Multiplayer Network Game Programming in MFC

-A Case Study of Video Poker

Li Zhang

A Major Report

In

Department

Of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montreal, Quebec, Canada

April 2004

© Li Zhang, 2004

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisisitons et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-91154-3
Our file Notre référence
ISBN: 0-612-91154-3

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Multi-Player Network Game Programming

- A Case Study of video poker

Li Zhang

On-line multi-player games have become popular in recent years. In this report, we review the techniques developed for improving networking game. We overview the feature of the multip-player game for real-time reactive system in Internet environment. The VPNG (Video Poker Networking Game) system is mainly responsible for multiple players playing game online. This report describers the design and implementation of VPNG system. C++ has been chosen as the implementation platform because is can achieve capability. A Client/Server model is used for this system. The key algorithms used are presented in details as well as the interactions among the underlying objects.

// Acknowledgement

I would like to express my deepest gratitude to my supervisor, Dr. Peter Grogono.

He gave me helpful guidance and advices all along the way.

Also, I would like to thank my family for the support during the years of my graduate studies.

Table Contents

Chapter 1 Introduction 1

1.1 A Brief History of Computer Games 1

1.1.1 History of Computer Game 1

1.1.2 History of Multiplayer Online Game 3

1.2 Status of Computer Game World 4

1.3 Purpose and Problem Statement 7

1.4 Report Outline 7

Chapter 2 Background and Related Technology 8

2.1 Multiplayer Game Overview 8

2.2.1 Single-Machine Multiplayer Games 12

2.2.2 Local Multiplayer Games 13

2.2.3 Networked Multiplayer Games 14

2.2 Multiplayer Networking Game Programming Models 17

2.2.1 The Client / Server Model 22

2.2.2 Peer-to Peer Model 26

2.2.3 Host-Terminal Architecture 27

2.2.4 The Comparison of these Models 28

2.3 Multiplayer Networking Game Programming Feature 28

2.3.1 Synchronization 29

2.3.2 Multiplayer Scalability 29

2.3.3 Robustness 29

2.3.4 Area of Interest Management 30
2.3.5 Dead Rechoning 30
2.3.6 Object Scalability 30
2.3.7 Bandwidth Management 31
2.3.8 Faireness 31
2.3.9 Persistence 31
2.3.10 Rapid Database Access 32
2.3.11 Reliable, but fast protocols 32
Chapter 3. Windows Sockets Network Programming 33
3.1 Windows Sockets Concepts **33**
3.2.1 What is Socket 33
3.2.2 What is Windows Socket 35
3.2.3 A Socket Data Type 38
3.2.4 Uses of Socket 38
3.2 Network Program Mechanics **38**
3.2.1 Open a Socket 38
3.2.2 Name the Socket 38
3.2.3 Associate with Another Socket 39
3.2.4 Send and Receive Between Sockets 39
3.2.5 Close the Socket 39
3.3 Sockets Programming Models **39**
3.3.1 Windows Sockets Classes 40
3.3.2 CSocket Programming Model 42
3.4 Using Sockets with Archives **44**

3.5 How Sockets with Archives Work ... **44**

Chapter 4. Video Poker Game ... 50

4.1 Rules of the Game ... **50**

4.2 Basic Video Poker Hands ... **51**

Chapter 5. VPNG System Design ... 54

5.1 Description of the System ... **54**

5.1.1 System Functionalities ... 54

5.1.2 System Characteries ... 54

5.2 System Architecture ... **56**

5.4 Structure Model ... **57**

5.4.1 Foundation Classes in MFC ... 57

5.4.2 Class Diagram of Server Subsystem ... 58

5.4.3 Class Diagram of Client Subsystem ... 58

5.4.4 Classes Description in VPNG System ... 59

5.5 Behavioural Models ... **71**

5.5.1 Sequence Diagram ... 71

5.5.2 Activity Diagram ... 72

6.1 Pseudo code of VPNG System ... **74**

6.1.1 Pseudo Code of Client Side ... 74

6.1.2 Pseudo Code of Server Side ... 79

6.2 Interface ... **83**

6.2.1 Game Interface : Server Side ... 84

6.2.2 Game Interface : Client Side ... 86

Chapter 7 Conclusion & Future Work ... 95

7.1 Conclusion **95**

7.2 Future Work **95**

References 98

Appendix A. Installation of Server and Client Application 108

Appendix B. Setting up of Server and Client Application 108

Appendix C. Procedures to run demo Application 108

Appendix D. Glossary 108

Lists of Figures

Figure 1. Peer-to-peer startup protocol state diagram 19

Figure 2 Client-Server Programming Models 25

Figure 3 Windows Socket Classes 41

Figure 4 The Architecture Diagram of the VPNG 56

Figure 5 Class Diagram of Server Subsystem 58

Figure 6 The Class Diagram of the Client Subsystem 59

Figure 7 The Sequence Diagram for Client Subsystem 71

Figure 8 The Sequence Diagram for Server Subsystem 72

Figure 9 The Activity Diagram of Client Subsystem 73

Figure 10 The Activity Diagram of Server Subsystem 73

Figure 11 Objects in a Running SDI Application 84

Figure 12 Setup one game group on server 84

Figure 13 Interface for connection with players 85

Figure 14 Interface when game is over 85

Figure 15 Setup interface for player 86

Figure 16 Interface of the first trun for player Mike 87

Figure 17 Interface of first turn for player Lilian 87

Figure 18 Interface of first turn for player Sophie 88

Figure 19 Interface of second turn for player Mike 89

Figure 20 Interface of second turn for player Lilian 89

Figure 21 Interface of second turn for player Sophie 89

Figure 22 Interface of third turn for player Mike... 90

Figure 23 Interface of third turn for player Lilian... 90

Figure 24 Interface of third turn for player Sophie... 91

Figure 25 Interface of fourth turn of player Mike... 91

Figure 26 Interface of fourth turn for player Lilian 92

Figure 27 Interface of fourth turn for player Sophie... 92

Figure 28 Interface of last turn for player Mike... 93

Figure 29 Interface of last turn for player Lilian... 93

Figure 30 Interface of last turn for player Sophie... 94

List of Tables

Table 1 Setting Up Communication Between a Server and a Client 47

Table 2 VPNG system characters 55

Table 3 An example for total number of players on one server.................................... 56

Chapter 1 Introduction

Since the release of the first multiplayer computer game, the level of attention game developers have devoted to the network aspect has dramatically increased as games continue to grow in quality and complexity. The size and behavior of the networks on which such games have been designed to run are also changing rapidly, and games are becoming increasingly demanding in term of resource.

The purpose of this document is to overview multiplayer game and related technology, to introduce the socket network programming , and to describe the of VPNG (Video Poker Networking Game) system design and implementation.

1.1 A Brief History of Computer Games

1.1.1 History of Computer Game

Playing games on computers was first made possible by the introduction of minicomputers in the late 1950s. Freed from the IBM punch card bureaucracy, programmers for the first time were able to explore the possibilities opened up by hands-on interaction with computers. Games were among the first programs attempted by the original "hackers," undergraduate members of MIT's Tech Model Railroad Club. The result, in 1962, was the collaborative development of the first computer game: Spacewar, a basic version of what would become the Asteroids arcade game, played on a $120,000 DEC PDP-1. (Levy, 1984; Wilson, 1992; Laurel, 1993) Computer designer Brenda Laurel points out this early

recognition of the centrality of computer games as models of human-computer interaction:

Why was Spacewar the "natural" thing to build with this new technology? Why not a pie chart or an automated kaleidoscope or a desktop? Its designers identified action as the key ingredient and conceived Spacewar as a game that could provide a good balance between thinking and doing for its players. They regarded the computer as a machine naturally suited for representing things that you could see, control, and play with. Its interesting potential lay not in its ability to perform calculations but in its capacity to represent action in which humans could participate (Laurel, 1993, p. 1).

As computers became more accessible to university researchers through the 1960s, several genres of computer games emerged. Programmers developed chess programs sophisticated enough to defeat humans. The first computer role-playing game, Adventure, was written at Stanford in the 1960s: by typing short phrases, you could control the adventures of a character trekking through a magical landscape while solving puzzles. And in 1970 Scientific American columnist Martin Gardner introduced Americans to LIFE, a simulation of cellular growth patterns written by British mathematician John Conway. LIFE was the first "software toy," an addictively open-ended model of systemic development designed to be endlessly tinkered with and enjoyed (Levy, 1984; Wilson, 1992).

The 1970s, of course, saw the birth of the video arcade, the home video game system, and the personal computer. By the early 1980s, computer game software production had become an industry (Wilson, 1992). And in the past fifteen years, as personal computers' capacities have continued to exponentially expand, computer games have continued to develop, offering increasingly detailed graphics and sounds, growing opportunities for multiple-player interaction via modems and on-line services, and ever-more sophisticated simulation algorithms.

The world of computer games today ranges from arcade-style games emphasizing hand-eye coordination, to role-playing games adding sound and video to the Adventure formula, to simulation games in which players oversee the growth and development of systems ranging from cities to galaxies to alternate life-forms. Computer game publications divide the contemporary field into seven genres: action/arcade, adventure, role-playing adventure, simulation, sports, strategy, and war. Within these categories, of course, there remains much overlap. An empire-building game like Civilization, for example, rests somewhere between a wargame and a simulation, while many adventure games contain arcade-style interludes.1

1.1.2 History of Multiplayer Online Game

This section gives a brief overview in the history of multiplayer online games.

1969 Rick Blomme wrote the famous *Specewar*. The first game that worked on a remote network.

1980 The first MUD (Multi User Dungeon) was released on ARPANet (the basic of the Internet).

1984 The beginning of commercial online gaming. Playing *islands of Kesmai* for an hour cost about $12.

1987 The first " real" graphics-based MMOG *Air Warrior* was released.

1993 ANet was becoming increassingly available to a wider audience and was becoming known by the public as the Internet.

1993 The famous 3D game Doom by ID Software was released. The first game which could be played over a network for up to four players.

1996 ID software released Quake, which featured for the first time a built-in internet-play capability.

1996 3DO Company launched *Mridian 59,* the first commercial 3D graphical MUD.

1997 Origin launched *Ultima Online*. The first major MMORPG (Massive Multiplayer online Online Role Playing Game). 50'000 players in the first three months.

1999 Verant Interactive launched *Evrquest.* The first 3D MMORPG.

1.2 Status of Computer Game World

The computer games have rapidly become a significant and expanding field of entertainment industry and modern culture. Various conceptual and theoretical models to understand games and how they work are being created, while the games themselves are growing into new dimensions with their online and multiplayer capabilities. The transition into the world of mobile gaming is creating even more challenges and further possibilities.

The computer games are a relatively new innovation in the overall scheme of things. They have been around in different forms since the beginning of computers and in a lot of ways were essential in the route that computers have taken in becoming a part of our every day lives.

The lack of research into network gaming was never a problem . Before Doom [1] released in 1993, nearly all networked games were text based and used telnet or similar protocols to transmit data from player to server and back. But even with the advent of Doom, networked gaming was still confined to a small portion of the population. However, in the last 5 years, with the growth of the Internet, this has changed drastically. In the Internet environment, the vast majority of networked gamers play card games, chess, checkers, and similar games. The genres of games that have the most players, after parlor games, are First Person Shooters (FPS) and Massively Multiplayer Online Role Playing Games (MMORPGs), followed closely by Real Time Strategy (RTS) games.

Since Doom, FPS have made up a large portion of networked gaming. In these

games, the player views the world through the perspective of this character (the first person part) and is usually required to move around various locations slaying monsters and other players, with an amalgamation of ranged weaponry found along the way (the shooter part). On an average night, there are well over 10,000 servers for games using the Half-Life engine supporting over 40,000 gamers. Other FPSs support slightly smaller user populations.

MMORPGs have been a rapidly growing field since Ultima Online's[7] release in 1996. A MMORPG can be safely thought of as a graphical Multi-User Dungeon.[8] All MMORPGs released thus far provide some mechanism for character advancement, large areas of landmass to travel across, and other players to interact with. The "big three," Asheron's Call,[9] Ultima Online,[10] and Everquest,[11] claim to have nearly 1 million subscribers combined, and while only a fifth of them login on any given day,[12] these players consume a non-negligible amount of bandwidth. In addition, several more MMORPGs have been released in recent months,[13] adding to this total.

The first RTS game was Dune 2,[14] which was based loosely on the world from the Frank Herbert series of novels. RTS games are generally characterized by resource collection, unit construction, and battles that consist of large numbers of animated soldiers standing a few feet apart going through the same animated attack motion over and over. All of these actions happen continuously, unlike earlier strategy games (most notably Civilization[15] and various war games from SSI and others) in which the player could take as much time as he or she needed

to plan his or her turn before pressing the process turn button. Since Dune 2, there have been several more games released,16 each with their own variation on the theme. Currently, the number of RTS fans playing Starcraft17 on an average night numbers at least 20,000 players.18

1.3 Purpose and Problem Statement

The main goal of this major report is to describe the design VPNG and its implementation. An environment for VPNG systems development based on Microsoft Visual C++6.0. The main contributions of the report are:

- Overview the multiplayer network game and programming;
- Design and Implementation of the one multiplayer network game;
- Testing the implementation on the local network.

1.4 Report Outline

The major report is organized as follows: Chapter 2 overviews multiplayer computer game, and introduces the models and features of multiplayer game. Chapter 3 introduces the socket programming to implement multiplayer game. Chapter 4 introduces the video poker game. The software architecture of VPNG system , design and implementation are documented in Chapter 5 and Chapter 6. The conclusions and the future work are outlined in Chapter 7.

Chapter 2 Background and Related Technology

Computer and video gaming is an area which has recently seen a vry rapid development. The gaming industry has evolved from single developers creating a toy for teenager boys, to massive companies offering entertainment to all social groups. The games offer an element of interactivity that allows the user to be the lead character in the entertainment, rather than passively watching the product of others.

Most modern games also include multiplayer options. Human opponents give the games a social aspect. The internet has given the games a new opportunity for expansion. The internet allows more players to play the same game than what would be practical to gather in one location, it also promotes contact between people with common interests.

2.1 Multiplayer Game Overview

As little as five years ago multi-player games were the exception rather than the norm in the video game world. The thought of playing online with friends or against them was something only talked or read about in computer magazines. Today, virtually every new computer video game supports some sort of multiple

player-ability. If it doesn't, watch out. Fans will be up in arms that their new game can't be played online with random people from the far off reaches of the globe.

The world of multi player games has exploded over the past couple years making sites such as Microsoft's The Zone widely popular with thousands of potential gamers ready and waiting to play the same game you want to.

It's tough to determine whether or not a new game that you buy will have the same, better, or worse play-ability between stand-alone and multi-player universes. A game such as Age of Empires 2: Age of Kings is excellent being played alone, but many times the AI is either too hard, or too easy, making the replay-ability of the game in very limited. Lets face it no one likes to get spanked every time they play a game, and in the same respect a game that is won too easy gets boring and dull. Taking this game and turning it loose on the Zone opens up a whole new world.

Before you know it it's 3:00am and you're on your 5th game trying to send your Persian War Elephants loose on the Turkish base while their janissaries and pike men are slowly picking apart your village. Playing a real person, whether it is, someone sitting next to you, or someone in Europe or Asia, makes the game more fun. Knowing that there is someone on the other end of your conquest (no matter how virtual and insignificant it may be) makes all the difference in the world.

Almost every major web portal features online gaming, from Yahoo to MSN, and the BBC to AOL. All these sites feature many games that are completely free of charge, just download a small file and you're playing checkers, chess, or some other puzzle game against some one.

For the really great multi player games like *Age of Empires, Star Wars Galactic Battlegrounds, Ashron's Call,* or *Ultima Online* you will have to purchase the game and then you're able to play. *Ashron's Call*, however charges a monthly subscription fee to play online with thousands of other fantasy gamers. Whether you want to play a quick game of checkers, a 2 hour game of *Age of Empires*, or a complete campaign that may take weeks to finish or reach your goal, online gaming is probably available for all of them. In any internet search engine just type in "online games or "multi-player games" and it's sure to come up with hundreds if not more matches. In the coming weeks make sure to check back for reviews, links, top picks, screenshots, and additional articles that feature action games that have multi-player capability.

