Poker game

(Design Document)

https://github.com/srhee91/PokerGame

Team 3

So Mi Choi, Bo Heon Jeong, Hanchen Li,
Sang Rhee, Yixin Wang, Yuying Wang

Design Document

Purpose:

We are designing a Texas Hold'em game for desktops and laptops designed to be played over a local network with friends. Each game can support up to 8 players and additional spectators with individual GUIs for each person. The GUIs will have simple animations and sounds to display each player's actions.

Non-Functional requirements:

If I want to make casual poker game that can be played with friends,

- Response time:
  - As a user, I would like a low latency connection between my client and the game server so I can see other players' actions in as close to real-time as possible.

- Throughput:
  - As a user, I would like the game server to be able to communicate with up to 20 clients (players or spectators) with no discernible increase in latency. This way, the server will be easily able to communicate with everyone in an ongoing game as well as a significant number of spectators.
  - As a user, I would like the GUI to run at a minimum of 40 frames per second at all times so the game animations appear smooth.

- Resource usage:
  - As a user, I would like this game to be light on resource usage in terms of CPU and network communications. Since the game will not a very computationally-intensive program and there can be a lot of waiting from one turn to the next, I should be able to do other tasks outside of the game while I'm waiting.

- Reliability:
  - As a user, I would like to be able to play multiple sessions of Poker in one sitting without the program crashing or my local game state going out of sync with that of the server.

- Availability:
  - As a user, I would want this program to be able to run on a variety of operating systems including Windows, Mac, and distributions of Linux. This way, I will be able to enjoy the game on most desktops and laptops.

- Recovery from failure:
  - As a user, I would like the game to be able to reconnect players to the game server if their connection is temporarily lost because I don't want to lose all the progress I've made in a game due to a spotty connection.

- Allowances for maintainability and enhancement:
  - As a developer, I would like the game to be written in a clear structure, because I want to modify the code and add more features in the future.

- Allowances for reusability:
  - As a developer, I would like some part of the game such as network and graphic can be reused easily, so we can develop other online poker games easily.

Design Outline:

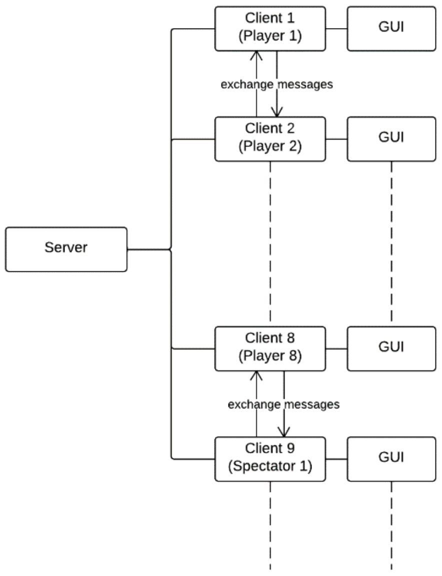
a. We'll be following the client-server model. The major components will be the server (game host), clients (players and spectators), and GUI (one per client).

Server (game host):

One server instance will be created per game. The purpose of the server is to keep track of the poker game state (e.g. chip amounts and cards for each player, the cards on the table, the pot and side pots, player actions). It will poll player clients in order and receive their responses and update the player state according to the user action it received.

Clients (players and spectators):

The client will run on each player's or spectator's systems and will send and receive messages from the server to update its local copy of the game state, which will include just enough information for the GUI of that player or spectator (spectators will have all players' cards face up, while players will only have their cards face up, except in the case of showdown).



GUI (one per client):

The GUI will run on each player's or spectator's system in its own thread. The GUI will be responsible for displaying the onscreen buttons and the actions of players through simple animations and accompanying sounds. The player's button presses will be detected and the corresponding action will be carried out.

b. Player actions will originate from their GUI, which will have methods to relay these actions to the client running on the same system. The client will send a message containing the player's action to the server (which may or may not be running on the same system, depending if this player is the game host or not). The server will then update the poker game state and send a message to the client belonging to the next player in the order of play.

