Improving Response Times in Client/Server 3D Mobile Games

Prapat Lonapalawong

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Engineering

Prince of Songkla University

2008

Thesis Title	Improving Response Times in Client/Server 3D Mobile Games
Author	Mr. Prapat Lonapalawong
Major Program	Computer Engineering

Major Advisor

.....

(Dr. Andrew Davison)

Co-Advisor

.....

(Asst. Prof. Dr. Pichaya Tandayya)

Examining Committee:

…….....Chairperson

(Asst. Prof. Dr. Suntorn Witosurapot)

.....

(Dr. Andrew Davison)

.....

(Asst. Prof. Dr. Pichaya Tandayya)

.....

(Asst. Prof. Dr. Chaiwat Oottamakorn)

The Graduate School, Prince of Songkla University, has approved this thesis as fulfillment of the requirements for the Degree of Master of Engineering in Computer Engineering

.....

(Assoc. Prof. Dr. Krerkchai Thongnoo)

Dean of Graduate School

ชื่อวิทยานิพนธ์	การปรับปรุงเวลาตอบสนองสำหรับการสื่อสารแบบไคลเอนต์/เซิร์ฟเวอร์ในเกมส์ 3 มิติบนโทรศัพท์มือถือ
ผู้เขียน	นายประพัฒน์ โลณะปาลวงศ์
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
ปีการศึกษา	2551

บทคัดย่อ

เกมแนวเดินยิงมุมมองบุคคลที่หนึ่ง (First Person Shooters หรือ FPSs) คือเกมที่ใช้มุมมองของบุคคลที่หนึ่ง, แสดงผลแบบภาพ 3 มิติ, มีปืนหรืออาวุธชนิดอื่นๆในการโจมตีคู่แข่ง ผู้เล่นทำการติดต่อสื่อสารกันด้วยสถาปัตยกรรมแบบไคลเอนต์/เซิร์ฟเวอร์ซึ่งมีข้อมูลส่วนกลางเกี่ยวกับผู้เล่นทั้งหมด (เช่นไอดีของผู้เล่น) โดยปกติใช้โปรโตคอล UDP เพื่อลดภาระในการส่งข้อมูลซ้ำใน โปรโตคอล TCP

เกม FPS มักประสบปัญหาเวลาตอบสนองโดยเฉพาะกับการใช้การติดต่อแบบไร้สายหรือโมบาย เนื่องจากการติดต่อแบบไร้สายมีความเสถียรน้อยกว่าการติดต่อแบบมีสาย และข้อมูลมีโอกาสถูกหน่วงเหนี่ยวหรือสูญหายมากกว่าซึ่งมีผลต่อเวลาตอบสนองภายในเกม อาจนำไปสู่ปัญหาอื่นๆ เช่น การอัพเดตเกมที่ติดขัด

เพื่อที่จะทดลองเทคนิคที่ใช้ในการปรับปรุงเวลาตอบสนอง เกมต่อสู้ได้ถูกพัฒนาขึ้น เรียกว่า PenguinM3G เป็นเกม FPS 3 มิติจำลองพื้นฐานบนโทรศัพท์มือถือด้วยการสื่อสารแบบไคลเอนต์/เซิร์ฟเวอร์และโปรโตคอล UDP เซอร์ฟเวอร์สามารถจำลองความเสถียรของการติดต่อสื่อสาร แล้วทำการทดสอบเปรียบเทียบเวลาตอบสนองรูปแบบต่างๆของเกมระหว่างเกมที่ใช้เทคนิคปรับปรุงเวลาตอบสนองกับเกมที่ไม่ใช้เทคนิคใดๆ ผลลัพธ์ของการทดลองบ่งบอกว่าการใช้เทคนิคปรับปรุงเวลาตอบสนองหลายๆแบบเข้าด้วยกัน จะช่วยปรับปรุงเวลาตอบสนองของเกมในหลายๆรูปแบบได้ดีขึ้นตั้งแต่ 20 ถึง 90 เปอร์เซนต์เมื่อเปรียบเทียบกับเกมที่ไม่ได้ใช้เทคนิคปรับปรุงเวลาตอบสนอง

Thesis Title	Improving Response Times in Client/Server 3D Mobile Games
Author	Mr. Prapat Lonapalawong
Major Program	Computer Engineering
Academic Year	2008

ABSTRACT

"First Person Shooters", or FPSs, are games that use a first person viewpoint, 3D rendering, and a gun or other weapon to shoot at opponents. The players communicate using a client/server model, utilizing centralized information about all the players (e.g. the players' IDs). Typically, the UDP protocol is employed to avoid the retransmission overheads inherent in the TCP protocol.

FPSs suffer from "response times" issues, especially if the game is running over a wireless or mobile network. A wireless network is less reliable than a wired version - packets may be delayed or dropped more frequently, which can impact on game response times, and cause other issues, such as flickering updates.

To test our techniques for improving response times, we develop a combat game called "PenguinM3G", a simulated mobile phone-based 3D FPS using a UDP client/server model, and a server which can simulate varying network reliable. Game response times are compared between a client utilizing various combinations our response time techniques, and a 'vanilla' version of the game with no use of our techniques. The results show that a mix of different techniques is required to produce across-the-board improvements of the game's response times, 20% to 90% better than the response time for the game with no techniques enabled.

ACKNOWLEDGEMENT

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the Department of Computer Engineering for giving me permission to commence this thesis in the first instance, to do the necessary research work and to use departmental PC and equipment.

I am deeply indebted to my supervisor Dr. Andrew Davison who helps and supervises me in all the time of the research and writing of this thesis, and my co-advisor, Asst. Prof. Dr. Pichaya Tandayya who gives me many useful advices and supports me through out this research. Asst. Prof. Dr. Suntorn Witosurapot and Asst. Prof. Dr. Chaiwat Oottamakorn of examining committees who help me pointing out many things which are useful to improve this research.

I want to thank all of my friends for support and help me with the tests, and especially, I would like to give my special thanks to my family who patiently support me to complete this work.

Prapat Lonapalawong

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

LIST OF TABLES

LIST OF TABLES (CONT.)

LIST OF FIGURES

LIST OF FIGURES (CONT.)

LIST OF FIGURES (CONT.)

LIST OF ABBREVIATIONS

3D	Three Dimensional
API	Application Programming Interface
DR	Dead Reckoning
FPS	First Person Shooter
GUI	Graphical User Interface
IP	Internet Protocol
LAN	Local Area Network
J2ME	Java 2 Micro Edition
J2SE	Java 2 Platform Standard Edition
JSR	Java Specification Request
M3G	Mobile 3D Graphics
NTP	Network Time Protocol
PDA	Personal Digital Assistant
PC	Personal Computer
PHBDR	Position History Based Dead Reckoning
SD	Standard Deviation
TCP	Transport Control Protocol
UDP	User Datagram Protocol
WTK	Wireless Toolkit

CHAPTER 1

INTRODUCTION

The mobile phone is becoming one of the most important types of communication, offering many kinds of services, including games. A popular game type is networked First Person Shooters (FPSs), such as “Robot Alliance”, a shooting game with massive multiplayer game play over mobile networks [1]. Though, this game is not about directly players battling each other. The multiplayer mode is the players join the same faction and try to win the same mission with a variety of sceneries and animated Three Dimensional (3D) characters. A screenshot of Robot Alliance is shown in Figure 1.1.

Figure 1.1: Robot Alliance Screenshot

This type of game is suitable for a client/server type architecture since centralized control is more organized (the information for all the players is stored at one place where it is easy to update and distribute) and it’s also easier to establish communication (all the players only need to know the server Internet Protocol (IP), no need to know each of the other players IP). To reduce the amount of communication, the architecture consists of a fat server and fat clients. The fat server, which typically is a Personal Computer (PC) cluster, deals with packets routing and clients’ information such as login/logout of all players, while the fat clients (which are mobile phones) will get some heavy tasks to do, apart from the communication to the server, such as 3D processing and game logic. This reduces the cost of the heavy contents of the 3D processing communication between the client and the server.

1.1 Problem Statement

In general, action games require fast interaction between players, and need fast response times. Response time can be divided into two types: 1) one-way response time, and 2) two-way response

time. One-way response time is how long before a local avatar (representing the local player) sees a remote avatar changing due to a change triggered by the remote avatar. E.g. a remote player moves and the local player sees him. Two-way response time is the period of time from when a local avatar does something to a remote avatar while the change is shown in the local space (representing the local game world). For example, a local player shoots his gun at the remote player and sees that the remote player is shot.

The example 3D FPS mobile game used in this thesis is Penguin Mobile 3D Graphics (PenguinM3G). The player is a penguin who can shoot bullets and try to kill the other players to get points. The other elements in the game are the bullets area which can refill the bullets of the player, and colored spots that can get the scores and increase the player lives. The client uses the Java 2 Micro Edition (J2ME) Wireless Toolkit (WTK) on a PC and the server is implemented using Java 2 Platform Standard Edition (J2SE) on the PC for central communication [2][3]. It simulates mobile network reliability by generating packets delay or loss. For example, 90% stable (chance of packets delay or loss is 10%) or 75% stable (chance of packets delay or loss is 25%) which will slow down the response time.

In a mobile environment there are many factors that can slow response time, such as high latency (the amount of time required to transfer a bit of data from one place to another) [4], limited bandwidth (the rate at which the network can deliver data from a sender to the destination host), unreliable connections or packet loss. These problems greatly affect the quality of game play. To address these problems, three kinds of techniques are introduced to improve the response time in the game: general techniques which allow any kind of avatar to be updated in the context of poor network communication [5], PenguinM3G-specific optimizations, and networking techniques (e.g. packet grouping).

1.2 Objectives

1. Develop a 3D FPS client/server User Datagram Protocol (UDP) mobile game using J2ME technology. The player represented by the penguin can move, rotate and shoot bullets at the other players, and also can move onto a bullets area to refill the bullets and onto colored spots to increase their scores and lives. These actions require different kinds of the response times which our techniques will try to improve.

2. Study and use different kinds of general techniques like “dead reckoning (DR)”, “smoothing” [6], “visual field updating”, game specific techniques, e.g. “avatar blinking” and “avatar dying”, and application-level networking techniques, such as “packets grouping” and “packets priority”, to help reduce response time in the game.

3. Develop a game server using J2SE technology and vary the network characteristics on the server side, e.g. drop or delay the packets at the rate of 10% or 25%, and see how our techniques maintain good response time.

4. Test and compare the results of the game between three cases: applying no techniques, enabling individual techniques and enabling multiple techniques together, in order to see which can most improve the response times of the game.

1.3 Scope

1. We are using a 3D FPS client/server based game for our case study because it involves avatars, frequent updates and needs fast response times. Many techniques will be needed to improve the response time, such as dead reckoning to help the game avatar updating, and avatar blinking to help give a fast response to the player while waiting for packets to travel over the network.

2. We will develop and compare various measures e.g. one-way response time, two-way response time, packet details, to decide how to apply our techniques to the game.

3. We will develop and test in a simulation environment using the J2ME WTK emulator. WTK copies will run as the clients, connected to a central PC server on a Local Area Network (LAN). Many PCs are already available and the LAN environment is stable enough to get average results. Real phones are not used since the aim of this thesis is to focus more on response time techniques rather than on the mobile game itself, so a simulation environment is suitable. The server will control the characteristic of the mobile environment (unreliable network), delay (wait before sending packets back to clients) and packet loss (random drop or denial of packets). For 3D development, the additional WTK Application Programming Interface (API) M3G for J2ME (Java Specification Request (JSR) 184) is used [2]. M3G has many essential classes for creating a 3D scene for a small device such as a mobile phone. The *FileConnection* API is also utilized; it is part of the Personal Digital Assistant (PDA) Profile for J2ME (JSR 75) [2]. The *FileConnection* API can access the local file system, which can be used to store the response time results after testing has finished.

4. We use the UDP transport protocol for data sending between the client and the server. In a wireless environment with an unreliable connection, as in a game like this, UDP is more suitable than Transport Control Protocol (TCP) because it does not spend time retransmitting lost packets, and there is less overhead than TCP in creating communication.

5. We will not develop new transport protocols because the standard hardware and the standard UDP communication are already sufficient.

1.4 Tools

1. Both client and server use the same type of PC: Intel Pentium 4 CPU 2.80 GHz. RAM 1 GB. The operating system is Microsoft Windows 2000 Service Pack 4, and the monitor card is Asus Extreme Ax550 256 MB.

2. J2ME WTK emulator version 2.2 with M3G, and *FileConnection* API for simulating the client side [2].

3. J2SE version 1.5.3 for the WTK compiler and server side development [3].

CHAPTER 2

LITERATURE REVIEW

This section gives background on dead reckoning and smoothing [6], the prototype 3D standalone game GunnerM3G [7], and TCP and UDP. WTK client profiling is used for analyzing the work load of the J2ME program [2], and the statistic significant of the results is measured using z-scores [8].

2.1 Dead Reckoning

To help the game reduce the effect of packet delay/loss due to the unreliable network, dead reckoning (DR) can be implemented on the client side to "predict the future movement" of a remote avatar. This technique can enable the remote avatar to keep moving even when no update packets arrive due to the packets delay or loss, thus maintaining its response time. For example, in Figure 2.1, there is an object whose current position is east 1000 feet, north 1000 feet, and it currently moving east. The predicted position one second later will be east 1050 feet, north 1000 feet, by using the previous position (east at 1000 feet, north at 1000 feet) and its current velocity (east at 50 feet/sec).

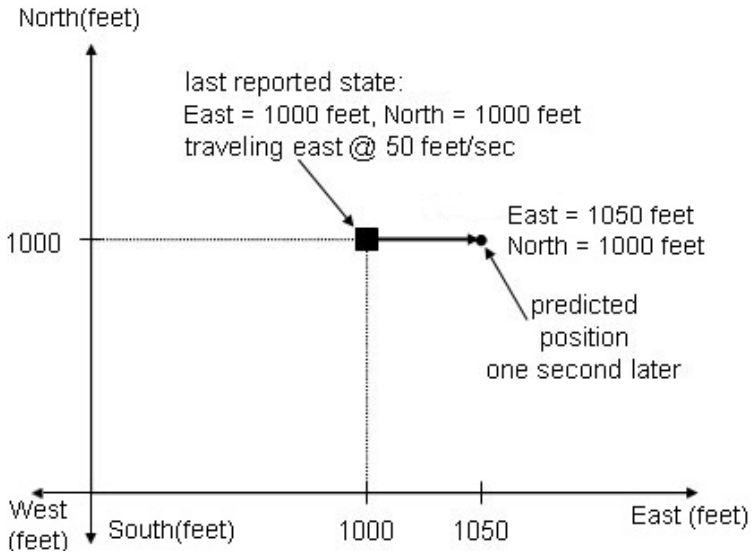


Figure 2.1: Predict Future Movement with Dead Reckoning [6].

There are many types (or 'orders') of DR: zero order uses only the previous position, first order uses velocity, and second order employs acceleration. DR techniques can also vary as how they employ previous input. The customized DR technique used in this thesis is called Position History Based Dead Reckoning (PHBDR) [9].

Position History Based Dead Reckoning (PHBDR)

PHBDR utilizes only positions, and calculates the velocity and acceleration from the previous few positions of the avatar. PHBDR consists of two steps:

- Step 1 is a tracking step which predicts the remote position from the available last few positions of the avatar until the next update arrives. This step also chooses which order of DR is used depending on the angle between the three most recent update positions. If the angle is small, or the avatar is turning rapidly, first order is chosen, otherwise, second order is employed.
- Step 2 is a convergence step that adjusts the current display position to converge upon the future predicted position obtained in step 1.

For example, in Figure 2.2, the gray dots are the real update positions of the avatar and the dashed line connecting the colored dot is the remote tracking path it is moving along based on the update history. The past (prior) displayed position (the long black line) is displayed and calculated from the PHBDR tracking step. After the current time (the vertical dash line), the convergence step predicts the convergence point. The real update position will be moving along the remote tracking line (dash and dot line) which should pass the predicted convergence point while the real display from PHBDR will try to converge to the convergence point along the convergence path (small dotted line).

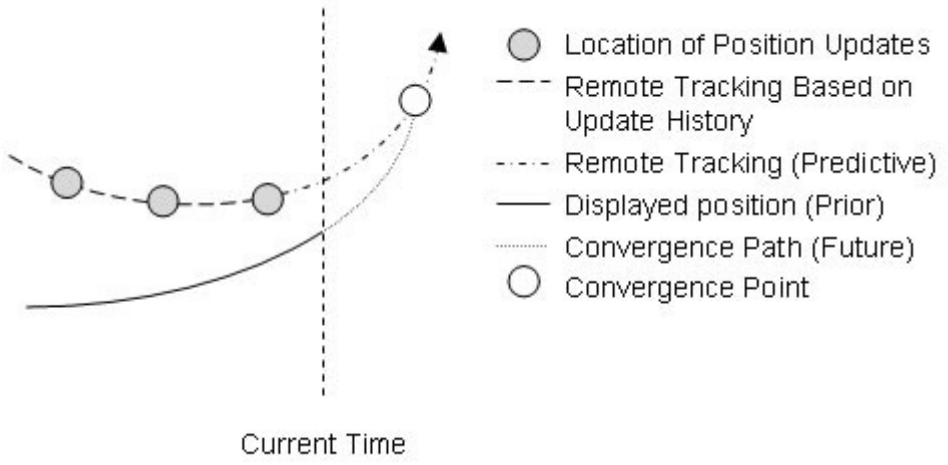


Figure 2.2: Tracking and Convergence Steps in PHBDR [6].

2.2 Smoothing

Smoothing reduces the discontinuities after updates by interpolating between the last two known points [6]. The trade-off of this technique is that the update accuracy is reduced in order to create a more natural display. Smoothing is used along with DR to gradually move/rotate an avatar from a predicted position to its correct position and angle.

An example of smoothing is shown in Figure 2.3. At time t1, the prediction is started from point A to point B and C. At time t2, a new state update message arrives with the real position. DR will calculate the next prediction based on the t2 position which is point E, with smoothing, point D will be added from interpolating the position between point C and point E.

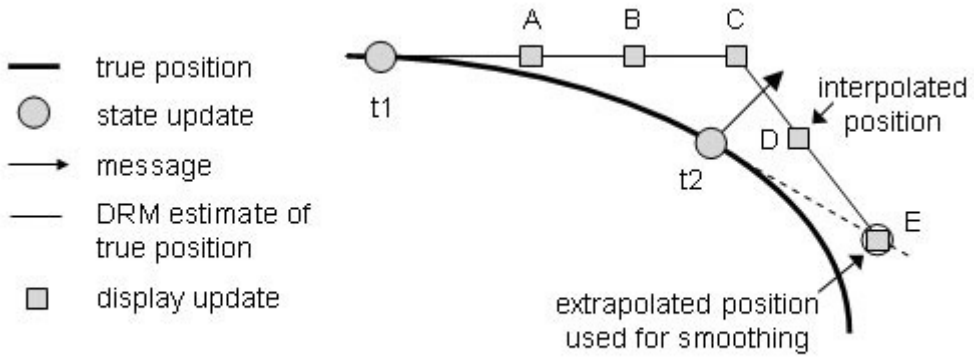


Figure 2.3: DR with Smoothing [6].

2.3 The GunnerM3G Program

PenguinM3G borrows some elements from a basic M3G game, GunnerM3G (see Figure 2.4), a standalone FPS demo running on WTK with no network capabilities. When the player fires the gun, a flash will appear from the gun. If the penguin is hit, an explosion-like fireball, the penguin ID, and the text "HIT!" are displayed. The penguin will disappear if you shoot it three times.

Figure 2.4: Original GunnerM3G Program [7].

The class diagram of GunnerM3G is shown in Figure 2.5:

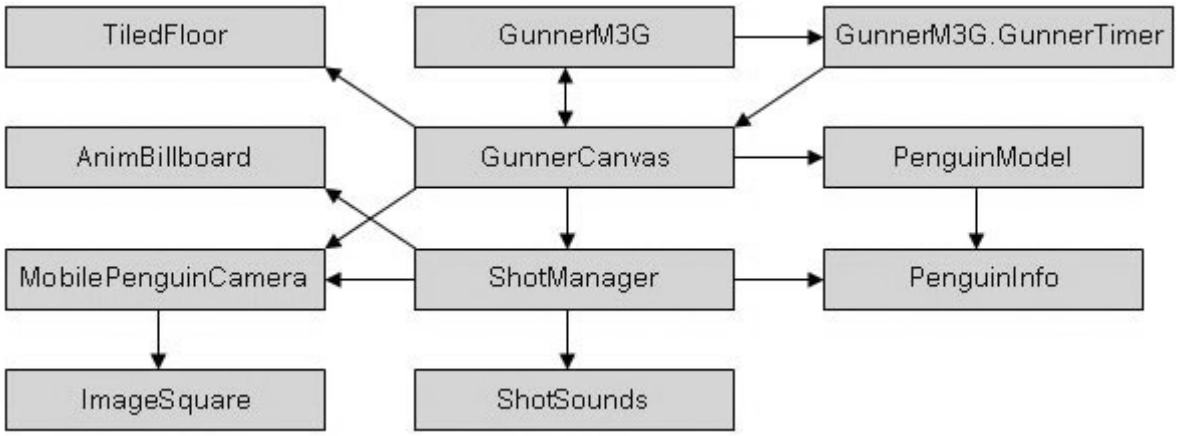


Figure 2.5: GunnerM3G Class Diagram [7].