For two games to participate in the same fame, it is crucially important that the gamers agree on the current state the game is in. Otherwise, you may end up talking to a person who is no longer present, or picking up an object the opponent has already taken. If the state changes very rapidly, like in aircraft simulators, ball games or combat scenes, this puts strong requirements on the

communication between the computers of two gamers. Today's Internet Protocols are poorly adapted to this type of communications, which are required to be both secure and extremely fast. Variations in the network bandwidth and latency will also affect how the game is experienced. In order to be well received, the game must offer the gamers a feeling of fair treatment, where everyone has the same probability of success, no matter how their connection varies.

There are a few distinct flavors of multiplayer games, all of which require different approaches. On one hand, there're games written for many players, but designed to be played on a single machine. On the other, there are games written for local play, or in general, game play across more than one machine with a local connection not reliant on a network (like a null modem cable or dial-up modem game). Finally, there are games which take place over large, wide-area networks. I shall refer to these as single-machine, local, and networked multiplayer games, respectively, from here on in.

In single-machine multiplayer games, the main game loop must call the engine to process each player involved in the game; this includes getting input from them, moving them, rendering their view, and computing any other logic associated with them in the game. In a local multiplayer game, the game loop only needs to process one player, the player on the machine which the game is running, then send information about the player out to the other machines and accept information about the other players (if needed) in return. Networked multiplayer

games are similar to local multiplayer games, but a client/server model is generally used whereby each machine transmits its data to one server, then gets information about other players back from the server.

2.2.1 Single-Machine Multiplayer Games

When designing single-machine multiplayer games, which are not very common on PCs these days, The player must take into account several questions. The most important of these would be, "How is each player going to be getting input to the game?" and "How is each player going to see their character move, with only one screen?".

Some common approaches include using two input devices, such as a joystick and a keyboard, or sharing one input device, such as two people using one keyboard (there would have to be seperate keys for every action for every player). There have been a few more worthy approaches to the display problem. Most single-machine-many-players-at-the-same-time games simply let players share the same view. As a rule, this will work well only if the view isn't first person, and only if (for 3D games) camera control is independent of player positions. In the old side-scroller days, we sometimes saw split screens, wherein each player's view was rendered on a different half of the screen. Split-screens are a big drag on rendering time for any kind of game, and they definitely can make a game cumbersome, but they do work.

About the most effective approach to the display and input problems associated with one-machine mulitplayer games, at least for PCs, is to use a turn-based system. Considering that they're still actually implemented, The turn-based systems have had the most luck over the ages. Turn-based systems work by switching between player and player and allowing only one player to play at a time.

Anyone that can write a single-player game and has an understanding of scalable design knows enough to write a single-machine multiplayer game. Single-machine multiplayer PC games are becoming progressively more extinct, but it fit to note that old single-machine techniques aren't useless. Consoles, are still very big on one-machine multiplayer.

2.2.2 Local Multiplayer Games

Local multiplayer games have been around for quite some time. Support for null-modem play is still very common in today's games. On the contrary, direct machine-to-machine modem connections are a lot less popular than they once were, due to the fact that it gets a bit limiting facing the phone charges associated with long-distance modem calls.

The architecture of a local multiplayer game is fairly simple. Each machine is responsible for updating the game for one player. Each machine then stores the changes in player state in a *packet* (a packet is data with a header to be

transmitted across a connection), which is sent to the other machines. The other machines in the game use this data to update their game-state, and send out information about their own players' movements, and the cycle continues.

Obviously, the specific implementation of this scheme is different between different games: In realtime games, asynchronous packet transmission is a must, and the game has to stay synchronized even if data packets are missed. In non-realtime games, The players can rely on sending and receiving packets at a definite time, and players also don't have to worry about overburdening their data stream.

So here's a basic recepie for a local multiplayer game. First, the players establish a connection to the other machine(s) in the game. This shouldn't be too hard, considering that a phyisical one (such as a LAN or a null-modem cable, or a direct phone line [modem]) must exist for game to be considered "local". Then, the players somehow negotiate the initial setup of the game. Finally, the game begins, and players begin the data-transfer process just described. Then, players close the "connection". It's conceptually simple.

2.2.3 Networked Multiplayer Games

Networked multiplayer games are conceptually the same as local-multiplayer games, with a few exceptions. First of all, each machine in the game still handles processing for the player using it. However, instead of sending packets out to all of the other players in the game (or just the other player), in a networked game,

machines send packets to one central server and receive packets from the server in return. The server is at the center of the game, storing all the game's information and keeping things running. If turn-based systems are used somehow, then the server is responsible for managing them. All the machines connected to the game's server are called clients. This model in general is called the client-server model of communication. Client-server is the communications model used in the real world.

Networks implement communication protocols in order to give meaning and structure to communications taking place on them. All communications on a network are carried out according to a standard set of these protocols; if you are to use a network effectively, then you must use its communications protocols. The Internet has TCP/IP (Transmission Control Protocol/Internet Protocol) to serve this purpose. TCP/IP is a family of protocols, both application-level (i.e., high-level, like FTP, HTTP, Gopher) and network-level protocols (such as IP, ARP, ICMP). In TCP/IP, packets are constructed and routed through the network to the appropriate machine, based upon the headers of these packets. The data-area of these packets is the acceptable place for you to put your game's data (you don't have to handle packets quite as was done above). The Internet gaurantees that your packets will be sent to the correct machine, that is, the machine running the server.

Client applications, or the version of the game that you would distribute to your end-users (this program is often called the "client"), send TCP/IP packets to the server, a program that dissects them and processes them; the server is a program that runs on a machine identified by an IP address. The server listens to a TCP/IP "port", where data comes in, and sends data back to its clients via TCP/IP.

For the amateur, client/server based games can present a lot of problems, when one tries to take them on from the ground up: First of all, you need a dedicated machine with a dedicated IP to run the server. Next on the list, the machine has to be able to actually run the server: Many commercial end-user operating systems like Windows 95 don't ship with TCP/IP server implementations. Fortunately, there are solutions such as Linux...

There's a lot involved in sending and processing a single packet of data in networked games. Fortunately, in most operating systems, we've got access to TCP/IP client implementations that allow us to avert the technicalities of low-level Internet communication. With Winsock, for instance, it's possible to come up with programs that do things like retrieve web pages from port 80 on any machine, in about a page of code. High-level operating system APIs for Internet functions are a blessing, not a hurdle, despite what the complaints may register. After all, is it really practical to implement around thirty years of Internet yourself?

2.2 Multiplayer Networking Game Programming Models

In our scenario, there are two or more players involved in a game over a network. This network may either be a LAN or the Internet. The players within the game should experience essentially the same reality, thus if player A shoots at player B, player B should see a shot incoming from player A's position. Thus the game-worlds should be synchronized in a sense that all players see each other's actions in exactly the same way. Hence, the coherence of our game reality is preserved. There exist two popular models of game would synchronization. In a peer-to-peer model, all the computers involved talk directly to all other computers. In a client/server model, every computer talks only to the server.

When this protocol is active all clients communicate in a strictly peer-to-peer manner. That is, there is no dedicated server (nor is there a client also acting as server). During startup a client broadcasts a connect request to find other clients that are already playing. If no other clients are found during this startup phase the client decides that it is on its own and enters the game alone. Others can then connect at any time later on. There cannot be more than four players in a single game at any one time, though.Packets that transmit the current game-state to other clients are sent to all other clients in a unicast fashion. Theoretically, it would be ideal to use multicasting for this purpose, but in this way the underlying network is not required to support any type of multicasting. Since the number of clients for peer-to-peer play is limited to four, this is perfectly feasible.

The most demanding problem of the peer-to-peer protocol is the startup phase (this part of the protocol is referred to as the startup protocol). Since there is no server there are a lot of potential problems with race conditions when two or more clients start up at approximately the same time. We have designed a quite involved startup protocol to resolve all of these problems and achieve predictable and consistent results in all cases.

Basically, in the peer-to-peer protocol clients find each other by broadcasting an initial connect request (multiple times if necessary) and waiting whether they get a reply by other clients that are currently running and therefore listening to such connect requests. If after a certain amount of time no reply has been received a client decides that it is alone in its world and enters without any further ado.

The following state diagram illustrates the peer-to-peer startup protocol using pseudo-code to describe what is done in each state:

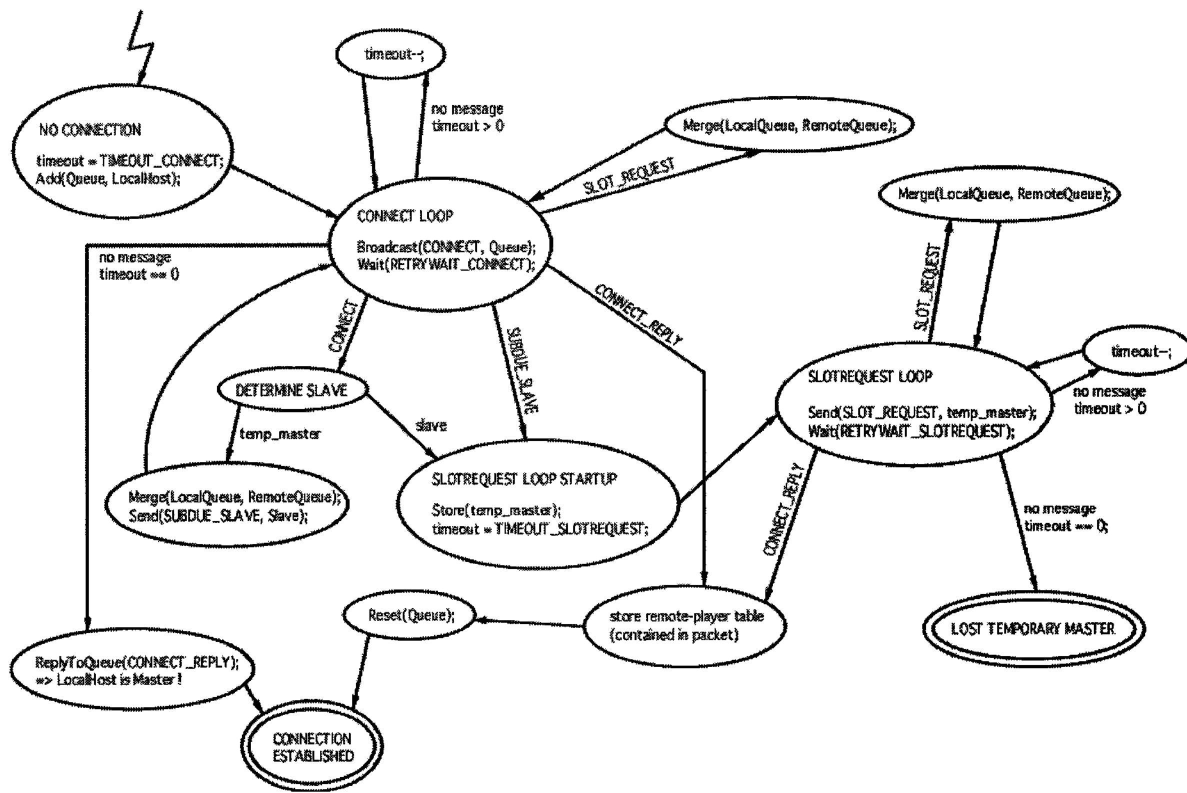


Figure 1. Peer-to-peer startup protocol state diagram

There are two fundamental approaches used here. First, conflicts of more than one client starting up at approximately the same time (which means at least on client is starting up during the startup phase of another client) are resolved by comparing the clients' node addresses. Since these should be unique, in case of conflict the client with the higher address wins. Second, clients that are starting up always queue connect requests they receive by other clients. When a client becomes the slave of another client that has just broadcast its connect request it forwards its queue to that client. This is necessary to ensure no connect requests get lost. Even though connect requests are sent multiple times if no reply has

been received, this alone is not sufficient to guarantee the entire request cannot be lost when yielding to another client without queuing up requests and forwarding them when necessary.

There are three major loops that are of importance to us here. Two of them are shown in the state diagram above and are part of the startup protocol. First, there is the connect-loop. Immediately after starting up a client enters this loop. It will be exited if this client either yields to another client with higher status (a client that is already connected, or a client also just starting up but with higher node address), or a certain amount of time has elapsed in which no clients of higher status have identified themselves. If this happens a client sends connect replies to all queued clients of lower status and enters the game-loop (the third loop, see below).

Second, there is the slot-request loop. A client enters this loop when it either implicitly decides for itself that it is a slave (a connect request of a client with higher node address has been received), or when it receives a SUBDUE_SLAVE notification, thus explicitly becoming a slave. A client being in the slot-request loop waits there until it receives a message by the master allocating a slot for the client. Normally, it is guaranteed that this will always happen for clients being in this loop; at least as long as the current (temporary) master doesn't crash in the middle of the startup phase while another client is in its slot-request loop. If this

should happen, though, the client will time out, display an error message, and the user has to issue another connect command.

Either way, after the startup phase a client will enter the game-loop. (This loop is the actual game (playing, rendering, etc.), as far as the networking-code is concerned.) There are two ways in which this is achieved. Most clients enter the game-loop as slaves, that is, someone already running has assigned them a slot. A client can also enter the game-loop as temporary master. Temporary in this case means that there actually is no master/slave relation as soon as all clients are in the game-loop. Nevertheless, during startup a temporary master is elected who assigns a slot to itself and each of the slaves in order to ensure race conditions cannot occur. After all slots have been assigned all clients are equal. There is a very good reason not to retain the already established master/slave relationship once everyone is in the game-loop. If the master exits another master has to be reelected. First, this is not quite trivial to do in a fail-safe way, and, second, there are all sorts of problems when clients try to start up while no new master has been elected yet.

Clients in the game-loop react to connect requests they receive in the following way. First, they look at their list of peers (including themselves) and establish a unique ranking order using the node addresses. They then wait their rank (starting with 0) times a certain wait-interval. If they are not able to snoop a connect reply of another client during this period, they will send a reply

themselves. This approach ensures that somebody will reply to the connect request, even if clients that are still in the list have already exited without sending a notification (i.e., crashed, or their host has been turned off without exiting). It also ensures that there are no conflicts, since the replicated client list can only be inconsistent with respect to crashed clients, but there cannot be clients that are connected but not part of the list. Therefore, no two clients will try to reply at the same time.

In essence, this approach amounts to some kind of dynamic master-reelection protocol each time a new client tries to connect.

2.2.1 The Client / Server Model

In the Client-Server model, there is a dedicated computer that takes on the role of a server. All clients connect to the server. All of the communication takes place between clients and a server, as clients never talk to each other. The server takes on the duties of the 'host', informing the clients of the joining and leaving players. The server also authenticates players' moves before sending them to the other clients. Thus the scheme functions as follows. ' A player wants to move forward. He sends the 'forward' command to the server. If the server allows the move, it broadcasts that move to all other players. If the command is invalid, the server messages the client and disallows the move.'

The advantages of the client-server approach are the improved security and excellent scalability. The improved security is achieved because the server validates all of the clients' moves. The scalability is good because each client only sees the up and down traffic from the server, which is essentially not dependent on the total number of clients. Further scalability may be achieved through load balancing, a principle that requires that you have a number of servers each handling a portion of the total number of connected clients.

The disadvantages of the client-server model are the need for dedicated hardware and reduced fault tolerance. The requirement for dedicated hardware is the primary disadvantage of the client-server approach. The server requires a very high bandwidth line, since it facilitates communication with all of the clients. Also the fault tolerance is reduced, since if the main server becomes unavailable, the game will stall.

In the online game context, a server refers to an entity that calculates and simulates the game states based on the players' actions, and a client refers to an entity that renders and presents the game states to the user.

When a server runs the game simulation or when a client presents the game state, they synchronize the events or states based on the time that they have been generated.

To synchronize the game play and interaction amongst the users, the online game system must take into account these latencies between the servers and the clients.

- "**Client-Server Architecture**" describes a network architecture in which each computer handles part of the processing work, but is designated as either a "client" or a "server" with respect to each process. "*Servers*" are shared, central computers which are dedicated to managing specific tasks for a number of clients. For example, "file servers" are dedicated to storing files, "print servers" are dedicated to printing, and "network servers" are dedicated to routing network traffic. "*Clients*" are workstations on which users run programs and applications. Clients rely on servers for resources, such as files, devices, and even processing power, but process independently of the servers. Client-server architecture can be used in any sized network; the distinguishing characteristic of client-server architecture is that all computers on the network participate in processing, but that certain computers are dedicated to specific services or tasks and do no other work.

