Faculty of Health, Science and Technology

Hugo Andersson
Simon Johansson

Chicago

A multiplayer card game based on Client – Server architecture

Computer science
C-level thesis

Date/Term: 13-06-05
Supervisor: Thijs Jan Holleboom
Examiner: Donald F Ross
Serial Number: C2013:12

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Chicago

A multiplayer card game based on Client – Server architecture

Hugo Andersson

Simon Johansson

© 2013 Hugo Andersson, Simon Johansson, Karlstad University

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Hugo Andersson

Simon Johansson

Approved, 2013-06-05

Advisor: Thijs Jan Holleboom

Examiner: Donald F Ross

Abstract

Chicago is a game based on a combination of poker and trick card games. We wanted to implement this game as a network based multi user system based on client – server architecture. To do this we had to create a client that would handle the interaction with the user, and a server that would handle all the clients. Users should be able to create, join, leave and play a game of Chicago. For the clients we had to create a GUI, handle user interaction and communicate with the server. The server had to be able to manage all the clients and games currently in the system. For every game there had to be a game procedure with specific game logic. To have the clients and server be able to communicate we had to decide how to structure the data sent. Both the client and server needed to use threads to be able to have multiple actions running at the same time. When the server and client had been created we tested the system with help of other users which gave us a different degree of feedback that was used to improve the system or gave us ideas for future upgrades.

Contents

1 Introduction ... 1

1.1 Multiplayer games online ... 1

1.2 Client –Server model ... 1

1.3 Disposition ... 3

2 Background ... 5

2.1 Chicago game ... 5

2.2 System Design ... 9

2.2.1 Client design
2.2.2 Server design
2.2.3 Game
2.2.4 Lobby

2.3 GUI Design ... 11

3 Client implementation ... 15

3.1 Lobby class ... 15

3.1.1 Lobby Constructor
3.1.2 New Game
3.1.3 Join game
3.1.4 Leave game
3.1.5 Update game
3.1.6 Game start

3.2 Lobby session class ... 19

3.2.1 Connect
3.2.2 Read message from server
3.2.3 Sending a message to server

3.3 Chicago class ... 22

3.3.1 Printing cards
3.3.2 User input
3.3.3 Print score
3.3.4 Opponent locations

3.4 Game session class ... 27

3.4.1 Init game
3.4.2 Changing cards
3.4.3 Trick game

3.5 Threads ... 29

3.6 Sockets ... 31

4 Server implementation 33
4.1 Deck 33
4.2 Hand 35
4.2.1 Sort hand
4.2.2 Find position in hand
4.2.3 Calculate hand strength
4.2.4 Compare hands
4.3 Other useful data structures 37
4.4 Chicago game procedure (CGP) 37
4.5 Lobby 41
4.5.1 Select
4.5.2 Client requests
4.5.2 Update game
4.5.3 Start game
4.5.4 Disconnect
4.6 Threads 44
4.7 Sockets 44
5 Client – Server communication 47
5.1 Lobby mode 47
5.1.1 Client lobby messages
5.1.2 Server lobby messages
5.2 Game mode 48
5.2.1 Game initiation
5.2.2 Changing phase
5.2.3 Trick phase
6 Feedback and future implementations 51
6.1 Chat 51
6.2 Settings 52
6.3 Timer 52
6.4 Feedback 53
7 Languages & Tools 55
7.1 C# and Visual Studio 55
7.2 C language 55
7.3 Putty and WinSCP 55
8 Result and evaluation 57
9 Conclusion 59
References 60

List of Figures

Figure 1: Client – Server model 2
Figure 2: Score system 5
Figure 3: Login screen 6
Figure 4: Creating a new game 7
Figure 5: Changing cards 8
Figure 6: Playing a trick card 9
Figure 7: Lobby window 12
Figure 8: Game window 13
Figure 9: Game string representation 15
Figure 10: Game list add or update 18
Figure 11: Starting the Chicago game window 18
Figure 12: Read message from server 20
Figure 13: Reading a list of names from server 21
Figure 14: Drawing a card 22
Figure 15: Calculate player location 23
Figure 16: Print score message 25
Figure 17: Print hand string 25
Figure 18: Opponents location 27
Figure 19: Run game thread 27
Figure 20: Next turn condition 29
Figure 21: Creating a thread (C#) 29
Figure 22: Changing controller thread safe 30
Figure 23: Creating socket (C#) 31
Figure 24: Connecting to a socket (C#) 31
Figure 25: Create network stream 31
Figure 26: Writing to network stream 31
Figure 27: Read all bytes 32

Figure 28: Calculate card 33
Figure 29: Calculate suit and value 33
Figure 30: Shuffle deck algorithm 34
Figure 31: Flow chart 39
Figure 32: Select function 41
Figure 33: Game start 43
Figure 34: Thread declaration 44
Figure 35: Creating a thread (C) 44
Figure 36: Creating a server socket 45
Figure 37: Listen on socket 45
Figure 38: Accepting a client 45
Figure 39: Chat model 52

1 Introduction

Chicago is a popular card game [11] that is a combination of a poker game and a trick game. In poker the goal is to get a hand with as high value as possible such as pairs, straights, flushes etc. A trick game is when one of the players plays a card and the others play a card in the same suit if possible. The player that played the highest card in the same suit as the first player will win the trick and start the next one. The Chicago game, as opposed to other card games, has been chosen for this project because we have not found a single distributed implementation of this game.

1.1 Multiplayer games online

As internet use has become more widespread, new possibilities have arrived. Today people are connected via their computers and phones. As internet usage has increased, online multiplayer games [13] have emerged. There are different kinds of games such as MMORPG (massively multiplayer online role-playing game) [2], FPS (first person shooter) [9], and RTS (real time strategy) [18], but we will discuss another genre of games, card games. People have been able to play card games online since the 1990s with games as spades and hearts, and at the beginning of the 21th century there was an explosion of poker clients. We wanted to add the Chicago card game to the list of online games. A big difference with Chicago as opposed to other poker games is that no money is involved. Chicago is played with the objective of socializing with other players and to have fun.

1.2 Client –Server model

When creating a network based computer system there are several models to choose from. The two most common are the Client – Server model [1] and the Peer to Peer (P2P) [19] model. In the Client – Server model, see Figure 1, there is a central point, the server, through which all clients communicate, while in the P2P model there is no central point and the clients communicate directly with each other. When we designed our multiplayer card game we chose to use the client server model. A reason for this is to remove the game logic from the clients since we want the client role to be a user interaction and graphical representation. To give an example of this we can think about the deck of cards used in a game. This deck is

used for all the clients involved in the game and it is logical that this deck should be on a central spot in the game system. We can think of the server as a dealer that will give the players their cards, replace the cards and decide what player has the best hand and also keep track of the score.

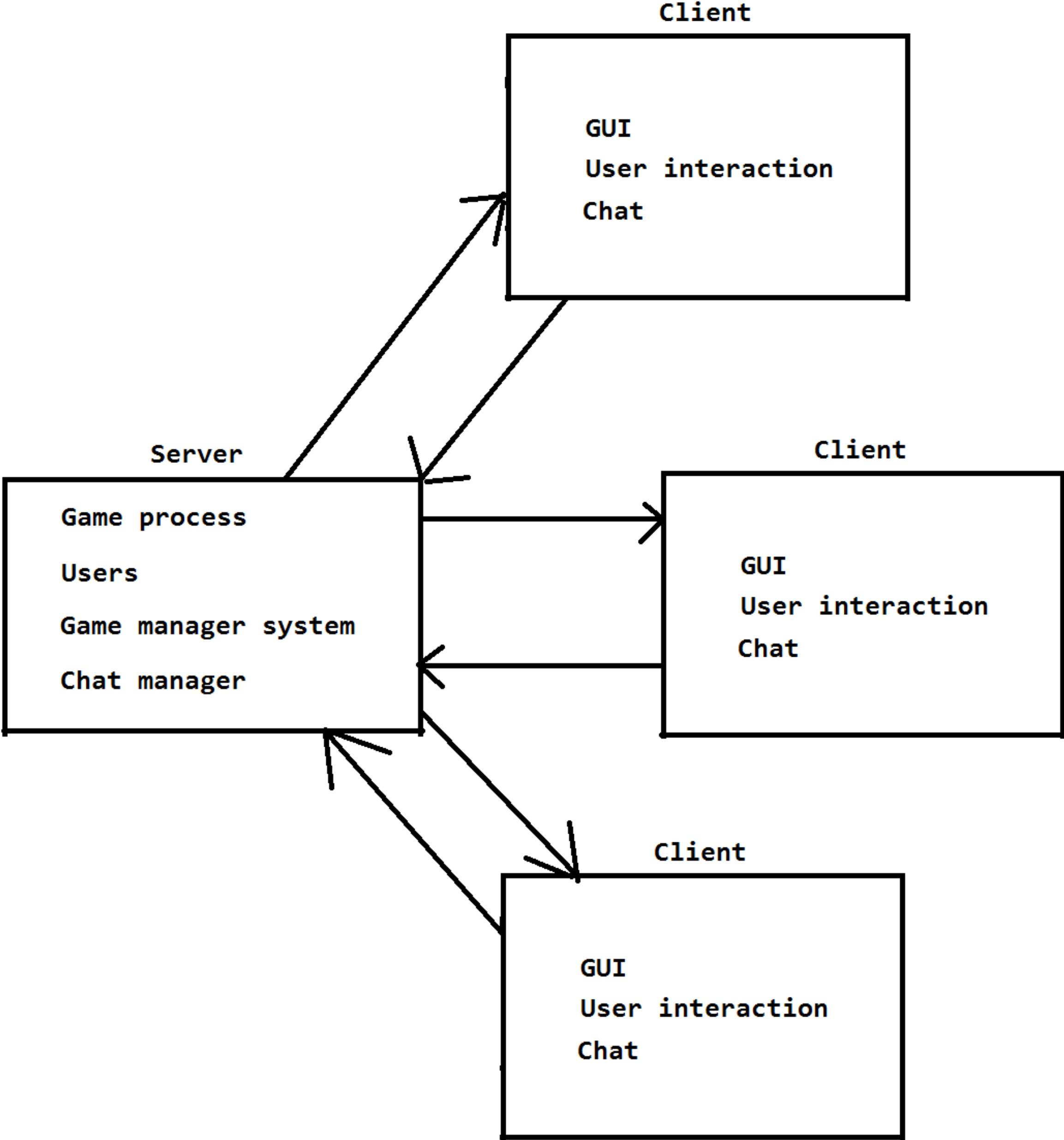

Figure 1: Client – Server model

Since we decided to use the client server model and we are two team members it became natural to distribute the work so that one person was responsible for the client and the other person for the server.

1.3 Disposition

We have structured this report as follows.

Chapter 2 – Background

In this chapter we want to familiarize the reader with the Chicago game by first going through the rules, and then demonstrating the game with an example game session. We also discuss how the system is designed.

Chapter 3 – Client implementation

This chapter is about how the client works. We will discuss the classes that make up the client and the main methods used. We will also discuss the usage of threads and sockets in the C# language.

Chapter 4 – Server implementation

This chapter describes how the server works. It is also about different data structures and algorithms we created, the main game function, and the Lobby system that handles multiple clients and manipulations of games. We also discuss the usage of threads and sockets in the C language.

Chapter 5 – Client –Server communication

This is a chapter about how the data that is sent between clients and the server is structured.

Chapter 6 – Feedback and future implementations

This chapter discuss some of the feedback we have received from users that have tested our system and also aspects we have found out ourselves by playing the game. We also discuss features that we have planned to add to the system in the future.

Chapter 7 – Languages & tools

This chapter presents some of the software that has helped us during the project and the programming languages that have been used for the client and server.

Chapter 8 – Result and evaluation

In chapter 8 we discuss how we have met the projects goal, problems that we have had and what we have learned.

Chapter 9 - Conclusion

This chapter summarizes the project and discuss our thoughts on what we have done.

2 Background

When we started to work on the Chicago project we knew it would give us a set of challenges such as designing game logic, relevant data structures and algorithms, designing a graphical user interface, process communication over the internet and threading.