GunnerM3G is the top-level of the application; it uses *GunnerTimer* to periodically update the canvas by calling *update()* in *GunnerCanvas*. *GunnerCanvas* creates the 3D scene, using *TiledFloor* for the floor, and two instances of *PenguinModel* for the penguins.

MobileGunCamera manages the camera, and utilizes *ImageSquare* instances for it's attached "gun hand" and "shot flash" images as shown in Figure 2.6. Gun hand or *GunMesh* will be placed at (-0.1, -0.055) on the yz plane and the shot flash or *shotMesh* will be placed at (-0.15, -0.05) on the yz plane. The flash is invisible by default but will appear when the player presses the fire key.

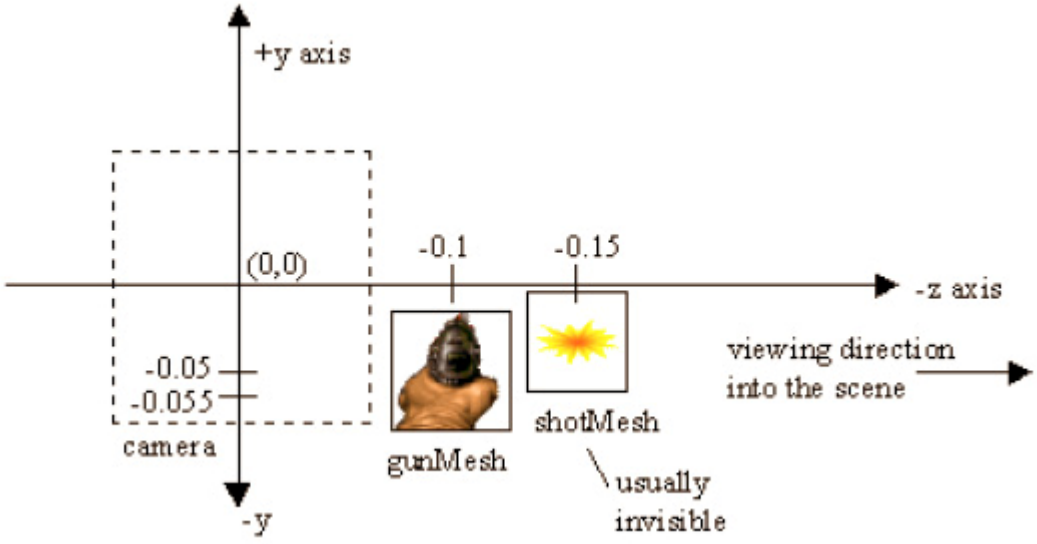


Figure 2.6: "Gun hand" and "Shot flash" [7].

ShotManager responds to the user pressing the fire key by making *MobileGunCamera*'s "shot flash" appear, and playing a laser noise through *ShotSounds*. The *ShotManager* sends a pick ray into the scene as shown in Figure 2.7, which may intersect with a penguin model. Relevant penguin data is stored in a *PenguinInfo* object stored in the model, which is used by *ShotManager* to report the hit.

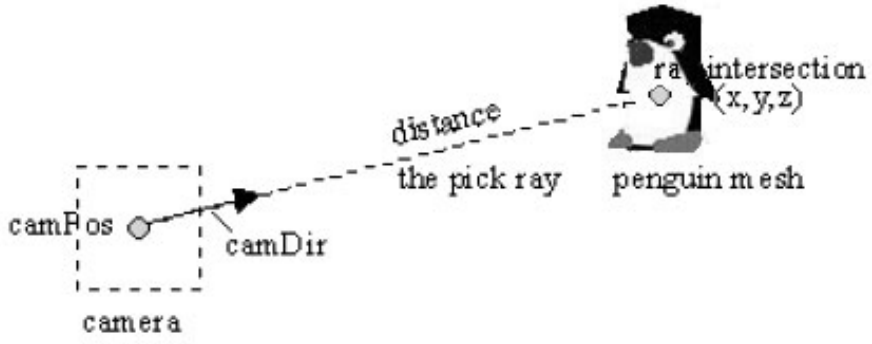


Figure 2.7: Shoot the Gun with the “Pick Ray” [7]

The pick ray requires a starting point and direction vector (*camPos* - camera current position and *camDir* - camera current forward direction) and progresses in that direction until it intersects a pickable mesh. The distance traveled is obtained from a *RayIntersection* object, and the (x,y,z) intersection point on the mesh can then be calculated.

The animated fireball is controlled by *AnimBillboard*, and the explosion sound by *ShotSounds*.

When implementing the test game for this thesis (PenguinM3G), some GunnerM3G classes will be reused and changed. All the classes in PenguinM3G are shown and briefly described below. They can be divided into reused GunnerM3G classes, modified classes, and new (added) classes.

The classes that are reused without change from GunnerM3G are:

- *TiledFloor* (creates the floor)
- *ShotSounds* (makes the sound when shooting)
- *ImageSquare* (displays the penguin beak and shooting flash)
- *AnimBillboard* (animated shooting explosion)

The modified classes:

- *GunnerM3G* (becomes *PenguinM3G*, the main class)
- *GunnerCanvas* (becomes *PenguinCanvas*, handles the game graphics)
- *PenguinModel* (creates the penguin avatar)
- *MobileGunCamera* (becomes *MobilePenguinCamera*, and handles the game camera)

- *ShotManager* (handles and checks if the penguin is shot by using a pick ray)
- *PenguinInfo* (contains penguin information, such as the penguin's previous positions used in DR, and timing information.)

The new *PenguinM3G* classes:

- *Spot* (creates the life spots and contains the spots status if the spot has already been visited or not)
- *Bullet* (creates the bullets area and contains the bullets area status if it's already been visited or not)
- *FileManager* (writes test results into a file)
- *PlayerInfo* (maintains player information, such as the number of player's lives, scores, bullets left)
- *TimeMeasure* (measures the update method)
- *Measuring* (calculates mean, Standard Deviation (SD), percentage of response times)
- *Sender* (sends UDP packets)
- *GeneralAndGameTechnique* (handles general techniques and game specific techniques)
- *ClientHandler* (sends/receives, handles packets and network techniques)

The more detailed of *GunnerM3G* classes are in chapter 4: Game Client.

2.4 TCP Client/Server

An example of network communication taken from "Threaded TCP Clients and Server" [7] was employed to create the game server, but changed to utilize UDP communication.

The server uses threads to communicate with its clients, and a shared object to maintain client information. Each client employs a thread to watch for server communication. The client consists of a Graphical User Interface (GUI) for user input and output that can send messages to the server, and a thread to wait for a message from the server which is then displayed in the GUI area. There are special messages, like "who" to retrieve the list of current chat clients. On the server side, there is a thread spawned to handle each new client since it uses the TCP protocol which needs to maintain a connection.

An example when two clients connect to the server is shown in Figure 2.8. The message sent from one client will be received by all the clients, and the client can receive a list of all the online clients via the Who command button.

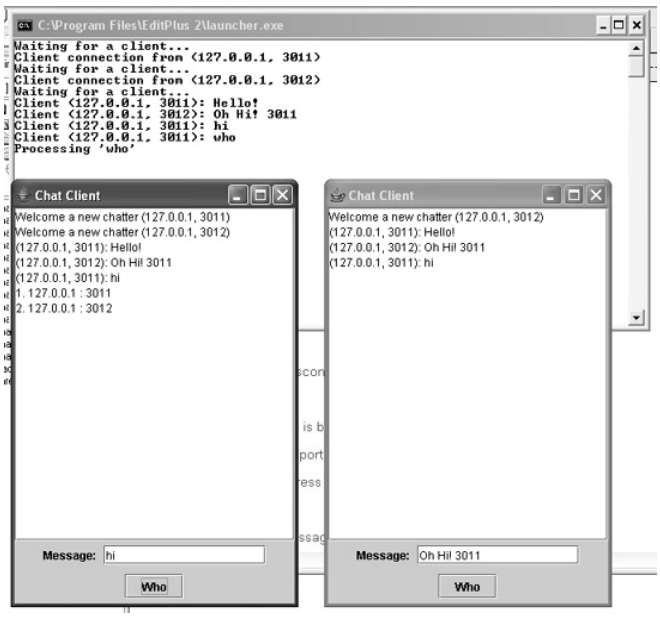


Figure 2.8: Example of Threaded TCP Clients and Server

The class diagram of Threaded TCP Clients and Server is shown in Figure 2.9:

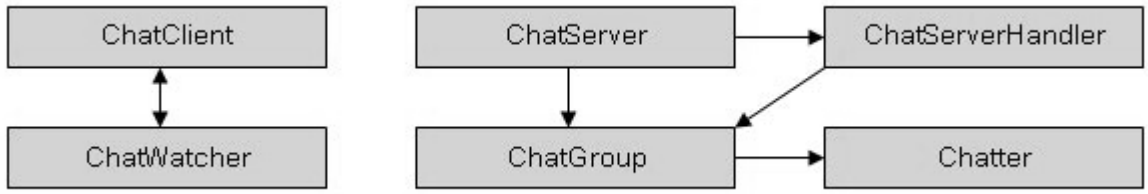


Figure 2.9: Threaded TCP Clients and Server Class Diagram

ChatClient interacts with the *ChatServer*. It can send the following messages:

who - a list of users is returned.

bye - client is disconnecting.

any text - which is broadcast to all clients.

In each client, there is a separate threaded *ChatWatcher* object for processing messages coming from the *ChatServer* object.

ChatServer waits for client connections and creates *ChatServerHandler* threads to handle them. Details about each client are maintained in a *ChatGroup* object which is referenced by each thread.

ChatServerHandler is a thread dealing with a client. Details about a client are maintained in a *ChatGroup* object, which is referenced by all the threads.

ChatGroup maintains info about all the current clients. It handles the addition/removal of client details, the answering of "who" messages, and the broadcasting of a message to all the clients.

Chatter stores information about a single client e.g. the client's IP address, port, and output stream. The output stream is used to send messages to the client. The address and port are used to "uniquely" identify the client.

When implementing the game server in PenguinM3G, these classes will be reused and changed. All the classes in the PenguinM3G game server are briefly described below. They can be divided into modified classes and newly added classes.

The modified classes are:

- *ChatServer* (the main class)
- *ChatServerHandler* (extracts packet details)
- *ChatGroup* (maintains information on all clients)
- *Chatter* (stores information on a single client)

The new classes are:

- *ReceiveMsg* (handles the simulation reliability (random delays packets or drops them))
- *PacketDelay* (the thread to delay packets)
- *RefreshObj* (the thread to re-enable the life spots and bullets areas after a player has visited them)

2.5 UDP Client/Server

The threaded TCP application of the last section is modified to use the UDP protocol. The first step is to create a J2SE UDP client and server, and the next is to implement a J2ME UDP client.

J2SE UDP Chat Client and Server

Several clients can connect to a server and broadcast messages to each other. A client can request a list of current chat users from the server by using a Who button. With no long term TCP connection, a client must now send an initial "join" message to the server when connecting for the first time. The original TCP client created a socket to connect to the server, without the need for a join message.

For example in Figure 2.10, Client1 joins the server by sending a join message to the server. When the server receives the join message, it broadcasts a welcome message with client1's IP address and port to all the other clients which informs them that there is a new client.

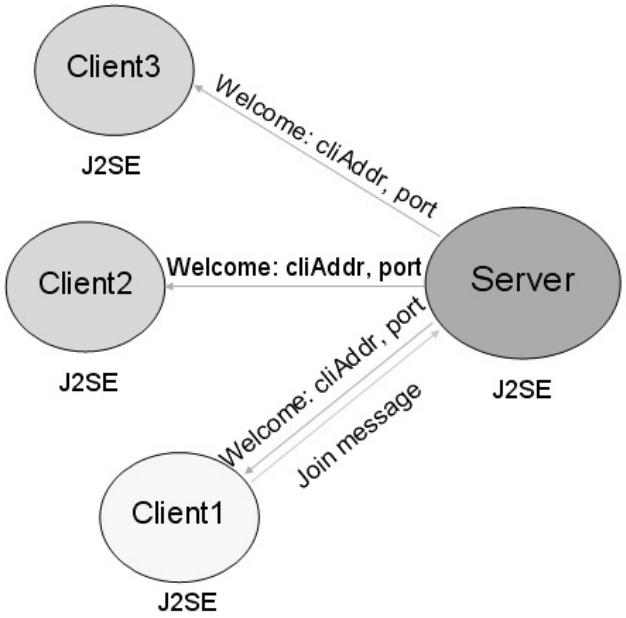


Figure 2.10: A Client Sends a Join Message and the Broadcast Response from the Server

J2ME UDP Chat Client

The J2ME UDP client is much the same as the J2SE version but it is reimplemented to run on a mobile device. It still connects to the J2SE UDP server, and the clients can send message to each other, as shown in Figure 2.11.

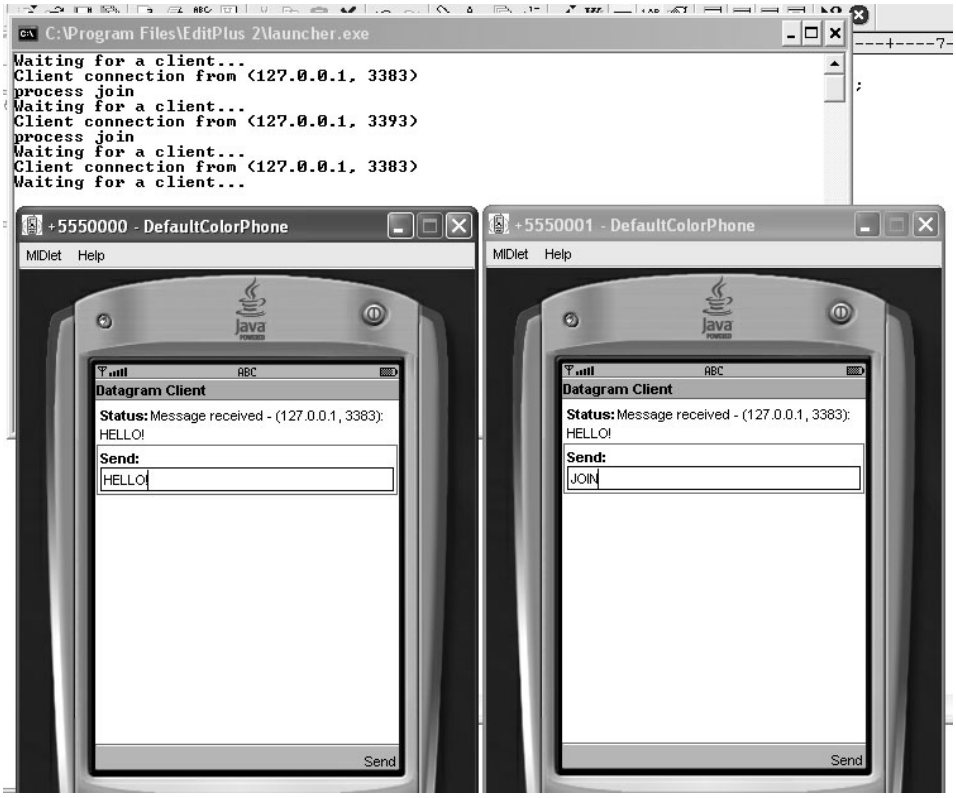


Figure 2.11: Example of J2ME UDP client and J2SE UDP server

An example of a J2ME client sending messages to the J2SE server is shown in Figure 2.12. Client1 sends a “hello” message to the server. The server receives the message, and broadcasts it to all the clients, along with client1’s address and port.

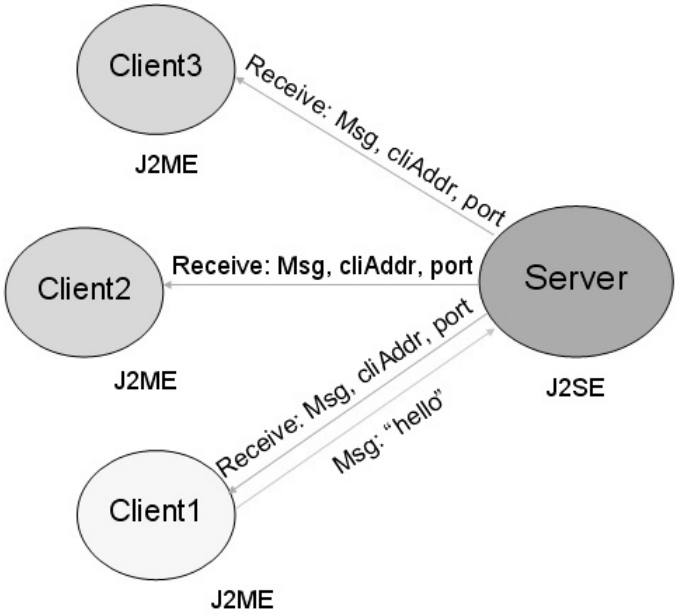
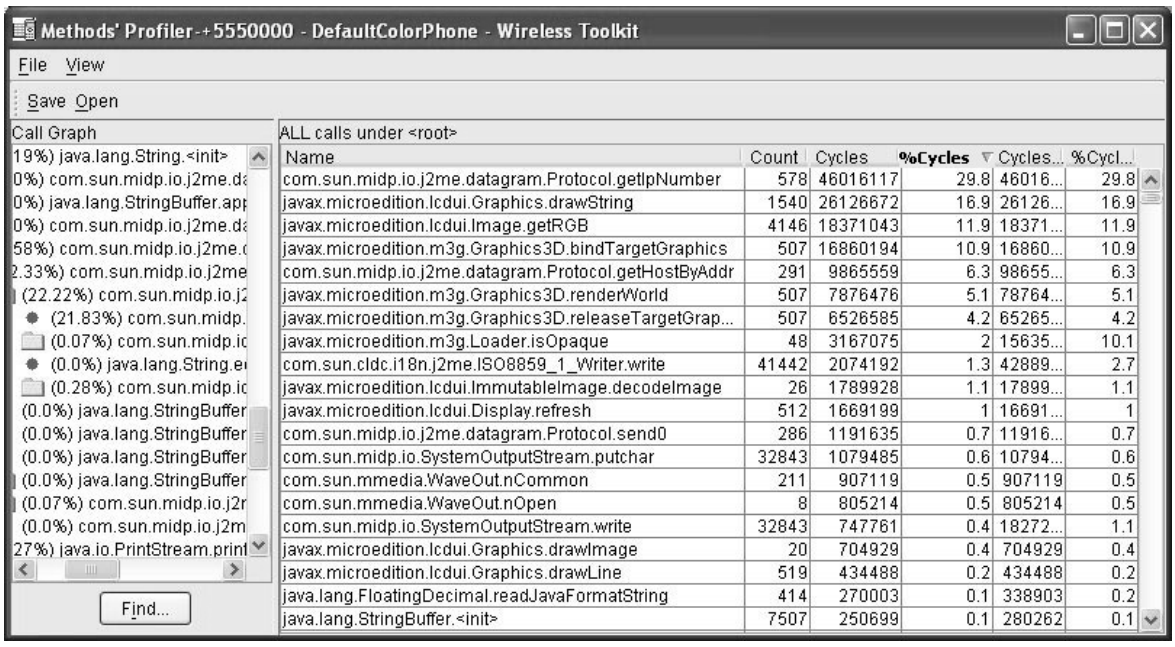


Figure 2.12: A Client Sends a Hello Message and the Broadcast Response from the Server

2.6 J2ME Client Profiling

The Profiler tool in the WTK can be used to find which parts of an application are slow so the code can be optimized. The profiler records the time used by each method at run time. After the application exits, a Profiler screen will appear as in Figure 2.13.



Name	Count	Cycles	%Cycles	Cycles...	%Cycl...
com.sun.midp.io.j2me.datagram.Protocol.getIpNumber	578	46016117	29.8	46016...	29.8
javax.microedition.lcdui.Graphics.drawString	1540	26126672	16.9	26126...	16.9
javax.microedition.lcdui.Image.getRGB	4146	18371043	11.9	18371...	11.9
javax.microedition.m3g.Graphics3D.bindTargetGraphics	507	16860194	10.9	16860...	10.9
com.sun.midp.io.j2me.datagram.Protocol.getHostByAddr	291	9865559	6.3	98655...	6.3
javax.microedition.m3g.Graphics3D.renderWorld	507	7876476	5.1	78764...	5.1
javax.microedition.m3g.Graphics3D.releaseTargetGrap...	507	6526585	4.2	65265...	4.2
javax.microedition.m3g.Loader.isOpaque	48	3167075	2	15635...	10.1
com.sun.cldc.i18n.j2me.ISO8859_1_Writer.write	41442	2074192	1.3	42889...	2.7
javax.microedition.lcdui.ImmutableImage.decodeImage	26	1789928	1.1	17899...	1.1
javax.microedition.lcdui.Display.refresh	512	1669199	1	16691...	1
com.sun.midp.io.j2me.datagram.Protocol.send0	286	1191635	0.7	11916...	0.7
com.sun.midp.io.SystemOutputStream.putchar	32843	1079485	0.6	10794...	0.6
com.sun.mmedia.WaveOut.nCommon	211	907119	0.5	907119	0.5
com.sun.mmedia.WaveOut.nOpen	8	805214	0.5	805214	0.5
com.sun.midp.io.SystemOutputStream.write	32843	747761	0.4	18272...	1.1
javax.microedition.lcdui.Graphics.drawImage	20	704929	0.4	704929	0.4
javax.microedition.lcdui.Graphics.drawLine	519	434488	0.2	434488	0.2
java.lang.FloatingDecimal.readJavaFormatString	414	270003	0.1	338903	0.2
java.lang.StringBuffer.<init>	7507	250699	0.1	280262	0.1

Figure 2.13: Profiler Screen Example

The important column in the output number is **%Cycles** (the third column from the right, the percentage of total execution time that is spent in the method). In Figure 2.13, *getIpNumber()*, which gets the IP address of the datagram, takes 29.8% of the processor time, and so should be optimized, if possible.