- **Client-Server Programming Models**

Client-server programs split functions and processes between client and server computers. In most cases, client-server programs create three functional divisions of "work":

- **Data Storage** -- storage of data used by the program, typically in a large database.
- **Business Logic** -- processes which do the "work", such as requesting data, sorting data, returning data, processing data into reports.
- **Data Presentation** -- display and user interface.

The following diagram illustrates the design issue:

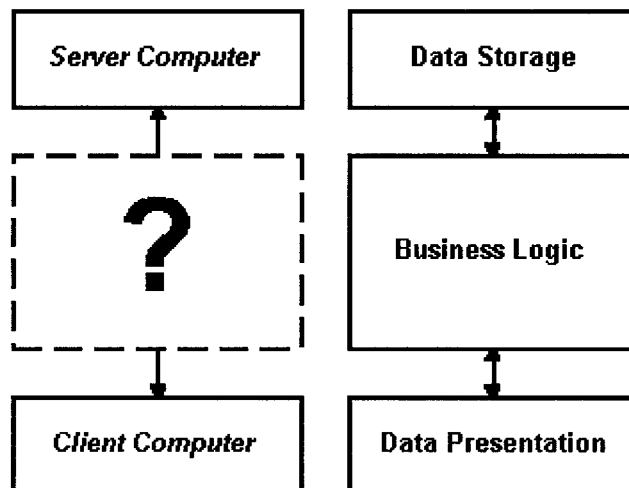


Figure 2 Client-Server Programming Models

Almost all client-server models locate *Data Storage* on the server computer and *Data Presentation* on the client. The difference between client-server models depends on how the Business Logic is split up between client and server.

2.2.2 Peer-to Peer Model

In peer-to-peer model, all clients talk to all other clients. Thus, individual state updates are transmitted to all the machines. Also, a single machine is designated 'host'. The host is almost like a server, in that it keeps track of the players and informs the other clients when a client joins or leaves the game. Here, however, the similarity ends, as the host does not serve as a central connection point for all the clients. The host is a required entity, so peer-to-peer games almost always support host migration, which is a protocol that is commonly handled. 'Host migration' automatically transfers the host responsibilities to another computer, in the case that the active host disconnects or becomes unavailable. An illustration of a typical peer-to-peer setup is following.

The advantage of the peer-to-peer method is that it is fault tolerant, easy to implement and does not require any special hardware. Since there is no central server to disrupt, if anything happens to one or more of the players, the rest will not be affected. Thus, this system is very fault tolerant. Implementation is also very simple as all clients are the same and there is no need for a specialized stand-alone server. Also, since the server is not present, there is no need for any special hardware.

The disadvantages of the peer-to-peer method are the high security risk and scalability problems. The issue of the security risk comes in because the clients are responsible for sending information about themselves to all other clients.

Thus, there is no external validation mechanism and if a client were hacked, it could send erroneous information to the rest of the clients, which they would take it at face value. Hence, in a game, the cheater could talk through walls, get infinite health, etc. Yet, the biggest disadvantages of peer-to-peer networking lies in scalability problems.

- "**Peer-to-Peer Architecture**" describes a type of network in which a number of workstations are connected together, but in which no computer is dedicated as a "server". Instead, each workstation can be (and usually is) both "client" and "server", depending on the task involved (for example, Workstation A may be connected to a printer and act as the "print server" for the other workstations, while Workstation B may be connected to a modem and act as the "communications server" for the others). Peer-to-peer networks are usually small (under 25 workstations) and do not offer the solid performance in larger installations or under heavy network traffic loads.

2.2.3 Host-Terminal Architecture

Host-Terminal Architecture" describes a type of network in which a central computer (the "*host*") is connected to a number of workstations ("*terminals*"), and in which the host handles all processing. Terminals are used to input data into the host and to review reports, but the terminals are "dumb" in the sense that the workstations do not participate in processing work. Host-terminal architecture is

suited to both large and small networks; the distinguishing characteristic is that a single, central computer handles the processing work.

2.2.4 The Comparison of these Models

The two approaches are both valid but on different scales. If the game world will consist of only a few players, then a peer-to-peer model is an acceptable method for synchronizing the game world. For example, Microsoft's Age of Empires uses the peer-to-peer networking scheme, but limits the number of active players to four. How ever, if there is a possibility of a moderate to large amount of players being present in the game-world at the same time then the client –server model should be used. It should also be noted that the peer-to-peer model is much less secure, so if the security is a factor then the model should be chosen acoordingly. An excellent example of the client-server approach is the ID software\s Quake III, which is a fast-paced 3D shooter that allows up to forty clients to participate in a single game.

2.3 Multiplayer Networking Game Programming Feature

The multiplayer game networking offers a wide variety of technological challenges. For some of the following points satisfactory solutions already exist, others are currently subject to research, while others again are likely candidates for future activities. For all of these problems, it must be kept in mind that the

solution of one problem should not complicate others, by introducing significant lag, extra processing or similar overhead.

2.3.1 Synchronization

Agreement on the time variable is an imperative in order to obtain a share state. For cases where the Network Time Protocol (NTP) is not applicable, due to firewalls or use of other networks than Internet, the games have to be considered to be developed a synchronization routine giving a precision on the order of the system clock.

2.3.2 Multiplayer Scalability

How many players can a game support? For a peer-to-peer game, communication demands scale as the number of players squared. With a server it scales linearly. In both cases, there is an upper limit on how many players can be supported. How can this limit be extended, or ideally, by distributed computing, be avoided entirely?

2.3.3 Robustness

With many players connected to a game using different connection types and hardware, it is unavoidable that some client connection will have a substandard performance, due to high latency, low bandwidth orhigh packet loss rates. In such case, it is important that as few of the other players as possible are affected

by this will have big troubles id only one client connection is poor. We hope to reduce the problem, so that only the players directly interacting with the substandard connection are affected.

2.3.4 Area of Interest Management

The Area of Interest defines a selection mechanism for what information a player needs to have , thus reducing the total bandwidth requirement in a relatively predictable way. For massive multiplayer games this may be a prerequisite for a functioning business model, as gaming companies operate with a limit on what bandwidth they can spent on each player and still earn money.

2.3.5 Dead Rechoning

Dead Reckoning is concerned with how to predict behaviour for situations where the actual information updates have not yes arrived due to network latency or bandwidth limitations. Dead Reckoning is essential to offering clients a feeling of immediate response to their actions.

2.3.6 Object Scalability

The world covers an enormous range of length scales. From huge planetary systems to the planet surface, to landscape features, cities, houses, down to details on the furniture and even further. If all the system should be as detailed as the finest resolution, it would include enormous amounts of data. Dynamic

terrain generation through a fractal routine with a known seed is one partial answer to a part of this, there may be others.

2.3.7 Bandwidth Management

Bandwidth is a limiting factor in many respects. Due to the heterogeneous nature of the Internet, clients cannot be expected to meet any quaranteed bandwidth requirements. If the game can dynmically adapt to the available bandwidth, the players can get a better gaming experience without excluding those with poor lines.

2.3.8 Faireness

Fairness is of course a very subjective concept, it is often possible to define serveral, mutually exclusive interpretations of what is " fair " in a given situation. Still, for computer game, giving the users a perception of faire treatment is essential to keeping customers, even when no gambling is involved. For auctions, or other types of interactions with economic consequences, fairness is an absolute requirement. In some cases this may be as little as giving the customer relevent information on how poor his connection is, in other cases it may be necessary to adapt the game to the slowest player.

2.3.9 Persistence

No one ever restated the World Wide Web. It would be impossible, but due to the way Web made, it is not necessary. When creating a truly massive multiplayer world, where people can log on the off as they please, the same flexibility is needed. Code must be downloaded while the game is running, to be able to introduce new functionality without asking all players to log off and restart the system.

2.3.10 Rapid Database Access

For huge worlds, database can be as bad a bottlenect as the netwoek bandwidth. Rapid, secure and flexible database access is therefore a primary concern in the development of larger game-world.

2.3.11 Reliable, but fast protocols

On the Internet, the most commonly used transport protocols are UDP, for repid, unreliable communications, and TCP, for reliable communications with virtually no upper limit on latency. This means you have to build your own protocol if you need all the messages, but cannot afford to wait for them. The Direct Play Protocol from Microsoft has attempted to solve some of these problems.

Chapter 3. Windows Sockets Network Programming

3.1 Windows Sockets Concepts

3.2.1 What is Socket

The basic building block for communication is the socket. A socket is an endpoint of communication to which a name may be bound. Each socket in use has a type and an associated process. Sockets exist within communication domains. A communication domain is an abstraction introduced to bundle common properties of threads communicating through sockets. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The Windows Sockets facilities support a single communication domain: the Internet domain, which is used by processes which communicate using the Internet Protocol Suite. (Future versions of this specification may include additional domains.)

Sockets are typed according to the communication properties visible to a user. Applications are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Two types of sockets currently are available to a user. A stream socket provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries.

A datagram socket supports bi-directional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as Ethernet.

What is a socket application?

A socket interface was first provided with Berkeley UNIX (BSD) in the eighties. It was designed as a network interprocess communication (IPC) mechanism for the built-in TCP/IP. A socket defines a bi-directional end point for combination between processes.

A socket has three primary components:

- The interface to which it is bound
- The port number to which it will send or receive data
- The type of socket--either stream or datagram

In TCP/IP, the interface is the IP address of the host. The port number is the software process address. In IPX/SPX, the interface is the combination of the IPX network ID and the MAC address of the network interface. The port number is the software process address (IPX socket number).

A server application listens on a well-known port over all installed network interfaces. A client generally initiates communication from a specific interface from any available port.

3.2.2 What is Windows Socket

A socket is a communication endpoint — an object through which a Windows Sockets application sends or receives packets of data across a network. A socket has a type and is associated with a running process, and it may have a name. Currently, sockets generally exchange data only with other sockets in the same "communication domain," which uses the Internet Protocol Suite.

Both kinds of sockets are bidirectional; they are data flows that can be communicated in both directions simultaneously (full-duplex).

Two socket types are available:

- Stream sockets

Stream sockets provide for a data flow without record boundaries: a stream of bytes. Streams are guaranteed to be delivered and to be correctly sequenced and unduplicated.

- Datagram sockets

Datagram sockets support a record-oriented data flow that is not guaranteed to be delivered and may not be sequenced as sent or unduplicated.

Windows Sockets 2 (Winsock) enables programmers to create advanced Internet, intranet, and other network-capable applications to transmit application data across the wire, independent of the network protocol being used. With Winsock, programmers are provided access to advanced Microsoft® Windows® networking capabilities such as multicast and Quality of Service (QOS).

Winsock follows the Windows Open System Architecture (WOSA) model; it defines a standard service provider interface (SPI) between the application programming interface (API), with its exported functions and the protocol stacks. It uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Windows in Windows Sockets 1.1, with which Windows Sockets 2 applications are backward compatible. Winsock programming previously centered around TCP/IP. Some programming practices that worked with TCP/IP do not work with every protocol. As a result, the Windows Sockets 2 API adds functions where necessary to handle several protocols.

What is Windows Sockets

The Windows Sockets specification defines a network programming interface for Microsoft Windows which is based on the "socket" paradigm popularized in the Berkeley Software Distribution (BSD) from the University of California at Berkeley. It encompasses both familiar Berkeley socket style routines and a set of Windows-specific extensions designed to allow the programmer to take advantage of the message-driven nature of Windows.

The Windows Sockets Specification is intended to provide a single API to which application developers can program and multiple network software vendors can conform. Furthermore, in the context of a particular version of Microsoft Windows, it defines a binary interface (ABI) such that an application written to the Windows Sockets API can work with a conformant protocol implementation from any network software vendor. This specification thus defines the library calls and associated semantics to which an application developer can program and which a network software vendor can implement.

Network software which conforms to this Windows Sockets specification will be considered "Windows Sockets Compliant". Suppliers of interfaces which are "Windows Sockets Compliant" shall be referred to as "Windows Sockets Suppliers". To be Windows Sockets Compliant, a vendor must implement 100% of this Windows Sockets specification.

Applications which are capable of operating with any "Windows Sockets Compliant" protocol implementation will be considered as having a "Windows Sockets Interface" and will be referred to as "Windows Sockets Applications".

This version of the Windows Sockets specification defines and documents the use of the API in conjunction with the Internet Protocol Suite (IPS, generally referred to as TCP/IP). Specifically, all Windows Sockets implementations support both stream (TCP) and datagram (UDP) sockets.

While the use of this API with alternative protocol stacks is not precluded (and is expected to be the subject of future revisions of the specification), such usage is beyond the scope of this version of the specification.

3.2.3 A Socket Data Type

Each MFC socket object encapsulates a handle to a Windows Sockets object. The data type of this handle is SOCKET. A SOCKET handle is analogous to the HWND for a window. MFC socket classes provide operations on the encapsulated handle.

The SOCKET data type is described in detail in the Win32 SDK. See the topic Socket Data Type and Error Values under Windows Sockets.

3.2.4 Uses of Socket

Sockets are highly useful in at least three communications contexts:

- Client/Server models
- Peer-to-peer scenarios, such as chat applications
- Making remote procedure calls (RPC) by having the receiving application interpret a message as a function call

3.2 Network Program Mechanics

3.2.1 Open a Socket

socket()

3.2.2 Name the Socket

- Name the Socket

- sockaddr Structure
- sockaddr_in Structure
- Port Numbers
- Local IP Address
- What's in a Socket Name?
  bind()
- Client Socket Name is Optional

3.2.3 Associate with Another Socket

- How a Server Prepares for an Association
  listen()
- How a Client Initiates an Association
  connect()
- How a Server Completes an Association
  accept()

3.2.4 Send and Receive Between Sockets

- Sending Data on a "connected" Socket
  send()
- Sending Data on an "unconnected" Socket
  sendto()
- Receiving Data
  recv()
  recvfrom()

3.2.5 Close the Socket

- closesocket()
- shutdown()

3.3 Sockets Programming Models

The two MFC Windows Sockets programming models are supported by the following classes:

MFC supplies two classes to support programming network applications with the Windows Sockets API. Class CAsyncSocket encapsulates the Windows Sockets API one-for-one, giving advanced network programmers the most power and flexibility. Class CSocket provides a simplified interface for serializing data to and from a CArchive object.

3.3.1 Windows Sockets Classes

- **CAsyncSocket**

  This class encapsulates the Windows Sockets API. CAsyncSocket is for programmers who know network programming and want the flexibility of programming directly to the sockets API but also want the convenience of callback functions for notification of network events. Other than packaging sockets in object-oriented form for use in C++, the only additional abstraction this class supplies is converting certain socket-related Windows messages into callbacks.

- **CSocket**

  This class, derived from **CAsyncSocket**, supplies a higher-level abstraction for working with sockets via an MFC CArchive object. Using a socket with an archive greatly resembles using MFC's file serialization protocol. This makes it easier to use than the **CAsyncSocket** model. CSocket inherits many member functions from **CAsyncSocket** that encapsulate Windows Sockets APIs; you will have to use some of these

functions and understand sockets programming generally. But **CSocket** manages many aspects of the communication that you would have to do yourself using either the raw API or class **CAsyncSocket**. Most important, **CSocket** provides blocking (with background processing of Windows messages), which is essential to the synchronous operation of **CArchive**.

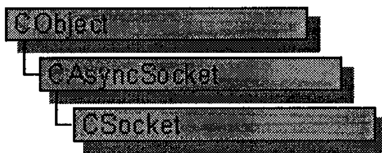


Figure 3 Windows Socket Classes

Class **CSocket** derives from **CAsyncSocket** and inherits its encapsulation of the Windows Sockets API. A **CSocket** object represents a higher level of abstraction of the Windows Sockets API than that of a **CAsyncSocket** object. **CSocket** works with classes **CSocketFile** and **CArchive** to manage the sending and receiving of data.

A **CSocket** object also provides blocking, which is essential to the synchronous operation of **CArchive**. Blocking functions, such as **Receive, Send, ReceiveFrom, SendTo**, and **Accept** (all inherited from **CAsyncSocket**), do not return a **WSAEWOULDBLOCK** error in **CSocket**. Instead, these functions wait until the operation completes. Additionally, the original call will terminate with the error **WSAEINTR** if **CancelBlockingCall** is called while one of these functions is blocking.