2.1 Chicago game

The game of Chicago [12] is played by 2-4 players. The goal of the game is to collect 52 points and in order to do that, every player tries to get the best poker hand by switching cards three times, and after every switch the best hand earns the player with that hand points. In the end of each round the players play a trick game where the aim is to get the last trick for 5 points. The trick game will be explained in more detail later in this chapter. A player can say the word Chicago before this trick game and then he needs to win all the 5 tricks for 15 points. If he does not take all 5 tricks, then he will lose 15 points. All possible hands with their respective score can be found in Figure 2.

Hand	Score	Description
Royal flush	Immediately wins the game	The highest straight flush. This is a straight flush from ten to ace.
Straight flush	10	A combination of a straight and a flush.
Four of a kind	7	Four cards with the same value.
Full house	6	A combination of three of a kind and a pair.
Flush	5	All cards in the same suit.
Straight	4	All five cards in sequential order such as 3, 4, 5, 6, 7.
Three of a kind	3	Three cards with the same value.
Two Pairs	2	Two different pairs.
Pair	1	Two cards with the same value such as two aces.

Figure 2: Score system

We will now follow two fictional players, Alice and Bob when they play a game of Chicago online. First of all, Alice and Bob start their Chicago client programs. They are met

with a dialog box (Figure 3) with a single text box and a button, and they both enter their username in their respective text box.

Figure 3: Login screen

Luckily their favourite usernames are available, so the server accepts the new clients and Alice and Bob gets a new window representing the game lobby. The game lobby is the place where a user can create a game of Chicago, or join a game that another client has created. Alice press the new game button and is met with a dialog box (see Figure 4) were she has the option to choose the number of players that should play the game. She can only choose two, three or four players, but today she wants to play a two player game so she chooses this option and press the begin game button and the client program return to the lobby.

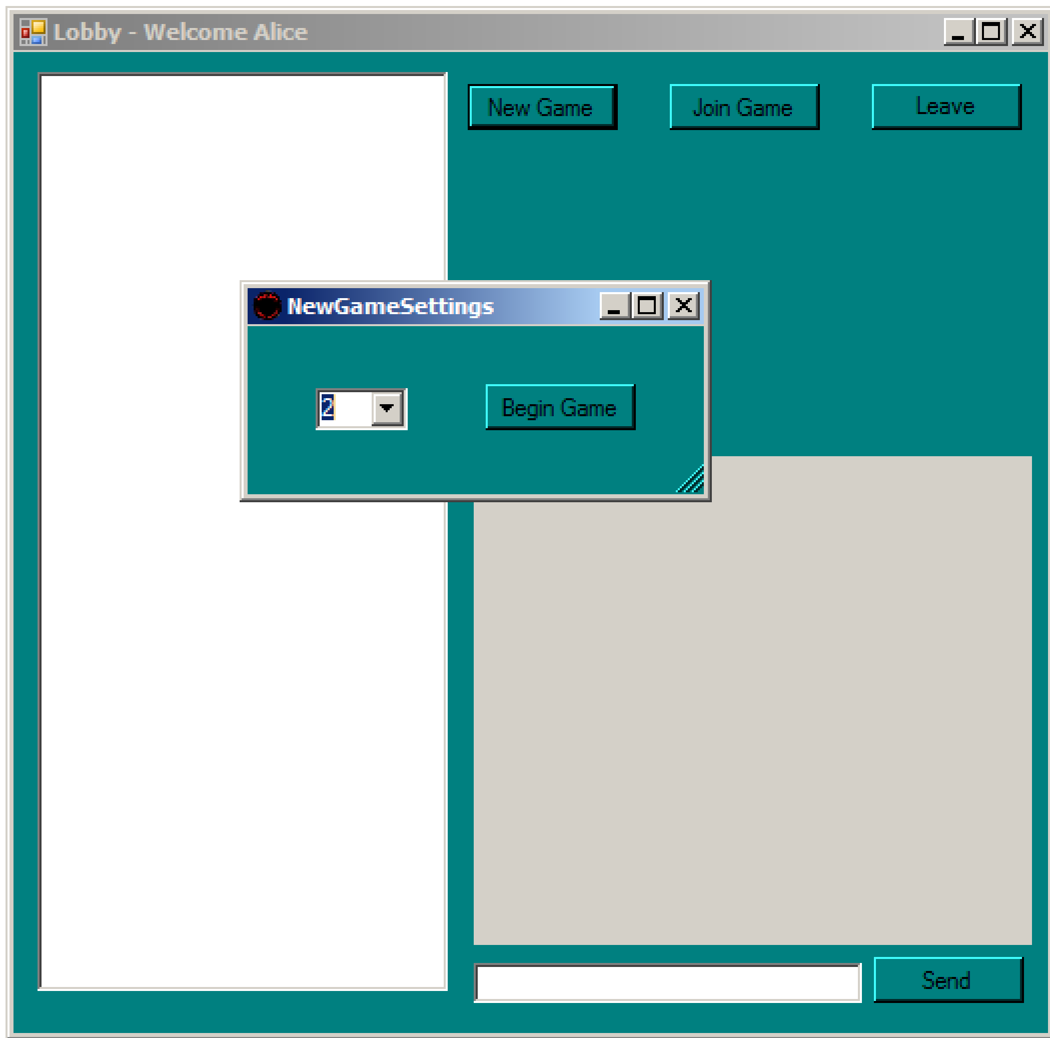

Figure 4: Creating a new game

In the chat window is a message that a new game was created and the game is found in the list of games, to the left of the screen. At the same time Bob sees the newly created game in his game list. He clicks on the game in the list and see that it is a two player game and he also notice that his best friend Alice is in this game. Therefore he clicks the join game button with the game marked in the list and since it is a two player game and two players have entered, the lobby window disappears and is replaced with a new window with the playing area.

Both Alice and Bob have received five cards each. Alice has got the ace of hearts, eight of hearts, seven of diamonds, six of clubs and the king of clubs. She decides to keep the two highest cards by clicking on the three other cards (see Figure 5). After clicking the cards there are red border surrounding the cards to mark out that they should be changed. To finally change the cards she presses the change cards button and after a short wait she receives three new cards, the three of spades, nine of spades and nine of clubs. She also reads in the chat

window and noticed that Bob has the best hand with a pair and he has been rewarded with one point for the hand. Alice once again tries to improve her hand by switching the three newly acquired cards and she receives the five of hearts, ace of diamonds and the jack of clubs.

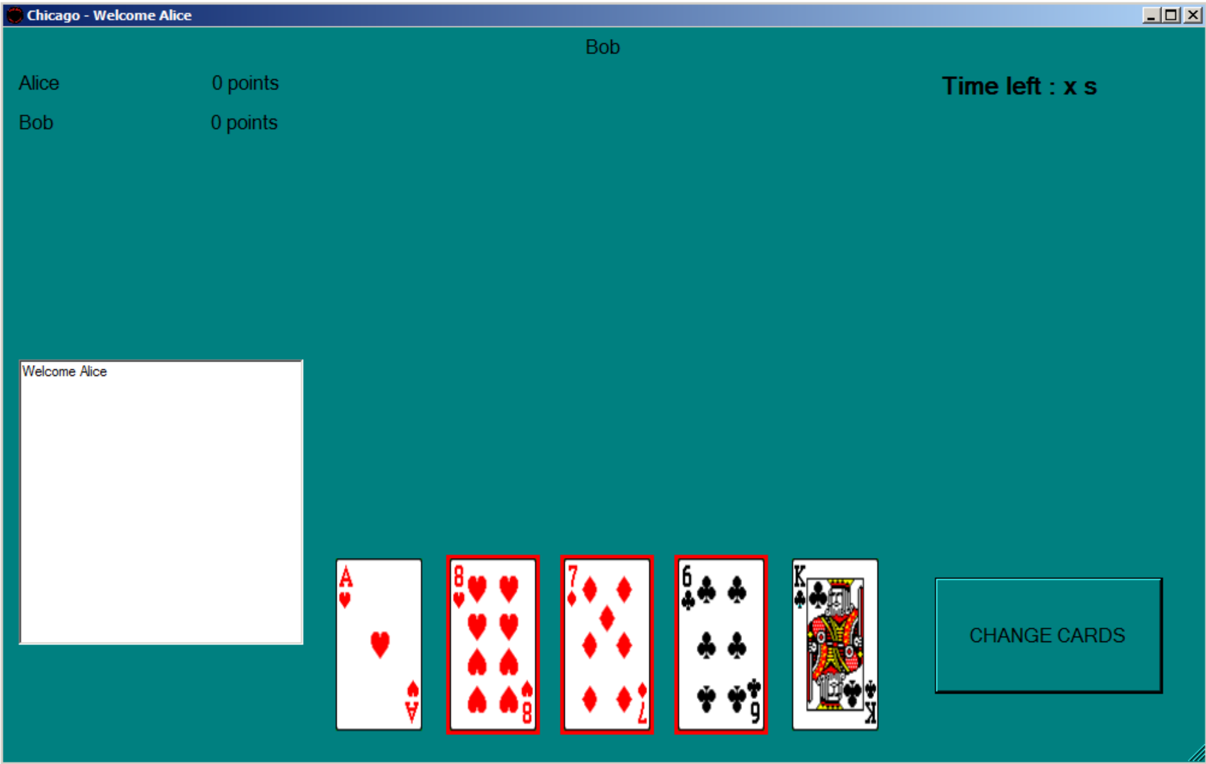

Figure 5: Changing cards

Alice now has a pair of aces and this hand is better than Bobs hand so this time Alice is the one to receive one point. This time Alice changes only one card, the five of hearts. It is replaced with the six of spades. At this time Alice and Bob need to choose if they want to have a normal trick round or if one of them should try to go for Chicago. A player going for Chicago will start the trick round and that player then needs to win all the five tricks. To play a trick the first player to act chooses one of the cards to play. The next player needs to play a card with the same suit as the first player and only if he cannot match the suit, another suit can be played. When all the players have played one card the player that played the highest card in the same suit as the first card wins the trick and will play the first card in the next trick. Alice knows that with a six as the highest card in spades there is not a good chance to be able to win all five tricks so she chooses no Chicago. Apparently Bob does not like to try his luck either, because in the chat window Alice can read that no player has gone for Chicago.

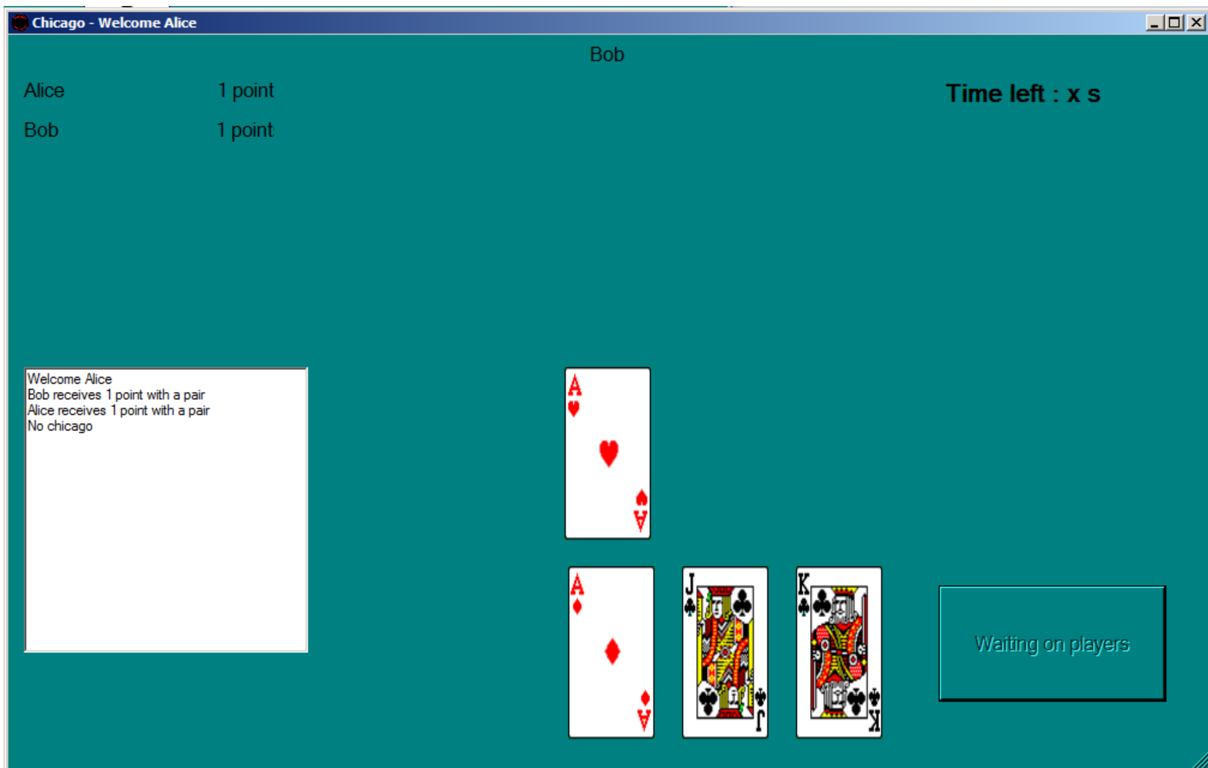

Figure 6: Playing a trick card

Bob is the first player to act and he plays the five of spades. Alice has to play the only legal card, the six of spades since she has to follow suit. Since she had the highest card in that suit she plays the next card the ace of hearts (see Figure 6). Bob does not have a heart so he instead plays his lowest card the ten of spades. Alice plays her other ace and Bob once again cannot follow suit but instead he plays the queen of spades. Alice plays the jack of clubs and this time Bob can follow suit with the better card, ace of clubs. Bob plays his last card which is the ace of spades and Alice has to play her last card knowing she has lost the trick game. In the end Bob gets five points for winning the last trick and Alice receives another point for her pair of aces. The score after the first round is three points for Alice and six points for Bob. Some seconds later they are dealt new cards and the second round starts. The game continues on for several rounds until finally Alice has got 52 points in the end of the round and this is the requirement to win the game. After the match she closes the game window and returns to the lobby.