PenguinM3G was optimized in several ways due to Profiler information:

- *getIpNumber()* has the highest process time because it creates a datagram objects every time the program uses *Sender.connect()* or *ClientHandler.run()*. In the original code, a new datagram is created when there is a message need to send to the server. This overhead can be fixed by reusing one datagram object instead of creating many new ones. The example of fixing result is shown in Table 2.1.

	Counts	Cycles	%Cycles
Original *getIpNumber()*	68	8903476	9.5
Modified *getIpNumber()*	2	58329	0.0

Table 2.1: getIpNumber Comparison

The original *getIpNumber()* creates a new datagram every time, while the modified version reuses a datagram. Table 2.1 shows that reusing the datagram can reduce the work load greatly, by 99.3% in this case.

- *drawString()* has a high process time because it displays the player information on the screen. Since it is displayed every frame, it is one of the most CPU intensive operations. The overhead can be reduced if StringBuffer is used instead of String.

2.7 Z-Score

In order to compare the response times between tests, the z-test is used to determine if the difference between the sample mean (when our techniques are enabled) and the population mean (when no techniques are enabled) is large enough to be statistically significant [8]. The z-test requires two means (X for the enabled techniques, μ for no techniques), two standard deviations (SD_x for the enabled techniques, SD_μ for no techniques) and the total number of tests (n_x for the enabled techniques, n_μ for no techniques).

The means (X) are calculated like so (the same formulae also applies to μ):

$$X = \Sigma x_n / n$$

x is the response time result of enabled techniques. n is number of test.

The standard deviations (SD_x) are calculated like so (the same formulae also applies to SD_μ):

$$SD_x = \sqrt{((\sum(x_n^2) / n) - ((\sum x_n)^2 / n^2))}$$

After obtaining the SD_x, σ_x^2, which is the variance of the mean (X), is calculated (also applies to σ_μ^2):

$$\sigma_x^2 = (SD_{x1}^2 * n_{x1}) + (SD_{x2}^2 * n_{x2}) + (SD_{x3}^2 * n_{x3}) + \ldots + (SD_{xn}^2 * n_{xn}) / n_{x1} + n_{x2} + n_{x3} + \ldots + n_{xn}$$

SD_{x1} is the SD of the first test on the response time result of enabled techniques. n_{x1} is the number of the first test, and so on.

The standard error (SE) from the σ is:

$$SE = \sqrt{\sigma_x^2 / n_x + \sigma_\mu^2 / n_\mu}$$

The z-score for the z-test is:

$$\mathbf{Z = (X - \mu) / SE}$$

X is the sample mean (when the techniques are enabled), μ is the population mean (when no techniques are used) and SE is the standard error.

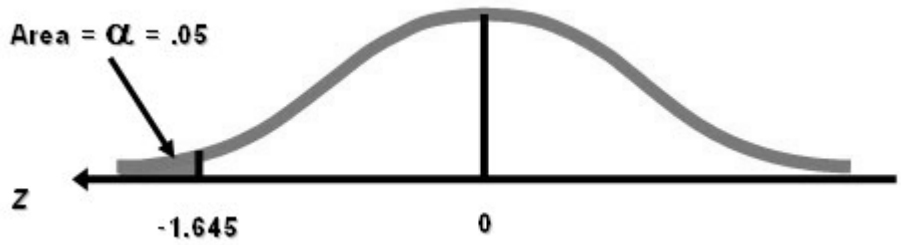


Figure 2.14: Level of Significant of Left-Tailed Test at 0.05 or 95%

The null hypothesis (H0) will be:

no techniques response time mean = enabled techniques response time mean

The alternative hypothesis (H1) will be:

no techniques response time mean > enable technique response time mean

Figure 2.14 shows a one-tailed test for whether the techniques make the response time significantly faster. At a 0.05 level of significant (α), if the z-score is less than -1.645, then the alternative hypothesis can be accepted.

Our results will be judged significant only if the z-score is less than -1.645, or more than 1.645 in the case of the enabled techniques comparison or the same technique with different parameters.

2.8 Summary

This chapter introduced PHBDR and smoothing, details of the test game prototype, and TCP and UDP client/server communication. WTK client profiling can be employed to measure and optimize J2ME applications, and z-scores will be utilized to judge the significance of our results.

CHAPTER 3

GAME DESCRIPTION

The test game used in this thesis is “PenguinM3G”, a 3D FPS client/server based game. The server employs J2SE and the clients use J2ME on WTK Emulator. The game involves “a local player” (who plays the game on their own simulated device) interacting with “remote players” (who play the game on other simulated devices). Due to the rapid play inherits in FPSs, the variation in the network characteristics (random delay or dropped packets), and a visual rich 3D GUI, the game must include optimization techniques to improve its response times. The test game is developed in order to test these response time issues, not as a marketable game, so some features such as a high scores table and help screens are left out.

The local player is a penguin that can shoot fire from its mouth at remote avatars (the penguins representing remote players on the local player’s device). The main objective of the game is to score as much as possible while being challenged by other players as shown in Figure 3.1. All the clients connect to a central server via a UDP based protocol, so they can exchange information (packets).

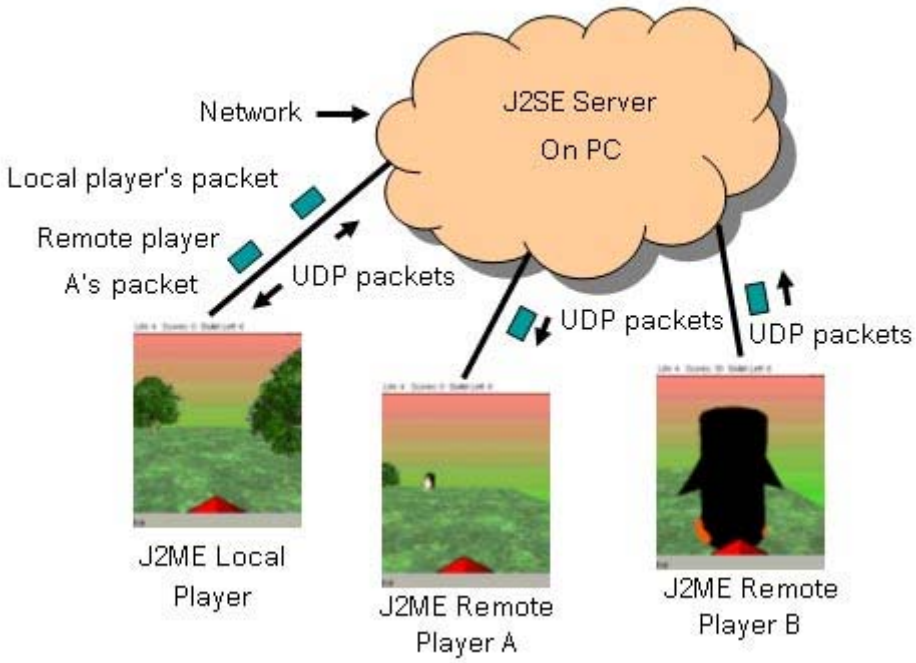


Figure 3.1: The Game Architecture

The following sections will describe more on the game architecture e.g. about the game elements, how the player controls the game, and about the game rules.

3.1 Game Elements and Controls

Figure 3.2 illustrates many elements of the game. There is a status bar at the top of the screen which displays the lives remaining, the score, and the bullets the player has left. On the bottom of the screen, there is a player beak which shoots at the other players. On the field, the player can move onto life spots which increase the player's score and regains one life as well. Also, there are bullets areas where the player can renew his bullets. The player sees other players as penguin-like remote avatars.

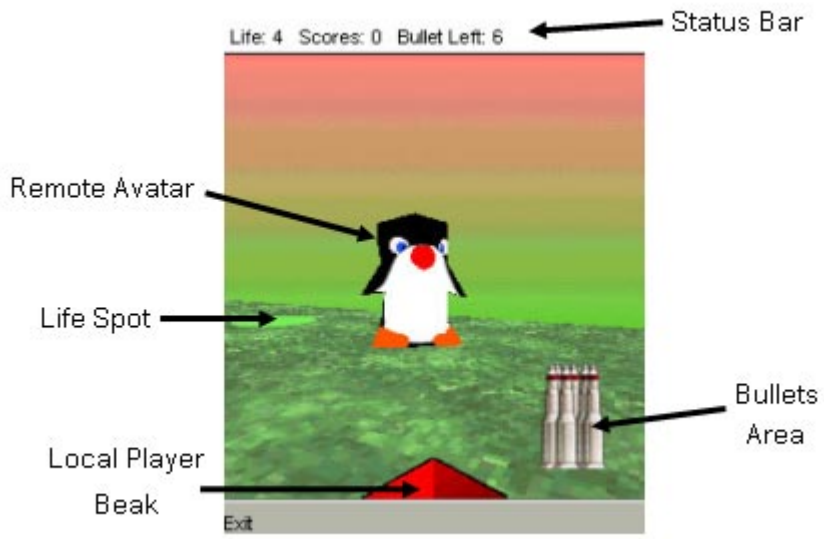


Figure 3.2: Game Elements

The player can move forward, backward, rotate left, and rotate right. The player can shoot bullets from his beak by pressing the Select key or the Enter key.

3.2 Game Rules

The primary aim of the game rules and elements (e.g. the life spots and the bullets areas) are to make the game's network packets behavior more complex, so that general and packets-based optimization techniques become useful.

- In order to play the game, the local player must first connect to the game server.
- After successfully connecting, the player will receive remote avatar information, ten lives, zero score points, and six bullets.
- The player can quit the game at any time.
- There is no time limit or 'winning' end to the game.

- When the local player shoots a remote avatar, the local player's bullets and the remote player's life are reduced by one, and the local player gets twenty points.
- If a player's life count is reduced to zero, the game is over for that player, and he will be automatically disconnected from the server.
- Moving onto a life spot gives the local player thirty extra points and one more life (up to a maximum of ten). Then the life spot changes from active (colored) to inactive (white). There are four life spots in the game.
- Moving onto a bullets area gives the local player a maximum of six bullets, and the bullets area changes from available to unavailable (the bullets area vanishes from the player's view). There are two bullets areas in the game.
- Inactive life spots and unavailable bullets areas will be reactivated after thirty seconds out of the game.
- Remote players can shoot at the local player at any time.
- Statistics related to response times are measured and collected by each client, but this activity is not shown in the game.

3.3 Summary

This chapter explained the basic PenguinM3G architecture, game elements such as the status bar and life spots, and game rules. These make the game's network behavior more complex.

CHAPTER 4

THE GAME CLIENT

The design and implementation of the game will be explained over two chapters. This chapter is concerned with the client side, and Chapter 5 with the server.

A game client is divided into two main parts: the game part and the network part. The prototype of the game part was developed using parts of the GunnerM3G program, described in section 2.3. The new game elements include trees, life spots, and bullets areas, as well as collision detection between objects and with the game boundary. The network part employs the UDP protocol to connect to the server, and to handle remote avatar events. An overview of the game architecture is shown in Figure 4.1.

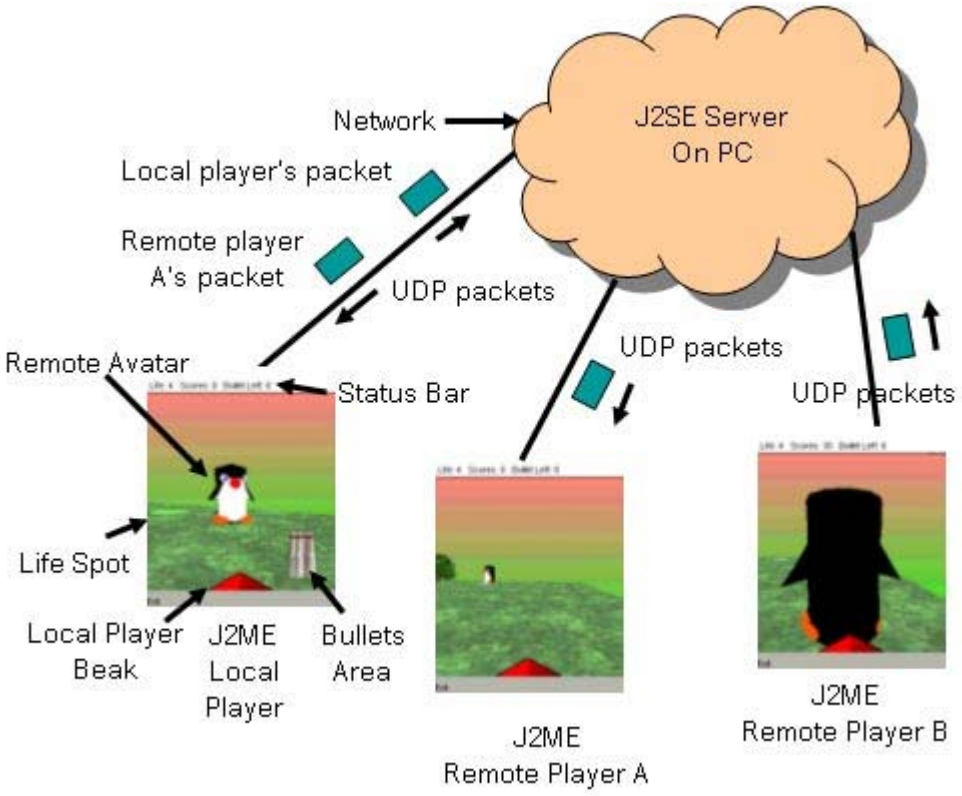


Figure 4.1: The Game Architecture Again

4.1 Overview of Game Client Classes

Class diagrams for the game client are shown in Figure 4.2. The classes are grouped into three categories which identify their relationship to the GunnerM3G example. The shaded boxes are the classes reused without changes, the striped boxes are changed classes, with new or modified methods, and the unshaded boxes are new classes.

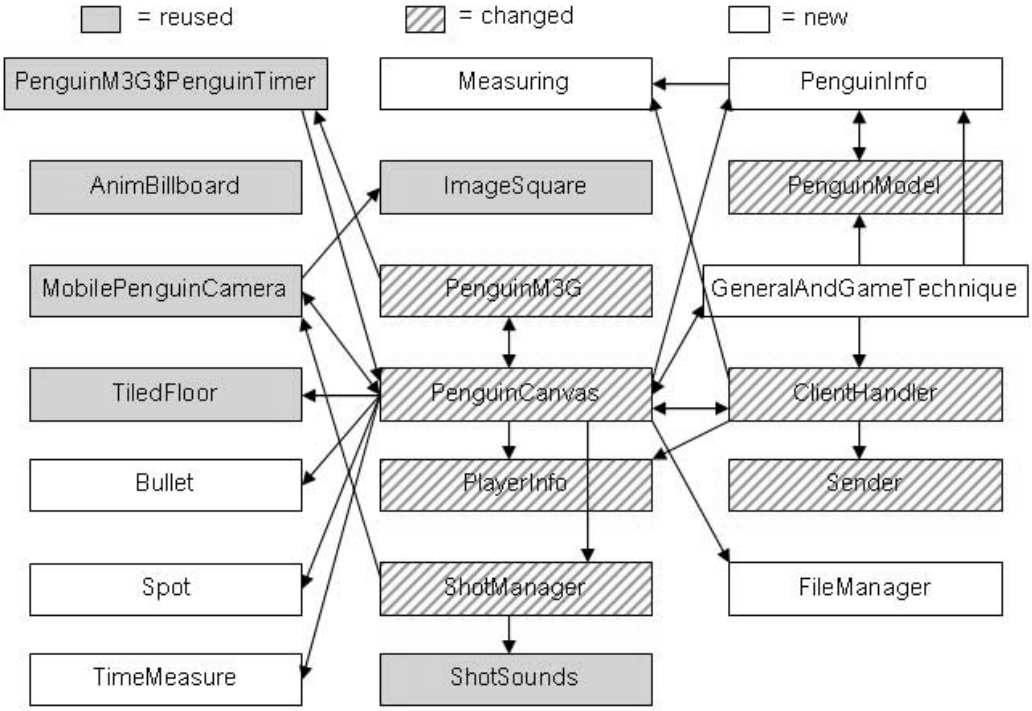


Figure 4.2: Game Client Class Diagram

PenguinM3G is the top-level class of the application. *PenguinM3G* uses the *PenguinTimer* class to periodically update the game canvas. The canvas is updated by calling the *update()* method in the *PenguinCanvas* class. The additional code in *PenguinM3G* is for a user interface that allows the player to enable different techniques, as shown in Figure 4.3.

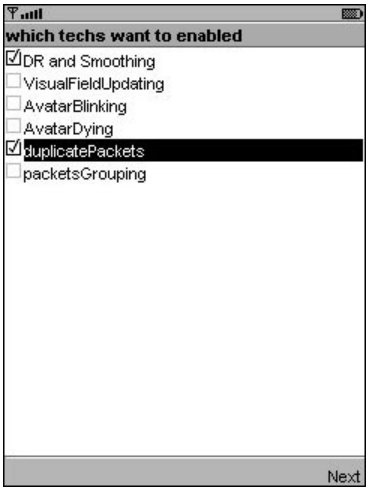


Figure 4.3: Techniques Check Lists

The *PenguinCanvas* class handles the 3D scene, using the *TiledFloor* class to create the floor. The *PenguinCanvas* class also creates game elements like trees, life spots, bullets areas, the status bar, and creates the penguins representing the remote avatars using the *PenguinModel* class.

The *PlayerInfo* class holds the information for the local player lives, scores, and bullets left.

The *PenguinInfo* class contains the information on the remote avatar id, position, angle, lives, and time statistics (e.g. the one-way response time between the local player and this remote player). It also stores the past moves and rotations of the remote avatar, used DR.

The *Spot* class and the *Bullet* class contain information on the available/unavailable state of the life spots and bullets areas.

The *MobilePenguinCamera* class deals with the game camera and attaches the *ImageSquare* class to make the penguin beak and the shot flash. This class also checks for collision detection with the life spots, bullets areas, and the boundary of the game field. More details on the original *MobileGunCamera* are available in section 2.3.

The *ShotManager* class makes a shot flash appear in response to the shot key, and triggers the *ShotSounds* class to play the shot noise.

The *AnimBillboard* class controls the animated fireball when the penguin is shot, and the explosion sound is controlled by the *ShotSounds* class.

The *GeneralAndGameTechnique* class processes general techniques (e.g. DR and smoothing) and game-specific techniques.

The *ClientHandler* class manages both incoming and outgoing network packets. Incoming packets are parsed and sent to the *PenguinCanvas* class to update the game state. Outgoing packets are sent to the *Sender* class for output as UDP packets. The *ClientHandler* class also handles the network techniques (e.g. packets grouping).

The *TimeMeasure* class measures the *update()* method of the *PenguinCanvas*.

The *Measuring* class calculates the means, standard deviations, and percentages necessary for measuring the techniques effectiveness.

The *FileManager* class automatically saves all the measuring results into a text file when the player quits the game.

The following classes which are the client's game elements will be described in more detail in section 4.2:

- The *Bullet* class is used to create the bullets area (section 4.2.1)
- The *Spot* class is used to create the life spots (section 4.2.2)
- The part of *PenguinCanvas* which creates the trees and status bar (sections 4.2.1 and 4.2.3 respectively)

- The part of *MobilePenguinCamera* class which handles collision detection (section 4.2.4)

4.2 Game Elements

The elements added to those originally in GunnerM3G are: trees, bullets areas, life spots, and a player status bar, as shown in Figure 4.4.

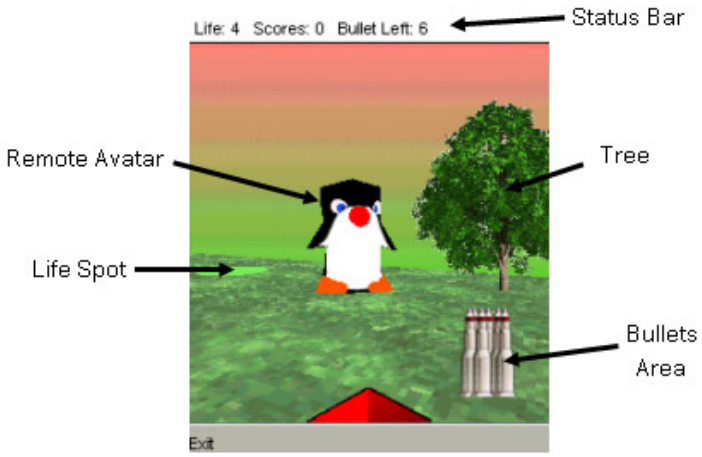


Figure 4.4: Game Elements

4.2.1 Trees and Bullets Area

The trees and the bullets area are not implemented with 3D models, instead, they are made from 2D images drawn on flat screens (called billboards) which stay oriented towards the camera. There are two ways to create a billboard: one is by using the Sprite3D class in the M3G library [7], the other is to implement a customized billboard class. For making the trees and the bullets area in this thesis, the Sprite3D class was chosen because Sprite3D automatically faces the camera while a customized billboard needs extra code to make it always face the camera. A bullets area is shown in Figure 4.4, and a tree is shown in Figure 4.5.