To use a **CSocket** object, call the constructor, then call **Create** to create the underlying **SOCKET** handle (type **SOCKET**). The default parameters of **Create** create a stream socket, but if you are not using the socket with a **CArchive** object, you can specify a parameter to create a datagram socket instead, or bind to a specific port to create a server socket. Connect to a client socket using **Connect** on the client side and **Accept** on the server side. Then create a **CSocketFile** object and associate it to the **CSocket** object in the **CSocketFile** constructor. Next, create a **CArchive** object for sending and one for receiving data (as needed), then associate them with the **CSocketFile** object in the **CArchive** constructor. When communications are complete, destroy the **CArchive**, **CSocketFile**, and **CSocket** objects. The **SOCKET** data type is described in the article Windows Sockets: Background in *Visual C++ Programmer's Guide*.

3.3.2 CSocket Programming Model

Using a **CSocket** object involves creating and associating together several MFC class objects. In the general procedure below, each step is taken by both the server socket and the client socket, except for step 3, in which each socket type requires a different action.

At run time, the server application usually starts first to be ready and "listening" when the client application seeks a connection. If the server is not ready when the client tries to connect, you typically require the user application to try connecting again later.

To set up communication between a server socket and a client socket

Construct a CSocket object.

Use the object to create the underlying **SOCKET** handle.

For a **CSocket** client object, you should normally use the default parameters to Create, unless you need a datagram socket. For a **CSocket** server object, you must specify a port in the **Create** call.

If the socket is a client, call CAsyncSocket::Connect to connect the socket object to a server socket.

-or-

If the socket is a server, call CAsyncSocket::Listen to begin listening for connect attempts from a client. Upon receiving a connection request, accept it by calling CAsyncSocket::Accept.

Create a CSocketFile object, associating the **CSocket** object with it.

Create a CArchive object for either loading (receiving) or storing (sending) data. The archive is associated with the **CSocketFile** object.

Keep in mind that **CArchive** does not work with datagram sockets.

Use the **CArchive** object to pass data between the client and server sockets.

Keep in mind that a given **CArchive** object moves data in one direction only: either for loading (receiving) or storing (sending). In some cases, you will use two **CArchive** objects: one for sending data, the other for receiving acknowledgments.

After accepting a connection and setting up the archive, you can perform such tasks as validating passwords.

Destroy the archive, socket file, and socket objects.

3.4 Using Sockets with Archives

Class **CSocket** supplies socket support at a higher level of abstraction than does class **CAsyncSocket. CSocket** uses a version of the MFC serialization protocol to pass data to and from a socket object via an MFC CArchive object. **CSocket** provides blocking (while managing background processing of Windows messages) and gives you access to **CArchive**, which manages many aspects of the communication that you would have to do yourself using either the raw API or class **CAsyncSocket**.

If you are writing an MFC client program to communicate with established (non-MFC) servers, don't send C++ objects via the archive. Unless the server is an MFC application that understands the kinds of objects you want to send, it won't be able to receive and deserialize your objects.

3.5 How Sockets with Archives Work

This section explains how a **CSocket** object, a CSocketFile object, and a CArchive object are combined to simplify sending and receiving data via a Windows socket.

The article Windows Sockets: Example of Sockets Using Archives presents the PacketSerialize function. The archive object in the PacketSerialize example works much like an archive object passed to an MFC Serialize function. The essential difference is that for sockets, the archive is attached not to a standard CFile object (typically associated with a disk file) but to a **CSocketFile** object. Rather than connecting to a disk file, the **CSocketFile** object connects to a **CSocket** object.

A **CArchive** object manages a buffer. When the buffer of a storing (sending) archive is full, an associated **CFile** object writes out the buffer's contents. Flushing the buffer of an archive attached to a socket is equivalent to sending a message. When the buffer of a loading (receiving) archive is full, the **CFile** object stops reading until the buffer is available again.

Class **CSocketFile** derives from **CFile**, but it doesn't support CFile member functions such as the positioning functions (**Seek**, **GetLength**, **SetLength**, and so on), the locking functions (**LockRange**, **UnlockRange**), or the **GetPosition** function. All the CSocketFile object must do is write or read sequences of bytes to or from the associated **CSocket** object. Because a file is not involved, operations such as **Seek** and **GetPosition** make no sense. **CSocketFile** is derived from **CFile**, so it would normally inherit all of these member functions. To prevent this, the unsupported **CFile** member functions are overridden in **CSocketFile** to throw a CNotSupportedException.

The **CSocketFile** object calls member functions of its **CSocket** object to send or receive data.

The following figure shows the relationships among these objects on both sides of the communication.

Up to the point of constructing a **CSocketFile** object, the following sequence is accurate (with a few parameter differences) for both **CAsyncSocket** and **CSocket**. From that point on, the sequence is strictly for **CSocket**. The following table illustrates the sequence of operations for setting up communication between a client and a server.

Server	Client
`// construct a socket` `CSocket sockSrvr;`	`// construct a socket` `CSocket sockClient;`
`// create the SOCKET` `sockSrvr.Create(nPort);` [1,2]	`// create the SOCKET` `sockClient.Create( );` [2]
`// start listening` `sockSrvr.Listen( );`	
	`// seek a connection` `sockClient.Connect(strAddr, nPort);` [3,4]

`// construct a new, empty socket`
`CSocket sockRecv;`
`// accept connection`
`sockSrvr.Accept( sockRecv );` [5]	

`// construct file object` `CSocketFile file(&sockRecv);`	`// construct file object` `CSocketFile file(&sockClient);`
`// construct an archive`	`// construct an archive`

CArchive arIn(&file,
    CArchive::load);
-or-
CArchive arOut(&file,
    CArchive::store);
- or Both -
	CArchive arIn(&file,
	    CArchive::load);
	-or-
	CArchive arOut(&file,
	    CArchive::store);
	- or Both -

// use the archive to pass data: arIn >> dwValue; -or- arOut << dwValue;[6]	// use the archive to pass data: arIn >> dwValue; -or- arOut << dwValue;[6]

Table 1 Setting Up Communication Between a Server and a Client

1. Where *nPort* is a port number.

2. The server must always specify a port so clients can connect. The **Create** call sometimes also specifies an address. On the client side, use the default parameters, which ask MFC to use any available port.

3. Where *nPort* is a port number and *strAddr* is a machine address or an Internet Protocol (IP) address.

4. Machine addresses can take several forms: "ftp.microsoft.com", "microsoft.com". IP addresses use the "dotted number" form "127.54.67.32". The **Connect** function checks to see if the address is a dotted number (although it does not check to ensure the number is a valid machine on the network). If not, **Connect** assumes a machine name of one of the other forms.

5. When you call **Accept** on the server side, you pass a reference to a new socket object. You must construct this object first, but do not call **Create** for it. Keep in mind that if this socket object goes out of scope, the connection closes.

MFC connects the new object to a **SOCKET** handle. You can construct the socket on the stack, as shown, or on the heap.

6. The archive and the socket file are closed when they go out of scope. The socket object's destructor also calls the Close member function for the socket object when the object goes out of scope or is deleted.

Additional Notes About the Sequence.

The sequence of calls shown in the preceding table is for a stream socket. Datagram sockets, which are connectionless, do not require the CAsyncSocket::Connect, Listen, and Accept calls (although you can optionally use **Connect**). Instead, if you are using class **CAsyncSocket**, datagram sockets use the **CAsyncSocket::SendTo** and **ReceiveFrom** member functions. (If you use **Connect** with a datagram socket, you use **Send** and **Receive**.) Because **CArchive** does not work with datagrams, do not use **CSocket** with an archive if the socket is a datagram.

CSocketFile does not support all of **CFile**'s functionality; **CFile** members such as **Seek**, which make no sense for a socket communication, are unavailable. Because of this, some default MFC **Serialize** functions are not compatible with **CSocketFile**. This is particularly true of the **CEditView** class. You should not try to serialize **CEditView** data through a **CArchive** object attached to a **CSocketFile** object using **CEditView::SerializeRaw**; use **CEditView::Serialize**

instead (not documented). The SerializeRaw function expects the file object to have functions, such as **Seek**, that **CSocketFile** does not support.

Chapter 4. Video Poker Game

4.1 Rules of the Game

Video Poker is a mixture of Stud Poker and a slot mahcine with a few wild cards thrown in. It's fasr, almost like playing Stud, and player has got a huge range of options. You can choose Jacks or Better, Deuces wild, all American, Joker Poker or a number of other variations. Each game has its own personality and rewards a particular kind of play.

Also, Video Poker is played between 2 –10 people. The player can play anything from 1 through 5 cards, with the payoffs improving at the high end of the scale. The players are dealt five cards in the first hand. When the players receive his first hand, choose the cards to hold or dicard. The cards he discard are replaced with new ones. So, if player knows his Poker hands and take the time to learn his game, video Poker can be fast, fun, and rewarding.

Objective: As with all forms of Poker, the player would like to get the best hand possible. The payoffs are dependent on the values of the hand players are holding.

Betting: Betting is pretty straighforward in Video Poker. The player can typically choose $0.25, $0.50, $1, or $5 games. And the bets are 1x, 2x, 3x, 4x, or 5x (Max Bet) .So if player is playing a $1 game, he can place bets of $1, $2, $2, $4, or $5.

Payoff: The first thing to note is that the game face shows the players for each betting level. Invariably playing Max Bet pays off better, overall, than any of the lower multiples. Smart players pick the betting level they are comfortable with and choose their game accordingly.

4.2 Basic Video Poker Hands

A poker hand consists of 5 cards. The Ace is considered the highest card, followed by Kings, Queens, Jacks, etc. The various poker hands are describered below from highest to lowest.

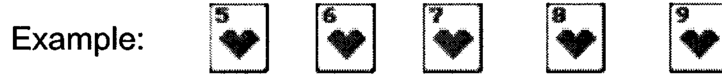
Royal Flush

This is a combination of the following cards of the same suit: Ace, King, Queen, Jack and Ten.

Example:

Straight Flush

This is a combination of five cards of the same suit in sequence.



Four of a Kind

This is a combination of four cards of the same denomination.

Full House

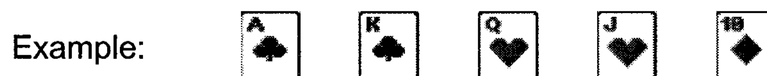
This is a combination of three cards of the same denomination, and two cards of another denomination.

Flush

This is a combination of five cards of the same suit.

Straight

This is a combination of five cards of any denomination in sequence.



Three of a Kind

This is a combination of three cards of the same denomination, and any two other cards.

Example:

Two Pairs

This is a combination of two pairs and any other card.

Example:

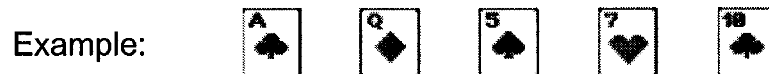
One Pair:

If two of the five cards are of the same values, it is a one pair.

Example:

No pair:

The lowest hand, containing 5 separate cards that do not match up to create any of the hands.

Example: 

Chapter 5. VPNG System Design

In this chapter, the system architecture and detail design of VPNG are presented.

5.1 Description of the System

5.1.1 System Functionalities

The inputs to this system are:

1. The amount of money players want to bet,
2. Betting or passing for the turn by players,

The system output are:

1. The balance of the players.
2. Cards's status of all players except the first turn.
3. In the last turn, all cards show up to all players.

VPNG system provides the following functionalities:

- Display historical information for each game player
- Display game records for each game player
- Display multiple game groups
- Calculate the result for each turn and display the results.

5.1.2 System Characteries

Because of the total 52 cards and 5 cards for each player, the number of the game players is limited to maximum 10. One player can play this game by his/herself without betting and win. At least two game players can play with betting and win.

Characters	**Minimum**	**Maximum**	**Note**
No. Players	1	10	
No. Game Groups	1	Unlimit	Actually, this is depending on the system limitation
Amount of money for betting	$0.00	$1000.00	

Table 2 VPNG system characters

The number of game group means how many group of players can be running on this server. Therefore, the number of players on this server is equal to sum of (number of players for each group).

For example, there are 5 groups and each group have different players as following.

Group	No. Players
0	5
1	3
2	4
3	6
4	10

Total	28

Table 3 An example for total number of players on one server

5.2 System Architecture

VPNG system is based on a client /server model. VPNG system is divided two subsystems: Client Subsystem and Server Subsystem.

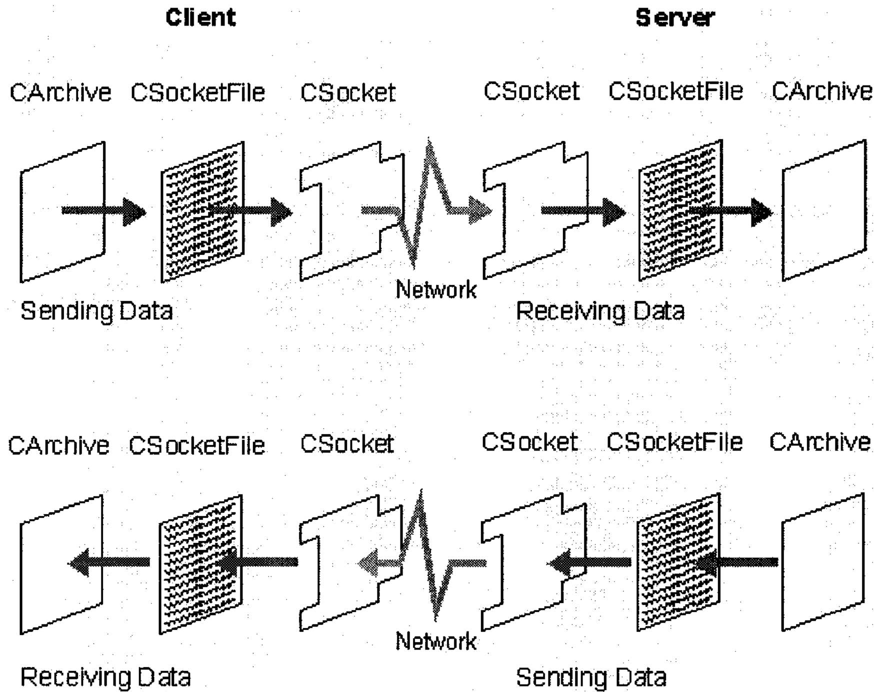


Figure 4 The Architecture Diagram of the VPNG

5.4 Structure Model

5.4.1 Foundation Classes in MFC

The CDocument class provides the basic functionality for user-defined document classes. A document represents the unit of data that the user typically opens with the File Open command and saves with the File Save command. Users interact with a document through the CView object(s) associated with it. A view renders an image of the document in a frame window and interprets user input as operations on the document. A document can have multiple views associated with it. When the user opens a window on a document, the framework creates a view and attaches it to the document. The document template specifies what type of view and frame window are used to display each type of document.

CEditView object is a type of view class that provides the functionality of a Windows edit control and can be used to implement simple text-editor functionality. The CEditView class provides the following additional functions: print , find and replace.

A CString object consists of a variable-length sequence of characters. CString provides functions and operators using a syntax similar to that of Basic. Concatenation and comparison operators, together with simplified memory management.

The CStringList class supports lists of CString objects. All comparisons are done by value, meaning that the characters in the string are compared instead of the addresses of the strings.

5.4.2 Class Diagram of Server Subsystem

The class CServerDoc is derived from Cdocument. The class diagram shown in Figure 11 illustrates the classes and their relationships in the context of VPNG.