Design Issues:

Issue 1. On which system will the server reside?

Option1: One of the players will run the host code in addition to the client code.
Option2: Use an existing server-hosting or server-rental provider.
Option3: Use an additional system as a dedicated server.

Choice: **option1** - Our game is aimed at people who want to get together and play. For the best usability, we decided not to have the server on a separate system to avoid the need to bring an extra system. Also, by not using a remote third-party server, it improves reliability, reduces latency, and allows players to play with just a LAN connection.

Issue 2. What if a player loses the game?

Option1: He becomes a spectator.
Option2: He may choose to either leave the game or become a spectator.

Choice: **option2** - For best usability, we should allow the player to choose between leaving the game or becoming a spectator when they lose. If the player simply wants to join a different game after the loss, they shouldn't have to become a spectator first and then leave, which requires an extra action for the user.

Issue 3. How should the game's tasks be distributed among processes and threads for the best performance?

Option1: Have the GUI class handle all the actions from the player and the server in one process.
Option2: Have the GUI and client in one process, and have the host in another process
Option3: Have the GUI and client in one process but two threads, and have the host in another process.

Choice: **option3** - In order for the GUI to have a high and consistent frame rate, the GUI should run in its own thread. As for the client and host code in the case of a host player, they should run in separate processes since they can be viewed as two independent programs with separate tasks. When they're both running on the host player's system, they should remain separate so the server-client interaction is similar to those of the other players.

Design Details:

- *Poker* is the main class. It is responsible for initializing the client code and the host code. It will instantiate a *Client* object every time, and it will instantiate a *Host* object only if the player wants to create a new game.

- *Host* is the class that's responsible for running the server of the game. It holds the overall game state (*PokerGameState* class*)*, handles all the calculations (*GameSystem* class*)*, and communicates to all the client (*HostMessageHandler* class*).*

- *GameSystem* is the class that has static methods for computing the next game state such as distributing pot, shuffling, and calculating the best hand.

- *HostMessageHandler* and *ClientMessageHandler* are classes contained in a *Host* instance and a *Client* instance respectively. They're responsible for the socket communication between a server and client, which consists of messages containing the game state or a player's action.

- *Client* is the class that's responsible for the client side of the game. It holds a copy of the local copy of the game state (*PokerGameState* class), handles communications from the host (*ClientMessageHandler)*, and contains code for the GUI *(GUI* class).

- The *GUI* class holds and controls the classes representing the four game modes.

- The *BasicGameState* class is a superclass for the four game mode classes. It will provide common functions, including initialize GUI, updating data, rendering screen.

- The *StartMode* class provides the interface when the user opens the application at the beginning. It will allow the user to input a name and to either create, join, or spectate a game (the game is specified by entering a port number into a textbox).

- The *LobbyMode* class provides the interface when the user joins an existing game. It shows the names of all the players who have joined and new players as they join. The host can start the game through a "Start" button when there are enough players.