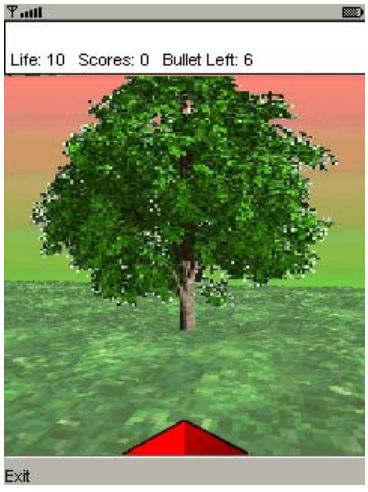


Figure 4.5: Example tree using Sprite3D

4.2.2 Life Spots

The spots on the field are implemented with a customized billboard which employs a camera alignment and a partially transparent flat mesh (a mesh is a 3D geometry of (x,y,z) points with associated appearance characteristics). Sprite3D can not be used since it always rotates about the y axis. The life spots need to lie on the floor, which is the xz plane (see Figure 4.6).

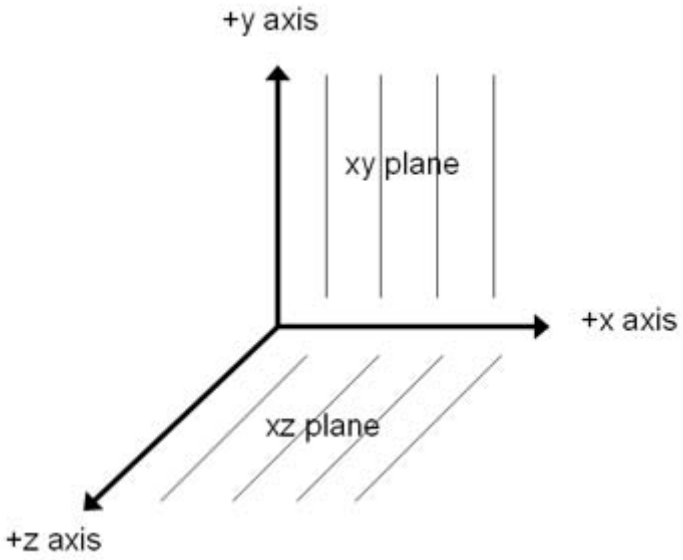


Figure 4.6: The 3D Axis

4.2.3 Status Bar

Player information (Lives, Score and Bullets Left) is shown in a status bar by using clipping. This uses repaint(x, y, width, height) which paints only a certain area of the screen, reducing redrawing costs for every frame. It also improves the *update()* method because less string objects are created in every frame. The status bar can be seen in Figure 4.4 at the top of the screen.

4.2.4 Collision Detection

There are two types of collision detection. The first detects the life spots and bullets areas on the field, and the second detects the boundary field of the game world. The trees are not involved in collision detection, so players can hide behind them and ambush other players.

After putting the life spots and bullets areas on the field, the next step is to detect if the player moves into them. There are two ways to detect something on the floor. The first way is by using a picking ray with M3G's *RayIntersection* class by rotating the camera down 90 degrees, firing a picking ray, then rotating the camera back to normal. The other way, which is easier, is to store the position of each life spots and bullets areas, and check them while the player is moving.

The other form of collision detection is to prevent the player traveling beyond the boundary of the game world. The position detection method used for detects life spots and bullets areas can also be employed here since the size of the floor is known. Collision detection is done before the player makes a move by testing the move to see if it goes out of bounds. If not, then the move is carried out.

4.3 Network Game Client

The local client sends packets to the server which forwards them onto the remote clients to update their game states. Section 4.3.1 describes the types of packets being sent. Section 4.3.2 explains how the packets are grouped into two categories: single packet per keypress and multiple packets per keypress, so that different types of optimization techniques can be employed. Section 4.3.3 discusses how remote avatars are integrated into the game client.

4.3.1 Types of Packets

A packet has a header which matches the role of the packet, an ID which identifies who sent the packet, and assorted other information. An example of the packet structure is shown in Figure 4.7. All the packets include a timestamp in order to calculate how long the packet takes to travel from one client to another. In the rest of this section, the timestamp field will be emitted from the figures to emphasize the more important fields.

Figure 4.7: General Packet Format

The game packets will be explained with the following categories: joining/quitting, moving/rotating, shooting, state change (when the player moves onto the life spots/bullets areas), and a reply code.

Joining/Quitting

Three kinds of packets are sent in order to join or quit the game: JOIN, JOINED and QUIT packets.

A JOIN packet is sent from the local player to the server to indicate that he/she wants to join the game. The position and the angle of the local player are sent (as shown in Figure 4.8) to allow the other players to create a penguin representing the player.

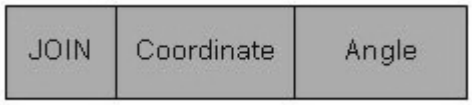


Figure 4.8: A JOIN packet

After the server has processed a JOIN packet, it broadcasts a JOINED packet to all the clients. The local player gets the status of current life spots and bullets areas state, while the other players get the local player's ID, position and angle, which they use to create a remote avatar for this player. The structure of a JOINED packet is shown in Figure 4.9.

Figure 4.9: A JOINED packet

The fields of a JOINED packet are:

- *ID*, a unique integer representing the player who has just joined the game, given by server.
- *Coordinate* represents the new player's (x, z) position. E.g. (2.31, -1.45)
- *Angle* represents the new player's angle in degrees relative to -z axis. E.g. 57.0
- *Spots state* are four numbers representing the game's life spots: 1 is active, 0 is inactive. E.g. 1110 means that the fourth life spot is inactive.
- *Bullets areas state* are two numbers representing the game's bullets areas: 1 is available, 0 is unavailable. E.g. 01 means that the first bullets area is unavailable.

An example join event is shown in Figure 4.10. Client A joins the game by sending a JOIN packet to the server. Then, the server broadcasts a JOINED packet to all the clients to notify them of client A's joining.

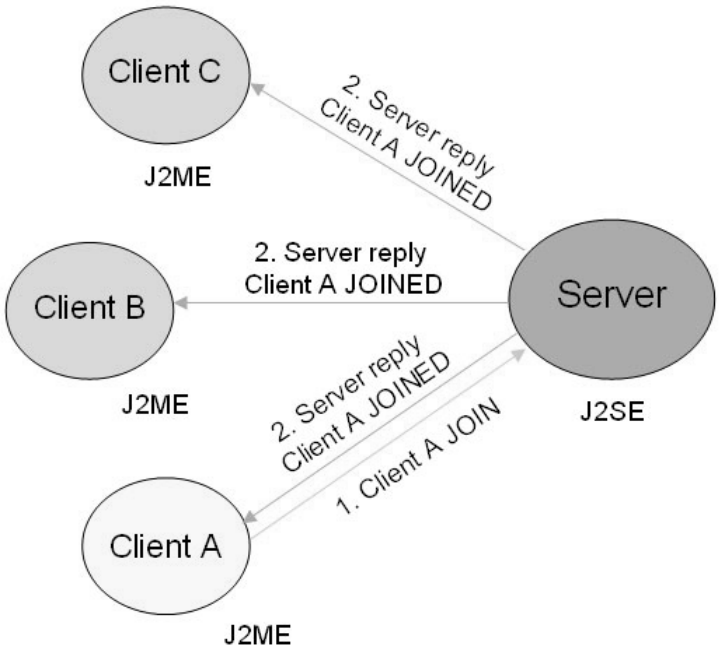


Figure 4.10: Client A Joins the Game

When the player wants to quit the game, or he loses all his lives, a QUIT packet is sent from the local player to the server. The server accepts the quit state and broadcasts the QUIT packet to all the clients to inform them of the departure of the player. They can then remove the remote avatar representing this player. The structure of a QUIT packet is shown in Figure 4.11.

Figure 4.11: A QUIT packet

Moving/Rotating

A remote avatar can move and rotate, represented by three kinds of packets: MOVE, ROTATE, and KEY_RELEASE.

A MOVE packet is sent from the local player to the server when the player moves forward or backward. The server then broadcasts the MOVE packet to all the other players. The structure of a MOVE packet is shown in Figure 4.12.

Figure 4.12: A MOVE packet

This packet includes an angle field for use in the smoothing, and a move type field.

- *Move type* is a number representing a player's possible moves: Forward = 1, Turn right = 2, Turn left = 3, Backward = 4.

A ROTATE packet is sent from the local player to the server when the player rotates left or right. The server then broadcasts it to all the other players. The structure of a ROTATE packet is shown in Figure 4.13.

Figure 4.13: A ROTATE packet

The move type in a ROTATE packet has only two possible values: turn right and turn left.

A KEY_RELEASE packet is sent when a local player releases the moving or rotation key to indicate the finish of the current move. This packet is used by the dead reckoning technique. This packet is broadcast by the server to all the other remote players. The structure of KEY_RELEASE is shown in Figure 4.14.

Figure 4.14: A KEY_RELEASE packet

Shooting

There are two kinds of shooting-related packets: SHOOT and SHOOT_RESPONSE. A SHOOT packet is sent from the local player to the server when the player thinks that a bullet has hit the targeted remote player. The server sends the SHOOT packet onto the remote player. The structure of a SHOOT packet is shown in Figure 4.15.

Figure 4.15: A SHOOT packet

A SHOOT_RESPONSE packet is sent by the remote player to the server in reply to a SHOOT packet, to indicate if the remote player was hit by the bullet or not. The server sends this packet onto the local player. The structure of the packet is shown in Figure 4.16.

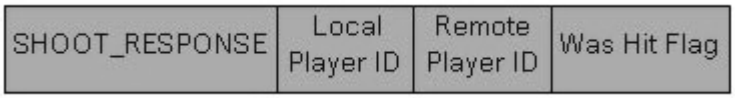


Figure 4.16: A SHOOT_RESPONSE packet

The SHOOT_RESPONSE‘s *was hit flag* represents whether the remote player was shot by the local player or not. E.g. was shot = 1, was not shot = 0

An example of shooting is shown in Figure 4.17. Client A shoots a bullet at client B, and sends a SHOOT packet to client B. Client B checks if it’s hit or not and replies with a SHOOT_ RESPONSE packet.

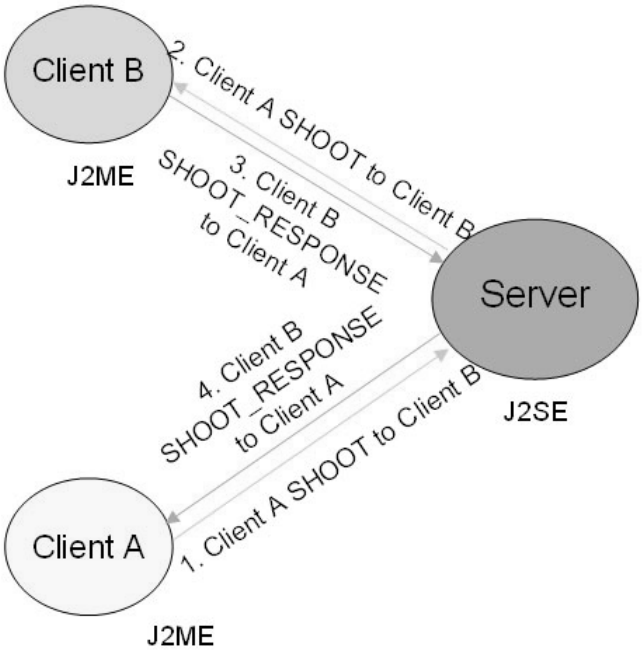


Figure 4.17: Client A Shoots a Bullet

State change

The game state is modified by changing the life spots and bullets areas with SPOTS_STATE and BULLETS_AREA_STATE packets.

A SPOTS_STATE packet is sent to the server when the player moves onto an active life spot. The server broadcasts it to all the remote players to change the global state of that life spot.

Another way of using this packet is when the server reactivates the life spots after 30 seconds. This packet originates at the server, and is broadcast to all the clients. The structure of SPOTS_STATE is shown in Figure 4.18.

Figure 4.18: A SPOTS_STATE packet

The ID field in a SPOTS_STATE packet is used to differentiate between two uses of the packet. ID 10001 indicates that this packet originates from the server, and its use for updating the life spots is shown in Figure 4.19. Client A moves onto the life spot and sends a SPOT_STATE packet to the server. The server then sends SPOT_STATE packets to all the clients to update their life spot states. Also, when the server wants to reactive a life spot, it sends a SPOT_STATE packet to all the clients.

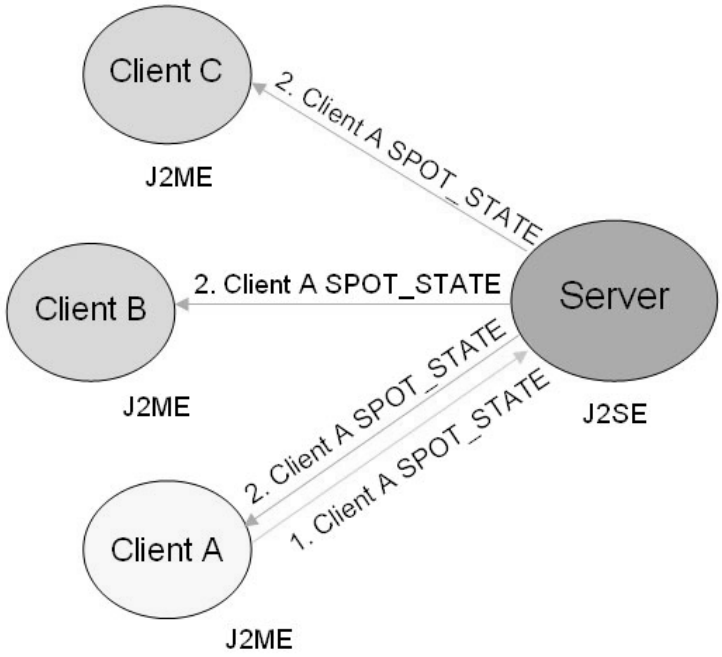


Figure 4.19: Updating the Life Spot

A BULLETS_AREA_STATE packet is sent from the local player to the server when the player moves onto an available bullets area. The server then broadcasts the packet to all the remote players. This packet is also used by the server to make a deactivated bullets area available again after 30 seconds. The BULLETS_AREA_STATE structure is shown in Figure 4.20.

Figure 4.20: A BULLETS_AREA_STATE packet

The ID field in BULLETS_AREA_STATE packet is used the same way as in SPOTS_STATE packet. An example of updating the bullets area is shown in Figure 4.21. When client A moves onto the bullets area, he sends a BULLETS_AREA_STATE packet to the server. The server then sends the BULLETS_AREA_STATE to all the clients in order to update their bullets areas. This is also done by the server when it reactivates a bullets area.

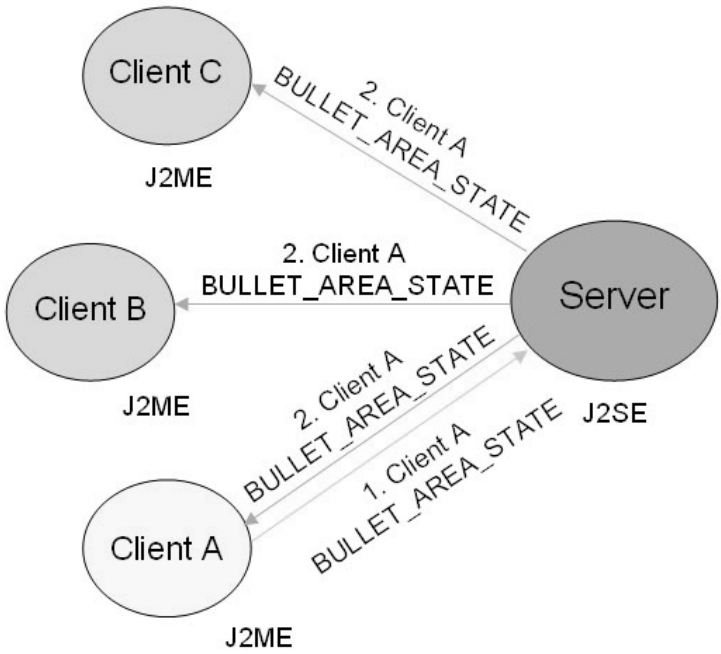


Figure 4.21: Updating the Bullets Area

Reply Code

A REPLY_CODE packet is used for measuring the response times of MOVE, ROTATE, BULLETS_AREA_STATE, and SPOTS_STATE packets. A REPLY_CODE packet is sent from a remote client back to the local client in reply to the one of these packets arriving at the remote client. The REPLY_CODE structure is shown in Figure 4.22.

REPLY_CODE	Source Player ID	Des Player ID	Source Timestamp	*Action Header

Figure 4.22: A REPLY_CODE packet

The action header field is the name of the packet this REPLY_CODE packet is measuring: MOVE, ROTATE, BULLETS_AREA_STATE, or SPOTS_STATE.

An example of how REPLY_CODE is used is shown in Figure 4.23. Client A sends a MOVE packet to client B. When client B gets the packet, it sends a REPLY_CODE packet back to client A along with client A's ID taken from the MOVE packet timestamp. When client A gets the REPLY_CODE, it can calculate the one-way response time of the MOVE packet by subtracting the REPLY_CODE's timestamp from the source timestamp, and dividing by two. The result becomes the one-way response time for the MOVE packet sent from client A to client B.

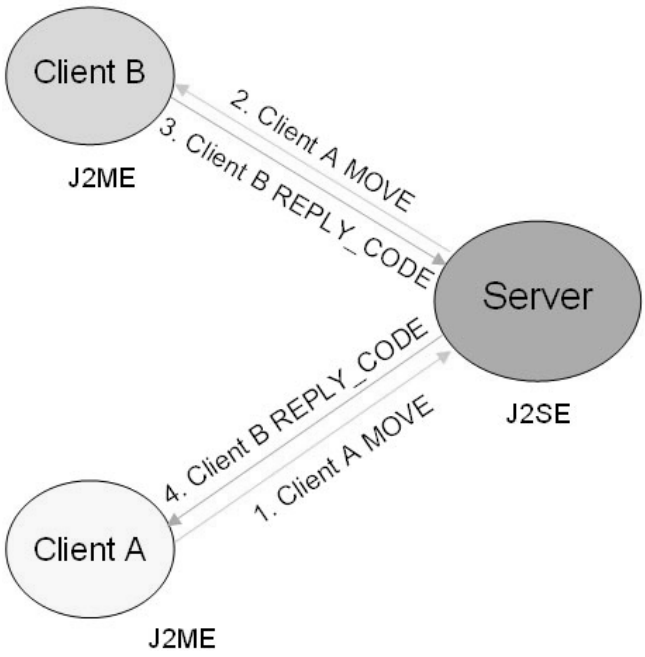


Figure 4.23: REPLY_CODE Example

4.3.2 Packets and Keypresses

When measuring response time, it's useful to distinguish between packets in terms of how many are generated when the user presses a key. We utilize two categories: "single packet per keypress" and "multiple packets per keypress".

Single packet per keypress is the situation when holding a key down generates only one packet. The packets in this category are SPOTS_STATE, BULLETS_AREA_STATE, JOIN, JOINED, QUIT, SHOOT, and SHOOT_RESPONSE.

Multiple packets per keypress occurs when holding a key down generates multiple packets. The packets in this type are MOVE and ROTATE, which are very common user behaviors, and so many such packets are generated during game play.

As we will see, different response time techniques are best suited to these different packets categories. For example, a technique very suitable for single packet per keypress is packet duplication. However, it is not much use to multiple packets per keypress since multiple packets are already being generated. In fact, duplicate packets will cause too much traffic in its case. Another example is dead reckoning which needs a lot of data to make its predictions, and so is more suitable for use with multiple packets per keypress.

4.3.3 Remote Avatar

The remote avatar represents the remote player, so duplicates all the moving and rotation actions of that player.

The move method (called *updatePosition()*) in *PenguinModel*, uses an existing *setTranslation()* method along with the position coordinate of the remote avatar to move the remote avatar to the specific location.

For the rotation method, the difference between the rotate avatar's new angle and its old one is needed. Once again, a pre-existing method, *preRotate()* can be utilized, which rotates the penguin around the y-axis (*preRotate()* means multiplies the current orientation component from the left by the given orientation).

4.4 Summary

This chapter gave an overview of the game's client side classes, and how to implement game elements such as trees, life spots, the status bar, and collision detection. The chapter also explained the network packets format, and described each packet type, such as JOIN, MOVE, and SHOOT_RESPONSE. The implementation of remote avatar moving/rotating was discussed.

CHAPTER 5

THE GAME SERVER

The game server is based on the "Threaded TCP Clients and Server" *ChatServer* (see section 2.2 [7]), modified to use the UDP protocol. The server uses threads to listen to multiple clients, and stores a shared object to maintain the clients information. The server does not have a GUI since its duty is to store clients details, e.g. client IDs, so it can tell a new client which clients are connected and forward messages among them.

The server's main task is to forward packets from one player/client to the other clients. It also simulates an unreliable network connection e.g. packet reliability of 90% or 75%. The packets can be delayed between 30 ms - 2000 ms, and be lost at the rate of 10% or 25%. The server supports this feature so that we can test how techniques such as dead reckoning help increase communication reliability.

The server also reactivates the game's life spots and bullets areas 30 seconds after they have been deactivated by a client moving onto them. The server notifies the other clients of a change by sending packets to them.

5.1 Overview of Game Server Class

Class diagrams for the game server are shown in Figure 5.1. The class boxes are colored differently to highlight the new classes and the modified classes.