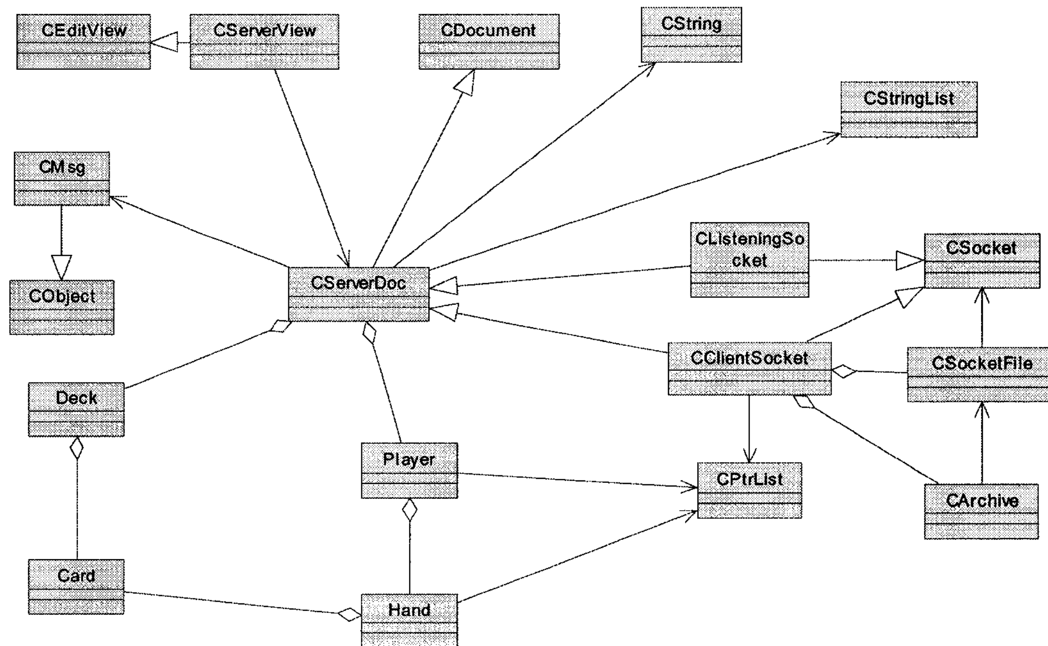


Figure 5 Class Diagram of Server Subsystem

5.4.3 Class Diagram of Client Subsystem

The aim of the class hierarchy implemented is to create at least one unique class for every client. This provides a high level of customization.

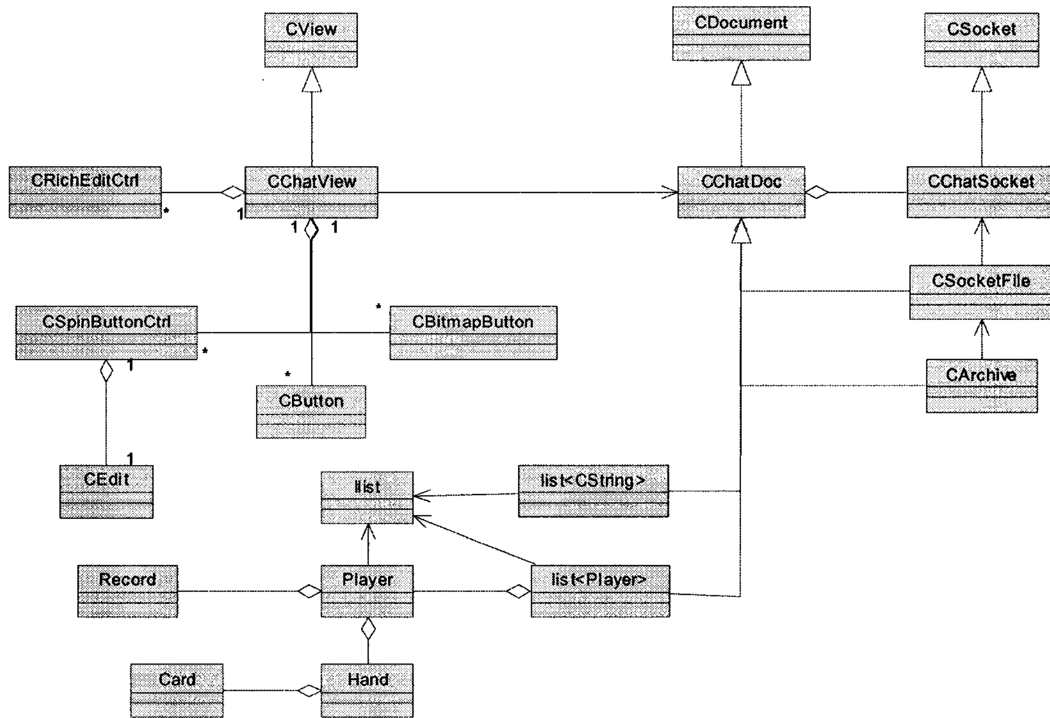


Figure 6 The Class Diagram of the Client Subsystem

5.4.4 Classes Description in VPNG System

Class definition:

```
enum CardSuit{club, diamond, heart, spade};

enum CardFace{two, three, four, five, six, seven, eight,
              nine, ten, jack, queen, king, ace};
```

class Card

{

```
public:
        Card(){}
        Card(const CardFace &f, const CardSuit &s);
        ~Card();
        CardFace GetFace() const;
        CardSuit GetSuit() const;
        void SetFace(const CardFace &f);
        void SetSuit(const CardSuit &s);
        bool isNextFaceOf(const Card &c);
        //bool isNextS
        CString getStringOutput();
private:
        CardFace Face;
        CardSuit Suit;
};
bool operator==(const Card &c1, const Card &c2);
bool operator<(const Card &c1, const Card &c2);
```

class deck

```
#include "card.h"
```

class Deck

```
{
        public:
                Deck();
                void Shuffle();
                void Sort();
                void Arrange();
                Card GetCard(int i)const;
                void SetCard(int i, const Card &c);
```

```
        void SwapCards(int i, int j);
    private:
        Card Cards[52];
};
```

class hand

```
#include "card.h"
enum Rank { nothing, Onepair, TwoPair, ThreeOfAKind,
            Straight, Flush, FullHouse, FourOfAkind, StraightFlush};

class Hand{
    public:
        Hand(){}
        Card GetCard(int i) const{return Cards[i-1];}
        void SetCard(int i, const Card &c){Cards[i-1]=c;}
        void Sort();
        void SwapCards(int i, int j);
        Rank Quality() const;
        CardFace getOnePairFace() const;
        CardFace gettwoPairFace() const;
        CardFace getThreeofAKindFace() const;
        CardFace getFullHouseFace() const;
        CardFace getFourOfAkindFace() const;
        bool HasOnePair() const;
        bool HasTwoPair() const;
        bool HasThreeofAKind() const;
        bool HasStraight() const;
        bool HasFlush() const;
        bool HasFullHouse() const;
```

```
        bool HasFourOfAKind() const;
        bool HasStraightFlush() const;
    private:
        Card Cards[5];
};
//auxiliary operators
bool operator==(const Hand &a, const Hand &b);
bool operator<(const Hand &a, const Hand &b);
ostream& operator<<(ostream &sout, const Hand &b);
CString outputRankStrName(Rank r);
```

class playingRecord

```
{
public:
    CString winner;
    CString suit;
    int win;
    int money;
};
```

class Record

```
{
public:
    int turn;
    CString borP;
};
```

class Player

```
{
```

```
public:
        Player();
        virtual ~Player();


        list<Record> resultList;
        int betUnit;
        int money;
        CString Name;
        CString cards;
        bool hasPut;
        Hand hand;
};
class CChatDoc : public CDocument
{
protected: // create from serialization only
        CChatDoc();
        DECLARE_DYNCREATE(CChatDoc)


// Attributes
public:
        BOOL m_bAutoChat;
        CString m_strHandle;
        CChatSocket* m_pSocket;
        CSocketFile* m_pFile;
        CArchive* m_pArchiveIn;
        CArchive* m_pArchiveOut;
        bool firstTime;
        bool secondturn;
```

```
        int betUnit;
        int total;
        int balance;
        int turn;
        list<CString> dispInfoList;
        list<playingRecord> playingRecordList;
        list<Player> myPlayerList;
        CString result;

// Operations
public:
        BOOL ConnectSocket(LPCTSTR lpszHandle, LPCTSTR lpszAddress, UINT nPort);
        void ProcessPendingRead();
        void SendMsg(CString& strText);
        void ReceiveMsg();
        void DisplayMsg(LPCTSTR lpszText);
        BOOL isDeplicated(list<Player> plist, CString str);

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CChatDoc)
        public:
        virtual BOOL OnNewDocument();
        virtual void DeleteContents();
        //}}AFX_VIRTUAL

// Implementation
public:
        CString getPlayingRecord();
```

```
        void restMessage();
        BOOL repeatRecord(list<Record> reList,int turn);
        void changeDeck(CString &str);
        int getTurn(CString &str);
        int toInteger(CString &string);
        void getPlayerInfo(CString & str);
        CString toString(int num);
        bool isStringDeplicated(list<CString> plist, CString str);
        virtual ~CChatDoc();
        virtual void Serialize(CArchive& ar);   // overridden for document i/o
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
        //{{AFX_MSG(CChatDoc)
        //afx_msg void OnGameAnothergame();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

class CChatView : public CView

```
{
protected: // create from serialization only
        CChatView();
```

```
	DECLARE_DYNCREATE(CChatView)

// Attributes
public:
	CChatDoc* GetDocument();
	CSpinButtonCtrl m_upDown;
   CEdit m_buddyEdit;
	CEdit m_balance;
	CListCtrl m_listView;
	CImageList m_smallImageList;
	CImageList m_largeImageList;
	CButton m_smallButton;
	CButton m_largeButton;
	CButton m_listButton;
	CButton m_reportButton;
	//CBitmapButton b1, b2,b3,b4, b5;
	CRichEditCtrl m_richEdit;
	int index;
	UINT m_uiTimer;
	list<Player> playerList;
// Operations
public:
	void Message(LPCTSTR lpszMessage);

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CChatView)
	public:
	virtual void OnDraw(CDC* pDC);  // overridden to draw this view
```

```
protected:
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
	//}}AFX_VIRTUAL

// Implementation
public:
	int getIndex(CString& str);
	void displayCard(CString& cardNameStr, int n);
	void creatButton();
	BOOL OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult);
	virtual ~CChatView();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:
	void CreateUpDownCtrl();
	void CreateListView();
	//void CreateTreeView();
	void CreateRichEdit();
	CString toString(int num);

// Generated message map functions
protected:
	//{{AFX_MSG(CChatView)
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	afx_msg void OnGameBetting();
	afx_msg void OnGamePassing();
```

```
        afx_msg void OnGameAnothergame();
        afx_msg void OnGameChangebetunit();
        afx_msg void OnTimer(UINT nIDEvent);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in chatvw.cpp
inline CChatDoc* CChatView::GetDocument()
   { return (CChatDoc*)m_pDocument; }
#endif
```

Server side class:

The card, deck, hand is same, below show the server doc and server view class

phototype

```
class CServerDoc : public CDocument
{
protected: // create from serialization only
        CServerDoc();
        DECLARE_DYNCREATE(CServerDoc)

// Attributes
public:
        CListeningSocket* m_pSocket;
        CStringList m_msgList;
        CPtrList m_connectionList;
        int max;//total player number
        int num;
        int step;
        int total;
        Deck deck;
        bool ready;
        bool sdisp;
        bool isBegin;
        bool isfinished;
        bool afterleft;
```

```
        CString cardlist;
        CString dispInfo;
        list<Player> myPlayerList;
        list<CString> dispInfoList;
        list<CString> playerList_a;

// Operations
public:
        void UpdateClients();
        void ProcessPendingAccept();
        void ProcessPendingRead(CClientSocket* pSocket);
        CMsg* AssembleMsg(CClientSocket* pSocket);
        CMsg* ReadMsg(CClientSocket* pSocket);
        void SendMsg(CClientSocket* pSocket, CMsg* pMsg);
        void CloseSocket(CClientSocket* pSocket);
        void Message(LPCTSTR lpszMessage);
        CString toString(int num);
        CString getPlayerName(CString msg);
        CString getCommand(CString msg);
        bool isRepeat(CStringList& csl, CString cstr);
        int toInteger(CString string);


// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CServerDoc)
        public:
        virtual BOOL OnNewDocument();
        virtual void DeleteContents();
        //}}AFX_VIRTUAL

// Implementation
public:
        bool isDeplicated(list<Player> plist, CString str);
        bool isStringDeplicated(list<CString> plist, CString str);
        void cardIni();
        virtual ~CServerDoc();
        virtual void Serialize(CArchive& ar);   // overridden for document i/o
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
```

```
protected:
	//{{AFX_MSG(CServerDoc)
	afx_msg void OnGameAnothergame();
	//}}AFX_MSG
	afx_msg void OnUpdateMessages(CCmdUI* pCmdUI);
	afx_msg void OnUpdateConnections(CCmdUI* pCmdUI);
	DECLARE_MESSAGE_MAP()
};
```

class CServerView : public CEditView

```
{
protected: // create from serialization only
	CServerView();
	DECLARE_DYNCREATE(CServerView)

// Attributes
public:
	CServerDoc* GetDocument();

// Operations
public:
	void Message(LPCTSTR lpszMessage);

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CServerView)
	public:
	virtual void OnDraw(CDC* pDC);  // overridden to draw this view
	protected:
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CServerView();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CServerView)
		// NOTE - the ClassWizard will add and remove member functions here.
		//    DO NOT EDIT what you see in these blocks of generated code !
```

