**Termination Project: Online Poker Game Design**

**Hao Fu**

**B00773431**

**Abstract**

As a very classic form of game, card games have always been popular, even in video games. Among the many card games, Texas Hold'em is the most famous one. In this project I designed and developed a 3-player online Texas Hold'em game.

This game is developed in C#, using Visual Studio 2019 and Unity Engine. The design is basically divided into three parts, server side, data base and client side. The server is responsible for data communication with the client and the database and handles the key logic in the game as well. The client is mainly responsible for the implementation of the game UI and data communication with the server. The role of the database is to store persistent data for the game.

This paper will introduce the design of the entire game from the above three parts. Important code will be shown for better explanation.

1. **Server**
   1. **Lower-level Framework Design.**

The lower-level framework of the server uses Socket class, the built-in API of C#. There are two main programming modes of the server: synchronous mode and asynchronous mode. In this project, I adopted the asynchronous programming idea. The advantage of asynchronous mode is to avoid blocking. There are two common asynchronous non-blocking models. One is a single-threaded event + callback model represented by node.js, and the other one is a coroutine method represented by python language. Here I used the first method, because the size of the project itself is small, and the efficiency of the server is not considered. Using the event + callback method to design is more intuitive and easier to develop. I will introduce some of the most important classes inside the framework.

* + 1. **ServerPeer Class**

The serverPeer class is an abstraction of the server itself. It consists of the following parts: Socket, client object pool, network message distribution center (application).

        private Socket serverSocket;

        private ClientPeerPool clientPeerPool;

        private IApplication application;

Socket is responsible for establishing a network port corresponding to the server for client objects to connect. Once the socket is successfully established, it will keep listening to requests from the client; the client object pool is used to improve the reuse efficiency of client objects. Creating objects every time is very performance consuming. The idea of the object pool is to create all the objects in advance, and only modify the value of the object when using it in the future. I will often use the idea of the object pool in the rest of the development.

Once the server is started, it will keep receiving client connection requests. I use an event + callback model. The code is as follows:

        private void StartAccept(SocketAsyncEventArgs e)

        {

            if (e == null)

            {

                e = new SocketAsyncEventArgs();

                e.Completed += E\_Completed;

            }

            bool result = serverSocket.AcceptAsync(e);

            if (result == false)

            {

                ProcessAccept(e);

            }

        }

        private void E\_Completed(object sender, SocketAsyncEventArgs e)

        {

            ProcessAccept(e);

        }

After the above process, the connection between the client and the server is established, and then the server will always monitor various requests from the client. In order to avoid the problem of sticking packets, I defined the length of the data in the header of the network packet. Therefore, the server will first unpack, and then distribute the data to the application layer, which is the network message center.

* + 1. **ClientPeer**

The ClientPeer is an abstraction of clients to server. Once a connection is established, server will maintain a ClientPeer object for this client. A ClientPeet mainly consists of the following parts: sockets arguments and a message buffer for communicate between server and client. The most important thing it does is to process receive and send net message.

        private void ProcessData()

        {

            isProcessingReceive = true;

            //decode packet

            byte[] packet = EncodeTool.DecodePacket(ref cache);

            if (packet == null)

            {

                isProcessingReceive = false;

                return;

            }

            NetMsg msg = EncodeTool.DecodeMsg(packet);

            if (receiveCompleted != null)

            {

                //delegate

                receiveCompleted(this, msg);

            }

            //loop check until queue is empty

            ProcessData();

        }

        public void SendMsg(int opCode, int subCode, object value)

        {

            msg.Change(opCode, subCode, value);

            byte[] data = EncodeTool.EncodeMsg(msg);

            byte[] packet =  EncodeTool.EncodePacket(data);

            SendMsg(packet);

        }

* + 1. **ThreadSafe Design**

Threadsafe is very important in server development because there can be multiple clients communicating with server at the same time. In another word, there is race condition everywhere. There are two classes to make sure thread safety: ThreadSafeInt and SingleExecute.

* + - 1. **ThreadSafeInt**

This class is used to generate thread-safe integers in order to get the object ID for higher-level logic development. It maintains only an interger inside an object. It will apply for a lock whenever there is a write operation to the interger.