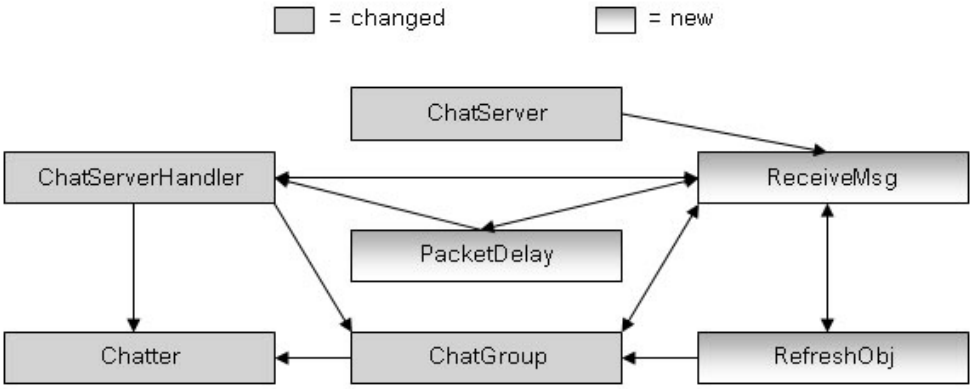


Figure 5.1: Game Server Class Diagrams

The *ChatServer* class is the top level console application. The program is started with one command line argument to specify the required network behavior: stable mode where there is no unreliability, or unstable mode where packets can be dropped or delayed. The *ReceiveMsg* class creates a thread to handle the clients' connection, while *ChatServer* waits for user input. If the user inputs the letter 'q' and presses enter, the server will shut down.

In an unreliable network simulation, *ReceiveMsg* randomly chooses between passing a packet normally, delaying it, or dropping it. *ReceiveMsg* also deals with the reactivation of the life spots and the bullets areas by sending a SPOTS_STATE or a BULLETS_AREA_STATE packet to all the clients.

ChatServerHandler extracts information from the client packets. If it sees a "JOIN" packet, *ChatServerHandler* will store the client address, port, and the client avatar's initial position in the *Chatter* class object via the *ChatGroup* class. *ChatGroup* maintains the *Chatter* objects, handles the adding/removal of client information, and creates UDP datagrams to send packets to the clients.

The *PacketDelay* class is used to delay packets for random specific durations.

The *RefreshObj* class creates a thread to handle the reactivation of the life spots and the bullets areas.

5.2 The Unreliable Network Simulation

The four main stages carried out by the server to simulate network unreliability are shown in the shaded boxes in Figure 5.2.

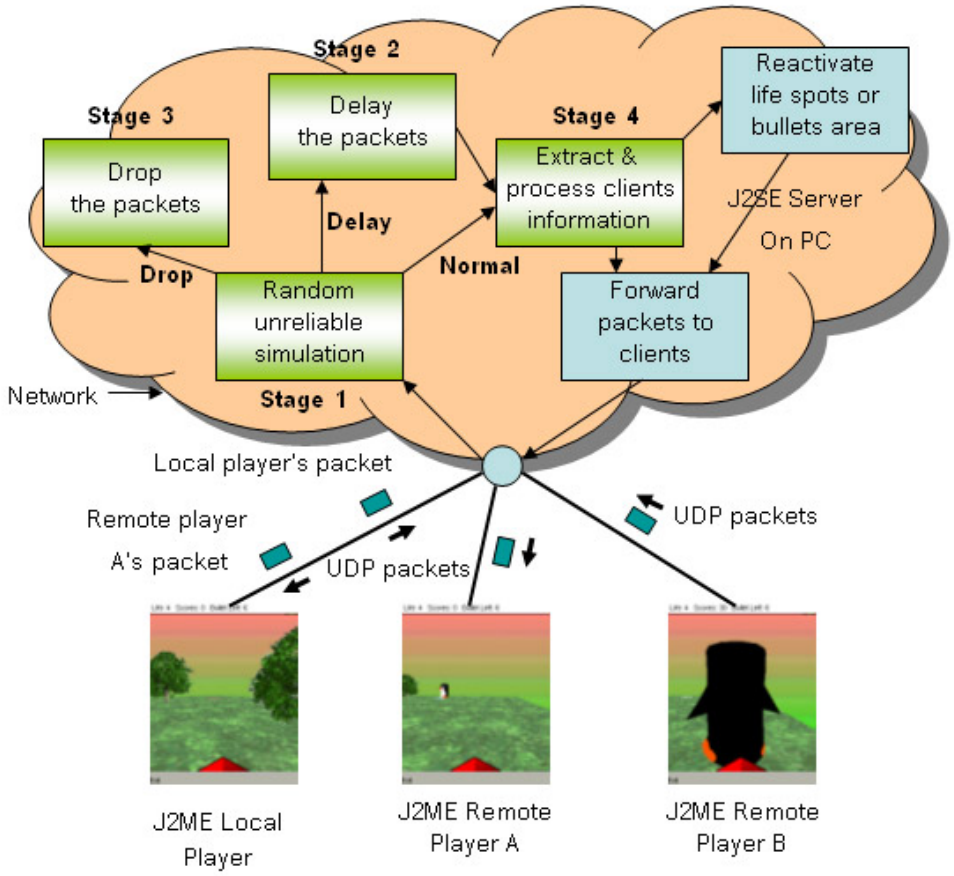


Figure 5.2: The Stages of a Network Unreliability Simulation

In stage 1, upon receiving packets from the clients, the server random chooses whether to let the packet pass normally, be delayed in stage 2, or be dropped in stage 3. If the server delays the packet, the packet will have to wait for between 30 ms and 2000 ms before being processed in stage 4.

There are two levels of reliability: 90% and 75% reliability, which are set in the *ReceiveMsg* class. 90% reliability means that the probability that the packet can pass normally is 90%, with 10% chance of it being delayed or dropped. 75% reliability means that a packet has a 25% chance of being lost or delayed. Two reliability levels allow us to test the effectiveness of our techniques under different circumstances.

In the *ReceiveMsg* class, *randompercent()* will be called to generate a random probability to decide whether the server will process a packet normally, delay it, or drop it. Here is the pseudo code for *randompercent()*:

```
// in ReceiveMsg class
private int randompercent()
{
    int choice = random in range 0 .. 100 ;

    if (choice < RELIABILITY (75 or 90))
      return NORMAL;

    else
      randomly return DROP or DELAY;
}
```

When a packet is processed normally, it is sent to the *ChatServerHandler* class to have its information extracted. If a packet is delayed, the delay time is randomly generated between 30 ms to 2000 ms (2 seconds), which is long enough to affect the game play.

The threaded *PacketDelay* class makes a packet waits for the specific delay time by using *Thread.sleep()*.

5.3 Reactivate Life Spots and Bullets Area

The two steps involved in reactivating the life spots and the bullets areas are shown in the shaded boxes of Figure 5.3.

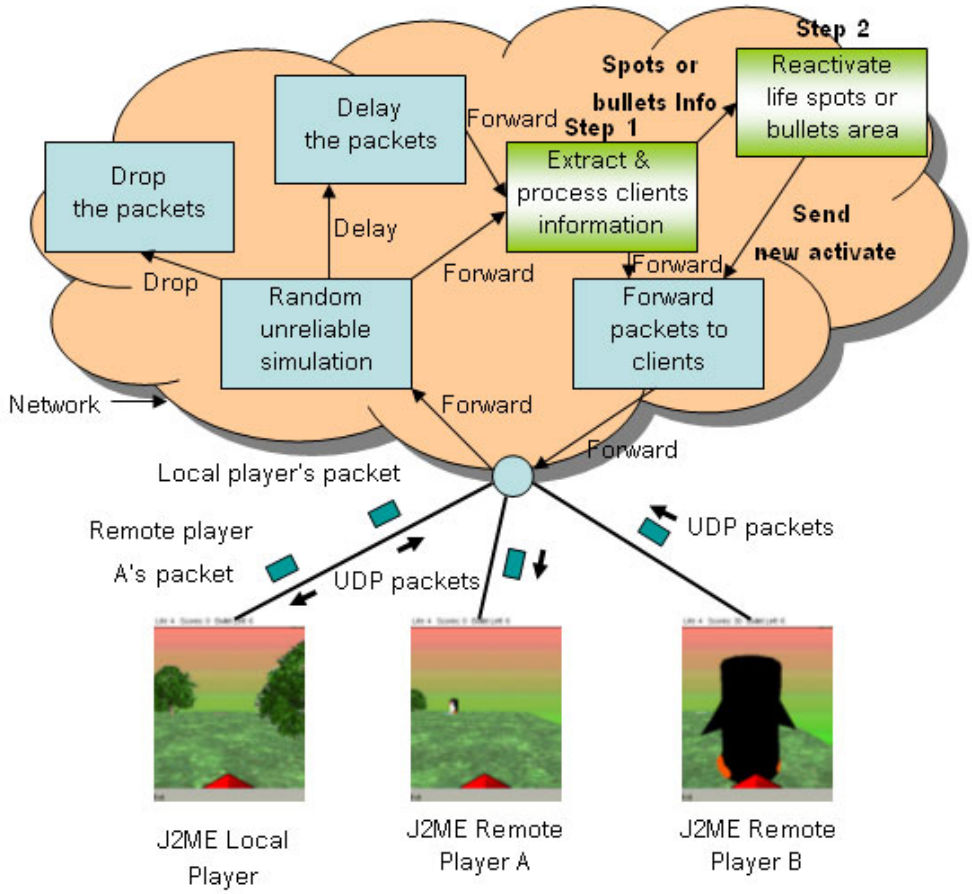


Figure 5.3: The Server Reactivates Life Spots or Bullets Areas

In step 1, the server usually forwards a packet to the client after receiving the packet. However, if the server receives a life spot or bullets area packet, then it processes to step 2 by checking which life spot or bullets area the client is located on by comparing the current modified state with the previous state. The server reactivates that life spot or bullets area after 30 seconds.

The *ChatServerHandler* extracts and processes the "SPOTS_STATE" and "BULLETS_AREA _STATE" packets (see Appendix S3). *ChatServerHandler* calls refresh() to send the life spot, or bullets area, packet to the *RefreshObj* thread class. *RefreshObj* reactivates the life spot or bullets area by sending a reactivation packet to all the clients after waiting 30 seconds.

5.4 Summary

This chapter gave an overview of the game's server side class. The main tasks of the game server are to simulate an unreliable network, and reactivate life spots and bullet area during the game. The server can simulate 90% and 75% packet reliability. The packet delay period is 30 ms and 2 seconds.

CHAPTER 6

RESPONSE TIME MEASURING

We classify response time into one-way response time (described in detail in section 6.1) and two-way response time (section 6.2). One-way response time is how long the local player must wait to see a remote avatar change due to remote client activity. Two-way response time is how long it takes for a local player to see a local change after doing something to a remote avatar. One-way response time can be further divided into timing statistics for packets where one packet is sent per keypress, and statistics when multiple packets are sent per keypress.

There are other elements worth measuring to help us judge our techniques: interval update time for remote avatar movement/rotation which is another way of one-way response time measuring (section 6.1.3), rendering update speed (section 6.3), and various packet statistics (section 6.4).

6.1 One-way Response Time

One-way response time is the time that an action takes to travel from a remote player to the local player, and update the remote player's remote avatar. An example of one-way response time is the time difference between a remote player moving and the update of his remote avatar on the screen of the device. One-way response time consists of networking time (from the remote player to the local player) and processing times on the server and the local player's device.

One-way response time can be divided into two cases depends on how many packets are sent when a game key is pressed: single packet per keypress, and multiple packets per keypress.

6.1.1 One-way Response Time for a Single Packet per Keypress

A single packet per keypress occurs when the player presses a key, and only a single packet is sent. Most of the packets in PenguinM3G are of this type, including JOIN, JOINED, QUIT, SPOTS_STATE and BULLETS_AREA_STATE. An example is shown in Figure 6.1, when the life spot is changed.

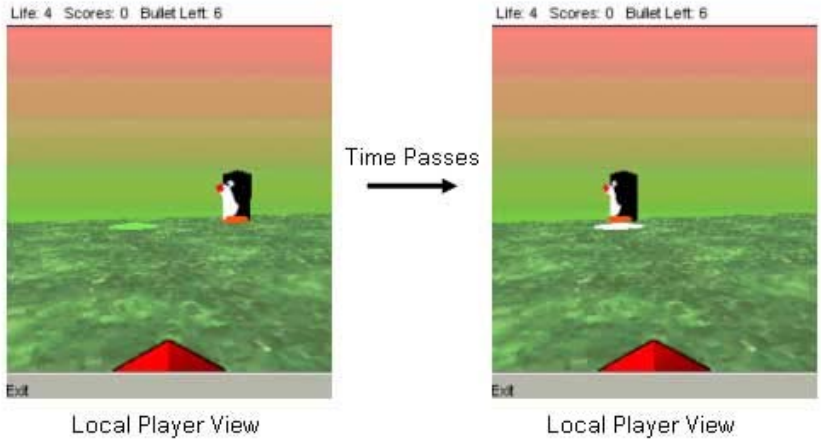


Figure 6.1: Example of Single Packet per Keypress (SPOTS_STATE)

When the remote player moves onto the life spot, a single SPOTS_STATE packet is sent to the other players to inform them that the life spot is now occupied, and the life spot changes to an inactive state.

6.1.2 One-way Response Time for Multiple Packets per Keypress

Multiple packets per keypress occur when the player presses a key and multiple packets are sent to the other players. This type includes the MOVE, ROTATE and KEY_RELEASE packets. KEY_RELEASE is in this group because it is sent after the MOVE and the ROTATE packets have finished indicating that the current move or rotation is over. An example of multiple packets per keypress is given in Figure 6.2.

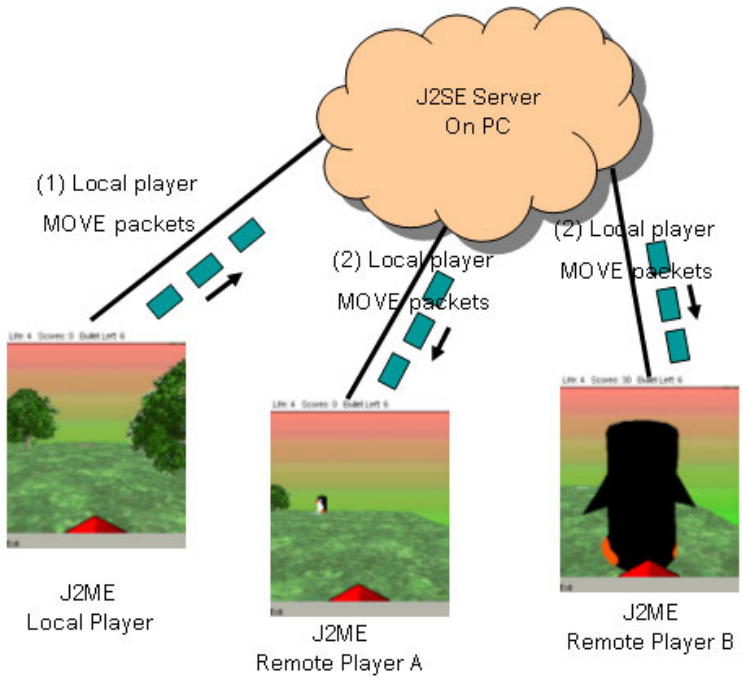


Figure 6.2: Example of Multiple Packets per Keypress

In step (1), when the local player moves, a series of MOVE packets are sent to the server. In step (2), the server forwards the packets to the remote players so they can update their avatar representing the local player.

6.1.3 Interval Update Time of a Remote Avatar Movement or Rotation

The interval update time of a remote avatar's movement is measured to indicate how well the game is being updated in an unreliable network. This measurement involves moving and rotation which is a one-way response time type. So, it is categorized in Section 6.1. The measure is the average time gap between updates of the avatar's position or rotation on the local player's device. An example of interval update for a remote avatar's movement is shown in Figure 6.3.

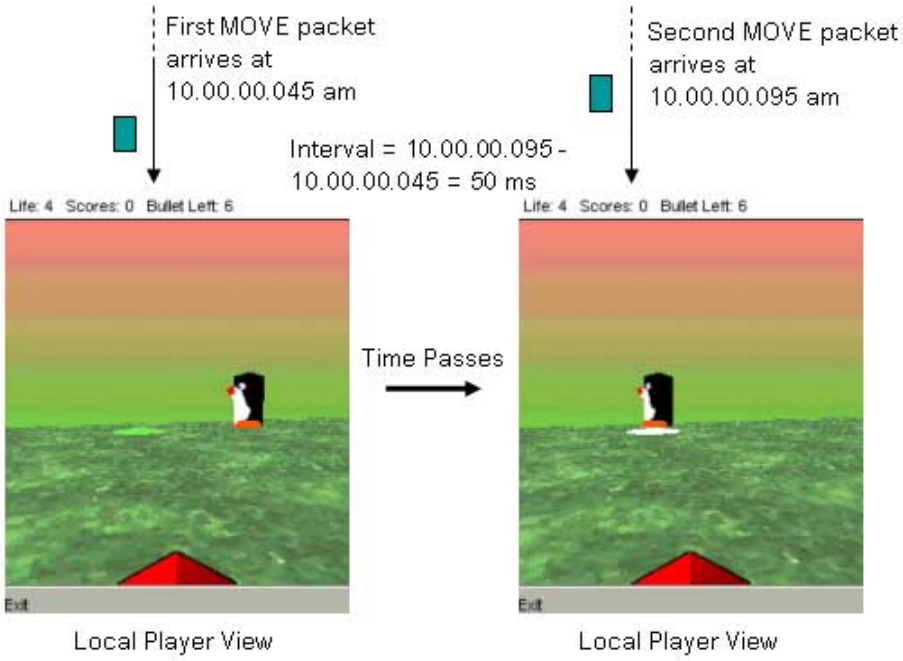


Figure 6.3: Example of Interval Update of Remote Avatar Movement

The first MOVE packet arrives from the server at 10.00.00.045 am, and the second arrives at 10.00.00.095 am, so the interval update time is 10.00.00.095 - 10.00.00.045 = 50 ms. The next interval will be measured between the second and the third MOVE packets, and so on, until a KEY_RELEASE packet arrives. There is also a threshold interval of 2500 ms to stop the measurement in case the KEY_RELEASE packet is lost.

The interval update time value should be close to the specified game frame rate. However, if packets are delayed or lost, then the interval will increase. Therefore, the closeness of the interval to the frame rate can be employed as another measure of our optimization techniques dealing with packet delay and loss.

6.1.4 One-way Response Time Implementation and Measuring

A packet timestamp field is needed to implement response time measuring. Timestamps are added at the source and destination clients, so that one-way response times can be calculated as shown in Figures 6.4 and 6.5.

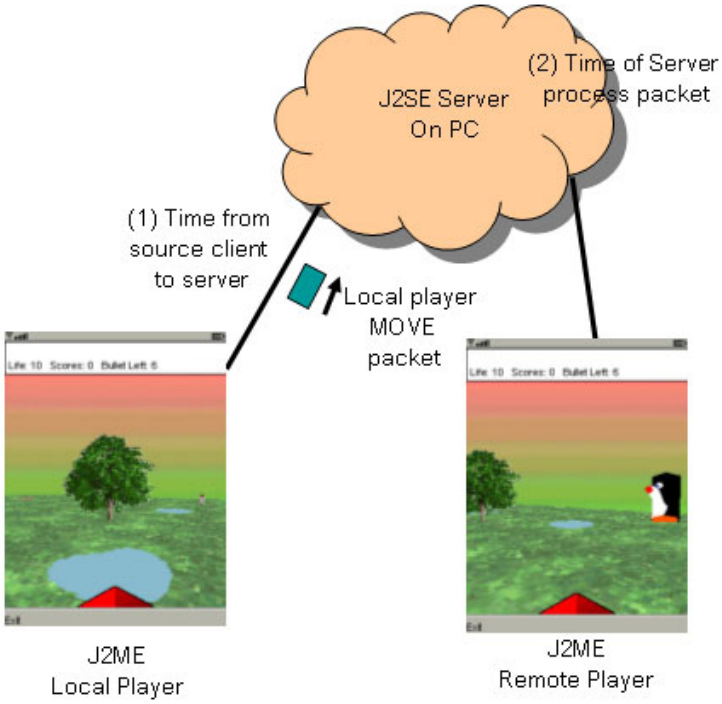


Figure 6.4: The Sequence of One-way Response Time Measuring (steps 1-2)

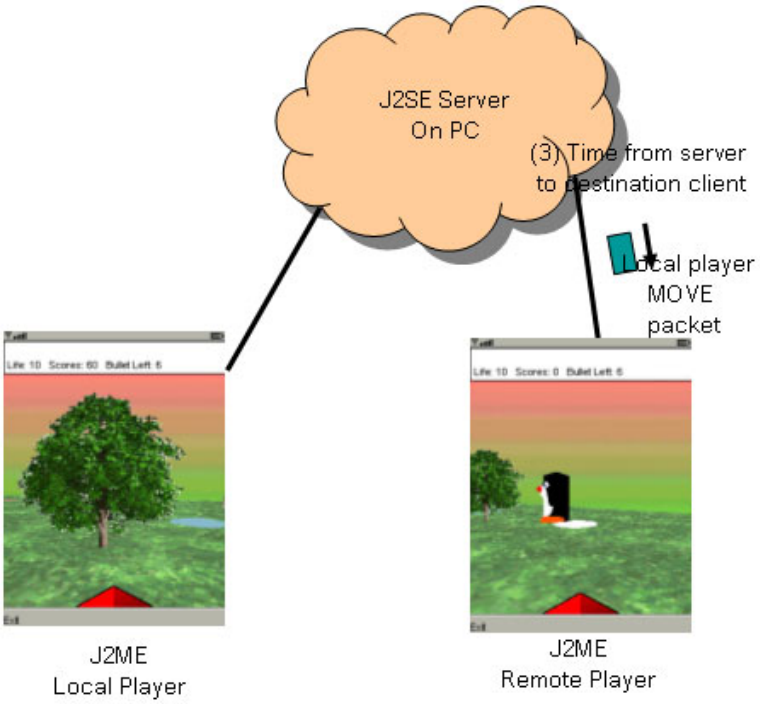


Figure 6.5: The Sequence of One-way Response Time Measuring (step 3)

Step 1 in Figure 6.4 shows a timestamped packet being sent from the local player to the server. and step 2 is the processing time on the server. Step 3 in Figure 6.5 shows the packet being sent from the server to the destination remote player. The difference between the arrival time and the timestamp extracted from the packet is the one-way response time.