```
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in srvrvw.cpp
inline CServerDoc* CServerView::GetDocument()
   { return (CServerDoc*)m_pDocument; }
#endif
```

5.5 Behavioural Models

5.5.1 Sequence Diagram

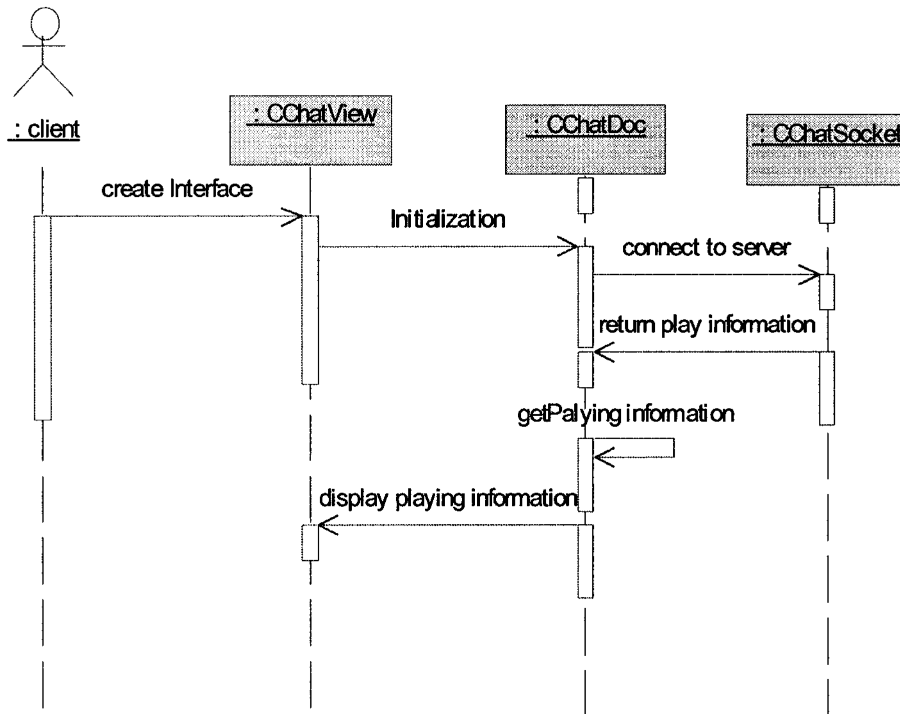


Figure 7 The Sequence Diagram for Client Subsystem

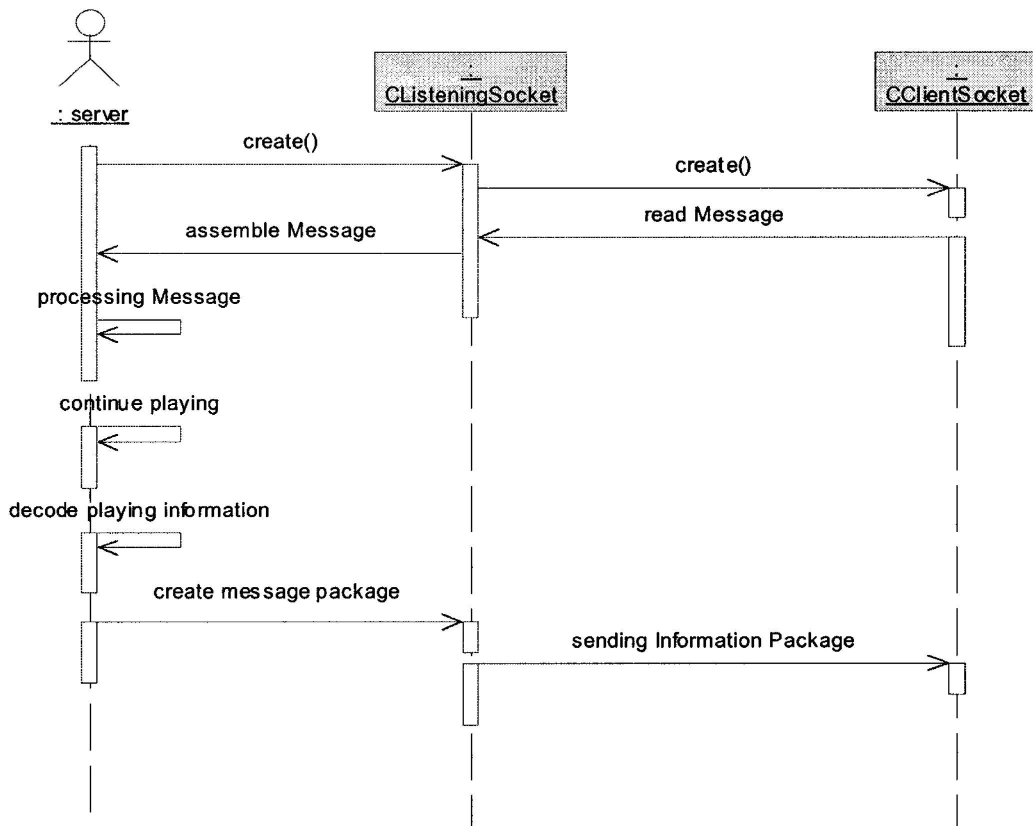


Figure 8 The Sequence Diagram for Server Subsystem

5.5.2 Activity Diagram

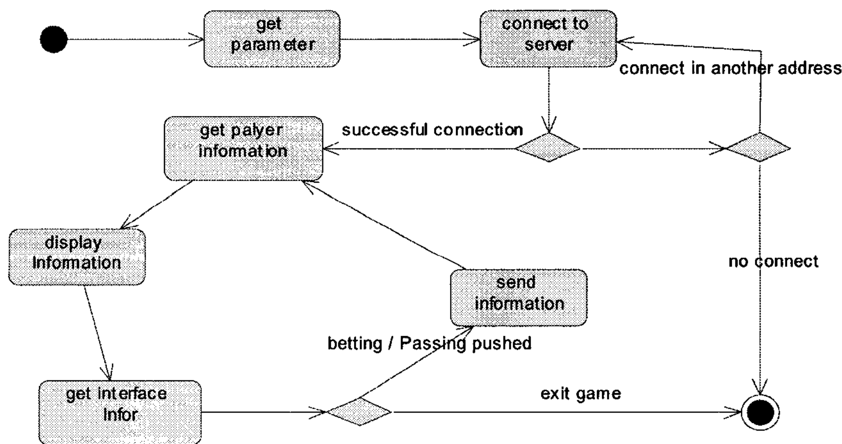


Figure 9 The Activity Diagram of Client Subsystem

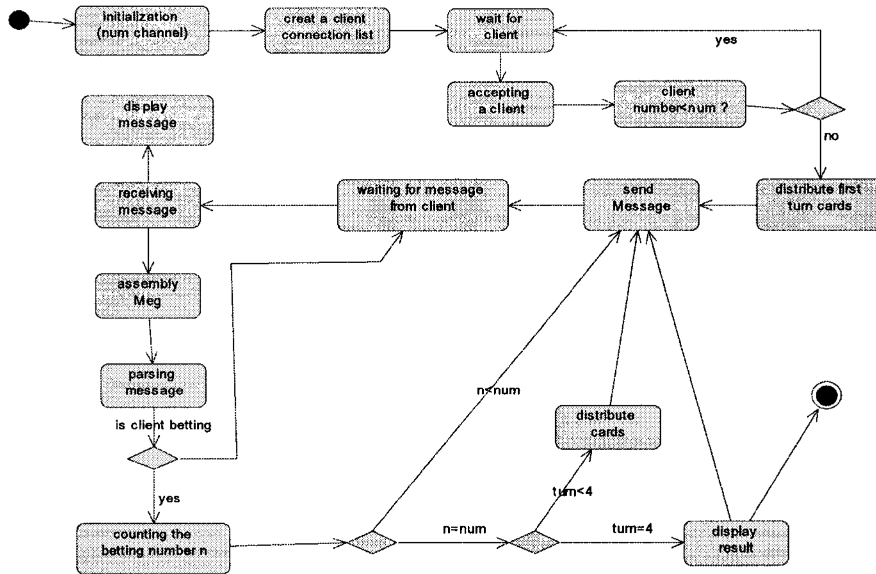


Figure 10 The Activity Diagram of Server Subsystem

Chapter 6. VPNG System Implementation

This chapter describes key algorithms used in VPNG. The whole process can be divided up into two main algorithms; they are used in the client side and server side.

6.1 Pseudo code of VPNG System

6.1.1 Pseudo Code of Client Side

```
Client side;
Create Client()
Begin:
Read in the connection information;
Connect to the server;
If successful, initial the client parameters;
Else ask user provide new server address and conent again;
End

Process Pending Read ()
Begin:
        Listening the socket
        {
                if socket is not empty
Receive Message ();
```

```
                else    return;
        }
End


Receive Message ()
Begin:
Read messages from socket archive file;
If the message is not empty;
Put all messages into the display information list;
Put the display information list point to the messages that belong to current turn
and preview turn;
For the point to current position to the end;
From the message that the pointer point to;
Get player information ();
Display Message ();
End


Get player information ()
Begin:
        Get the turn from the message and put into k;
        If  it is a new turn ( k>turn );
                Update the turn value;
If it is command join;
```

Get the player name;

If the player is new player;

Create his profile;

And add it into the players data list;

End if;

If it is command betting;

Get the player name;

Get the betting money;

Find the player in the player profile;

Update his profile;

End if;

If it is command passing;

Get the player name;

Find the player in the player profile;

Update his profile;

End if;

If it is running result;

For every running record;

Get the player name;

Get the user's betting money;

Get the player's playing record;

Find the player in the player profile;

Update his profile;

End if;

If it is final result;

Get the winner's information;

If the winner is the current player;

The player balance =total betting – the player betting;

else the player balance= – the player betting;

Get the user's betting money;

Get the player's playing record;

Find the player in the player profile;

Update his profile;

Update the player's balance;

End if;

End;

Display Message ()

Begin:

For every user create his title;

For every user's playing record;

Display in rich test field;

For every player's hands;

If it is not current player, show first card as a deck;

Else finding image to show the card;

End

Other functions in the window message loop:

```
Socket checking ()
Begin
        If some information coming
                Active Process Pending Read();
End


Betting button clicked ()
Begin
        If Betting button is clicked
                Reset the timer;
                Reading the current betting money;
                Send the message;
End


Passing button clicked ()
Begin
        If passing button is clicked
                Reset the timer;
                Send the message;
End
On timer triggered ()
Begin
```

Disconnect to the server;

End

6.1.2 Pseudo Code of Server Side

```
Server sides
Create listening Socket()
Begin:
        Lunch the initialization Dialog;
        If the confirm button of the dialog is pushed;
        Read in the initialization parameters;
        Create the server listening socket;
End

Process Pending Accept ()
Begin:
        If client request coming and the number of client < max
                Create a client socket;
Put the socket into the connection list;
End

Process Pending Read ()
Begin:
```

Read message from a client socket;

If the socket to be opened

Close and delete the socket;

Update clients ();

End

Update clients ()

Begin:

For every client socket in the connection list;

Read the message from it to a Message object;

Assemble and process the message from a client();

Put the message into a new message;

Display the message in the server ();

Send the message to every client ();

End

Assemble and process the message from a client ()

Begin:

Read the message list from the client socket;

For every message from the list;

Get player name and put into name;

Get command and put into the command;

If the command is “has just joined the game”

Create a client side message for it;

If it is new one;

Put the client message into the display information list;

End if

If the player is new

Create a profile for him and add into the player list

End if

End if;

If the command is “Pass”

Create a client side message for it;

If it is new one;

Put the client message into the display information list;

End if

Fid the player in the player list

Update his profile from the player list;

Add his profile to the temp playing player list

End if;

If the command is “has just left the game”

Create a client side message for it;

If it is new one;

Put the client message into the display information list;

End if

Fid the player in the player list

Delete his profile to the player list;

Delete his profile to the temp playing player list

End if;

If the command is betting

Create a client side message for it;

If it is new one;

Put the client message into the display information list;

End if

Read in the betting money;

Fid the player in the player list

Update his profile from the player list;

Add his profile to the temp playing player list

End if;

If all the user finish a turn of betting/passing;

Set ready to be true;

Turn number increase one;

End if

If it is fifth turn

Distribute the fifth hand to every client;

Decide who is winner;

Create a client side final result message for it;

If it is new one;

Put the client message into the display information list;