2.2 System Design

The system is built on a client server architecture where multiple clients can connect with a central server. We have tried to put as much of the game logic as possible on the server side

while the client is a state machine that interacts with the user. The implementation languages we chose for the client was C# and for the server we chose C. The reason for these choices was because C# is a good environment for creating windows application and the client has low requirements on performance and C is a fast language and perfect for a server that will handle multiple clients. The clients will run on windows and the server will run in a Linux environment.

2.2.1 Client design

The client is a Windows Forms based application [20] with a graphical view of the Chicago game. It is composed of two parts, the lobby and the game. The lobby is a window form where the player can create a game of Chicago or join a game already created by another player. The lobby uses a lobby session class that handles the communication with the server and a window form that processes the input from the user. When a game starts, the lobby window closes and the Chicago window form is created. In this window the user will be able to make all the actions that are needed to play the game. This window will show the players own cards, the current score, the time left to act and all cards that are played during the last trick session. There are two main classes at work in the game view and they are the Chicago class and the game session class. The Chicago class is the window form that is used and the game session class is interacting with the server.

In both views there are chat windows which are used to print messages to the player. There is also a dialog box that will show at the start-up of the program and this is used to set a username for the session.

2.2.2 Server design

The server is divided into two parts. The first part is the card game. There will be multiple threads, one for each game in session, running on the server. The other part is the interface to create and join games. This part is more about handling users and start game threads. We first created the game part and only when it was completely finished and tested did we implement the second part. The server is implemented in C because it is a fast language suitable for a server that should handle this kind of computations. The server runs on Linux since it is a suitable environment for a server and the best reason is that it is easy to operate the system remotely.

2.2.3 Game

The game part is a procedure called the CGP (Chicago game procedure) and this is a function that is created as a thread for every single game that is run on the server. This procedure consists of several rounds played until one of the players wins the game. For this procedure to function there are some specific data structures and algorithms used. These are the deck and the hand structures. The deck is a set of cards that is in a random order and there is a function that picks a card from the deck and another function that returns a card to the deck. If the first 52 cards have been dealt the returned cards will be reshuffled and used again. The hand is a structure that holds five cards. There are also algorithms such as sorting the hand, calculating hand strength and comparing hands.

2.2.4 Lobby

The other part is the lobby, which is the interface for creating games. This is a single thread that will process a message from a client and act upon that message. If for example a client sends a "create game" message a new game will be created. The lobby will also wait for new clients to connect and make sure that they have a valid username that is not already used. There are some data structures that are used in both parts of the program. These are the player and game data structure. There is also a module that is used for the client communication.

2.3 GUI Design

The lobby is where clients can create or join a game of Chicago. We will now explain the lobby window using the numbers in Figure 7. First of all there is a button (1) with the text new game on it. When a user presses this button he will create a new game and this game will show up in the list (2) to the left of the screen. If the client is already in a game the new game button will be disabled. If a user presses on a game in the game list (2) then there will be some information of the selected game under the buttons (3). The information includes the game id, a unique number to identify a game, and player names.

To join a game a user needs to first select a game in the game list (2) and then press the join game button (4). To leave a game the user has to press the leave game button (5). When the lobby is changing some messages to the user will be output in the chat window (6).

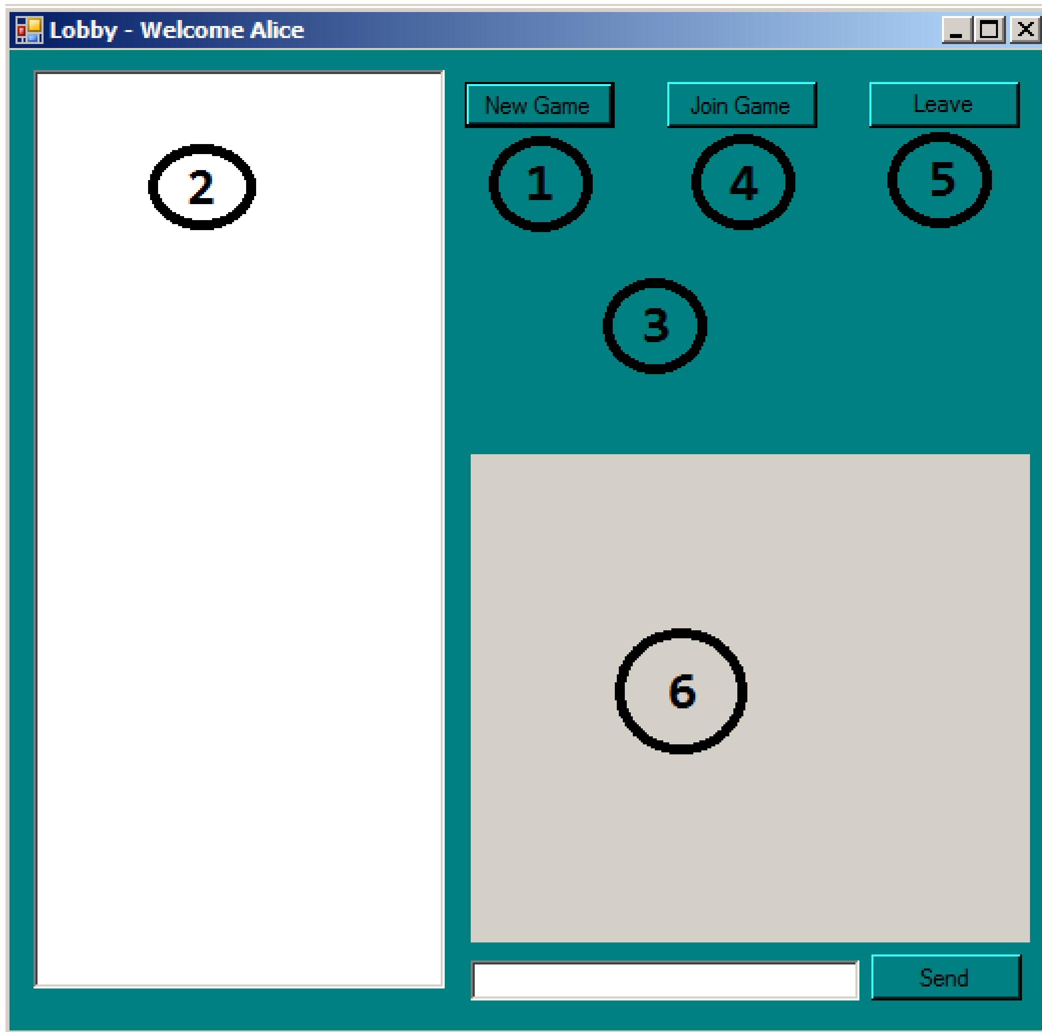

Figure 7: Lobby window

We will now go through the graphical parts of the game using Figure 8. First a list of the players and their score is shown (1) at the upper left corner. This will update after the first and second switch of cards and also after the trick game. In the left bottom (2) is the chat window. Here the user can read messages from other players or from the system. This is where the user will be able to follow the events throughout the game such as who had the last scoring hand or who won the last trick. At the bottom (3), the user's cards are shown. These can be marked or unmarked by clicking on them. If the trick game is on, then only one card can be selected and when it is time to change cards, all cards can be selected. To change the marked cards, or play the single card selected in a trick game, the change card button (4) should be pressed. It is in the middle of the screen (5) that the cards will appear when they are played in the trick game. Here the user will see its own cards as well as the opponent's cards. At the upper right corner (6) is a timer that will count down when it is the user's time to act. If this reaches zero the

current marked card will be changed or played. If no cards are marked then no cards are changed and if there is no marked card to play in the trick game, the game will pick a card to play. In the middle right (7) is a hidden button that will appear between the card switching phase and the trick game. If this button is pressed the user is requesting to go for Chicago, and he will start the trick game and have to win all the five tricks. To not select Chicago at this time the change button (4) should be pressed.

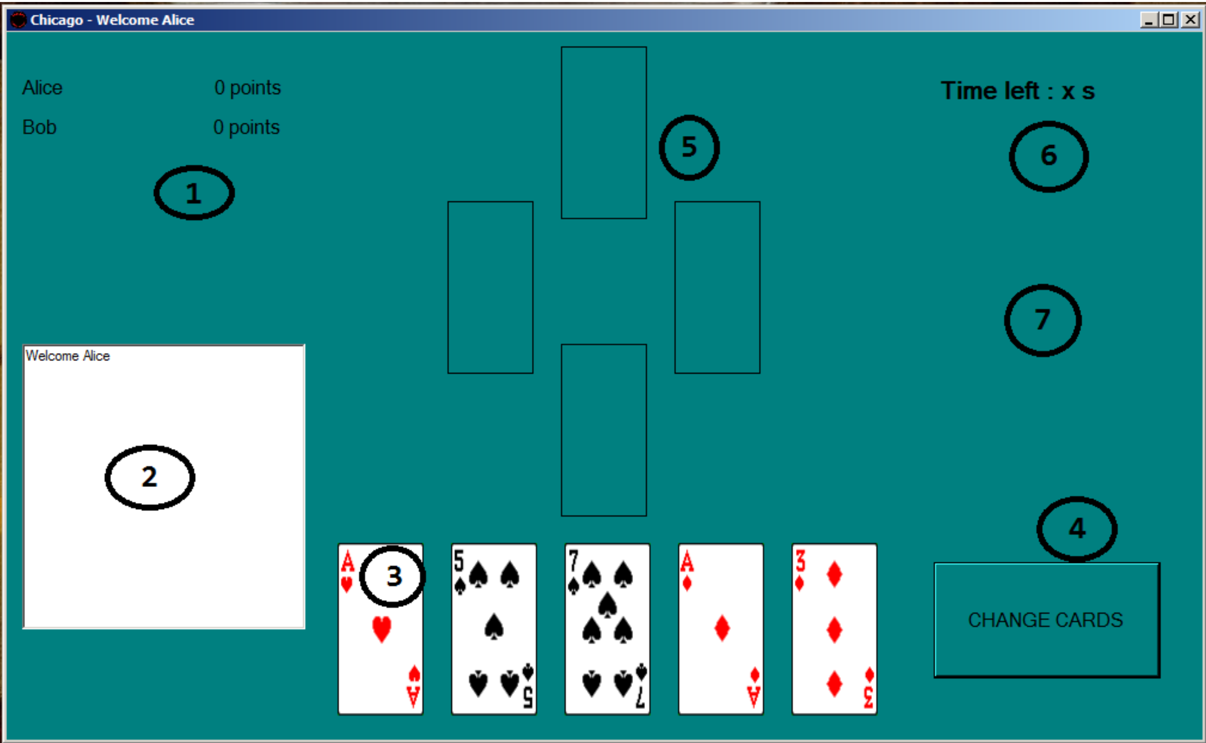

Figure 8: Game window

3 Client implementation

The client is built up of several classes. The Lobby, LobbySession, Chicago, GameSession, GUICom and some dialog boxes. When the program is in lobby mode the Lobby is used for user input and graphical output and the LobbySession is used to communicate with the server. When the client is in game mode, the Chicago class is used for user input and graphical output and GameSession is used for communication with the server. The session classes communicate with the Lobby and Chicago classes through the GUICom class. The GUICom class's only purpose is to organize the calls to the Lobby and Chicago for increased readability and we will therefore not describe this class in detail. We also have a Game class to contain some variables used to keep track of created games. The variables are, a unique game id to identify the game, the number of players that the game should have, the number of players that currently have entered the game and a list of the player names that are in the game. There is also a method that is used to create a string representation (see Figure 9) of the game. This method is also used to override the Game class default ToString method because the game list will use this when it display the game.

```
public static String gameString(uint id)
{
    return "Game: " + id.ToString();
}

public override String ToString()
{
    return gameString(id);
}
```

Figure 9: Game string representation

3.1 Lobby class

The lobby class is the class that handles user interaction and graphical output when the game is in lobby mode. It will take care of input such as creating a new game, joining an existing game, leaving a game and selecting a game. It will also handle server responses from the LobbySession class such as new game, join game, leave game, update game and start game.