* + - 1. **SingleExecute**

This class is also used for higher-level logic operation. I designed it as a singleton to make sure there is only one instance throughout the whole lifetime of the server. Whenever we need to do anything threadsafe, we apply for a lock, and call a delegate for the corresponding operation.

* + 1. **NetMsg**

NetMsg class is the net message between server and client. It consists three parts: opcode, subcode and value.

Opcode represents three main modules of the game: Acount, Match and Fight. Subcode subdivides the specific operations of each module. Value is the data. I will introduce module in detail in the later context.

* + 1. **TimerTool**

The game needs to time the player's operations. Because this is an online game, the server cannot wait for a certain player forever. If the player does not operate for a certain period of time, the server will perform the default operation for him and move to the next player.

* + - 1. **TimerModel**

It declares a delegate. When the time for a certain player is up, the delegate is called.

* + - 1. **TimerManager**

There could be more than one TimerModel objects. Therefore, we need a manager for these instances. It is designed as a singleton to make sure thread safety.

It maintains a object queue, and check the queue every 1000ms. If any timerModel object’s time is up, its delegate will be called.

        public TimerManager()

        {

            //1000ms

            timer = new Timer(1000);

            //calls every 1000 ms

            timer.Elapsed += Timer\_Elapsed;

            timer.Start();

        }

        private void Timer\_Elapsed(object sender, ElapsedEventArgs e)

        {

            //iterate tasks

            foreach (var item in idModelDic.Values)

            {

                //if dateTime is beyond the time we set before

                if(DateTime.Now.Ticks >= item.time)

                {

                    //delegate

                    item.Run();

                }

            }

        }

* 1. **Network Protocol**

In order for the client and server to communicate , we need to develop a set of protocols. Protocol is a class library. It is a library shared between server and client. It is used to encode and decode NetMsg. As we mentioned before, a NetMsg has an opcode, a subcode and value. For the value, I developed a set of data transfer objects to carry the data.

* + 1. **OpCode**

**There are three types of OpCode: account, match and fight. AccountCode is related to the user's account operations, such as login and registration. MatchCode is related to user matching in the lobby. FightCode is the related operation of the user in the game. We will discuss in more detail later.**

* + 1. **Constant**

Consant contains some enumerators constants in the game. These definitions are related to the rules of the game.

We have Identity, which is used to indicate the identity of the player in this game: small blind, big blind or dealer. The other one is RoundType, representing the current round of the game.

* + 1. **DTO ( Data Transfer Object )**

**There is a set of DTOs for different purposes.**

* + - 1. **UserDTO**

UserDTO is related to user login and register. It consists of the user’s id, username and the coin the user has.

* + - 1. **RankItemDTO and RankListDTO**

These two DTOs are two DTOs supporting the leaderboard feature in the game.

* + - 1. **PlayerDTO**

PlayerDTO is related to the game play. It consists of the player’s hand, the community cards, etc.

* + - 1. **MatchRoomDTO**

MatchRoomDTO is related to users’ matching.

* + - 1. **AccountDTO**

AccountDTO is related to users’ account.

* + - 1. **BetDTO**

BetDTO consists of information of a single bet being placed.

* + - 1. **CardDTO**

CardDTO is related to the information of a card such as the value and the suit to the card.

* + - 1. **GameOverDTO**

GameOverDTO consists of information from server to client when a game is over and client need to know what action to take.

* 1. **GameServer**

The two we mentioned above are class libraries, GameServer is the runnable program. It implemented the higher-level logic of the server. It consists of 3 parts: NetMsgCenter, Handlers to corresponding message and DatabaseManager. I designed an IApplication interface in Server class library, and NetMsgCent, the net message center will implement the interface; There is a set of handlers for the message center as specific services are concerned.

    public void Receive(ClientPeer client, NetMsg msg)

        {

            switch (msg.opCode)

            {

                case OpCode.Account:

                    accountHandler.Receive(client, subCode, value);

                    break;

                case OpCode.Match:

                    matchHandler.Receive(client, subCode, value);

                    break;

                case OpCode.Fight:

                    fightHandler.Receive(client, subCode, value);

                    break;

                default:

                    break;

            }

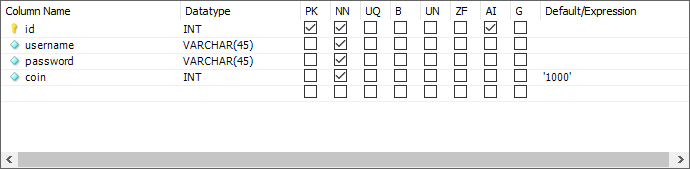
        }

DatabaseManager communicates with database. In the following chapters, I will introduce the remaining server logic combined with client logic in detail from a respective of game play.