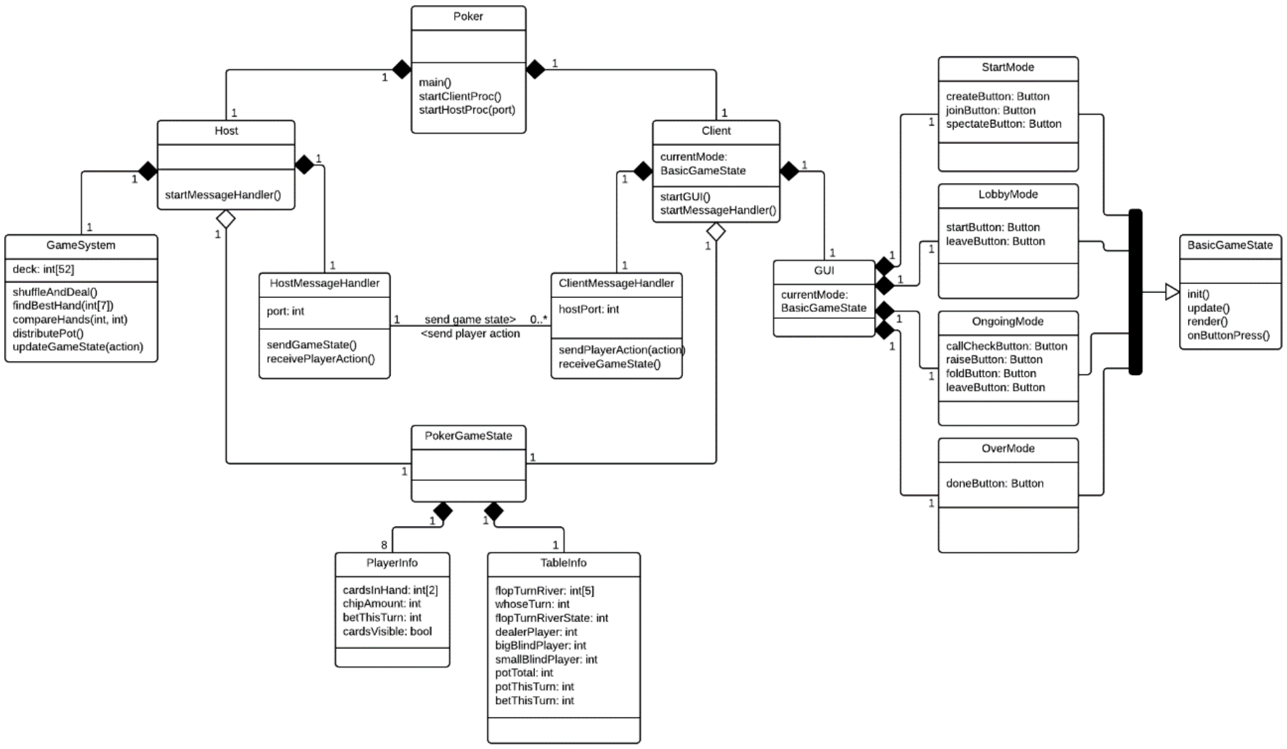
- The *OngoingMode* class provides the interface for the actual game as it plays. It shows each player's cards (face up or face down), the flop, turn, river, the pots and side pots, the chip amounts, etc. It also provides buttons for user actions like raise or fold.

- The *OverMode* class provides the interface for when the player loses or the game ends. It will show game stats and options to either become a spectator (if the game is still going) or return to the start screen.

- *PokerGameState* is a structure class for saving game state variables, which will be used in the *Host* and *Client* classes. It contains one instance of the *TableInfo* class and eight instances of the *PlayerInfo* class.

- The *TableInfo* class is a structure class for saving variables used for table information, such as player ID, dealer, big blind, small blind, card.

- The *PlayerInfo* class is a structure class for saving variables used for player such as the cards in their hand and their chip amount.



Poker
main()
startClientProc()
startHostProc(port)
Host
startMessageHandler()
Client
currentMode: BasicGameState
startGUI()
startMessageHandler()
GameSystem
deck: int[52]
shuffleAndDeal()
findBestHand(int[7])
compareHands(int, int)
distributePot()
updateGameState(action)
HostMessageHandler
port: int
sendGameState()
receivePlayerAction()
send game state>
<send player action
0..*
ClientMessageHandler
hostPort: int
sendPlayerAction(action)
receiveGameState()
GUI
currentMode: BasicGameState
StartMode
createButton: Button
joinButton: Button
spectateButton: Button
LobbyMode
startButton: Button
leaveButton: Button
OngoingMode
callCheckButton: Button
raiseButton: Button
foldButton: Button
leaveButton: Button
OverMode
doneButton: Button
BasicGameState
init()
update()
render()
onButtonPress()
PokerGameState
8
PlayerInfo
cardsInHand: int[2]
chipAmount: int
betThisTurn: int
cardsVisible: bool
TableInfo
flopTurnRiver: int[5]
whoseTurn: int
flopTurnRiverState: int
dealerPlayer: int
bigBlindPlayer: int
smallBlindPlayer: int
potTotal: int
potThisTurn: int
betThisTurn: int