3.1.1 Lobby Constructor

When starting the Chicago client the user will be presented with a dialog box for name input. After hitting return or pressing the send button the client will connect to the server and send the username that was requested. This is done in the `Lobby` constructor by creating a `UserID` dialog box. This dialog needs a reference to the `LobbySession` in order to call the method that connects to the server.

When the user either clicks the ok button or enter, a method will check if there is text in the text box. The method will just return if there is no text entered but in case there is, it will call the connect method with the text as input. If the text was accepted as a username by the server a true value will be returned and the dialog will be closed. If not, the method will just return and the user has to try a different name.

When the `UserID` dialog box has closed we read the username and store it in the `Lobby` class. Lastly we create a network thread with the method `readMsgFromServer`, which is part of the `LobbySession` class.

3.1.2 New Game

Most of the methods in the Lobby class are called by the `LobbySession` class through the `GUICom` class or from the event methods that is called when a user press on a controller. When the user press the new game button the `newGameButton_Click` method is called. At first this method checks if the program is in a valid state to create a game. There are two variables that are used and these are the Boolean variable `inGame`, and the `gameID` variable that is an unsigned integer. The `inGame` variable is true if the user is already in a game or is waiting for a request to create a game. The `gameID` is nonzero if the user is in a game or waiting for a request to join a game. If the player is not in a game and the `gameID` is zero it is ok to start a game and a `NewGameSettings` dialog box is created. This dialog has got a combo box with three alternative numbers of players to choose from, and a button to accept. When the dialog box is closed, the value in the combo box is read and parsed to an integer value. At last this value is passed on to the `createGame` method in the `LobbySession` class.

The newGame method is called when a new game has been created by the server. It will first check if the game id is a valid number, (not zero), and stores the id in the gameID variable. If the id is zero it means that the new game request was refused by the server and therefore the inGame variable will be set to false.

3.1.3 Join game

If the user clicks on the join game button the joinGameButton_Click method runs. This method will also control the state the program is in. If the inGame variable is set to false and the gameID is set to zero it is ok to join a game. Next the method must control if the user has selected a game in the gameList list box. If a game is not selected, a message is printed out in the chat window and the method returns. In the other case the selected games id is used with the joinGame method in the LobbySession class.

After the server has responded to the request to join a game the joinGame method will be called with an answer as in parameter. This answer is either a one, if it was ok to join the game, or zero if it was not ok.

3.1.4 Leave game

When the user clicks the leave game button the leaveButton_Click method is called and this method checks if the user is in a game by making sure that the inGame variable is set to true and the gameID is not zero and if this is the case it calls the leaveGame method in the LobbySession class. At a later point when the server has handled the request the LobbySession class will call the leaveGame method that will set the inGame variable to false and gameID to zero.

3.1.5 Update game

Every time there is a change in the game, or a new game has been created or removed, the LobbySession will call the updateGame method with the game to update as in parameter. This method will call two methods, addListBox and gameListUpdated. The first method will first try to match the game to the gameList list box [7] by using the list box FindStringExact method with the games sting representation as in parameter (see Figure 10).

```
int index = gl.FindStringExact(g.ToString());
if (index == ListBox.NoMatches)
{
    gl.Items.Add(g);
}
else
{
    gl.Items[index] = g;
}
```

Figure 10: Game list add or update

This will return an index that can be used to address the game item where it is in the list box or the index will show that the game did not exist in the list box. Depending on which we will either add the game or just replace the game that already exist.

After the game list is updated we need to call the `gameListUpdated` since we have to print out the game details if it has changed. The game details should always show the details for the game that is selected in the game list or it should not show anything if no game is selected. Therefore the first thing we do is to find out if there is a selected game. If there is not a selected game we just print out some empty strings. In case a game is selected we create a string to print out as a title containing the games string representation and the number players entered and the total number players that can enter. Next we create a multiline string with all the usernames that is in the games list of names.

There is also a method `listOfGames` that is used to prepare for updating the list from scratch. What this method does is to remove all items in the `gameList`.

3.1.6 Game start

When the last player required in a game has entered, the `LobbySession` will call the `gameStart` method. There the program will stop the `readMsgFromServer` thread by calling the disconnect method in the `LobbySession`. After doing so the lobby windows `visible` and `enabled` variables is set to false and a `Chicago` object is created as in Figure 11.

```
Chicago c = new Chicago(this);
c.ShowDialog();
```

Figure 11: Starting the Chicago game window

By calling the `ShowDialog` method on the `Chicago` object the lobby will stop running until the Chicago window has closed. Following this code is a check if the lobby should close or not after the game has ended by checking a Boolean value. If the value is true the lobby is closed and if the value is false the lobby will again enabled and visible. Lastly the network thread is created as in the constructor and a call to update the list of games is made to the `LobbySession`.

3.2 Lobby session class

In order to handle the communication with the server while in lobby mode we use the `LobbySession` class. This class has methods that send messages to the server and it has a method that will read messages received from the server and do the proper action depending on the message.

3.2.1 Connect

To connect to the server we use the `connect` method with a username as in parameter. This method will first look at a `connected` variable to make sure that there is not already a connection with the server, then it will try to connect to the server, and when connected it will send the username and wait for the server response. If the response is a zero then the username could not be used and the connect method will return false. If we read the value one from the server we set the `connected` variable to true and return true.

3.2.2 Read message from server

To be able to receive messages from the server the `Lobby` will start a network thread [17] using the method `readMsgFromServer`. This method reads one byte (see Figure 12) when available on the network stream and depending on the value in the byte it runs the correct method.

```
switch (msgBuffer[0])
{
    case NEW_GAME:
        fromServerNewGame();
        break;
    case JOIN_GAME:
        fromServerJoinGame();
        break;
    case LEAVE_GAME:
        fromServerLeaveGame();
        break;
    case LIST_OF_GAMES:
        fromServerListOfGames();
        break;
    case GAME_START:
        fromServerGameStart();
        break;
    case GAME_UPDATE:
        fromServerGameUpdate();
        break;
    case REMOVE_GAME:
        fromServerRemoveGame();
        break;
    default:
        break;
}
```

Figure 12: Read message from server

NEW_GAME is received after the server has handled a new game request and if this is received the LobbySesson class will read 4 more bytes from the network stream and parse them to an integer value that is used as the game id. This id is then passed to the lobby through the Lobby newGame method.

JOIN_GAME will be the server response after it has handled a request for joining a game. The LobbySession will read a second byte that it passes on to the Lobby with the newGame method.

LEAVE_GAME is received if the server has allowed the client to leave the game it is currently in. No additional data is required to read from the network stream. The LobbySession class will pass the message to the Lobby with the leaveGame method. A note here is that a leave game request does not always have to end up in a leave game response since the server will deny the client the possibility of leaving the game if the game happens to start before the leave game request has reached the server.

LIST_OF_GAMES will also not require more data to be read. This message is passed on to the Lobby with the listOfGames method that is used to prepare for a new list of games.

GAME_START will not require more data to be read. The message is passed on to the Lobby with the gameStart method.

GAME_UPDATE is a message from the server with all the game data of a single game. First the LobbySession will read 6 additional bytes from the network stream. 4 of these bytes are the game id and the other two are the number of players the game holds and the number of players that have currently entered the game. Now when the number of players entered is known, the LobbySession can read in a username for every player in the game (see Figure 13). This is done by first reading a single byte with the length of the name, and then read the name and put it in a name list. All the data that has been read is packed into a game object and sent to the Lobby through the method addGameList.

```
List<String> names = new List<string>();
for (int i = 0; i < playersEntered; i++)
{
    byte[] namelen = new byte[1];
    readFromServer(namelen, namelen.Length);
    byte[] namebuffer = new byte[namelen[0]];
    readFromServer(namebuffer, namebuffer.Length);
    names.Add(Encoding.ASCII.GetString(namebuffer));
}
```

Figure 13: Reading a list of names from server

REMOVE_GAME is a message from the server that tells the client that the game has been removed from the system. The LobbySession class have to read four additional bytes and store them as the id of the game to be removed. The id is passed to the Lobby with the removeGameList method.

3.2.3 Sending a message to server

The LobbySession class has four methods for sending a message to the server. The first is createGame and it will send two bytes. The first byte is the NEW_GAME value and the second byte is the number of players that the game should hold. Next is the joinGame method that will send five bytes of data. The first byte is the JOIN_GAME value and the other

four is the id of the game to join. The third is the `leaveGame` method, and this will send one byte with the `LEAVE_GAME` value. The last method is `listOfGames`. This will send one byte of data with the `LIST_OF_GAMES` value.

3.3 Chicago class

The `Chicago` class is a window form that is the graphical view to the user and also the class that handles the user interaction. It is the `Chicago` class that creates an instance of the `GameSession` class and also starts the `GameSession` thread. The `GameSession` class can communicate with the `Chicago` class through the `GUICom` class that we created mainly to organize all the communication in that direction. We will go through all the important tasks that the `Chicago` class has.

3.3.1 Printing cards

First we needed to implement the ability to print cards. We tried to reuse the cards.dll that Windows has used for its card games but we found out that this DLL-file has not been used since windows XP. So we had to try a different plan and started to look for cards on the internet. We found out that it is very easy to get a set of pictures for a deck of cards so we randomly selected the images for a deck. To print these cards we first needed an object to print it on. We also knew that the cards have to be clickable. With all these requirements in mind we found out that what we needed was a picture box. The picture box's main feature is that it is clickable and it is possible to draw a picture on it [6]. How to do this is shown in Figure 14.

```
cardPic[i].Image = Image.FromFile(path + card.ToString() + ".gif");
```

Figure 14: Drawing a card

We create an image from a file and compose the filename from a file path, the card value and the .gif extension. Then we set this image as the image of the picture box. To be able to mark a card we use padding so that the picture box is slightly bigger than the image and the image is centred in the box. Then if we want to mark the card we set the picture box background to red and create the illusion of a red frame around the card.

Another important part of drawing the card is how to calculate the location of the card. There are two different aspects here, where the easier one is to draw the cards that should be

switched. We have an array of five picture boxes and we draw the cards in the same order as they arrive from the server. The second aspect is about drawing the trick cards. We have an array of the trick cards but the positions of players are different depending on which client is drawing them. The server has arranged the player in one order but the client always set its own user at the bottom. We created a short algorithm (see Figure 15) to calculate the position that should be used to draw a trick card.

```
pos = (gs.getMyPos() - pos) % gs.getNumberPlayers();
pos = (gs.getNumberPlayers()-pos) % gs.getNumberPlayers();
```

Figure 15: Calculate player location

At first `pos` is the position that is the server's perspective. At the end of this calculation the `pos` will be that position but in the perspective of the client.