1. **Client & Server Logics**
   1. **Database Design**

The first thing to do as a user open the game is to login or create a new account, which is register. Considering the characteristics of this game, the database we need requires the following three types of data: username, password, and the number of coins owned by the user. In addition, we also need the user's ID as a unique identifier and the primary key of the table.

The database management system I chose is MySql. The table is design as below:



* 1. **EventCenter**

On client side, the network message sent by the server(or service within client itself ) may be responded by multiple modules at the same time, so if we call the methods of different modules immediately, the code coupling will be very high and it is not easy to maintain. In order to reduce the coupling degree of the code, I designed a message center model. When a certain service of a certain module needs to be processed by other modules, the corresponding event code is broadcasted. In this way, the module that monitors this event code will Receive the broadcast and process it accordingly.

//dictionary

private static Dictionary<EventType, Delegate> m\_EventTable = new Dictionary<EventType, Delegate>();

//add to dictionary

private static void OnListenerAdding(EventType eventType, Delegate callBack);

//get the corresponding call back to a event

private static void OnListenerRemoving(EventType eventType, Delegate callBack);

//remove from dictionary

    private static void OnListenerRemoved(EventType eventType);

    public static void AddListener(EventType eventType, CallBack callBack)

{

        OnListenerAdding(eventType, callBack);

//add a call back to the event

        m\_EventTable[eventType] = (CallBack)m\_EventTable[eventType] + callBack;

    }

    public static void RemoveListener(EventType eventType, CallBack callBack)

    {

        OnListenerRemoving(eventType, callBack);

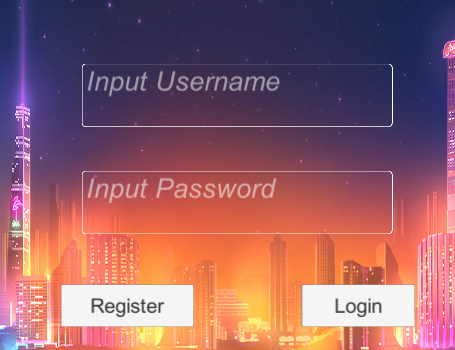
//remove the call back from the event

        m\_EventTable[eventType] = (CallBack)m\_EventTable[eventType] - callBack;

        OnListenerRemoved(eventType);

    }

* 1. **User Register & Login**



* + 1. When the user starts the game, if he plays for the first time, he must create a new account, so after opening the registration UI window, enter the username and password, and then click register. At this time, the client will send a network request to the server.

AccountDTO accountDTO = new AccountDTO(inputUsername.text, inputPassword.text);

NetMsgCenter.Instance.SendMsg(OpCode.Account, AccountCode.Login\_CREQ, accountDTO);

Here we use AccountCode in Opcode, subCode is Login, and AccountDTO is used as data. AccountDTO only contains the username and password.

When the server receives this message, because the OpCode is the Account Code, the message is distributed to the AccountHandler. In the AccountHandler, OnReceive distributes the message to the corresponding method.

      public void Receive(ClientPeer client, int subCode, object value)

        {

            switch (subCode)

            {

                case AccountCode.Register\_CREQ:

                    Register(client,value as AccountDTO);

                    break;

                case AccountCode.Login\_CREQ:

                    Login(client, value as AccountDTO);

                    break;

……

}

Inside Register method, the server will communicate with database, it will check whether the username exists or not, while the databaseManager executes corresponding SQL command to database. If not, it will create a new data in database and send back a respond to client.

if (DatabaseManager.IsUsernameExist(accountDTO.username))

{

    client.SendMsg(OpCode.Account, AccountCode.Register\_SRES, -1);

    return;