Device Clocks Synchronization Problem

This timestamping approach conduces a problem if the two client devices have involved different clock settings.

The clocks on client A and client B may be out of synchronization (e.g. at the same instance, the client A time is 10:00:00:00 but client B shows 10:00:00:05). If a packet takes 300 ms to travel from client A to client B, then the one-way response time between A and B will be mis-reported as 305 ms.

I attempted to solve this "out of synchronization" problem in several ways. I tried using a network time protocol (NTP) server to synchronize the PC times, but its accuracy was over 100 milliseconds which is too poor. This out of synchronization comes from my tests on Windows 2000 machines on a LAN network, where the NTP server is a windows 2000 built-in service.

I solved the problem by using only one client to calculate the one-way response time. A message is sent round trip between the clients and then divided by two to get the one-way response time, as shown in Figures 6.6 and 6.7.

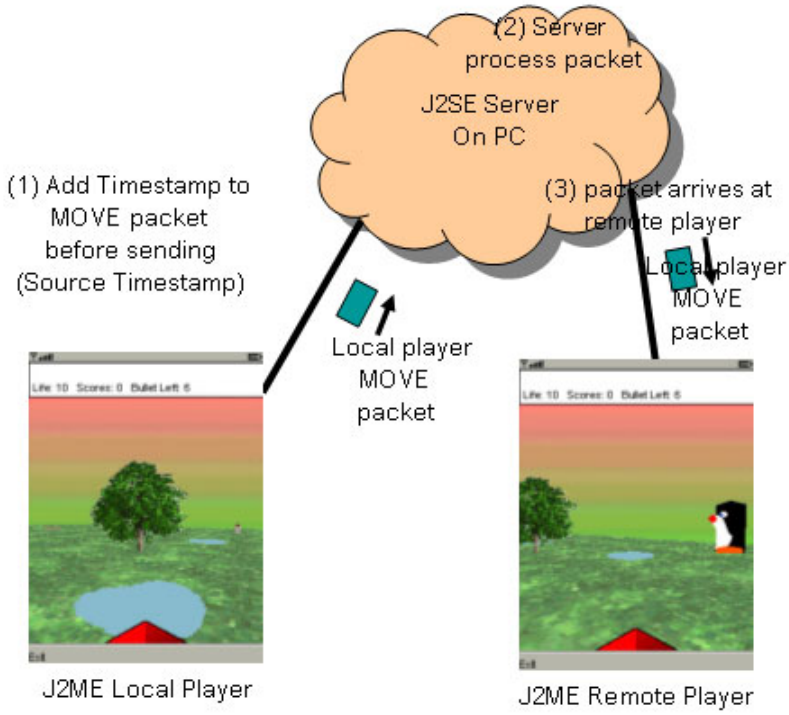


Figure 6.6: One-way Response Time at One Side of Client (Steps 1-3)

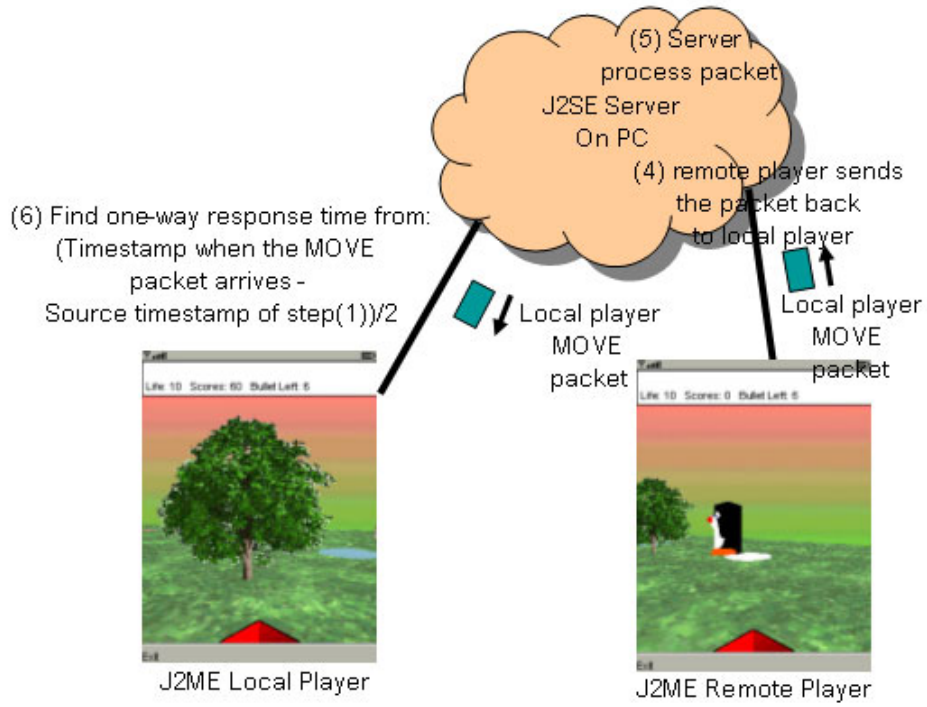


Figure 6.7: One-way Response Time at One Side of Client (Steps 4-6)

In step 1 of Figure 6.6, a timestamp is added to the MOVE packet sent to the server. The server processes the packet and forwards it onto the remote client in step 3. Instead of the remote client calculating the one-way response time, it sends the packet back to the sender with a "REPLY_CODE" header. The local client receives this packet in step 6, and calculates the one-way response time by subtracting the original timestamp from the current time, and dividing by two.

Varying network time can affect this calculation, but such variations are very uncommon on the LAN where I carried out my tests.

The implementation steps for one-way response time measuring are shown in Figure 6.8.

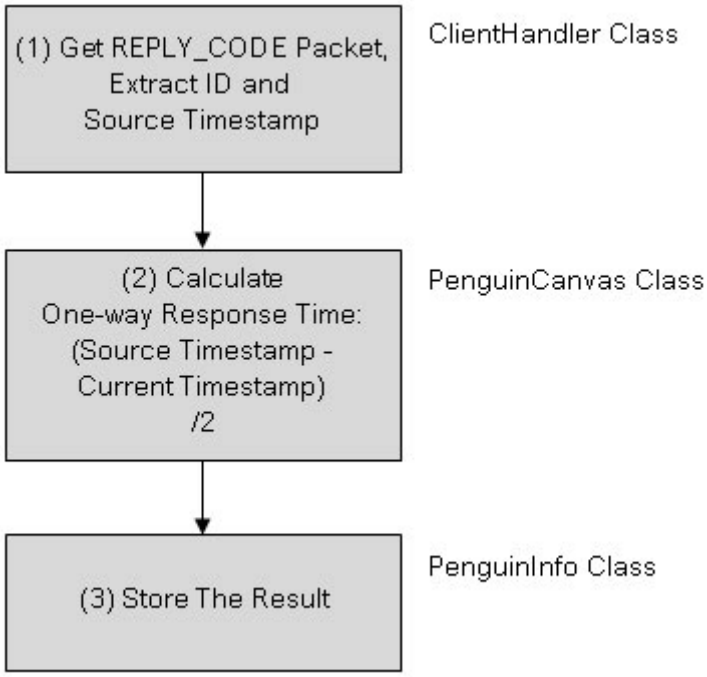


Figure 6.8: One-way Response Time Implementation

When *ClientHandler* gets a reply packet (indicated by the "REPLY_CODE" header), it extracts the destination client ID and source timestamp. These are sent to *PenguinCanvas* where the one-way response time is calculated as the current timestamp minus the source timestamp, divided by two. *PenguinInfo* collects these response times in order to calculate the mean and SD.

6.2 Two-way Response Time

Two-way response time is the time that a local player action takes to travel to a remote player, affect it, and for the remote player's new state to travel back to the local player and change the remote avatar. An example of this type of response time is when the local player shoots at a remote avatar. A packet is sent to the remote player which decides if the bullet hits the player or not. The remote player sends a reply packet back to the local player. The example is shown in Figures 6.9 and 6.10.

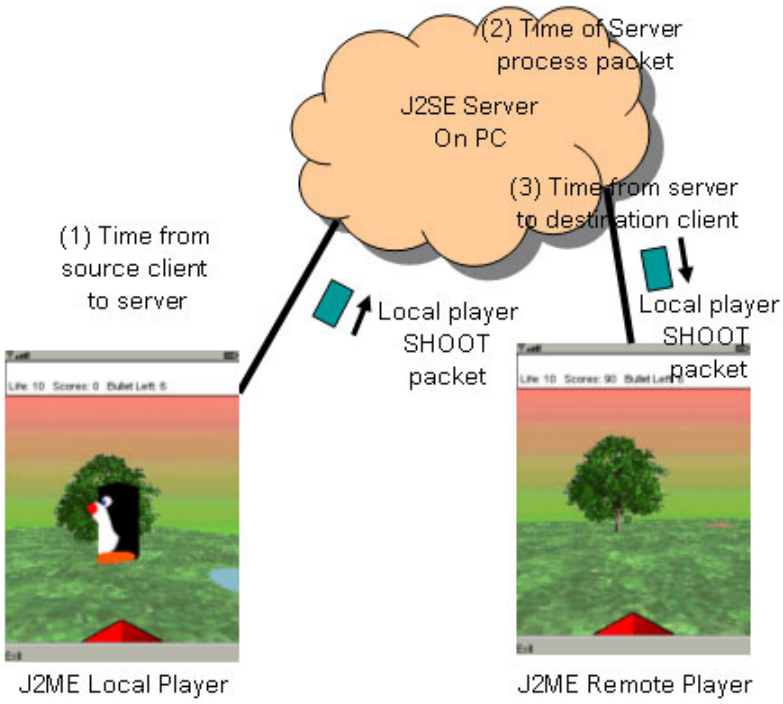


Figure 6.9: Two-way Response time (Steps 1-3)

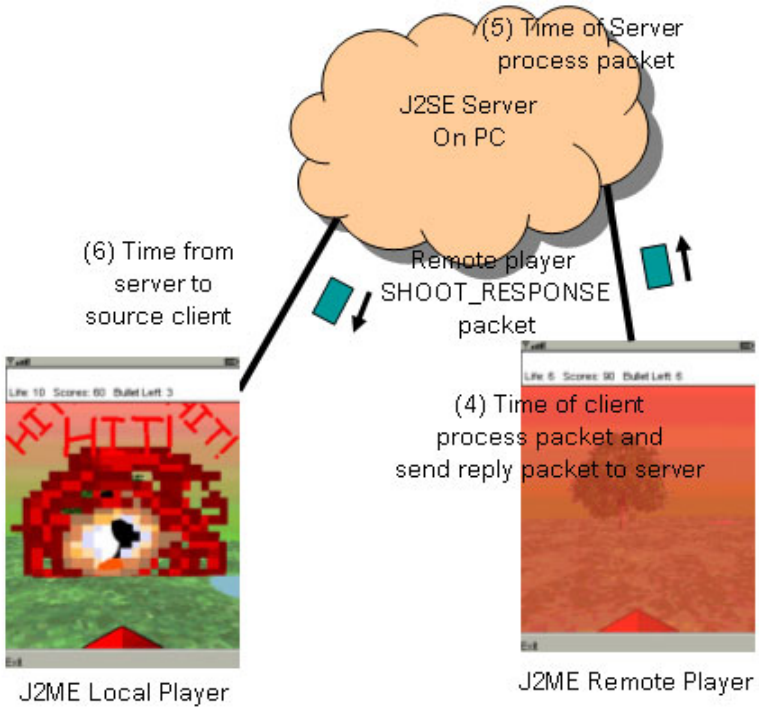


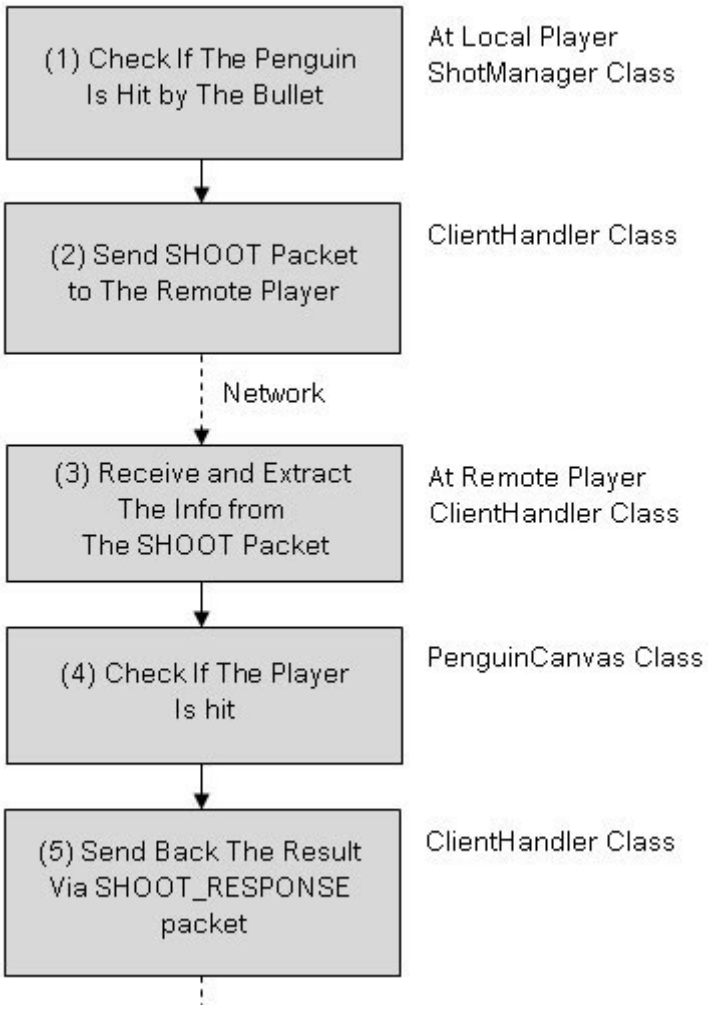
Figure 6.10: Two-way Response time (Steps 4-6)

In step 1 of Figure 6.9, a timestamp is included in the SHOOT packet sent to server. The server processes the packet and forwards it to the remote client. In step 4 (Figure 6.10), the remote client processes the SHOOT packet and sends a reply packet (SHOOT_RESPONSE) back via the server.

The server processes the packet and forwards it to the local client. The time duration, from when the local player sent the SHOOT packet until it receives the SHOOT_RESPONSE packet and updates the remote avatar, is the two-way response time.

Two-way Response Time Implementation and Measuring

Measuring two-way response time is similar to measuring one-way response. The implementation is outlined in Figure 6.11.



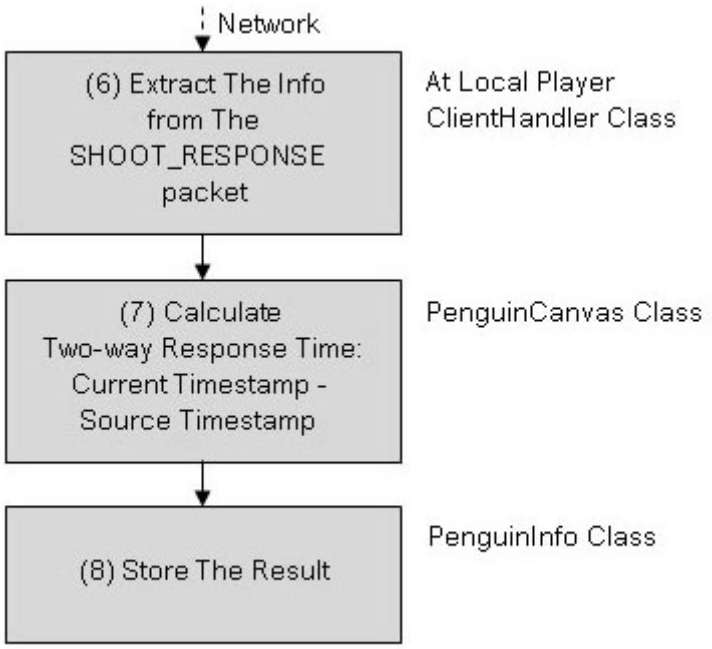


Figure 6.11: Two-way Response Time Implementation

ShotManager checks if the penguin has been hit by a bullet. If it has, then *ClientHandler* sends a SHOOT packet to the remote player. When the remote player receives a SHOOT packet in its *ClientHandler*, the information is extracted. The hit coordination is compared with the player's location to check if the player was actually hit or not. The reason for checking is that the penguin position, and the remote player position, may be different (due to delay). There is a threshold so that even if the penguin has moved a bit, the shot may still count as a hit.

After checking if the remote player is hit, a SHOOT_RESPONSE packet is sent to the local player. SHOOT_RESPONSE indicates whether the remote player has been hit or not and includes the source timestamp from the SHOOT packet. The local player receives the SHOOT_RESPONSE packet at its *ClientHandler*, which extracts the information. *PenguinCanvas* displays an explosion if the penguin was hit and calculates the two-way response time by subtracting the current time from the source timestamp. Step 8 adds the result to the *PenguinInfo* object.

6.3 Rendering Update Method

The client processes tasks related to response time which may affect the game's frame rate. The effect can be measured by recording the slowdown in the game's *update()* method in *PenguinCanvas* which deals with the player's input and scene rendering.

Update Method Implementation and Measuring

The *update()* implementation is illustrated in Figure 6.12.

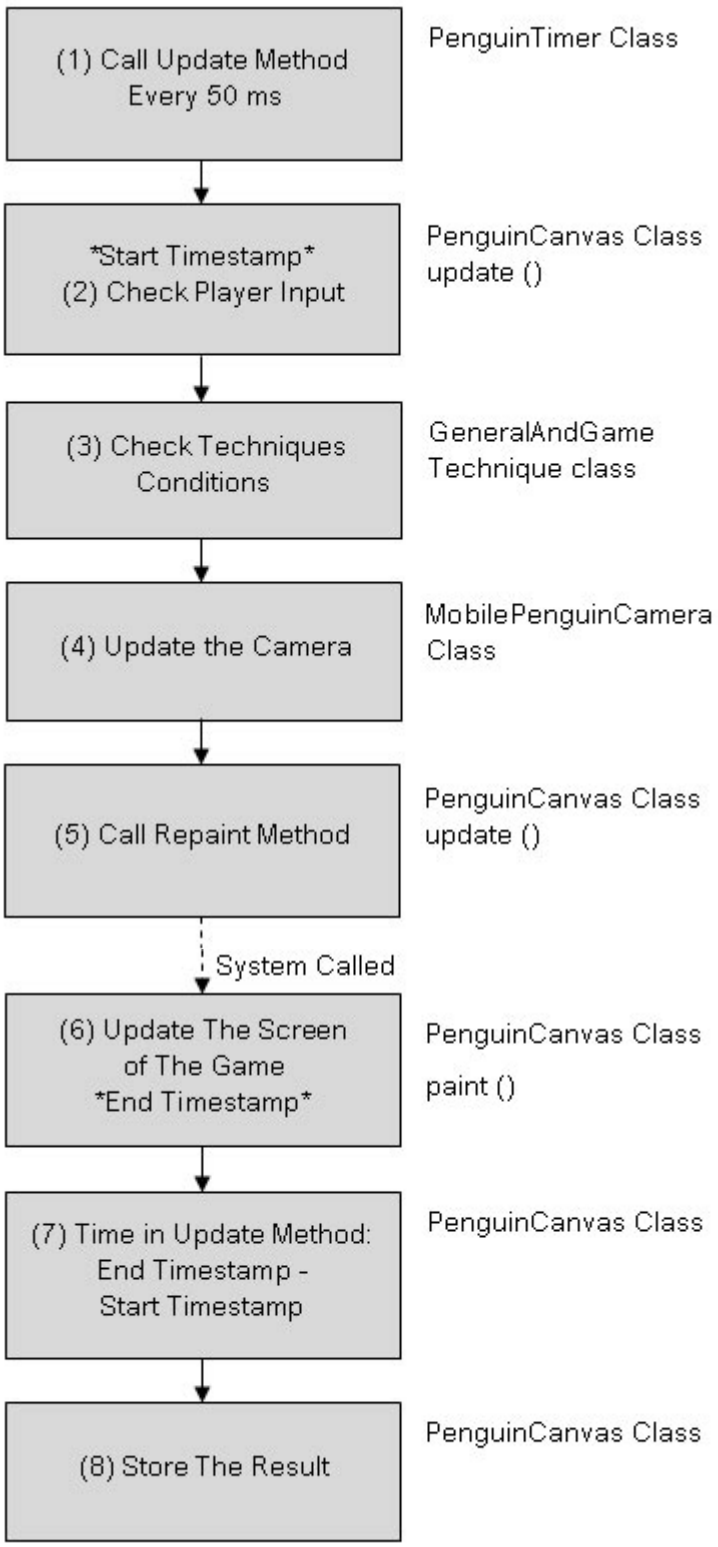


Figure 6.12: Update Method Implementation

PenguinTimer calls the *update()* every 50 ms, which makes the game executes at 20 frames per seconds (20 fps =1000/50). *update()* checks the keypresses and releases and calls *GeneralAnd GameTechnique* to check the techniques conditions. In step 4, the game camera is updated by *MobilePenguinCamera*, and the *repaint()* is called. The *paint()* is not immediately called by *repaint()*. So, there may be a delay at this point.