3.3.2 User input

Next important feature is to handle the user input. This is when the user clicks on one of the cards or on a button. First we will discuss about clicking on a card. We have indexes for the five cards numbered 0-4 from left to right. When the user clicks on a card a method will run. This method will then call a method called `cardClickAt` with the argument `i`, that is the index of the card that was clicked. This method is where the clicks are processed depending on the game's state. There are three possible states that the game can be in at this point. First of all, if we have disabled the send button we do not allow clicks at all so the method just returns. If enabled we check if the trick game is on. If both of these are false then we know that we are changing cards.

When we change cards or play the trick game we need to be able to mark the cards. The only difference is that when we change cards we can mark multiple cards. To be able to tell if a card is marked we use an integer for every card and increase it by one for every click on that card. Then we can see if it is marked by investigating whether the integer is odd or even. It is a little more complicated when the trick game is active because then we need to also check if there is a chosen card or not. We did have a strange bug here where multiple cards could be marked if we clicked fast enough and we solved this by resetting the background on all the cards before printing the new situation. The `Chicago` class will notify the `GameSession` class with the `changeCardAt(Boolean, int)` where a true value will change the card

and a false value will not change the card. The integer is used to identify the position of the card.

The next important input is the send button. This is used to change cards, play a card during trick game or decline Chicago. The method that handles these clicks will find out which of these states it is in and execute the correct task depending on that state. If the Chicago button is enabled it will just start the trick game by changing a Boolean value. If that same Boolean value is true it will further check if there is a chosen card and also the players turn to act. If all of these are true, then there will be a control if the card chosen is a legal card to play, and if it is so, then the game session class will be notified that that card should be played.
If the game is in the state of changing cards all integer values of the cards will be checked and the marked ones will be set to invisible.

A last note on the send button is that for every click made on this button the method will always lock the button so the user cannot use it again until the game allows it to. It will also notify the `GameSession` class that the button has been pressed with the method `pressSendButton`.

3.3.3 Print score

In the game of Chicago players will receive points when they have the best hand and when they win the last trick. These points have to be presented so that the users can see the current score and also be able to see why a player received a particular score. We have three main methods that are used to fulfil these tasks.

Every time a player receives points from the best hand we call two of the methods. The first is the `pointMsg` method and this is used to print out a message that will tell the user the amount of points a player has received and why he received it.

```
if (lastPoint == 1)
{
    msg = names[lastPlayer].Text + " receives " + lastPoint.ToString() + " point with " + printHandString(lastPoint);
}
else
{
    msg = names[lastPlayer].Text + " receives " + lastPoint.ToString() + " points with " + printHandString(lastPoint);
}
printMsg(msg);
```

Figure 16: Print score message

This is done with some string manipulation based on the scoring player and the value of the point the player has received (see Figure 16). The `printHandString` method (see Figure 17) is used to get a string representation of a hand depending on the hand value.

```
case 1:
    return "a pair";
case 2:
    return "two pairs";
case 3:
    return "three of a kind";
case 4:
    return "a straight";
case 5:
    return "a flush";
case 6:
    return "a full house";
case 7:
    return "four of a kind";
case 8:
    return "a straight flush";
case 9:
    return "a royal flush";
default:
    return "an illegal hand";
```

Figure 17: Print hand string

There are also some other possible outputs such as when there is no scoring hand, the hands are drawn or when the player has got a royal flush. It is probably justified to explain why the royal flush needs a special string. It is because a royal flush does not give points since it just wins the game immediately.

After the program has printed out the message it will also have to update the score in the top left corner of the screen. This is achieved in the `points` method that takes a char array as input. This array has got the current score for every player and all we need to do is to print this out.

The third method we use when dealing with the score is to print out the result of the trick game. We call the `trickPoints` method and use two integers as input. The first integer is the `failval` variable used to indicate if a Chicago has succeeded or failed. If the value is zero then Chicago has failed, if it is one it has succeeded and if it has another value then there was not a Chicago attempt in that trick round and no message about Chicago will be printed out. The other integer value is the index of the last trick winning player. If this value is set to the number of players in the game then there was no player receiving points for last trick. This is a special case that can occur when a player has selected Chicago, failed the attempt but still wins last trick.

3.3.4 Opponent locations

There are four picture boxes that are used to draw the trick game cards (see nr 5 in Figure 8). The box in the bottom is the player's cards and then there are the opponent's cards to the left, right and the top. We store these picture boxes in an array and order them from bottom, left, top to right. We did not like the look of the game when we played a two player game since the single opponent was sitting to the left so we created a method, which we run during the initiation of the game, that change the position of the picture boxes in the array. This only affects where a card will be drawn and not the indexes that are used when drawing the cards. When we had implemented this change (see Figure 18) for a two player game we also became aware of that the same problem existed for a three player game, so we changed that so that the top player was drawn to the right in that kind of game. We also needed to do the same adjustment to the labels with the opponent names.

```
if (numPlayers == 2)
{
    opponentsNames[0] = oppositionName;
    opponentsNames[0].Visible = true;
    trickCards[1] = opponentcard;
}
else if (numPlayers == 3)
{
    opponentsNames[1] = rightName;
    opponentsNames[0].Visible = true;
    opponentsNames[1].Visible = true;
    trickCards[1] = leftcard;
    trickCards[2] = rightcard;
}
else
{
    opponentsNames[0].Visible = true;
    opponentsNames[1].Visible = true;
    opponentsNames[2].Visible = true;
}
```

Figure 18: Opponents location

3.4 Game session class

When the game is in progress the client needs to communicate with the server in an organized matter. That is what the `GameSession` class is created to do. This class has a method called `runGame` (see Figure 19) that is executed as a thread as long as the game is in session. This method follows the same structure as the CGP function in that it is running at the server end of the system. There are three main parts of the `runGame` thread [17]. There is the initiation, the changing cards phase and the trick game part. The last two parts is what makes a round.

```
public void runGame()
{
    if (!init_game())
        return;
    guic.printPlayerNames(names);
    while (!round()) ;
    guic.end_game(winner);
}
```

Figure 19: Run game thread

3.4.1 Init game

At the start of the game the client needs some important data about the game in session. The data received at start is the number of players in the game, the position the client has in perspective of the server and the names of all the players in the current game. This data is then passed on to the `Chicago` class. After the initiation the round will start with the `wait_for_cards` method that will read the starting hand from the server.

3.4.2 Changing cards

The user can change cards three times and therefor the implementation for changing cards involves a three round loop. Every round in the loop will follow the same pattern except a small variation on the third round. The steps are as follows.

1.) **Change cards.** To do this the `GameSession` class sets the `sendButtonPressed` variable to false and tells the `Chicago` class that cards should be changed. Then it polls this variable every 200 ms until it is marked as true by the `Chicago` class. Then there is an array with a Boolean for every card in the hand that is investigated and every true value indicates that the card should be changed. We also have an array with the values of the cards and we change the card values that should be changed to the `EMPTY_CARD` value and send it to the server.

2) **Wait for cards.** To get new cards the client reads five bytes from the server and stores them in the card array. At this point all the variables in the Boolean array associated with the cards is set to false assuming no cards should be changed by default. Then the `GameSession` class sends the new cards to the `Chicago` class.

3) **Wait for score.** To receive the score two bytes are read and these represent the player that has scored and the value of the points that the player has got. Then there are additional bytes read, one for every player in the game, and these are the current score values. All the score data are sent to the `Chicago` class the first two rounds of the loop but not the third since the scoring hand will be rewarded after the trick game.

4) **Check for royal flush.** Since a royal flush wins the game immediately we need to check this at this point and if it would occur the game will end.

3.4.3 Trick game

There are five tricks that should be played to make up the trick game. To play one of these tricks the client first needs to wait for a card from server. At first one byte is read and if this value is the `EMPTY_CARD` value then the client will not receive a card at that time. Instead this is in fact a signal that this player is the one to start the trick by playing a card. If the value is not the `EMPTY_CARD` it will represent the player that has played the card and another byte has to be read with the card that was played.

If the player has received the start signal he will send a card. The send card method first set a char variable to the `HAND_SIZE` value, which is the number five. Then the `Chicago` class is informed that a card should be sent. The variable will be polled until it has changed value to the card that should be played and a single byte is sent to the server. If the client did not receive the start signal but instead a card, then this card is sent to the `Chicago` class to be drawn. Then there is a control (see Figure 20) if the client is the next player in turn to play a card and if so a card is sent to the server.

```
if (((myPos == 0) && (turn + 1 == numberPlayers)) || (turn + 1 == myPos))
```

Figure 20: Next turn condition

At the end of the trick game the trick score is read from the server and there is also a message indication if the game is won or if a new round should start.

3.5 Threads

The lobby and the game view use threads [17]. In Figure 21 is an example how we create the game session thread.

```
gameThread = new Thread(new ThreadStart(gs.runGame));
gameThread.Start();
```

Figure 21: Creating a thread (C#)

As can be seen the Thread needs a `ThreadStart` class and this class needs a method that will run as the thread. After the creation of the thread we need to call the threads start method to start the threads execution.

When we first started to use threads we found some strange errors at run time. At first we did not understand what created these errors since the code seemed to be correct but after some research we found out that there is a problem with writing to a windows control if it is created in another thread because it is not thread safe [5]. This is built into the C# language and an exception is thrown when the wrong thread is trying to write to for example a text label. Luckily there is a solution to this problem. We had to write set methods for all the controllers that should be able to be set from another thread and these methods all had to follow the pattern seen in Figure 22.

```
delegate void setSomeControllerCB(String text, int i;)
private void setSomeController(String text, int i)
{
     if (controller[i].InvokeRequired)
     {
         setSomeControllerCB d = new setSomeControllerCB(setSomeController);
         controller[i].Invoke(d, new object[] { text, i });
     }
     else
     {
         controller[i].Text = text;
     }
 }
```

Figure 22: Changing controller thread safe

First of all we can look at the `setSomeController` method. This is a general method only used here as an example and not actually used in the `Chicago` program. There is however a number of methods with this pattern implemented, one for every controller that we write to. The method has got two in parameters. There could also have been any other number of in parameters with any data types. A call back function declaration is also needed and it is important that this has received the exact number of parameters as the method.

In the method we check to see if the controller needs to be invoked, that is, if we are trying to write to this controller from a different thread. If not, then we can just write to the controller. But what if we are? Then instead of writing to the controller we create a delegate function that we invoke. What happens is that the method will be executed in the thread that is created in where it is allowed to write to the controller.

3.6 Sockets

The client uses a socket [8] to communicate with the server. This is created with the `Socket` constructor as in Figure 23.

```
server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
```

Figure 23: Creating socket (C#)

We set the socket to function over the internet and to be a stream socket using the TCP protocol. We use TCP because it is important that all the data we send will be correct when it arrive at the other end. To connect to the server we use the `connect` method on the socket (see Figure 24).

```
server.Connect(serverAddr, serverPort);
```

Figure 24: Connecting to a socket (C#)

We connect to the IP-address that the string `serverAddr` contains and the port that is in the integer `serverPort`. We then use the socket to create a `NetworkStream` (see Figure 25).

```
ns = new NetworkStream(server);
```

Figure 25: Create network stream

This stream can then be used to write to the server or to read from it. When we write we just fill up a byte buffer and call the write method as in Figure 26.

```
ns.Write(buffer, 0, buffer.Length);
```

Figure 26: Writing to network stream

But when we read it is possible that all the data is not ready to be read. To solve this problem we created a method (see Figure 27) that reads from the network stream over and over again until all the bytes specified is read.

```
while (bytesLeftToRead > 0)
{
    bytesRead += ns.Read(buffer, bytesRead, bytesLeftToRead);
    bytesLeftToRead = bytesToRead - bytesRead;
    if (bytesLeftToRead > 0)
        Thread.Sleep(200);
}
```

Figure 27: Read all bytes

We let the thread sleep for 200 milliseconds before attempting to read again if we could not read all the bytes, so we do not have to get to many failed attempts. This time is short compared to how long time users have to commit to their actions.