}

DatabaseManager.CreateUser(accountDTO.username, accountDTO.password);

    client.SendMsg(OpCode.Account, AccountCode.Register\_SRES, 0);

* + 1. Login is similar to Register except that server has to check if the user is already online.

if (DatabaseManager.IsUsernameExist(accountDTO.username)==false)

{

    client.SendMsg(OpCode.Account, AccountCode.Login\_SRES, -1);

         return;

}

if (DatabaseManager.IsMatch(accountDTO.username, accountDTO.password) == false)

{

    //password incorrect

    client.SendMsg(OpCode.Account, AccountCode.Login\_SRES, -2);

          return;

}

if (DatabaseManager.IsOnline(accountDTO.username))

{

      //account is online

      client.SendMsg(OpCode.Account, AccountCode.Login\_SRES, -3);

           return;

}

    DatabaseManager.Login(accountDTO.username, client);

    client.SendMsg(OpCode.Account, AccountCode.Login\_SRES, 0);

* 1. **Leaderboard**



When the user successfully logs in, he enters the game lobby. In the game lobby, users can view the current server leaderboard.

The logic is simple: Client sends a request to server. When server receive the request, it communicate with database and get a list of users descend ordered by coin.

* 1. **Start & Match**

When the user clicks the Start button, the game will let the player choose a room with different ante bets. After the selection is made, the game will enter the next stage: the matching stage. On the client side, the user will enter the gambling scene and send a request to the server side. The MatchHandler on the server side will respond to the request and process accordingly.

* + 1. The match handling logic in the server involves MatchRoom and MatchCache. MatchRoom is a class abstracted from matching rooms. Each MatchRoom instance manages the players in the current room. In MatchCache, in order to improve object reuse and server efficiency, I still use object pool technology. When all players in a matching room leave, the object enters the pool, waiting to be used by other users.
    2. When a user enters, leaves or gets ready in a match room, the server will broadcast each client ( except for this particular client ) over the event.

room.Broadcast(OpCode.Match, MatchCode.Enter\_BRO, userDTO, client);

* + 1. When everyone in a match room is ready, the game shall start. MatchHandler will call a delegate to move to FightHandler to manage the game.

//if everyone is ready, the game shall start

if (room.IsAllReady())

{

   startFight(room.ClientList, roomType);

   room.Broadcast(OpCode.Match, MatchCode.StartGame\_BRO, null);

   matchCacheList[roomType].DestroyRoom(room);

}

* 1. **Fight**

The fight handling is the most complicated one. It consists of CardLibrary, FightCache, FightRoom and RoundModel. CardLibrary maintains a deck of cards and is in charge of dealing cards and shuffling. Every FightRoom instance possess one instance of CardLibrary, and the FightCache manages all FightRooms. FightRoom manages current players inside the room and an instance of RoundModel, which is responsible for keeping track of current betting player and current round the game is in.

* + 1. **Deal Card**

When the game starts, FightRoom will deal cards to the players in the room and the community cards. These data are stored in the PlayerDTO, and then distributed to the players in the room by broadcasting. Once the client receives the response, it obtains the card of its own player according to the ID and processes the corresponding UI logic.



* + 1. **GetHandRank**

The server knows each player's hand and community cards after the cards are dealt. Therefore, we need to design a mechanism so that the server can measure the quality of the player's hand, in order to compare each player's hand after the game is over. I used a double variable to quantify the quality of a player’s hand.

public double handRank;

* + - 1. **EvaluateRankByHighestCards() method**

Given a K sorted arrays with integers between 2 and 14, score it so values with high indexes will always be preferred on lower ones.

            int i = 0;

            double sum = 0;

            int fixedSize = cards.Count() - 1;

            for (int j = fixedSize; j >= 0; j--)

            {

                int cardValue = cards[j].Value;

                if (cardValue == excludeCardValues)

                    continue;

                int normalizedValue = cardValue - 2;

// since CardValue is an integer between [2,14]

                sum += normalizedValue \* Math.Pow(13, fixedSize - i);

                if (i == limitCheck - 1)

                    break;

                i++;

            }

            return (double)sum / normalize;

        }

* + - 1. **To get the hand rank**

We need to know some characteristics about player’s hand.