```
                End if
End if
 If ready is true
        Distribute hands for a new turn;
Create a client side result message for it;
        If it is new one;
Put the client message into the display information list;
                End if
End if
Clear the message list;
Add all messages from display information list into the message list
End
```

6.2 Interface

The VPNG system is implemented with Visual C++ and MFC. Microsoft Foundation Class Library (MFC) framework is based largely on a few major classes and several Visual C++ tools. Some of the classes encapsulate a large portion of the Win32 application programming interface (API).

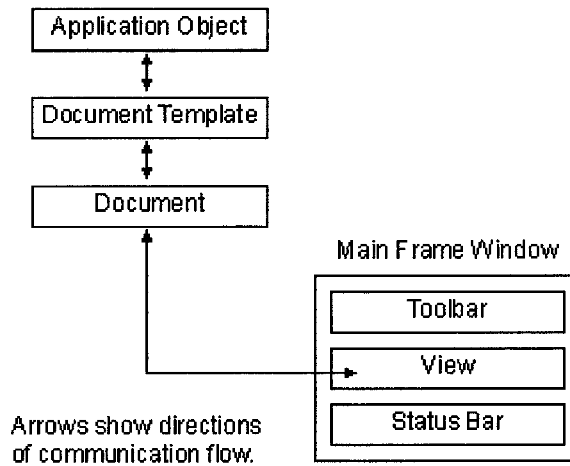


Figure 11 Objects in a Running SDI Application

6.2.1 Game Interface : Server Side

The function of server subsystem is like game controller which can decides how many players play it for this time game. For example, Mike, Lilian and Sophie decide to play Video Poker Game on Intranet or Internet by MSN chat and email. The 3 of players is input into below interface on server and choose the Game Group is “0”.

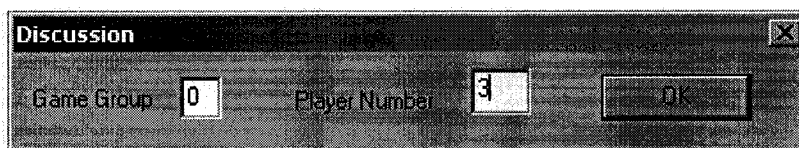


Figure 12 Setup one game group on server

During the execution of the game, the interface of server is displaying all information from each player.

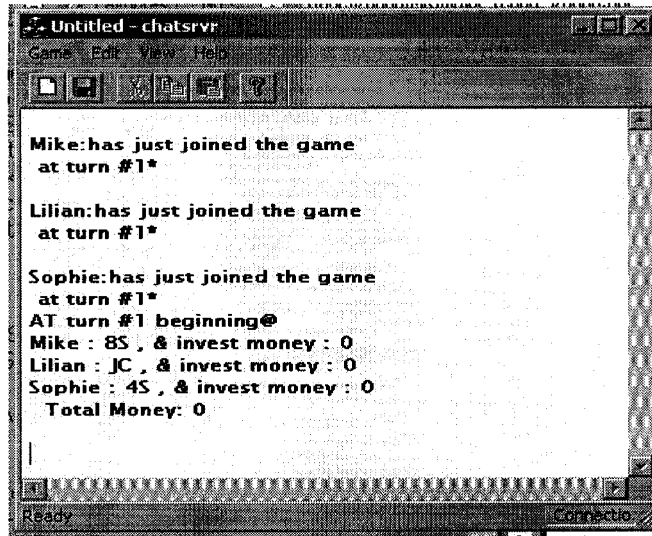


Figure 13 Interface for connection with players

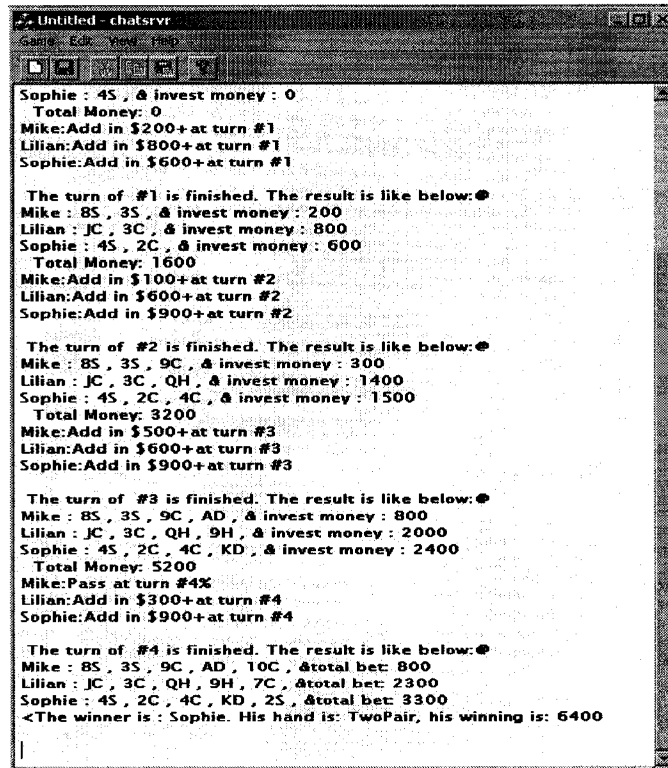
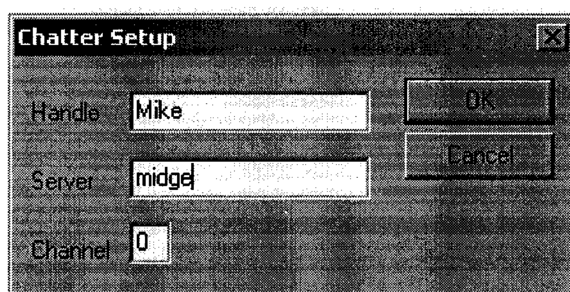


Figure 14 Interface when game is over

6.2.2 Game Interface : Client Side

When the players, for example, Mike, Lilian and Sophie decided to play a Video Poker Game together, they will run Setup interface of VPNG system on client side, enter his/her name, server name and group / channel number.



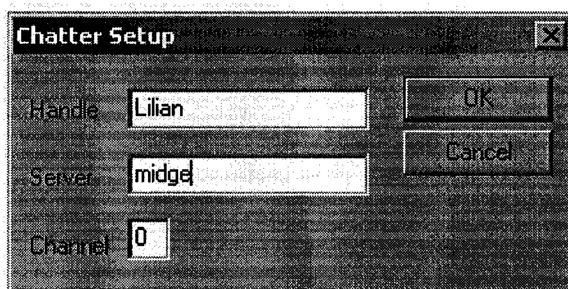


Figure 15 Setup interface for player

Then the interface “chatter” will be poped up on each clinet side. For the first turn, the card only show up to the player who is runing on his/her side and hide the others. Each game player start to bet or pass this turn depending on the situation skill, and experience.

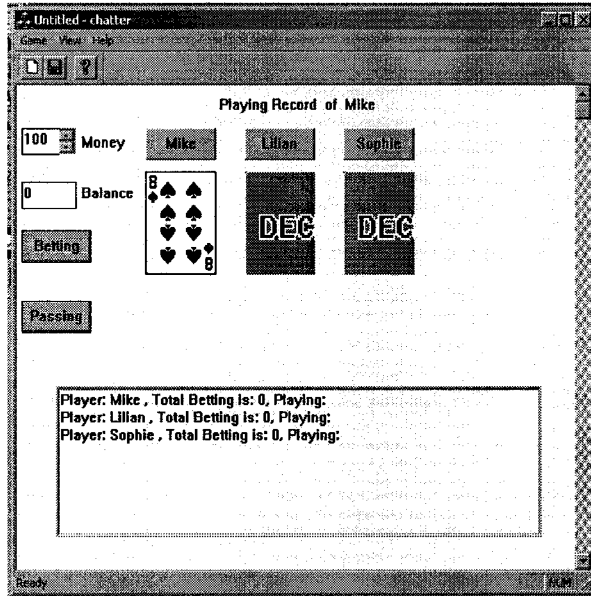


Figure 16 Interface of the first trun for player Mike

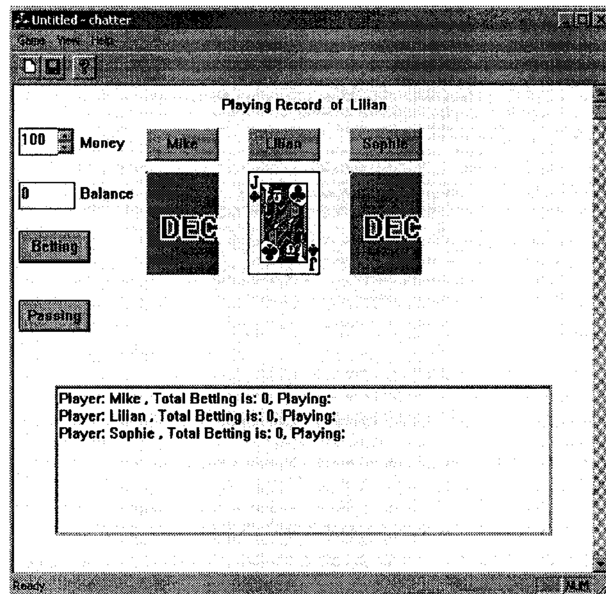


Figure 17 Interface of first turn for player Lilian

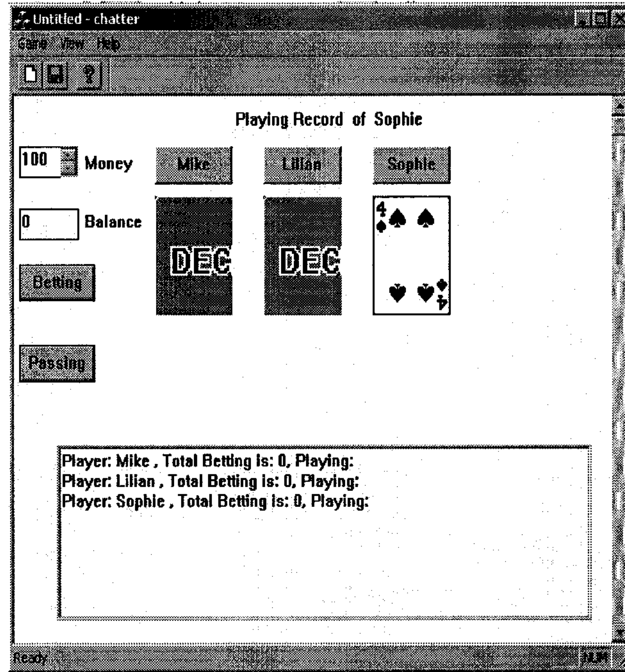


Figure 18 Interface of first turn for player Sophie

From the second turn to end, all cards will show up to every players. The game players then decide to bet or pass this turn. The current balance will be displayed.

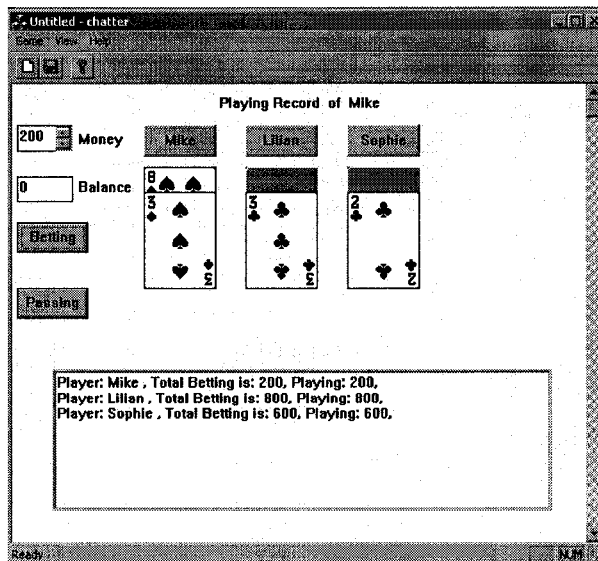


Figure 19 Interface of second turn for player Mike

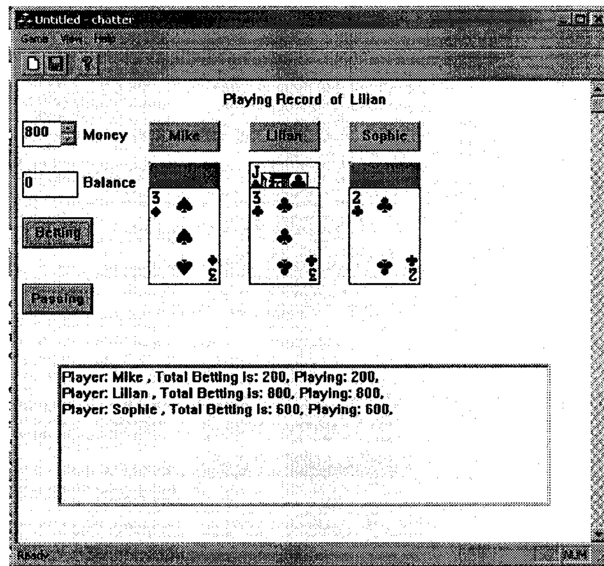


Figure 20 Interface of second turn for player Lilian

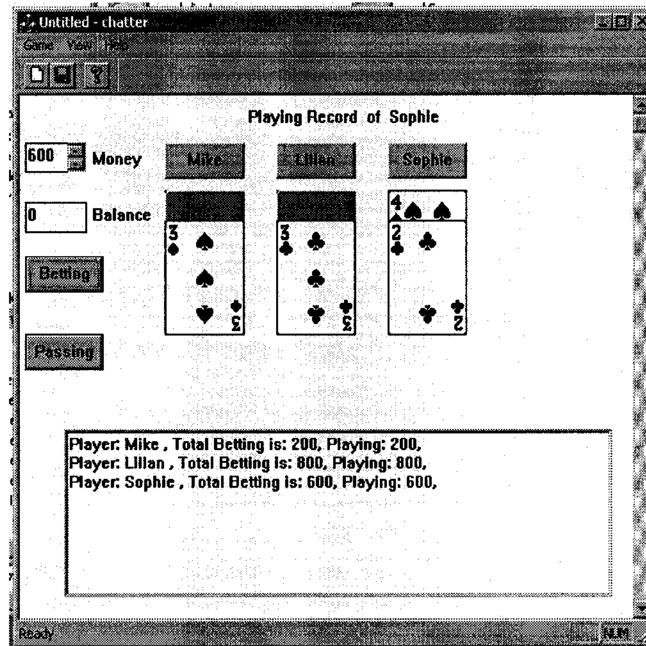


Figure 21 Interface of second turn for player Sophie

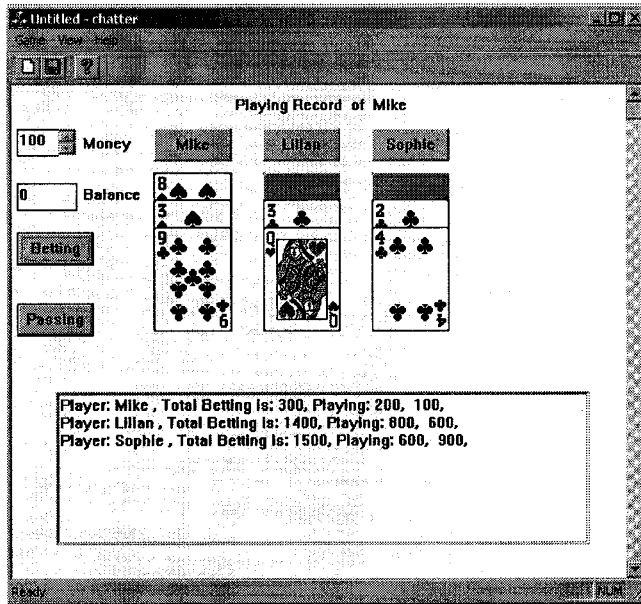


Figure 22 Interface of third turn for player Mike

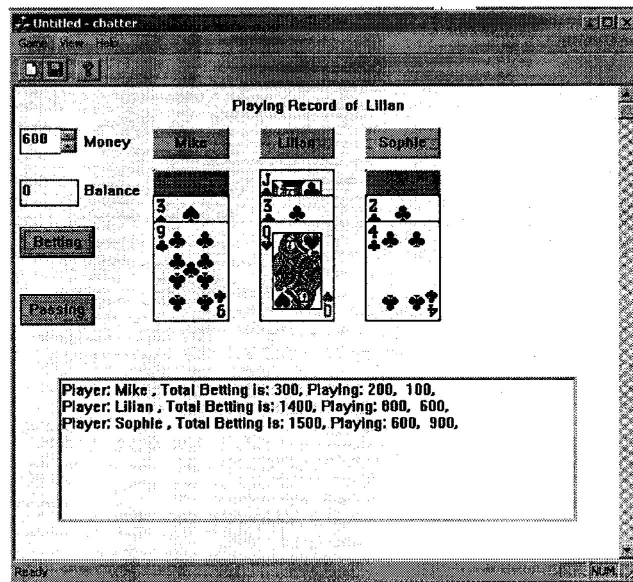


Figure 23 Interface of third turn for player Lilian

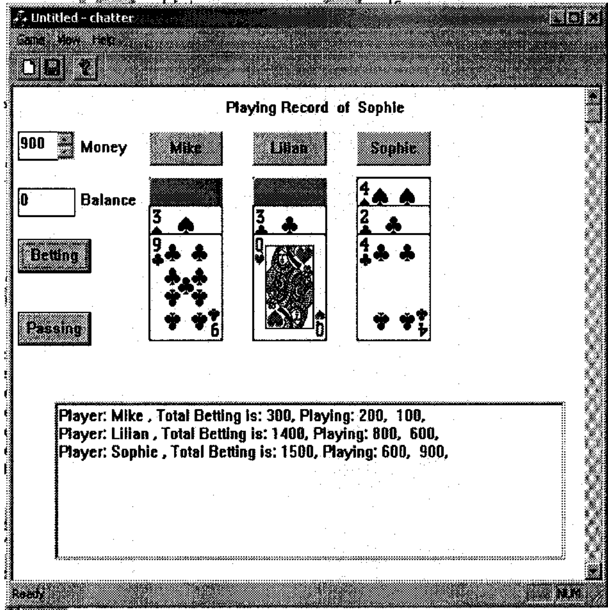


Figure 24 Interface of third turn for player Sophie

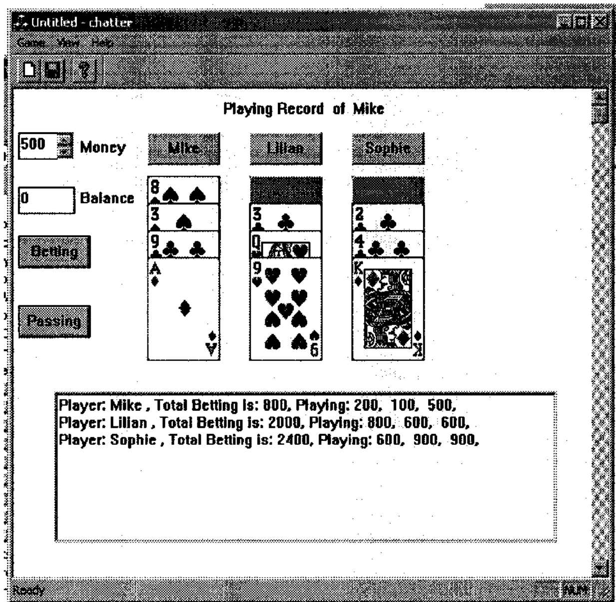


Figure 25 Interface of fourth turn of player Mike

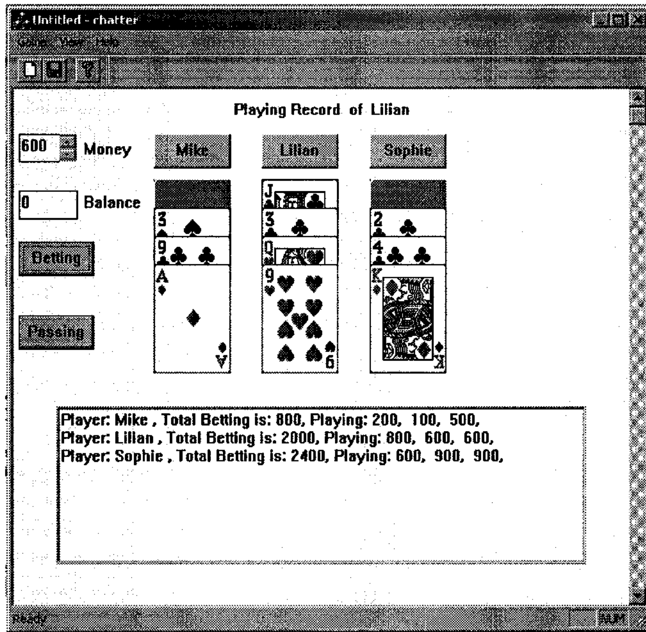


Figure 26 Interface of fourth turn for player Lilian

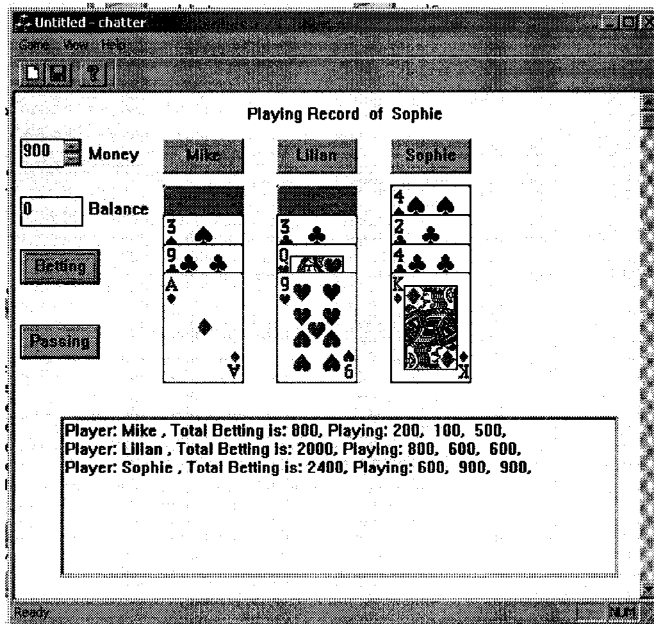


Figure 27 Interface of fourth turn for player Sophie

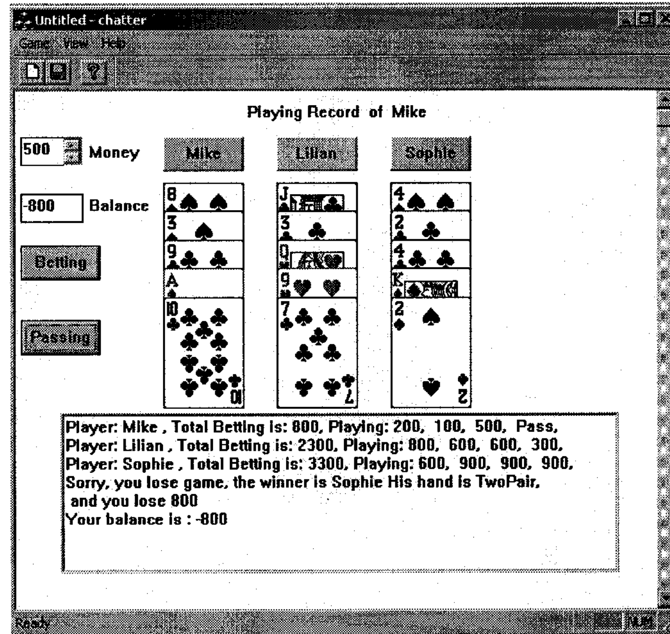


Figure 28 Interface of last turn for player Mike

Finally, all cards will displayed to every game players. The total amount of balance for every game player will be displayed. The winner and the reason will also displayed.

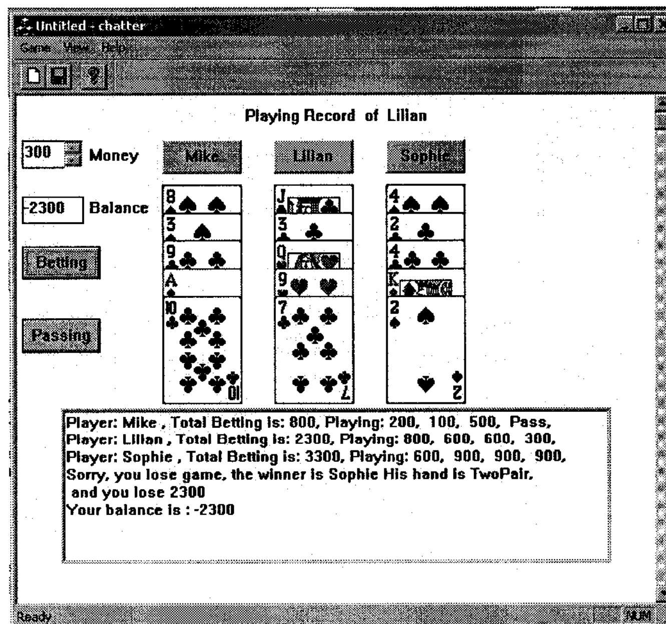


Figure 29 Interface of last turn for player Lilian

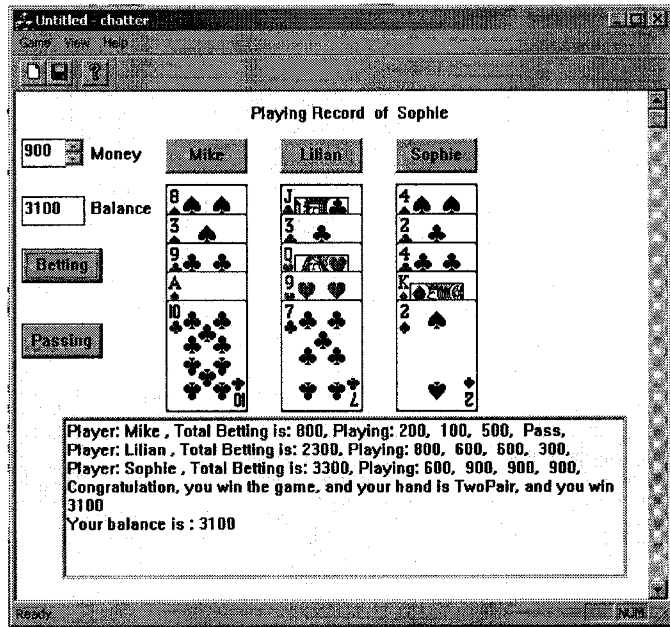


Figure 30 Interface of last turn for player Sophie

Chapter 7 Conclusion & Future Work

7.1 Conclusion

The amount of research devoted to network games is lacking, even if the popularity of network games is growing. We intended to fill some of this knowledge gap with a study on how games behave over the Internet, and provide a means of facilitating future research.

The report presents concepts related to multiplayer network game and multiplay network game programming. It presents a Video Poker game based on client/server architecture. This architecture takes advantage that has been shown to provide better services to clients and extends it to a system which changes at a very high rate. It also describes a new synchronization mechanism to provide efficient consistency between clients and server.

Our research is also useful for the design of non game related application, such as webcasts and online events with large numbers of viewers, community websites.

7.2 Future Work

Despite the number of tasks we accomplished, there are always a number of opportunities for improvement. Choices were made throughout the project that required discarding options. A few possibilities work are introduced in this section. Finally, there was a rather large set of topics that while interesting and useful, were outside the scope of our particular project.

There were three tasks yet need to be completed. We wanted to work a bit more on them so they would be as polished as the rest of the project. First, we could improve the interface of game since it should be easy to learn and use. However, a lot of MFC work could be needed while MFC is not popular tools currently in game development.

Second, we were able to design a basic structure for a client/server game and partially implement it. However, the functionality for playing this game on Internet has not yet been fully implemented. This could become fully functional.
Third, we should do more testing work to debug the system. There are some bugs to be fixed in this system.

There were several areas that while within the scope of the project, we lacked the time or resources to pursue. For example, we can add chating function to game in order to the players can talk each other during game playing. Also, when only one game player access the server and send a request to setup the game for his/her, he /she can play game hisself/herself without any win or lose. Nobody like to play game without and result. If the VPNG system could be added

a function which any player can player with server , that could be more interesting. However, there are a lot of coding work on server subsystem since some algorithms even Artificial Intelligence knowledge and algorithms could be needed.

References

[1] QuakeCon game web site: http://www.idsoftware.com/

[2] Ted Friedman "Making Sense of Software: Computer Games and Interactive Textuality", http://www.duke.edu/~tlove/simcity.htm

[3] Bong-Jun Ko, Dan Rubenstein, and Kang-won Lee, "Dynamic Server Selection for large Scale Interactive Online Games (Work-in-Progress paper) http://www.nyman-workshop.org/2003/papers/Dynamic server selection for large scale interactive online games.pdf

[4] J. Orwant, EGGG:Automatedprogramming for game generation, http://www.research.ibm.com/journal/sj/393/part2/orwant.pdf

[5] Lars Aarhus, Knut Holmqvist, and Martin Kirkengen, Generalized TwoTier Relevance Filtering of Computer Game Update Events, http://www2.nr.no/dart/projects/gisa/download/gisa-aoim-netgames2002.pdf

[6] Semion S. Bezrukov, Methods for Multiplayer Gameworld synchronization, http://www.cs.umd.edu/Honors/reports/Semion/MethodsXforXMultiplayerXGameWorldXSynchronization.doc

[7] Knut Håkon T. Mørch, Cheating in Online Games –Threats and Solutions

Version 1.0, Januar 2003, Note DART/01/03

http://www2.nr.no/dart/projects/gisa/download/Cheating in Online Games.pdf

[8] Ghita Kouadri Mostéfaoui and Soraya Kouadri Mostéfaoui , Java Shared

Data Toolkit for Multi-Player Networked Games ,

http://diuf.unifr.ch/~kouadrim/publications/JSDTpaper.pdf

[9] Programming FORUM NOKIA Version 1.0; October 29, 2003Java™

Multi-Player MIDP Game

http://ncsp.forum.nokia.com/downloads/nokia/documents/Multi_Player_MIDP_Ga

me_Programming_v1_0_en.pdf

[10] SUMEA - Game Developer Point of View ,Nokia Developer Hub - Technical

Day for Java (August 20, 2003) presentation

http://ncsp.forum.nokia.com/download/?asset_id=437;ref=ncspsupport