4 Server implementation

When the server starts it will first set up a socket to listen for clients and then it will start a lobby thread that will accept clients and communicate with them. Then the main thread will accept input from the administrator that can be used to terminate the server or print out server status. The lobby thread will create additional threads for each game that is created by the clients. To achieve this there are several data structures such as `Deck`, `Hand`, `Player`, `Client`, `Game`, `Serverdata` and `Gamedata`.

4.1 Deck

An important item in a card game is the deck. A deck is a set of 52 unique cards in four different suits, hearts, clubs, diamonds and spades. The cards have 14 different ranks from 1 to 14 and in the game of Chicago, the lowest card, the ace, can be used as rank 1 or rank 14. The suits have got their own values from 0 to 3 and the ranks are represented with the values 0 to 12 (0 is representing rank 1 and 14). With this system we can calculate what card a value represents. As an example given that diamonds have the value 2 and we want to know the card 10 of diamonds. A rank 10 card has the value 9 so using the formula in Figure 28 we get the card 35.

card = suit * 13 + value

Figure 28: Calculate card

Even more important is to calculate the reverse (see Figure 29). If we have the number 35 how will we find that it is the 10 of diamonds.

suit = card/13

value = card (mod 13)

Figure 29: Calculate suit and value

In addition we have defined the `EMPTY_CARD` as the value 52 since it is the next value after the highest value in the deck. Our goal when creating a deck was that the deck should be reusable in other programs and that it should be simple to use. We identified some actions that

are important when using a deck where the most important is to deal a card. You can deal cards as long as there are cards left in the deck but if all 52 cards are dealt, the deck will deal you a -1 to indicate that the deck is empty.

Another important action is to return a card to the deck. In most games when cards are returned they are put in a separate pile next to the deck and if the deck is empty, the cards in the pile can be reshuffled and reused. With this in mind we created the deck data structure with two arrays of 52 chars. The reason for using chars is that we only need 52 values for the cards and one extra for the empty card. The structure has two variables that contain the size of the arrays, one variable that tells us which of the arrays is active and lastly a variable that gives us the position in the active array.

To use a deck you first have to declare it and pass it to the `init_deck` function. This function will set all the variables, it will fill the active array with the values from 0 to 51 and at the end it will shuffle this array. The shuffle algorithm in Figure 30 will randomly select a card and swap it with the card at the first position. Then it will select another card randomly among the cards that are left and swap it with the card on the second position and so on.

```
for(i=0;i<size;i++){
  r=(rand()%(size-i))+I;
  swap(card[i],card[r]);
}
```

Figure 30: Shuffle deck algorithm

The shuffle function will shuffle the array that is active and it will take into consideration the size of the number of cards that is put into that array with its size variable. The `getCard` function will return the next card in the active array. If the array is empty it will check if there are cards in the other array and if there are, it will change active array, shuffle and return the first card. If there are no cards left in the deck the `EMPTY_CARD` is returned.

The last function that will be mentioned is the `returnCard` function and this function will take a card as input and try to put it in the array that is not active. The function will return 0 if it has succeeded or `FULL_DECK` if the deck is full. `FULL_DECK` is also defined as the number 52.

4.2 Hand

We found that a common structure needed for the Chicago game procedure is the hand. A hand is made up of 5 cards. The combination of cards gives the hand a strength, such as two cards of the same value will make a pair and all the cards in the same suit will make a flush, where a flush > pair. The hand module has got a `Hand` data structure and four public functions. The structure contains the 5 cards, the hands strength and two values that can be used to compare hands with the same strength.

4.2.1 Sort hand

In many situations it will be useful to sort the hand and therefore we have made a `handSort` function. This function uses the bubble sort algorithm and the reason for this choice is that it is a really simple algorithm to implement and it does not matter that it is not the fastest sorting algorithm since a hand only contains 5 items.

4.2.2 Find position in hand

Next is the `handFindPos` function. This function will loop through the five cards and try to find a match. If there is a matching card, then its position will be returned and if not the function returns `NOT_IN_HAND`. We have again chosen a linear algorithm that is not the fastest but by far the simplest with the same motivation that the search is only through 5 items.

4.2.3 Calculate hand strength

The `handCalcStrength` function will calculate the strength of the hand and store the result in the hand data structure. This function should be run before comparing hands with `handCmp` and the hand should be sorted. To calculate the strength we first check if the hand is a flush. In order to do this we compare the suits on the first card with the second one, the second with the third and so on. If we find a pair with different suits, then we return 0, and only if we can match all 5 cards do we return 1.

After the flush check we investigate if the hand is a straight. For this we exploit the fact that the hand is sorted and we subtract the value of the first card from the second, the second

from the third and so on. If the result is not 1 then it is not a straight and we return 0. If we find it is a straight we return 1. There is a special case here that needs to be checked as well. That is because an ace can represent both the value 1 in a low straight and 14 in a high straight. We have solved this special case by looking at the first card and if it is an ace we check the second card. If the second card is a 10 or a 2 then it can be a straight and the algorithm continues as above with comparing second and third.

If the `handCalcStrength` function has found both a flush and a straight then it has also found a straight flush and if it also finds an ace on the first position and a 10 on the second position it is a royal flush. If any of these hand strengths are found the function returns since it is impossible to have any other hand strength at the same time. If not the search goes on.

The function will create a temporary hand that contains only the values of the original hand's cards. It will then sort the temporary hand and after this it will start to go through it from the beginning and compare the first and second and so on and if it finds two values are the same it will count up an `issame` variable. If it in one try does not find a match or if it is matching the last pair of cards the function will check this variable and depending on the value it will see if a pair, three of a kind or a four of a kind is found. It will also set the `value1` and/or `value2` variables to store the value of for instance a pair. It will also, by looking at the current strength when finding a pair or three of a kind, find out if it is a two pair or a full house. As for the latter if a pair is found and the strength is set to three of a kind then it must be a full house. The values are set so that the `value1` is the first tie breaker and `value2` is the second one.

4.2.4 Compare hands

The last function is the `handCmp` function. As the name implies it will compare hands and return -1, 0 or 1 depending on which hand is the best or if it is a draw. It starts by comparing the hand's strength variables and if they are different the highest one wins. If they have equal strength then depending on the strength the values are compared. For all strengths, except two pair, only `value1` is compares, and for two pair `value2` is compared only if both hands have the same value in value1. If this has not helped to find out which hand has the better strength then the cards are compared one by one, first both hand's highest card, then their

second highest and so on. Here the function must check if a losing card is an ace since the ace is in value the lowest card but in reality the highest card and in this case the winning card.

4.3 Other useful data structures

We have created some data structures that are needed only to pack data that belongs together. Here we will discuss each of them briefly.

The `Player` structure represents a player and this structure contains a socket used to read and write to the player, a location used to save some searching when we need to find the player in the `Serverdata` structure.

The `Client` structure contains a pointer to a `Player` and a `Game` if the player is in a game. Also the structures used to create the client sockets are stored in the `Client` structure.

The game data structure contains a game thread variable, a location for finding the game in the `Serverdata` structure, an `ok` variable to indicate if all players are connected to the server, the `running` variable to indicate if the game is started or not. It also has a unique game id, the number of players the game can hold, the number of players currently in the game and the players in the game.

We have mentioned the `Serverdata` structure and this is used to share some variables used throughout the server. It contains a `done` variable that is used to tell if the program is about to stop executing, server socket and port number, a set of file descriptors for all clients and the clients in the lobby, an array with all the clients and an array with all the games.

In the Chicago game procedure the `Gamedata` structure is used to store all the variables used throughout the game.

4.4 Chicago game procedure (CGP)

The CGP module contains only one public function and that is the `chicago_game_process`. This function is used as a thread [4] for every game that is currently running on the server. It will receive game data as input. The game data includes the

name and number of players that is involved in the game. To be able to create this function we first created a flow chart (see Figure 31) with server, client and the traffic between them. The function starts with some initialization and after that, the game starts. Since the game is made up by rounds the function is used in a constant loop that evaluates whether the round ended the game or if a new round should start.

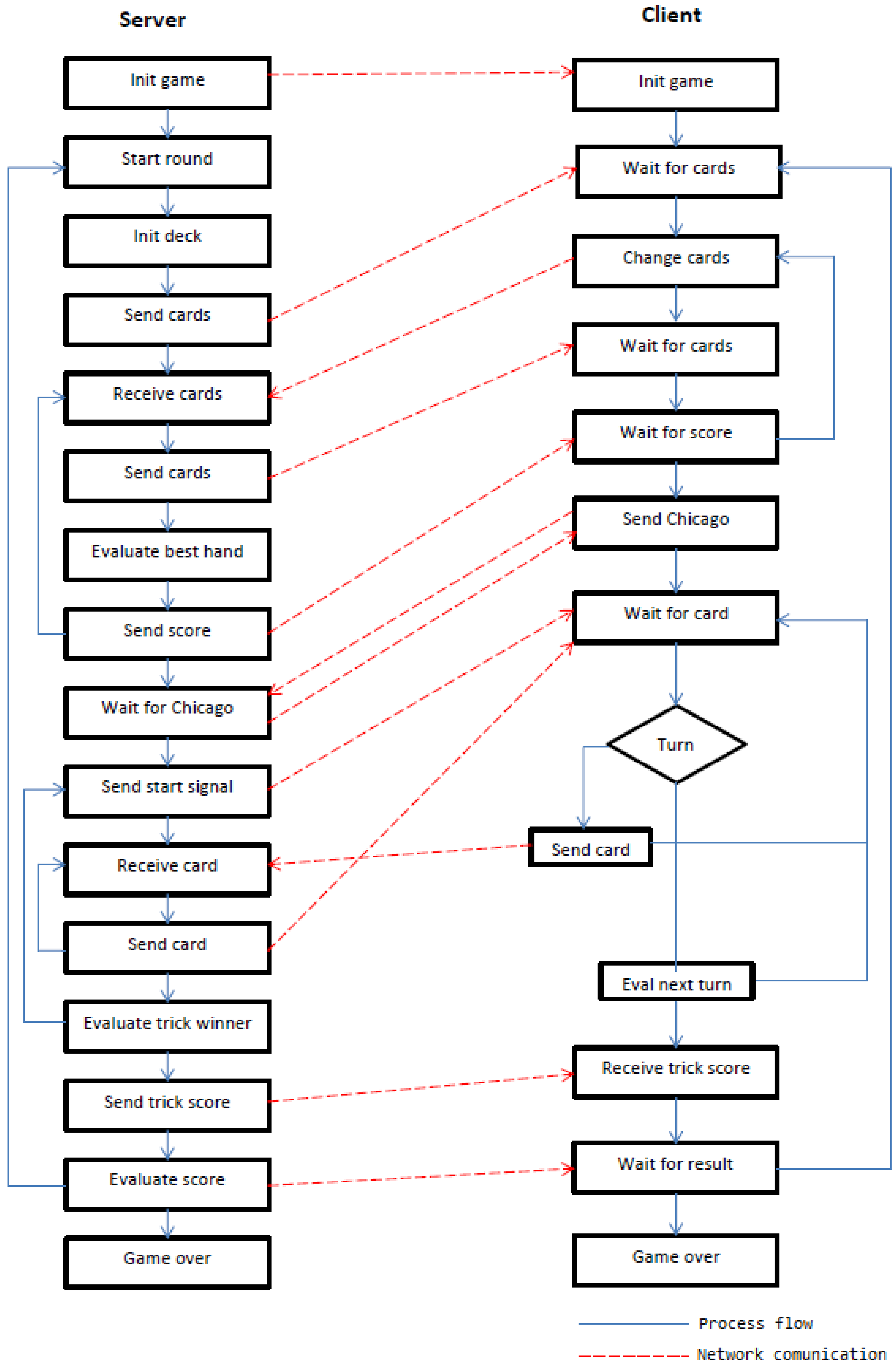

Figure 31: Flow chart

The round starts with the initialization of a deck. Then it picks 5 cards for every player in the game and sends the cards to respective client. The cards are always sent as a stream of 5 bytes. After sending all the cards the game waits for the clients to send back the cards they want to change. The returning cards are also sent as a 5 byte stream, were the values are set to the card values that should change or `EMPTY_CARD` for the bytes that are left over. Then the cards are changed and new hands are sent to the clients and then the function will calculate what hand is the best one, update the score and send it to the clients. The last steps are then repeated 2 more times. Now the round enters the next phase but first it will wait for all the clients to signal if they want to go for Chicago or not. If they send 1 they will go for it, otherwise they send 0. If two clients will send a 1 then the first client in turn will be the one going for Chicago.