           int dupCount = 1, seqCount = 1, seqCountMax = 1;

           int maxCardValue = -1, dupValue = -1, seqMaxValue = -1;

         int currCardValue = -1, nextCardValue = -1;

           int currCardSuit = -1, nextCardSuit = -1;

* + - * 1. dupCount: Count of duplicate cards
        2. seqCount: Count of sequential cards
        3. seqMaxValue: max card value of the sequence(if any)
        4. curr\* and next\* are used in a for loop, after the for loop we shall get everything we need.
        5. Duplicates are stored inside:

List<double[]> duplicates = new List<double[]>();

* + 1. **Hand Rank Enumeration**
       1. **Royal Flush: 900**

This is the easiest situation to be checked because the cards are constant, we need to check that the values are: 10, J, K, Q, A and the suits of every one of those cards equal identical.

* + - 1. **Straight Flush Group Range: [800, 900)**

We'll go through every suit and check if its count bigger than 4, if it does, we'll make another check if the numbers are in ascending order. EvaluateRankByHighestCards() the highest of the cards.

* + - 1. **Full House Group Range: [600, 700)**

Main check: if duplicates.Count > 0, it can occur for example if we have 22 and 55,, and the 2nd condition is if the count of duplicates is 3 and then 2, so it will happen in case of 22255 for example. EvaluateRankByHighestCards() the cards.

* + - 1. **Flush Group Range: [500, 600)**

We walk through every cardList.Where(x => x[1].Equals(suit)); which means every group of cards collected by their suit, and if the count is bigger or equals to 5, we'll save it as suitCards. We will check if suitCards.Count >= 5; EvaluateRankByHighestCards() the highest of the cards.

* + - 1. **Straight Group Range: [400, 500)**

We already have a variable from the pre-evaluation that saved for us the sequence max count, and if it is bigger or equals to 5 we'll check of the highest card in the sequence to evaluate the rank's hand.

* + - 1. **Three of a kind Group Range: [300, 400)**

We will check if duplicates.Count >= 1 and that the duplicates amount is 3. EvaluateRankByHighestCards() the rest of the cards.

* + - 1. **Two Pairs Group Range: [200, 300)**

We'll check for two duplicates and find the highest duplicate’s value. EvaluateRankByHighestCards() the rest of the cards.

* + - 1. **Pair Group Range: [100, 200)**

There's only one pair, otherwise we would have entered the "Two Pair" case. EvaluateRankByHighestCards() the rest of the cards.

* + - 1. **High Card Group Range: [0, 100)**

Otherwise, if we couldn't find anything else, we'll just EvaluateRankByHighestCards() the rest of the cards.

* + 1. **Call & Raise**

When a player calls, it means that he wants to make his bet equal to others’ bet. Server will update the pot variable of FightRoom, update the player’s coin in database and broadcast every player the event through BetDTO. When a player raises, it is similar except the player wants to make his bet X times as much as others. When everyone’s bet is the same, the game will move to the next round, which could be the flop, the turn, the river depending on the situation.

* + 1. **Turn**

Turn is a method to make players take turn to place bet.

Once Turn is called, it will firstly find the player that should player next and add a timer task upon the player. There is a timer on client side as well but it is to remind the user. It is actually the server side to decide when to take default action.

timerClient = DatabaseManager.GetClientPeerByUserId(nextId);

TimerManager.Instance.AddTimerEvent(60, TimerDelegate);

Console.WriteLine("current betting player: " + client.Username);

room.Broadcast(OpCode.Fight, FightCode.StartBet\_BRO, nextId);

* + 1. **Fold & Leave**

When a player fold or leave, it means that he is not in game anymore. Therefore, the Turn method should skip him. And server will also broadcast the event.

* + 1. **GameOver**

When the current round is the river and it needs to move on, the game shall be over. The server will first figure out the winners by each player’s hand rank, then update database and broadcast information about winners and losers through GameOvereDTO. Next, if nobody leaves, the game restarted as button moves. Otherwise, player can choose either go to match room or back to lobby.

1. **References**
   * 1. <https://github.com/danielpaz6/Poker-Hand-Evaluator>
     2. <http://www.sikiedu.com/my/course/356>