[11] Wu chang Feng, Wu chi Feng, On the Geographic Distribution of Online

Game Servers and Players

http://www.cse.ogi.edu/sysl/projects/cstrike/netgames03_geo.pdf

[12] Wu-chang Feng, Francis Chang, Wu-chi Feng, Jonathan Walpole, "Provisioning On-line Games: A Traffic Analysis of a Busy Counter-Strike Server", *in Proceedings of the Internet Measurement Workshop,* November 2002. http://www.cse.ogi.edu/sysl/projects/cstrike/CSE-02-005.pdf

[13] Eric Cronin Burton Filstrup Anthony Kurc, Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109-2122, A Distributed Multiplayer Game Server System, f*ecronin,bfilstru,tkurc*g*@eecs.umich.edu* , May 4, 2001 http://warriors.eecs.umich.edu/games/papers/quakefinal.pdf

[14] Eric Cronin Burton Filstrup Anthony R. Kurc Sugih Jamin_Electrical Engineering and Computer Science Department University of Michigan Ann Arbor, MI 481092122, An Efficient Synchronization Mechanism for Mirrored Game Architectures http://warriors.eecs.umich.edu/games/papers/netgames02-tss.pdf

[15] Eric Cronin, Anthony R. Kurc, Burton Filstrup and Sugih Jamin_ (fecronin,tkurc,bfilstru,jaming@eecs.umich.edu), Electrical Engineering and Computer Science Department University of Michigan, **An E_cient Synchronization Mechanism for Mirrored Game Architectures** (Extended Version) http://warriors.eecs.umich.edu/games/papers/mtap-tss.pdf

[16] Eric Cronin Burton Filstrup Sugih Jamin, Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109-2122, **Cheat-Proofing Dead Reckoned Multiplayer Games** (Extended Abstract)
http://warriors.eecs.umich.edu/games/papers/adcog03-cheat.pdf

[17] Aditya Mohan, Maurice Herlihy, *Department of Computer Science, Brown University, PO BOX 1910, Department of Computer Science, Providence, RI 02912 USA*,**Peer-to-Peer Multiplayer Gaming using Arrow Multicast: *Peer-to-Peer Quake***
http://www.cse.msu.edu/icdcs/posters/final/12_s.pdf

[18] Debanjan Saha Sambit Sahu Anees Shaikh, Network Services and Software IBM TJ Watson Research Center, Hawthorne, NY 10598, **A Service Platform for OnLine Games**
http://www.research.ibm.com/people/a/aashaikh/papers/netgames03.pdf

[19] D. Saha, S. Sahu, and A. Shaikh, *Proc. of NetGames 2003 Workshop*, May 2003.**A Service Platform for On-Line Distributed Games,**
http://www.research.ibm.com/people/a/aashaikh/papers/netgames03.pdf

[20] A. Acharya, A. Shaikh, and R. Tewari, *Proc. Open Signaling for ATM, the Internet, and Mobile Networks (OPENSIG 2000) Workshop*, Invited talk, October 2000. **Local and Wide-Area Server Selection: Techniques and Challenges** http://www.research.ibm.com/people/a/aashaikh/papers/opensig2000.pdf

[21] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos *Department of Electrical and Computer Engineering, University of Toronto* **Behavior and Performance of Interactive Multi{player Game Servers** http://www.eecg.toronto.edu/~moshovos/research/quake_ispass00.pdf

[22] Paulo Guedes_ Daniel Julin, **Writing a Client-Server Application in C++** http://citeseer.nj.nec.com/cache/papers/cs/186/ftp:zSzzSzftp.cs.cuhk.hkzSzpubzSzmach3zSzsrczSzmach_uszSzsrczSzdoczSzusenix-czPzzPz-92.pdf/guedes92writing.pdf

[23] Albert Parent, Amr Hafez, and Dan Keefe, BGS Systems Inc. **Comparative Performance Analysis of Client-Server Applications developed in Java and C++** http://www.bmc.com/offers/performance/whitepapers/docs/1997/comparative_perf_analysis_clientserver_apps_java_c.pdf

[24] Jered Wierzbicki, **The Essentials of Multiplayer Games**

or, "Some categorization, problems, solutions, general stuff about multiplayer-games"

http://www.gamedev.net/reference/articles/article722.asp

[25] David Michael, **Designing for Online Communities**

http://www.gamedev.net/reference/articles/article889.asp

[26] *Yanna Vogiazou, Research Proposal*

Presence Based Massively Multiplayer Games:Exploration of a new concept

http://kmi.open.ac.uk/publications/papers/kmi-tr-123.pdf

[27] Tobias Baumann ,**Why do we play multiplayer games?**

An analyse of today's goals and ideas behind multiplayer online games.

2003,

http://www.the2ndsky.ch/inquiry/T.B. Study (Why do we play multiplayer games).pdf

[28] Dave LaPointe Josh, **Analyzing and Simulating Network Game Traffic,**

December 19, 2001

http://www.cs.wpi.edu/~claypool/mqp/net-game/game.pdf

[29]Amund Tveit, Øyvind Rein, Jørgen V. Iversen and Mihhail Matskin. **Scalable Agent-Based Simulation of Players in Massively Multiplayer Online Games.** In *Proceedings of the 8th Scandinavian Conference on Artificial Intelligence (SCAI'03)*, Frontiers in Artificial Intelligence and Applications, IOS Press, Bergen, Norway, November, 2003.
http://www.abiody.com/people/amund/publications/2003/zereal.pdf

[30] Amund Tveit and Gisle B. Tveit. **Game Usage Mining: Information Gathering for Knowledge Discovery in Massive Multiplayer Games.** In *Proceedings of the 3rd International Conference on Internet Computing*, CSREA Press, Las Vegas, USA, June 2002, pp. 636-642.
http://abiody.com/gamemining/publications/2002/GameUsageMining.pdf

[31] Yahn W. Bernier, **Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization,**
http://www.gdconf.com/archives/2001/bernier.doc

[32] Lov Pater, **Multiplayer Gaming Popularity**, Feb, 2003.
http://www.planetblackandwhite.com/features/articles/staff/iovpater/mpex.shtml

[33] Massive multiplayer online games:

Ultima Online by Origin Systems, 1997

http://www.ultimaonline.com/, Feb 2003

Everquest by Verant Interactive, 1999

http://everquest.station.sony.com/, Feb 2003

Dark Age of Camelot by Mythic Entertainment, 2001

http://www.darkageofcamelot.com/, Feb 2003

Asherons Call 2 by Turbine Entertainment,2002

http://www.asheronscall2.com, Feb 2003

[34] FPS and other multiplayer online games:

Half-Life by Value Software, 1998

http://www.half-life.com, Feb 2003

Counter-Strike a Half-life Modification by the CS team, 1999 till today

http://www.counter-strike.net, Feb 2003

Unreal Tournament 2003 by Epic Games, 1999

http://www.unrealtournament.com, Feb 2003

Diablo 2 and **Warcaft 3** by Blizzard Entertainment, 2000 and 2002

http://www.blizzard.com/diablo2/, Feb 2003

http://www.blizzard.com/war3/, Feb 2003

Battlefield 1942 by DICE, 2002

http://www.battlefield1942.com, Feb 2003

[35] Briam “Beej” Hall, **Beej's Guide to Network Programming**

http://www.ecst.csuchico.edu/~beej/guide/net/bgnet.pdf

[36] Winsock Programmer’s FAQ,http://tangentsoft.net/wskfaq/

[37] Jim Frost, **Windows Sockets: A Quick And Dirty Primer**, Last modified December 31, 1999

http://world.std.com/~jimf/papers/sockets/winsock.html

[38] Online MSDN: http://msdn.microsoft.com/

[39] Windows Sockets 2 Application Programming Interface-An Interface for Transparent Network Programming Under Micooft Windows, Revision 2.2.2, Augest 6, 1997.

ftp://ftp.microsoft.com/bussys/winsock/winsock2/WSAPI22.DOC

[40] WinSock 2 Information, December 5, 1998

http://www.sockets.com/winsock2.htm#Docs

[41] Windows Sockets, An Open Interface for Network Programming Under Microsoft Windows, version 1.1, 20 January 1993

http://www.cs.concordia.ca/~comp445/labs/webpage/winsock.htm

[42] WinSock 2 Information,http://www.sockets.com/winsock2.htm

[43] Francis Chang, Wu-chang Feng, "**Modeling Player Session Times of On-line Games**", *in Proceedings of NetGames 2003*, May 2003, http://www.cse.ogi.edu/sysl/projects/cstrike/netgames03_flow.pdf

Appendix A. Installation of Server and Client Application
Appendix B. Setting up of Server and Client Application
Appendix C. Procedures to run demo Application

Appendix D. Glossary

FPS	First Person Shooter game
RPG	Role Playing Game
MMOG	Massive Multiplayer Online Game
MMORPG	Massive Multiplayer Online Role Playing Game
MUD	Multi User Dungeon
MFC	Microsoft Foundation Class Library
VPNG	Video Poker Networking Game
BSD	Berkeley Software Distribution
IPS	Internet Protocol Suite
QOS	Quality of Service
WOSA	Windows Open system Architecture
SPI	Service Provider Interface
API	Application Programming Interface
RPC	Remote Procedure Calls
TCP	Transmission Control Protocol
FTP	Internet file Transfer Protocol
LAN	Local Area Network

UDP	User Datagram Protocol
IP	Internet Procotol
ARP	Address Resolution Protocol
ICMP	Internet Control Message Protocol
HTTP	Hypertest Transfer Procotol