The server will then calculate which client is the one to start and send a start signal, `EMPTY_CARD`, to that client. All the clients are in that moment waiting for cards but the one starting will read the card and find that it is a start signal and instead send back a card that it wants to play. This card, and the number of the client that played the card, is then sent to all the clients and the clients will calculate if they are the next in turn. The server will receive another card and send it etc. When all clients have played one card each the server will calculate who is next to play and a new start signal is sent and so on. All this is done 5 times. After this the game will calculate the new score and update the clients. Finally it will calculate if there is a winner and if it is then the round function will return 1. If there is no winner it will return 0.

The score that is sent to the client is made of an unsigned char for each player. We have chosen this data type for two reasons. First we do not need a bigger data type because the goal is to get up to 52 points and the unsigned char has a maximum value of 127. The second reason is that a player can lose points if he fails a Chicago attempt and therefore end up in a negative score. It is theoretically possible to reach the maximum or minimum values and therefore the program do check for overflow and underflow of these variables and just do not allow it to happen. If for example an addition 126 + 3 is made, then the result will end up as 127.

4.5 Lobby

The server will have to handle requests from multiple clients such as joining, creating and removing games. To do this we created the lobby. The lobby is a thread [4] that loops through the same sequence over and over until the server shuts down. The sequence is as follows.

1) If server is running, run the `select` function using a timeout of ten seconds.

2) If `select` [3] returned a value greater than zero there is data to be read from at least one socket.

3) Check if the server socket can be read and if it can accept a new client connection.

4) Loop through all the client sockets and if a socket can be read, read one byte and store it as the `request` variable. If the byte could not be read, the client is disconnected.

5) If the client was not disconnected handle the request.

4.5.1 Select

When handling multiple sockets there is the problem that some clients send data while others are idle and the `read` function call will block until data is read or the client is disconnected. It would be good to have a method to check which sockets have data to read in order to only apply the `read` function on those sockets. To handle this situation we use the `select` function [3] (see Figure 32).

```
retval=select(FD_SETSIZE,&read_fds,NULL,NULL,&tv);
```

Figure 32: Select function

This function has five arguments. The first argument needs the maximum size of the set that is used. We use the `FD_SETSIZE` here, which is the maximum size of the set. The second argument is a pointer to a set of file descriptors that should be checked for reading. In our program we have all the client sockets that are currently in the lobby and the server socket in this set. The third argument is a set of file descriptor that should be checked for writing and the fourth argument is a set of file descriptors that should be checked for errors. We only need to control whether the sockets can be read and therefore set the third and fourth argument to `NULL`. The last argument needs a time value structure and this is used to set the time the `select` function should wait for sockets to be ready for reading. If the `select` function should not timeout, but instead block until one of the sockets can be read, this argument can

be set to `NULL`. As soon as at least one socket can be read the `select` function returns the number of file descriptors in the set that is ready. The file descriptor sets can be manipulated with the macros `FD_SET` to add a file descriptor, `FD_CLR` to remove a file descriptor, `FD_ISSET` to check if a file descriptor is ready, `FD_ZERO` to make the set empty.

4.5.2 Client requests

The server will read a byte from the lobby and depending in this bytes value it will do one of four actions.

`NEW_GAME`: If the request was a new game then the server will read another byte from the client representing the number of players that should be in the game. Then it will try to find an open spot in the game array and if there is room for more games it will create a new game and store it at that free location. The creation of a new game is made by first finding a unique game id. This is done by using an unsigned static integer variable that will increase its value by one every time a new game is created. We need to check all other games so that there is no game already using that id. This is because the integer will overflow when it has reached its maximum value and start over from zero, and in theory a game could still be in progress even if many thousands of games have ended. In the worst case scenario the server needs to try as many ids as there are games, so if there are n games it will have to perform n times n operations. When a game has been created a new game message is sent to the client or if an error arises an error message is sent. In the end the game will be updated to all clients in the lobby.

`JOIN_GAME`: When the server has received a join game request it must read 4 more bytes that represent the game id to join. Then the server must find the game in the game list and add the player to the game if it exists. After this it will compare the number of players entered in the game to the number of players that the game holds and if these values are the same the games `running` variable will be set to true. Next the server will send a join game message to indicate that the client joined the game successfully. After this reply the server will check the `running` variable and if it is true it will start the game. If the game could not be created an error message is sent to the client instead of the join game message. Also after the game has changed it will be updated to all the clients in the lobby.

LEAVE_GAME: After the server has received a leave game request it will first check which game the client is in. If the game's running variable is set to true or if for any reason the game does not exist, the leave game request is just ignored. Otherwise the client is removed from the game and a leave game reply is sent to the client. After this the server checks the number of players still in the game and if it has become zero the game is removed. If the game is removed all the clients are informed of the removal of the game and if it is not all clients will receive an update of the game.

LIST_OF_GAMES: If the client has sent a list of game request the server will first send a list of games reply followed by a game update for every game currently in the lobby.

4.5.2 Update game

When a game has been created or changed, the server needs to update the clients. The server contains a function that will send all the game data of a game to all the clients that are in the lobby. This function uses a function that will update only one client with the relevant game data. This function is also used when the client request a list of all the games currently in the lobby. There is also a remove game message that can be sent to all clients if the game has been removed.

4.5.3 Start game

When a game has enough players entered it is time to start the game. To do this the server first removes all the clients from the lobby by clearing the client sockets from the lobby file descriptor set. Then a GAME_START message is sent to all the clients involved in the game and when all clients have been notified the game thread [4] is created with the pthread_create function as in Figure 33.

```
pthread_create(&g->gameThread, NULL, chicago_game_process, (void *) cgpin);
```

Figure 33: Game start

The last argument to the pthread_create function is the input to the chicago_game_process and this structure has got a pointer to the game data and the server data. After the game thread has been created a remove game message is sent to all clients that are still in the lobby.

4.5.4 Disconnect

If the server reads from a client and the expected amount of data could not be read, the client has disconnected. When the server finds a disconnected client in the lobby it has to remove all data connected with that client. The first action the server takes is to remove the clients socket from the lobby file descriptor set. Then it has to control whether the client was currently in a game. If the client was in a game then the client is removed from that game and all lobby clients will be updated. The last action the server takes is to close the disconnected clients socket and free all data connected to the client.

4.6 Threads

The server uses threads [4] to be able to run several functions simultaneously. To create a thread in c we first need to declare a thread variable (see Figure 34).

```
pthread_t lobbyThread;
```

Figure 34: Thread declaration

To create the thread we run the `pthread_create` function call (see Figure 35). This function takes 4 arguments as input. The first argument is a pointer to the thread variable that should be associated with the thread. The second argument is used to set specific attributes for the thread but it is also possible to use the default attribute by setting this argument to `NULL`. The third argument is the function that should run as the thread and the last argument is a pointer to the data structure sent as argument to the thread function.

```
pthread_create(&lobbyThread, NULL, lobbyThreadFunction, (void *) &sd);
```

Figure 35: Creating a thread (C)

All the threads we use will run to the end of the functions used, or they will stop if the `exit` function is called.

4.7 Sockets

To be able to communicate over the internet we need some way to abstract the network. We also need the data to arrive at its destination without getting lost or corrupted. The answer is to use sockets [3]. We use two kinds of sockets, a server socket and several client sockets. The server socket is used to listen for new clients in order to receive client sockets. To create

a server socket (see Figure 36) we call the socket function. We use TCP sockets since they will guarantee that the data sent will be correct and that no data will be lost as long as there is a connection. It is important since all the data must have the correct values or the system would fail. After the creation of the server socket we need to bind it to a port so that the packets sent to the server will find the correct process.

```
sd.server=socket(AF_INET,SOCK_STREAM,0);
if(sd.server<0){
    error("ERROR opening socket");
}
bzero((char *)&serv_addr,sizeof(serv_addr));
sd.port=atoi(argv[1]);
serv_addr.sin_family=AF_INET;
serv_addr.sin_addr.s_addr=INADDR_ANY;
serv_addr.sin_port=htons(sd.port);
if(bind(sd.server,(struct sockaddr *)&serv_addr,sizeof(serv_addr))<0){
    error("ERROR on binding");
}
```

Figure 36: Creating a server socket

With this socket we can now listen for clients as in Figure 37.

```
listen(sd->server,5);
```

Figure 37: Listen on socket

After we have called the listen function we can accept a client (see Figure 38) and this will return a client socket that we can use to send and receive data on with the `read` and `write` system calls. These functions need a socket, a buffer and the size to be read or written as arguments.

```
c->clilen=sizeof(c->cli_addr);
c->p.socket=accept(sd->server,(struct sockaddr *)&(c->cli_addr),&(c->clilen));
```

Figure 38: Accepting a client

We wrapped these functions into a `receiveFromClient` and a `sendToClient` function that has the same arguments, but they also print out the data sent, so it is easier to monitor the traffic for debugging, and it also returns true or false depending of if it read or wrote as many bytes as the size argument. We can then use this to find disconnected clients in the system and then remove them. It is rather easy as long as the clients are in the lobby but it is a little harder if they are involved in a game. We solved this by adding an `ok` byte that we

sent before every message during a game. If the server finds a disconnected client in a game then the games `ok` variable will be flagged to false and the other clients will be informed of this the next time data is sent to them. Then the server will add those clients to the lobby, remove the game and end the game thread.

5 Client – Server communication

To be able to coordinate the server and the multiple clients we had to decide which data to send and how to structure the data. When a client is in the lobby it needs to be able to execute requests like creating a game or joining a game, and while a client is in a game then the data must follow the game's flow such as sending cards to change and receiving new cards.

5.1 Lobby mode

When a client is in the lobby it will communicate with the server by sending and receiving messages. We call them client lobby messages and server lobby messages.

5.1.1 Client lobby messages

The client can send four messages.

1) The Create game message is two bytes where the first byte is the `NEW_GAME` value and the second is the number of players that the game should hold.

2) The Join game message is five bytes long, where the first is the `JOIN_GAME` value and the other four represent an integer value which is the game id.

3) The Leave game message is one byte with the `LEAVE_GAME` value.

4) The List of game message is on byte with the `LIST_OF_GAME` value.

5.1.2 Server lobby messages

The server can send six messages.

1) The New game message is five bytes where the first byte is the `NEW_GAME` value and the other four represent an integer value which is the game id.

2) The Leave game message is one byte with the `LEAVE_GAME` value.

3) The List of game message is one byte with the `LIST_OF_GAME` value.

4) The Game start message is one byte with the `GAME_START` value.

5) The Game update contains a variable amount on bytes depending on how many players that are currently in the game and how many letters there are in the player names. The message is created starting with seven bytes where the first byte is the `GAME_UPDATE` value. The next four bytes represent an integer value which is the game id. The sixth byte is the number of players the game holds and the seventh byte is the number of players entered. Then

for every player currently in the game, one byte with the players name and then that amount of bytes holding the characters of the name is sent.

6) The Remove game message is five bytes where the first is the `REMOVE_GAME` value and the other four represent an integer value which is the game id.

5.2 Game mode

We will here explain which data is sent between the server and one of the clients during the initiation of a game and the first round of the game. All the other clients will communicate in exactly the same way with the only difference being that some of the values, such as card values, are different. Added to all the messages from the server to the clients we have one byte informing the clients if the status of the game is ok, that is that no client is disconnected. If the server finds a disconnected client it will set this byte to false and after sending it to the remaining clients it will put them back in the lobby, remove the game and end the game thread.

5.2.1 Game initiation

The first message that is sent from the server during the initiation of the game is one byte with the number of players in the game. Next is a byte telling the client what position the client is on in the perspective of the server. At the end the server will send one byte with the length of the name of a player followed by that player name for every player in the game.

5.2.2 Changing phase

The cards are sent to the client as five bytes representing the five cards. The client sends five bytes back to change the cards and the values of the cards should be the cards that should be changed or the `EMPTY_CARD` value if it should not be changed. When cards have been changed the score is sent to the client by sending two bytes with the player that received the last score and the number of points acquired. These two bytes are followed with one byte for every player in the game where every single byte is a player’s total score. After the changing phase is completed the client send one byte with a decision of going for Chicago or not and the server responds with a byte that either represents the player going for Chicago, or a value that tells the client that no one has gone for Chicago.