A timestamp is obtained at the beginning of the *update()* (before step 2) and subtracted from the time at the end of *paint()* in step 6. The result is stored in *PenguinCanvas* at step 8.

6.4 Packet Statistics

The packet statistics gathered include the size of all the packet bodies (in bytes) and the packet sending frequency (packets/sec). These are used to compare the techniques based on their reduction of packet size and packet resending compared to the original game.

6.4.1 Packet Size and Measuring

Packet size is measured by the *connect()* in *Sender* before sending the packets. *Byte.length* is obtained from the byte array that stores the packet information.

6.4.2 Packet Sending and Measuring

The packet sending rate is measured by *ClientHandler* in term of the interval between sending two packets.

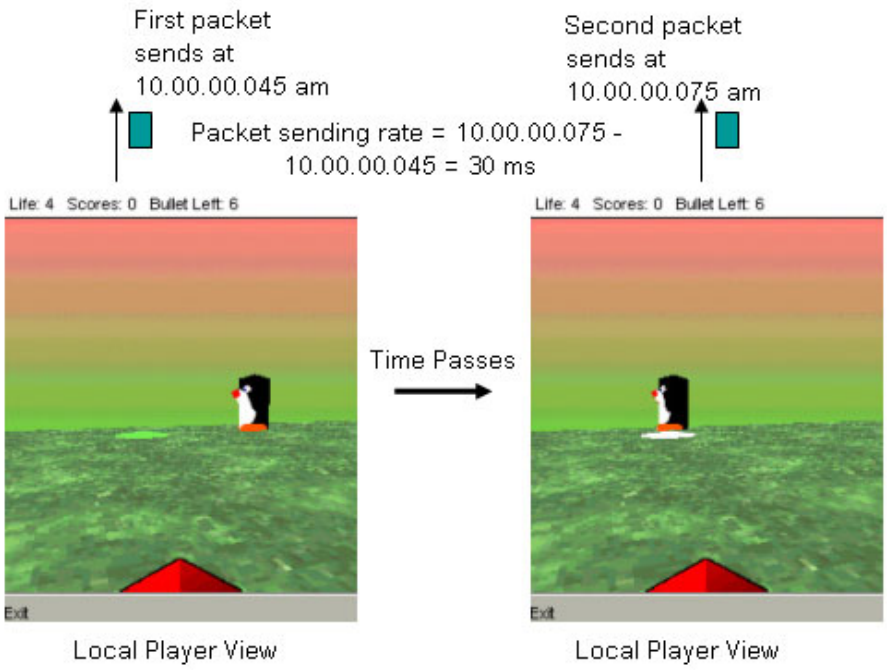


Figure 6.13: Packet Sending Rate

For example, in Figure 6.13, the first packet is sent at 10.00.00.045 am and the second at 10.00.00.075 am, making the sending rate 10.00.00.075 - 10.00.00.045 = 30 ms. There is a threshold packets sending rate of 3 seconds, which causes packets if which sending rate exceeds three seconds to be ignored. The threshold of three seconds is chosen from the randomly delay time of the server is two seconds, plus another second added on for the network time and processing time of the client. For instance, if a packet is sent at 10.00.00.100 am and the next at 10.00.04.500, then the packet sending rate is 10.00.04.500 - 10.00.00.100 = 4400 ms which, because it exceeds the threshold of three seconds, will not be counted. This threshold is needed so that if the player moves, stops, then moves again, then the second move will not be counted towards the packet sending rate if the player stopped for longer than three seconds.

6.5 Summary

This chapter described response time types: the one-way response time for a single packet per keypress, the one-way response time for multiple packets per keypress, the interval update time of a remote avatar movement or rotation, and two-way response time. The problem of device clocks synchronization was discussed. Other elements are measured to judge our techniques effectiveness on rendering update time, packet size, and packet sending.

CHAPTER 7

TECHNIQUES FOR IMPROVING RESPONSE TIME

The techniques applied to the game to improve response time can be divided into three groups: general techniques can be applied to any networked game using avatars, game-specific techniques are applicable only to this game, and, packets based techniques focus on game packets.

7.1 General Techniques

General techniques can be used in networking game involving avatars. I consider three in this chapter: dead reckoning, smoothing [6], and, visual field updating.

7.1.1 Dead Reckoning

Dead reckoning (DR) uses previously sent move/rotate packets to predict a remote avatar's next movement/rotation when the packets holding that information are lost or delayed.

A penguin can move or rotate, but not both at the same time. This form of movement allows first order dead reckoning to be used (see section 2.1).

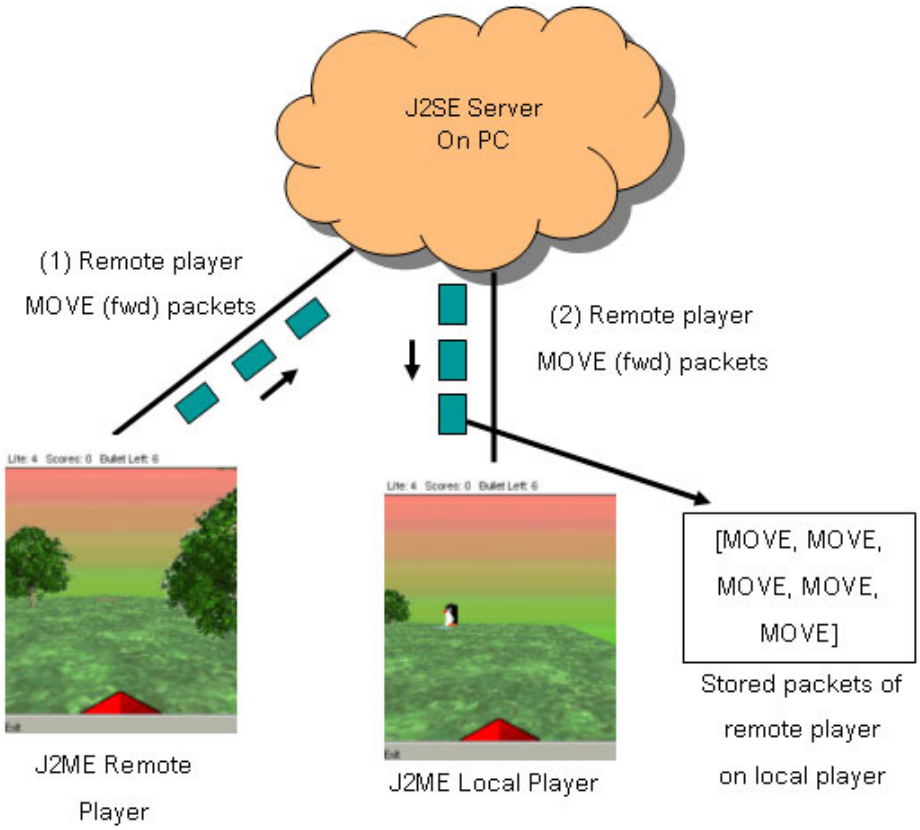


Figure 7.1: The DR Sequence (Steps 1-2)

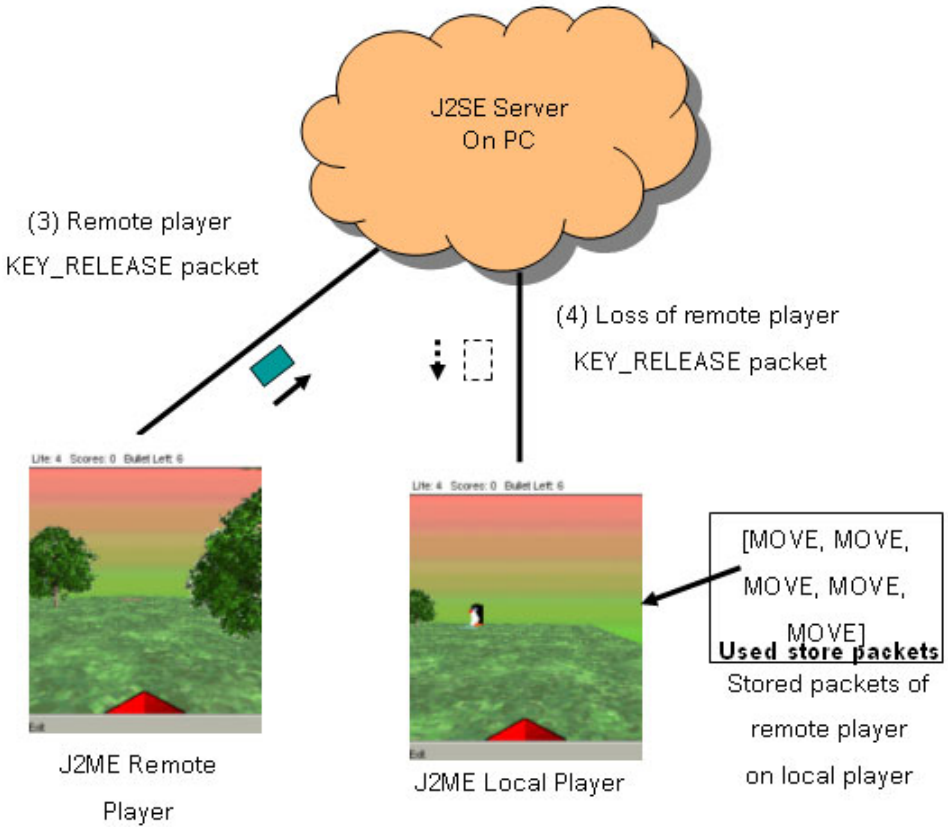


Figure 7.2: The DR Sequence (step 3-4)

For instance, in Figures 7.1 and 7.2, when the remote player presses moving or rotating keys, MOVE or ROTATE packets are sent to the local player at 1 packet per frame. When the remote player releases the key, a KEY_RELEASE packet is sent to indicate that this series of moves or rotates are finished. DR will be activated if the local player gets a series of MOVE or ROTATE packets but no KEY_RELEASE packet within a threshold time of 1 or optional 2 frames (50 or 100 ms). The DR prediction makes a remote avatar move or rotate based on its history of moves and rotates.

For example, if the last stored command is MOVE, then first order DR makes the remote avatar move forward as shown in the Figure 7.3. DR keeps generating moves or rotates until a new MOVE, ROTATE, or KEY_RELEASE packet arrives, or DR reaches a maximum threshold prediction set at 10 frames (500 ms) which should be long enough or prediction for too long will have the chances to make more mistake of prediction. After this, DR is deactivated.

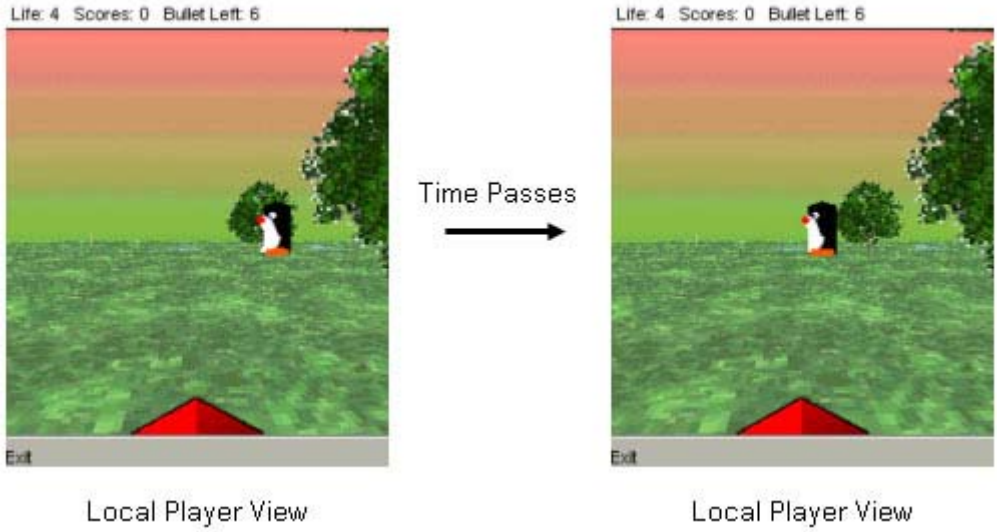


Figure 7.3: DR Activation

DR keeps one-way response time small since an update occurs even when some packets are lost or delayed. The avatar does not need to wait for the next packet to arrive.

The DR implementation stages are shown in Figure 7.4.

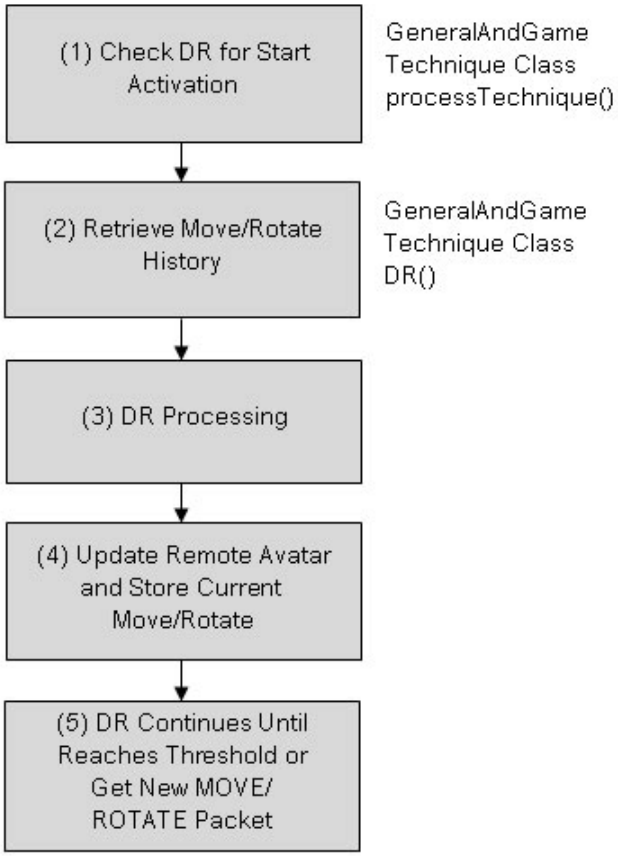


Figure 7.4: DR Implementation

The DR status is monitored by *processTechniques()* in *GeneralAndGameTechnique*. DR is activated (*DR()* is called) if the arrival of MOVE/ROTATE packets is not continuous, or some packets are missed during the processing of the move/rotate. The non-arrival of KEY_RELEASE within its threshold will also trigger DR.

DR() retrieves the histories of moves/rotates from *PenguinInfo*. There are two types of history: one made from the previous MOVE/ROTATE packets, and one based on the history of the DR processing. When DR is first activated, the move/rotate history is examined, but as DR continues, the DR history is examined.

In step 4, the remote avatar is moved/rotated according to the DR prediction. This DR prediction is stored in *PenguinInfo* separate from the history of MOVE/ROTATE packets. If there is not a MOVE/ROTATE or KEY_RELEASE packet received in the next frame, DR continues, it makes use of the history of DR predictions. DR continues until a timing threshold is reached (10 frames since the start of DR) or until a MOVE/ROTATE or KEY_RELEASE packet arrives.

7.1.2 Smoothing

Smoothing is utilized after DR processing finishes in order to gradually correct the DR generated move/rotation of a remote avatar to bring it to its actual position/angle.

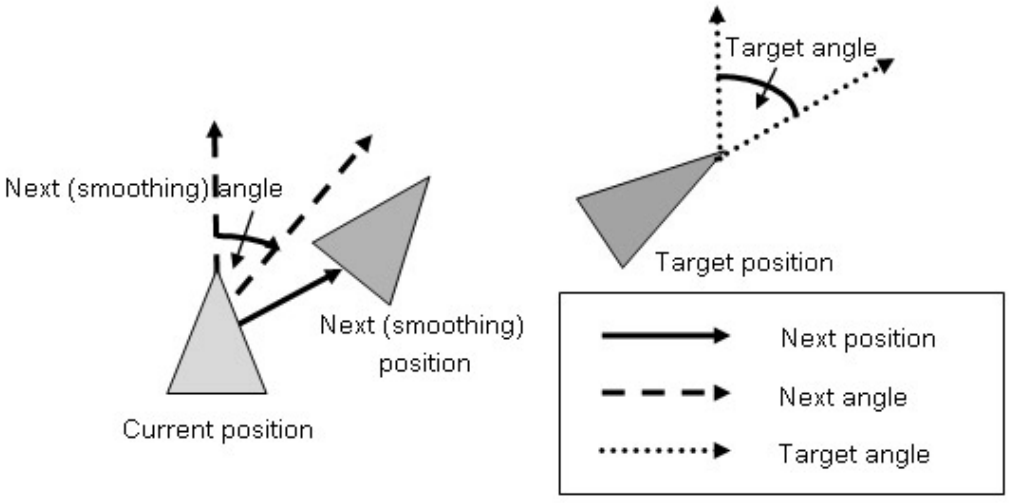


Figure 7.5: Smoothing

In each frame, smoothing checks the angle and position of the remote avatar against its required position, and adjusts it to be closer to that position. Smoothing continues until the current avatar position/angle is equal (or very close) to the required position/angle. If MOVE or ROTATE packets arrive during smoothing, they are stored and smoothing calculates the position based on the newest packet.

The implementation steps for smoothing are shown in Figure 7.6.

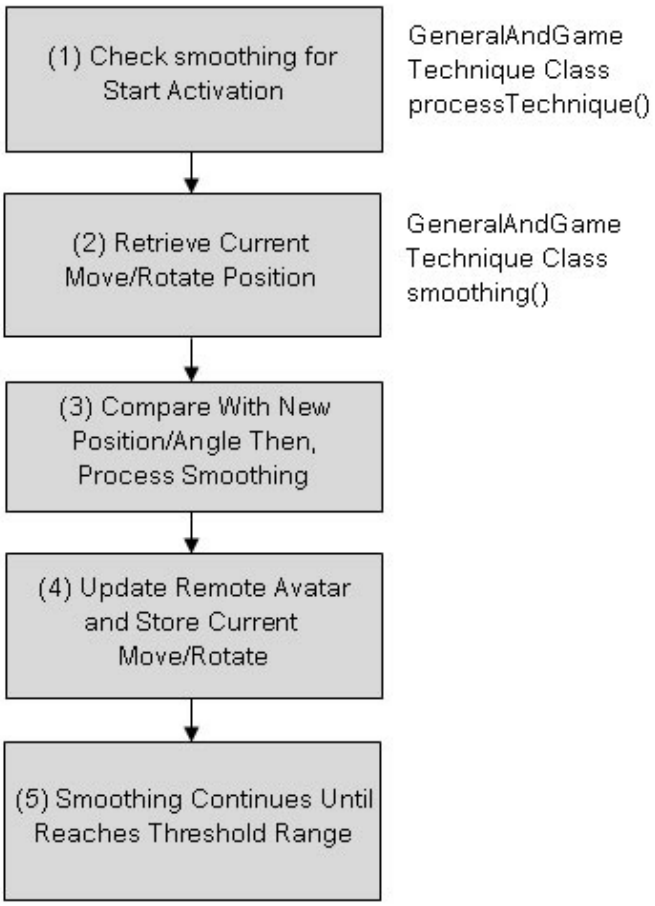


Figure 7.6: Smoothing Implementation

Smoothing is checked in *processTechnique()* in *GeneralAndGameTechnique*, and activated when DR is activated and new MOVE/ROTATE packets arrive. At this point, DR stops and smoothing takes over. *smoothing()* in *GeneralAndGameTechnique* gets the current position/angle of the remote avatar and compares it to the position/angle of the new MOVE/ROTATE packet in order to adjust the avatar's move/rotation toward that of packet. In step 4, the remote avatar is updated and the current position/angle is stored so it can be used by future smoothing processing. Smoothing continues until the remote avatar is near enough to the new position/angle.

Combining DR and Smoothing

Since smoothing is used to improve DR prediction, DR and smoothing are treated as one technique in the measurement performed in chapter 8. DR predicts the next move when there is no packet and, smoothing corrects the remote avatar position when a packet arrives. It is the most effective to activate smoothing after the end of DR so that the remote avatar performs natural movement instead of sudden jumps to a new position.

7.1.3 Visual Field Updating

Visual Field Updating uses the player's visual range to reduce the amount of updates to remote avatars. If a remote avatar is outside the viewing range, e.g. behind the player as in Figure 7.7, then there is no need to update its position or angle.

