:

- **cards**: An array of Card structs, each representing a card in the deck. The array size is defined by the constant NUM_CARDS, typically 52 for a standard deck.
- **topCardIndex**: An integer representing the index of the top card in the deck. This index changes as cards are drawn from the deck.

4.1.3 Game

The Game struct represents a game of poker. It includes:

- **players**: An array of Player structs, each representing a player in the game.
- **numPlayers**: An integer representing the number of players in the game.
- **communityCards**: An array of Card structs representing the community cards on the table.
- **deck**: A Deck struct representing the deck of cards in the game.
- **stage**: An integer representing the current stage of the game. The values 0-3 represent Pre-Flop, Flop, Turn, and River, respectively.
- **pot**: An integer representing the total number of chips in the pot.
- **currentBet**: An integer representing the current bet that players need to call or raise.
- **currentPlayerIndex**: An integer representing the index of the player whose turn it is to act.

4.2 Detailed description of functions and parameters

- **createCard(int rank, int suit)**: This function creates a card with a given rank and suit. The parameters include an integer for rank and another for suit. Returns a Card struct.

- **initGame(Game *game)**: This function initializes the game with the given players. It accepts a pointer to a Game struct, which it modifies to initialize the game. Does not return a value.

- **dealCards(Game *game)**: This function deals cards to the players. It accepts a pointer to a Game struct, which it modifies to deal two private cards to each player and to set the initial community cards. Does not return a value.

- **goToNextStage(Game *game)**: This function advances the game to the next stage. It accepts a pointer to a Game struct, which it modifies to advance the game stage, deal additional community cards if necessary, and move the turn to the next player. Does not return a value.

- **evaluateCard(Card *hand, Card *communityCards)**: This function evaluates the given cards and returns the score of the best hand. It accepts pointers to two arrays of Cards: one representing a player's private cards and another representing the community cards. Returns an integer score representing the strength of the best possible 5-card hand from the given cards.

- **createDeck()**: This function creates a deck of cards. It has no parameters. Returns a Deck struct.

- **shuffleDeck(Deck *deck)**: This function shuffles a given deck of cards. It accepts a pointer to a Deck struct, which it modifies to randomize the order of the cards. Does not return a value.

- **drawCard(Deck *deck)**: This function draws a card from a given deck. It accepts a pointer to a Deck struct, which it modifies to remove the top card. Returns the Card struct that was drawn.

- **isRoyalFlush(Card *cards, int numCards)**, **isStraightFlush(Card *cards, int numCards)**, **isFourOfAKind(Card *cards, int numCards)**, **isFullHouse(Card *cards, int numCards)**, **isFlush(Card *cards, int numCards)**, **isStraight(Card *cards, int numCards)**, **isThreeOfAKind(Card *cards, int numCards)**, **isTwoPair(Card *cards, int numCards)**, **isPair(Card *cards, int numCards)**: Each of these functions checks if the given cards contain a specific hand rank. They all accept a pointer to an array of Cards and an integer representing the number of cards. Each function returns a boolean value indicating whether the given cards contain the specific hand rank.

4.3 Detailed description of the communication protocol (still in progress)

- The server-client communication is based on a request-response model.
- The client sends a request to the server containing the player's action.
- The client then updates the UI based on the received game state.
- The server processes this request, updates the game state, and sends a response back to the client with the updated game state.
- Socket Initialization: The server starts by setting up a TCP/IP socket and begins listening for client connections on a specified port.
- Multiplexing with Select: The server uses the select system call for I/O multiplexing. This allows a single thread to monitor multiple file descriptors (sockets in this case). When any of the monitored sockets have an event (e.g., client sending a message), the select will return and notify the server.
- Event-Driven Architecture: The server uses an event-driven architecture, where actions are triggered based on events, such as receiving a message from a client or a client closing the connection. This ensures efficient use of resources as threads are not actively waiting for events to occur.
- Game State Management: Each client is associated with a player in the game, and the server maintains the state of the game, such as the

current bet, the pot, the deck, and each player's hand. The game state updates based on actions performed by the players (clients), and the updated state is communicated back to the relevant clients.

- Game Logic Execution: The server also executes the game logic, such as dealing cards, handling bets, and determining the winner. In response to these events, the server sends updates to the clients to inform them about the changes in the game state.
- Disconnection Handling: If a client disconnects, the server handles this gracefully by removing the client from the list of active clients and updating the player state to indicate that the player has folded.

5 Development plan and timeline

5.1 Partitioning of tasks

- **Client**
  - Client Module
  - UI Module
  - Automated Client
- **Server**
  - Server Module
  - Game Logic Module

5.2 Team member responsibilities

- Yongye Li: GUI
- Mary Campbell: GUI
- Arjun Sivakumar: Game Logic
- Joseph Principe: Client/Server Protocol
- Cristian Pina Bravo: GUI
- Zijie Huang: Client/Server Protocol, Game Logic

6 Back matter

6.1 Copyright

- Copyright ©2023 Team rand(). All rights reserved.
- Redistribution or modification of this program is prohibited with the exception of personal/non-commercial use.
- Distribution of this program for commercial use must have our written approval.

6.2 Error Messages

The online poker game includes a robust error-handling system. If an error occurs during gameplay, an error message will be displayed on the screen. The error message will provide a description of the error and, if applicable, suggest corrective actions. Here are some common error messages and their meanings:

- "Network Connection Error": This error occurs when the program is unable to establish a connection with the server. Make sure your internet connection is working and try again.
- "Server Full": This error occurs when the server has reached its maximum number of connected clients. Try again later when a spot becomes available.
- "Invalid Argument Error": This error occurs when the player sends a message to the server that it understands but with an invalid value. The server will reply that the given value is invalid.
- "Invalid Message Error": This error occurs when the player sends a message to the server that it does not recognize. The server will reply that it did not understand the message it received.
- "Invalid Move Error": This error occurs when the player attempts to do an action that they are not allowed to, like performing a check after a bet has been made.

- "Out of Memory Error": This error occurs when the system does not have enough memory to run the program. It is recommended to close other applications and try again.

6.3 Index

- G
  - Goals
  - Glossary
  - graphical user interface (GUI)
- H
  - High Card
- I
  - Installation
- M
  - Multiplayer
  - Makefile
- O
  - Operating System
- P
  - Pair
  - Poker Hand
  - Pot
- R
  - Raise
  - RAM
  - Royal Flush
  - Raise
- S
  - Suit
  - Scoring and Winning Conditions

    - Showdown
    - Small Blind
    - Straight
    - Straight Flush
    - System Requirements
- T
    - Three of a Kind
    - Turn
    - Timer
- U
    - Uninstalling
    - Usage Scenario
    - User Interface
- F
    - Flush
    - Four of a Kind

6.4 References

[1] \Sockets tutorial," Linux Howtos: C/C++ -> Sockets Tutorial, https://www.linuxhowtos.org/C_C++/socket.htm.

[2] GTK+ 2 reference manual: GTK+ 2 reference manual, https://developer-old.gnome.org/gtk2/stable/index.html.