5.2.3 Trick phase

Every trick in the trick phase starts with the server sending a start signal to the starting client. This start signal is one byte with the `EMPTY_CARD` value. When a client plays a card it will send one byte with the value of the card and the server will send the value of a played card as one byte to the client. When the trick game is done the server will send one byte that can have three different values depending on if Chicago has succeeded or not or if no player went for Chicago in that trick game. Then one byte with the trick winner is sent followed by one byte for every player with the players total score. At the end of the round two additional bytes are sent to the client where the first tells if there is a winner of the game or not and the second tells who the winner is if the game is over.

6 Feedback and future implementations

The Chicago program can always be improved in the future and we will here discuss some of the features that we think should be implemented in the future such as chat, settings and timer. We will also discuss some of the feedback we have received from users who have tested the Chicago client.

6.1 Chat

Since the user always plays the Chicago game with other users it would be good for the users to be able to communicate with each other. This would improve the experience for the users and it would also help the users to organize themselves while creating and joining games. It would also be good if there were different chats depending on which game the user is in, or if the user is in the lobby. In order to implement a chat for the Chicago system we have discussed two ways of implementing it. One way would be to send chat messages on the current TCP connections in between the lobby traffic or game traffic currently going on during execution. To make this idea work we would have to change the implementation in a number of places on the server and we would have to add an extra byte to mark whether the data is a chat message or not. We think that this is an impractical idea since we would have to do a number of changes throughout the code in both the client and server. The second way of doing this is to create a new chat process on the server that will accept new TCP connections. We also create a chat class on the client that will connect to the chat process. Then we have all the chat traffic on its own connection. A model of this solution is shown in Figure 39. Depending on how we choose to send the chat messages we think that the chat needs a system involving chat channels so that there can be one private chat for every game. The chat class will have methods for joining and leaving a chat channel. The client will either be in the lobby channel or a channel with the id corresponding to the id of the game the user is currently in. The chat class will also have methods for sending chat messages and the server will send this message to all the clients that are in the same chat channel.

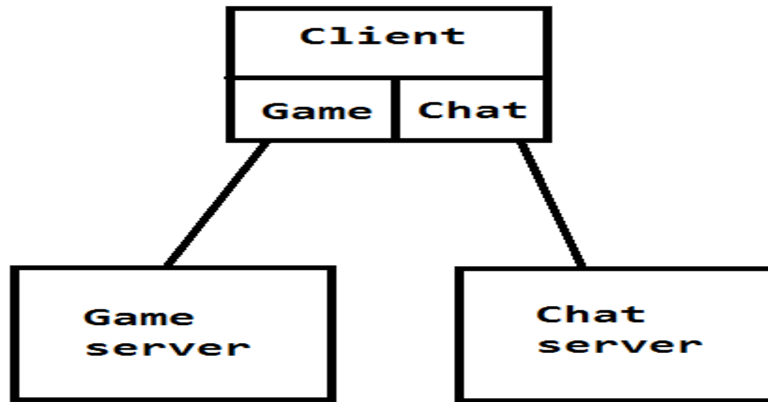

Figure 39: Chat model

6.2 Settings

It would be good to let the user have some control over the client in order to change the client's appearance. Aspects that could be changed are the background colour, card images, sounds and fonts. To implement these we would first create a menu with these kinds of options. To change the background colour or fonts we only need to change the variables that control this. As for the other options we need to change the path string, to the folder were images or sounds are located.

6.3 Timer

A problem we thought about from the start was the event of a user taking too long to make his moves. It will not be a pleasant game session if every change of cards takes several minutes or if a user will leave his computer to let the other users wait. To address this problem we want to implement a timer that will tick down every time a player has to make a decision and if the timer runs out the program will make the decision for the user. Aspects to take in mind here are how long the timer should tick and what decisions should be made. Our thoughts on this are that the time should be around 20 seconds to interact, but that time could be changed if a majority of users experience this to be too long or short. When the client changes cards the default decision should be to change the currently marked cards. If the choice is to go for Chicago or not, the default behaviour should be to not go for Chicago since this is the most common choice. If the choice is to play a card in the trick game the card chosen could be the lowest card, the highest card or a random card. We think the lowest card

should be the default option since the user might want to have the high cards left to have a chance to win last trick. In any case the card has to be in the correct suit if such a card exists and if the user is the first to act then the suit could be selected randomly or by the suit value.

6.4 Feedback

We have been playing the Chicago game and we have also had other people play the game and this has given us a certain amount of feedback from ourselves and the other persons. Here we will discuss the feedback and our reflections.

The first aspect we found out was that the game needed some way to tell the user when it is time to act. We and others often found ourselves in a situation when we did not know that it was our turn to act and therefore the game stalled. To solve this problem we have implemented a sound that will be heard when it is time to act. We have also thought about adding a flashing colour that could attract the attention of the player, but we decided that the sound is enough.

The next problem was that some users did not like the way the cards were played in the trick game. As it is now, the user has to click on a card to mark it and then click the send button to play it. Some users wanted to be able to drag the card to the middle of the window. It is probably possible to implement this feature but we found that a compromise could be to play the card just by double clicking on it. This is a really easy solution compared to be able to drag the card.

At first it was very confusing to know how the opponents were seated compared to the user. This was solved by adding labels that shows the user names and their location.

7 Languages & Tools

In this chapter we will briefly comment on some of the programs and programming languages we have used during this project.

7.1 C# and Visual Studio

We used C# as our programming language for the client since it is an easy to use for making windows application. To our help we had visual studios so that we could use the built-in window designer. The designer is an easy way to create window forms by dragging controllers to the form and changing their properties in a list of settings.

7.2 C language

For the server we used the C language because it is a fast language and a good choice for making programs for Linux systems. Speed is an important part since there will be a lot of pressure on the server if we increase the number of clients and games allowed.

7.3 Putty and WinSCP

To be able to run and test the server we used a Linux computer that has access to a 100 Mbit network. In order to upload files to this computer we used WinSCP [16] that is a graphical file manager application that use the SSH protocol [15]. We also used the Putty [10] application to be able to compile and run the server. Putty is a console application that also uses the SSH protocol.

8 Result and evaluation

Our main goals with this project were to create a server process and a client to be able to play a game of Chicago, and to add a lobby system for managing multiple clients and games. We have met these goals even though the Chicago system always has room for improvements in the future. We also had secondary goals such as implementing a chat system and time control in the game but these have not yet been implemented.

First we created a server process that could handle a single game and a Client to play the game on. Then we tested the game so that the game logic would be correct and we also made changes due to feedback we received. Next step was to use this process and client in a multiplayer system that also could handle multiple games. To this end we created the lobby system where users can create, join and leave a game of Chicago.

On the client side we created the Lobby class for user input and graphical output and the `LobbySession` class for server communication while in the Lobby. We also created the Chicago class for user input and graphical output and the `GameSession` class for server communication and controlling the game flow during a game session.

On the server side we identified and implemented some important data structures that we used such as `Deck`, `Hand`, `Client` and `Game`, and algorithms to calculate hand strength, compare hands and shuffle a deck of cards. We created the Chicago game procedure to use for every instance of a game running on the server and a lobby to communicate with the clients so that games could be managed.

We used threading on both client and server to be able to have multiple actions made simultaneously and sockets to send data over the internet.

During the project we have had some problems where most of them were easy solved with common debugging, but there were two problems that gave us some unexpected setbacks. The first was the problem with threads [5] in combination with windows controllers. This problem was found during runtime when an exception was thrown and it was not obvious at

first how to solve this issue. The second problem was that we did not add disconnection detection to the system until late in the project. This gave us about 10 hours of extra work and complication and this could have been much easier and had a better design if we would have spent more time and planning on this matter from the start.

When we have tested the game ourselves and with others we have received feedback and we have used this to improve the system especially on the client. We have found small bugs just by playing the game and trying all kinds of scenarios. One of the bugs we found was when one of the players had acquired a straight flush for the first time resulting in a system crash. This was not a hard problem to solve since it was due to a typing error that still could compile. Feedback has also made us aware of problems we missed at the planning stage such as notifying the user that it is time to act and that users have different expectations on how to interact with the client.

By working on the Chicago project we have learned how to design a server system that handles multiple clients and what difficulties it can bring. We have learned how to handle multiple clients with the use of the select function [3]. We have also learned how to set up a server behind a router with the use of port forwarding [14], and using SSH [15] to upload files, compile and start the server remotely.

9 Conclusion

The project resulted in a multiplayer card game based on client – server architecture. The server was developed in C and the client in C#. As a whole we are happy with what we have achieved during the project. The next time we undertake a project like this one we would probably organize the network parts better and have a plan for disconnections and other problems that have to be addressed from the beginning. We will also know that we have to do specific set and get methods for window controllers so that they will work from other threads and this should probably be done as soon as the controller is added to the project. Also we would have liked to test the system a little bit more than we have, but this will be done as the project will continue.

References

[1] Blair, G., Coulouris, G., Dollimore, J. & Kindberg, T. (eds.) (2007). *Distributed Systems: Concepts and Design.*(5th edn.). Addison-Wesley.

[2] Wikipedia. *MMORPG.* [Online] Available from: http://en.wikipedia.org/wiki/MMORPG [2013-05-20].

[3] Hall, B. *Beej´s Guide to Network Programming.* [Online] Available from: http://beej.us/guide/bgnet/output/html/multipage/index.html [2013-05-20].

[4] Ippolito, G. *POSIX thread (pthread) libraries.* [Online] Available from: http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html [2013-05-20].

[5] Microsoft. *How to: Make Thread-Safe Calls to Windows Forms Controls.* [Online] Available from: http://msdn.microsoft.com/en-us/library/ms171728.aspx [2013-05-20].

[6] Microsoft. *Image.FromFile Method (String).* [Online] Available from: http://msdn.microsoft.com/en-us/library/stf701f5.aspx [2013-05-20].

[7] Microsoft. *ListBox Class.* [Online] Available from: http://msdn.microsoft.com/en-us/library/system.windows.forms.listbox.aspx [2013-05-20].

[8] Microsoft. *Socket Class.* [Online] Available from: http://msdn.microsoft.com/en-us/library/system.net.sockets.socket.aspx [2013-05-20].

[9] Wikipedia. FPS*.* [Online] Available from: http://en.wikipedia.org/wiki/First-person_shooter[2013-05-20].

[10] Tatham, S. *Putty.* [Online] Available from: http://www.putty.org [2013-05-20].

[11] Wikipedia. *Card game.* [Online] Available from: http://en.wikipedia.org/wiki/Card_game [2013-05-20].

[12] Wikipedia. *Chicago (poker card game).* [Online] Available from: http://en.wikipedia.org/wiki/Chicago_(poker_card_game) [2013-05-20].

[13] Wikipedia. *Online Game.* [Online] Available from: https://en.wikipedia.org/wiki/Online_game [2013-05-20].

[14] Wikipedia. *Port Forwarding.* [Online] Available from: http://en.wikipedia.org/wiki/Port_forwarding [2013-05-20].

[15] Wikipedia. *Secure Shell.* [Online] Available from: http://en.wikipedia.org/wiki/Secure_Shell [2013-05-20].

[16] WinSCP. *WinSCP.* [Online] Available from: http://www.winscp.net [2013-05-20].

[17] Microsoft. *Threading Tutorial.* [Online] Available from: http://msdn.microsoft.com/en-us/library/aa645740(v=vs.71).aspx [2013-05-20].

[18] Wikipedia. *RTS.* [Online] Available from: http://en.wikipedia.org/wiki/Real-time_strategy[2013-05-20].

[19] Wikipedia. *Peer to Peer.* [Online] Available from: http://en.wikipedia.org/wiki/Peer-to-peer [2013-05-20].

[19] Wikipedia. *Peer to Peer.* [Online] Available from: http://en.wikipedia.org/wiki/Peer-to-peer [2013-05-20].

[20] Microsoft. *Windows Forms Application.* [Online] Available from: http://msdn.microsoft.com/en-us/library/system.windows.forms.application.aspx[2013-05-20].