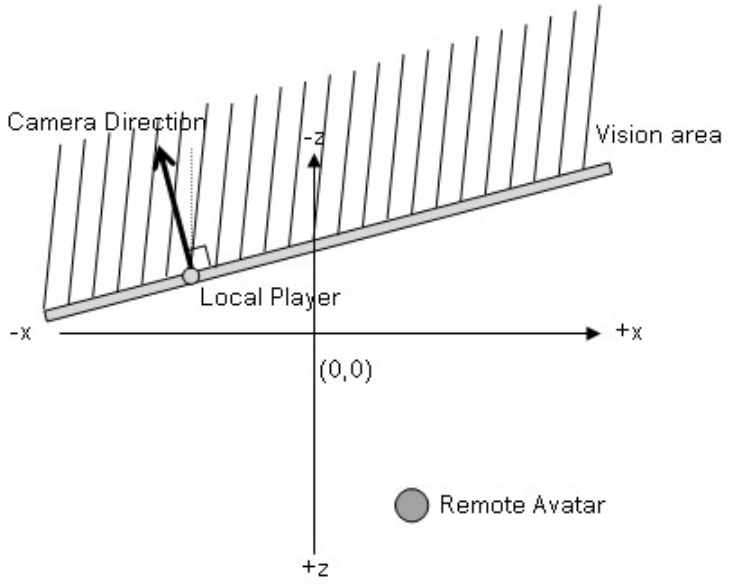


Figure 7.7: Visual Field Updating Vision Area

Visual Field Updating checks every remote avatar to see if it is inside the player's camera viewing range. If a remote avatar is outside the vision area, then it is not updated and the incoming MOVE, and ROTATE packets for that avatar are stored. Also, techniques related to the avatar, such as DR and smoothing, are deactivated. The implementation steps of Visual Field Updating are shown in Figure 7.8.

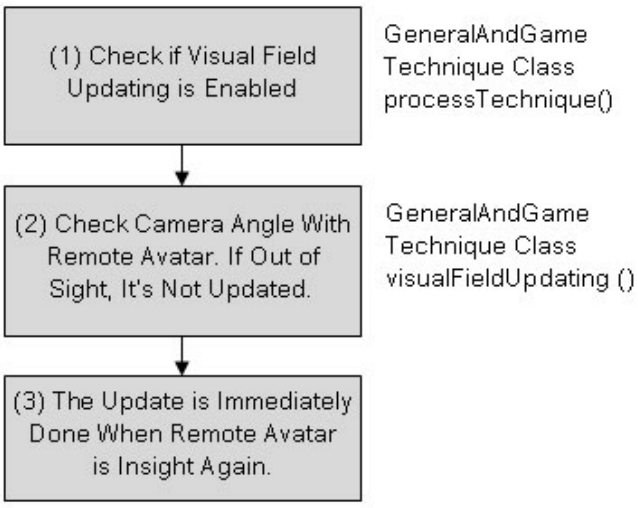


Figure 7.8: Visual Field Updating Implementation

Visual Field Updating is enabled by *processTechnique()* in *GeneralAndGameTechnique*. The vision range is calculated from the current angle of the camera, plus and minus 90 degrees around the xz plane (as shown in Figure 7.7). If the remote avatar is outside the vision area, it will not be updated and the MOVE/ROTATE packets for that avatar are stored for later. When the remote avatar does appear in the vision area, it will be updated to its current position/angle.

7.2 Game-Specific Techniques

Game-specific techniques are specified to this game's remote avatars. I consider two techniques in this group: Avatar Blinking and Avatar Dying.

7.2.1 Avatar Blinking

The remote avatar blinks (as shown in Figure 7.9) when there is a chance that it has been shot by the local player. This activity improves the response time (when avatar blinks, it counts as response to the player) while the local player waits for a shot result packet to arrive from the remote player. Also, the blinking indicates the chance that the shot may have missed the remote player since its position on the local client is not its real position due to MOVE/ROTATE packet delay or lost.

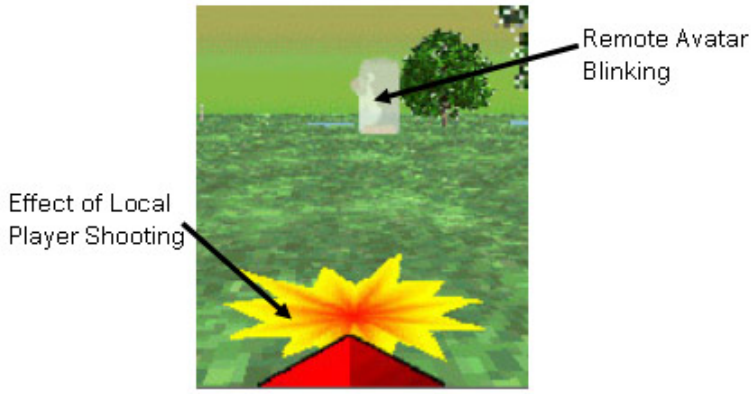


Figure 7.9: Avatar Blinking

When the player shoots, the flash effect is displayed on the screen and the remote avatar blinks. The blinking is displayed until either a shot result packet arrives or until a blink time threshold of 2 seconds is reached. The threshold may be reached if the result packet is lost on its way from the remote client or it is delayed for more than 2 seconds which will be discarded.

The implementation steps for avatar blinking are shown in Figure 7.10.

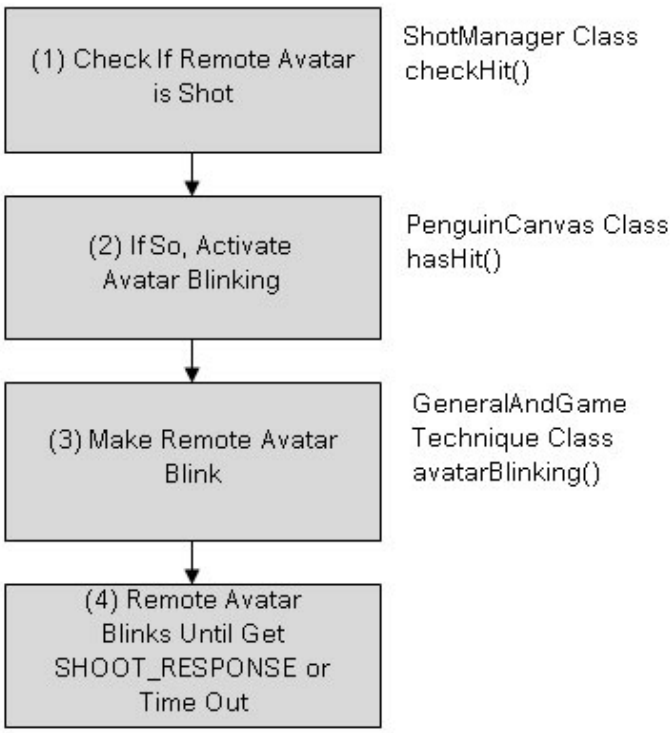


Figure 7.10: Avatar Blinking Implementation

The shooting of a remote avatar is checked by *checkHit()* in *ShotManager*. *hasHit()* in *PenguinCanvas* determines if the local version of the remote avatar was hit in order to calls *avatarBlinking()* in *GeneralAndGameTechnique*. The blinking is implemented by moving the penguin under the floor on alternative frames. The maximum threshold for blinking is two seconds, or until a SHOOT_RESPONSE packet arrives.

7.2.2 Avatar Dying

Avatar dying makes a transparent skull image appear in front of remote avatar (as in Figure 7.11) when no packet is received from that player for more than 30 seconds. If another 30 seconds passes without a packet, then the remote avatar is remove from the game.

Figure 7.11: Avatar Dying

This technique handles the situation when a remote client exits the game without telling the server first either intentionally or because of machine or network failure.

The implementation steps for avatar dying are shown in Figure 7.12.

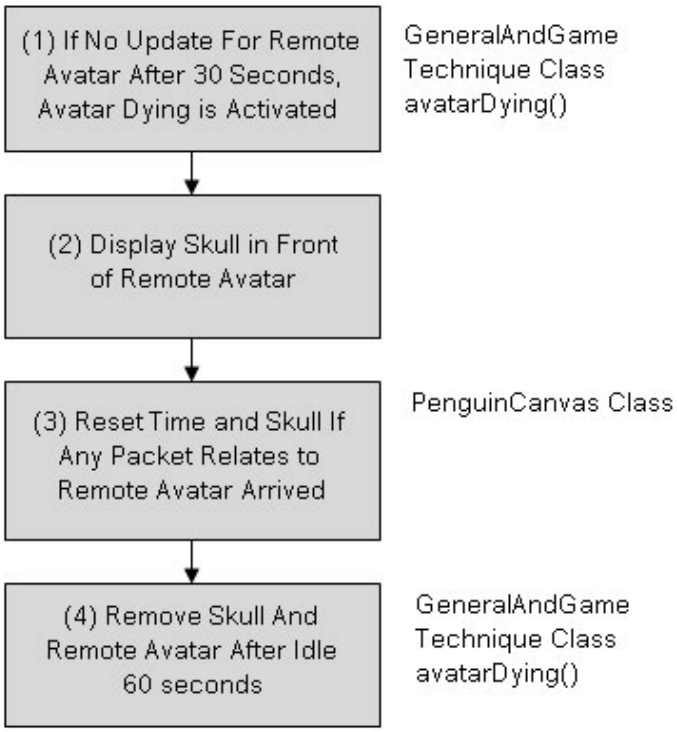


Figure 7.12: Avatar Dying Implementation

avatarDying() in *GeneralAndGameTechnique* counts every frame when there is no update of a remote avatar. When the count reach 600, which is equivalent to 30 seconds, a skull image is displayed in front of the remote avatar. If no update packet arrives (e.g. moving, rotating, shooting, or quit) for another 30 seconds (i.e. a total inactivity of 1 minute), then the remote avatar is removed from the game.

7.3 Packets Based Techniques

Packet based techniques deal with game packets. There are two techniques in this group: Packets Grouping and Duplicate Packets.

7.3.1 Packets Grouping

Multiple packets are grouped into one packet and sent out at once. This reduces the frequency of packet sending and the use of the UDP header which is more than 100 bytes, and so much bigger than the information of a single packet. However, packets grouping lengthens one-way response time due the delay in grouping packets before sending them out.

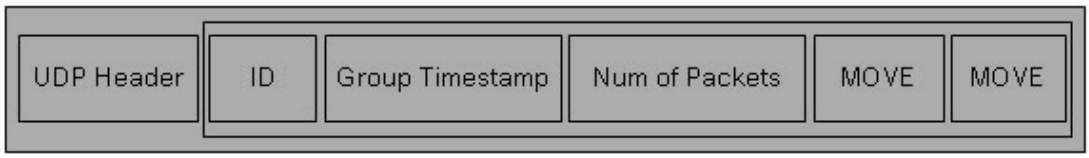


Figure 7.13: Packets Grouping

Packets Grouping has two modes: where a maximum of 2 packets or collected within 100 ms (2 frames), or where a maximum of 3 packets are collected within 150 ms (3 frames). These are called Packets Grouping (2) and Packets Grouping (3) respectively. An example of Packets Grouping (2) is shown in Figure 7.13. The packet has an ID, group timestamp (for checking packet order) and number of packets (in order to extract the packets correctly). It may be that less than 2 or 3 packets are sent per group if grouping would otherwise take too long.

Packets Grouping is used for MOVE, ROTATE, and KEY_RELEASE packet types. These types of packet are sent most frequently so grouping them is an effective optimization. Grouping does not employ 4, 5 or more packets since it would significantly slow down the response time.

The implementation steps for Packets Grouping are shown in Figure 7.14.

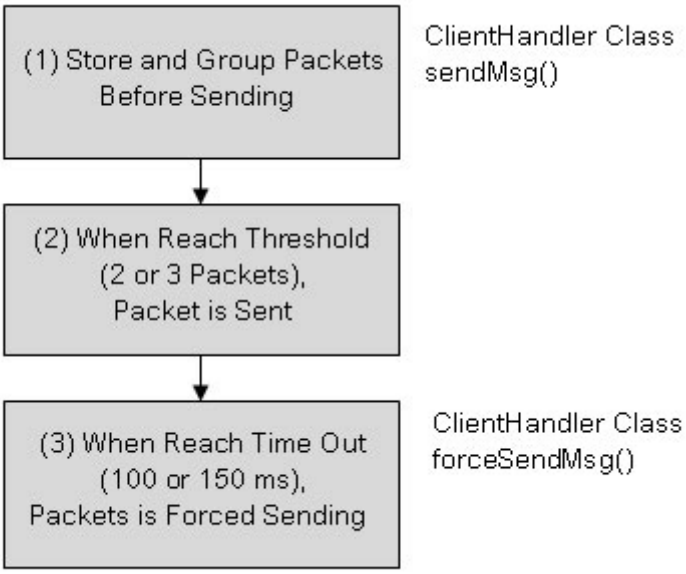


Figure 7.14: Packets Grouping Implementation

Packets are stored in a string buffer by *sendMsg()* at *ClientHandler*. When enough packets have been collected (2 or 3), the group packet is built and sent. There is a time-out threshold in case the group takes too long to collect (100 or 150 ms), in which case a group packet is created with the existing buffer contexts and sent immediately. This prevents grouping waiting too long.

7.3.2 Duplicate Packets

The same packet is sent twice in order to reduce the chance of packet lost. Duplicate Packets are set for JOIN, JOINED, QUIT, SHOOT, SHOOT_RESPONSE, SPOTS_STATE, and BULLETS_AREA_STATE, which are all the game packets except for those utilizing Packets Grouping. Loss of Packet Grouping packets is handled by DR (see section 7.1.1).

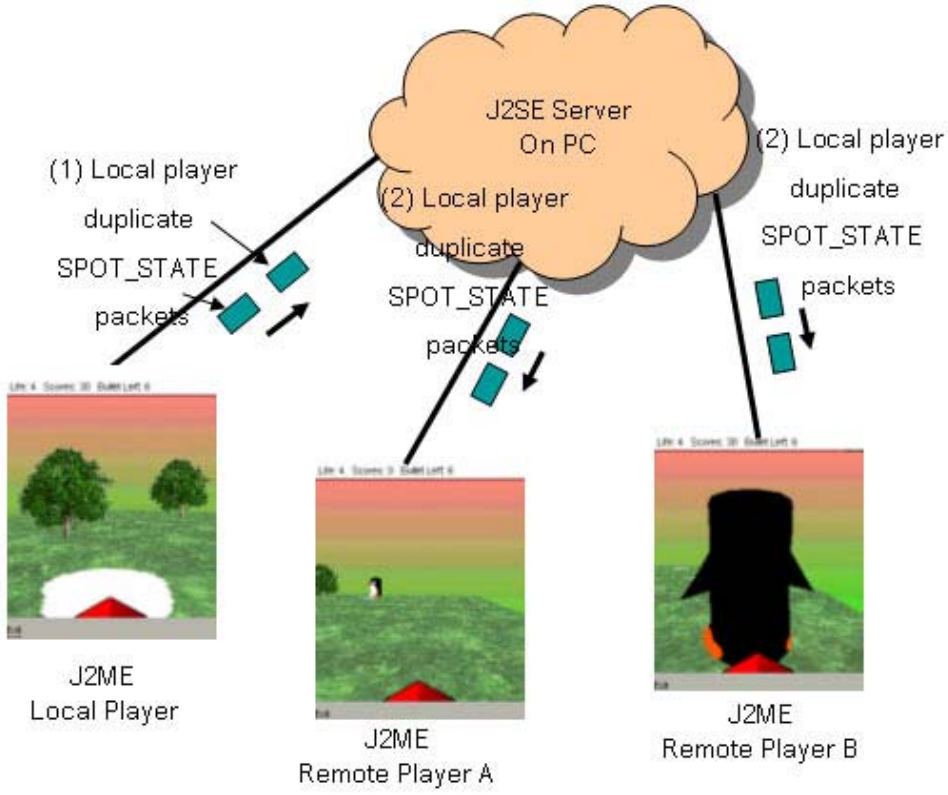


Figure 7.15: Duplicate Packets Example (Move onto a Life Spot)

For example, in Figure 7.15, when the local player moves onto a life spot, a SPOT_STATE packet is sent to the other players. Duplication means that the SPOT_STATE packet is sent twice. If one of the SPOT_STATE is lost, the receivers (remote players) can process the other SPOT_STATE packet. If both packets arrive, the receivers will use their timestamps and IDs to discard one of them. This requires that each player keeps a history of <ID, timestamp> pairs, so if two copies of the same packet arrive (based on its ID and timestamp), then the later one is discarded.

There are two types of duplication in the game: Duplicate Packets, and Triplicate Packets. Duplicate Packets means sending the same packet twice, and Triplicate Packets means sending the same packet three times. Triplicate Packets makes it much less likely that any given packet will be lost. However, Triplicate Packets also increase the network traffic more than Duplicate Packets.

The implementation steps for Duplicate Packets are shown in Figure 7.16.

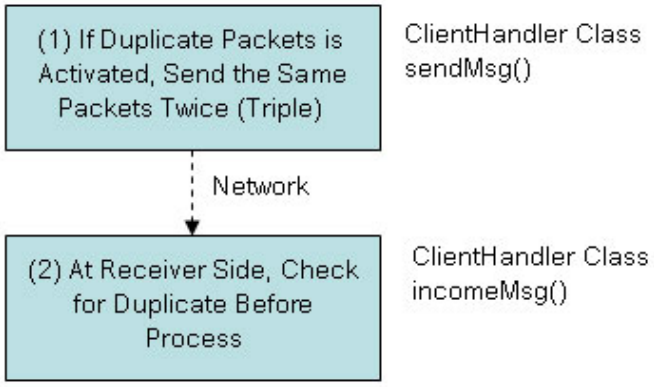


Figure 7.16: Duplicate Packets Implementation

If Duplicate or Triplicate Packets is activated, *sendMsg()* in *ClientHandler* will send a packet twice or three times. On the receiver side, incomeMsg() in *ClientHandler* receives a packet and stores its <ID, timestamp> pair. When the next packet arrives, its ID and timestamp are compared with the stored pairs, and is discarded if it is found to be a duplicate.

7.4 Summary

The game client techniques are divided into three groups: general techniques, game-specific techniques, and packets based techniques. This chapter explained each technique and its implementation: DR, Smoothing, Visual Field Updating, Avatar Blinking, Avatar Dying, Packets Grouping, and Duplicate/Triplicate Packets.

CHAPTER 8

GAME MEASUREMENT

The techniques explained in chapter 7 were tested on a local area network (LAN) with Java Wireless Toolkit (WTK) clients on three PCs and a J2SE server simulating 90% and 75% network reliability.

90% network reliability means that the chance of packet being delayed or lost is 10%. A delay can vary randomly between 30 ms and 2 seconds. 75% network reliability means the chance of packet delay or loss is 25%.

Z-tests are used to compare the results of enabling various combinations of techniques with the system with no techniques utilized (z-tests are described in section 2.7). The means of the results using techniques are compared with the mean of no techniques and the difference is deemed significant if the z-score is less than -1.645, which is a 0.05 level of significance in a one-tailed test.

Techniques

The tests consist of 11 combinations of different techniques. The first (T1) is the basic system with no techniques enabled, which will act as a basis for comparison. T2 to T7 are individual techniques, and T8 to T11 combine several techniques to see if a combination produces better results.

T1: No techniques enabled.

T2: *Dead reckoning (DR) and smoothing.* DR predicts the movement or rotation of a remote avatar when no update packets arrive (Section 7.1.1). Smoothing corrects the DR prediction by moving or rotating the remote avatar gradually to its correct position or angle (Section 7.1.2). There are two versions of this technique:

- **T2a**: *DR (1,10) and smoothing.* (1,10) means wait for 1 frame of packet loss before DR is activated and stay activated for a maximum of 10 frames.
- **T2b**: *DR (2,10) and smoothing.* (2,10) means wait for 2 frames of packets loss before DR is activated and stay activated for a maximum of 10 frames.

T3: *Visual field updating.* Disable the update of the remote avatar if it is outside the player's view (Section 7.1.3).

T4: *Avatar blinking.* Switch on the blink response for remote avatar for at most 3 seconds while waiting for a shoot result packets to arrive (Section 7.2.1).

T5: *Avatar dying.* Show a skull head image in front of the remote avatar after no response of 30 seconds, and remove the avatar if there is still no response after a further 30 seconds (Section 7.2.2).

T6: *Packets grouping.* Group packets together before sending them out (Section 7.3.1). There are two variants of this technique:

- **T6a**: *Packets grouping (2).* Group 2 packets or wait at most 2 frames before sending a single packet.
- **T6b**: *Packets grouping (3).* Group 3 packets or wait at most 3 frames before sending a single packet.

T7: *Duplicate packets.* Send the same packet multiple times in order to reduce the effect of packets loss (Section 7.3.2). There are two versions:

- **T7a**: *Duplicate packet.* Send the same packet twice.
- **T7b**: *Triplicate packet.* Send the same packet three times.

T8: *All techniques enabled.* Combine T2, T3, T4, T5, T6, and T7. The "a" or "b" cases are chosen depending on which is best for the response time result.

T9: *All techniques except avatar dying (T5).* Avatar dying is disabled because it does not have an effect on the response time but uses processing time.

T10: *All techniques except avatar dying (T5) and packets grouping (T6).* Packets grouping is disabled because it delays the response time.

T11: *All techniques except avatar dying (T5), packets grouping (T6), and visual field updating (T3).* Visual field updating is disabled because it only improves response time by a small amount compared to the other techniques.

Measurements

The following measurements are used to judge the effectiveness of the techniques:

M1: *One-way response time for multiple packets per keypress.* See if the technique improves the response time (Section 6.1.2).

M2: *Interval update time of a remote avatar movement or rotation.* Shows how a technique affects the interval update of the game (Section 6.1.3).

M3: *Average % of DR moving and rotating prediction error rate.* How accurate is DR for predicting the movement and rotation of a remote avatar (More detail in Section 8.1.3).

M4: *One-way response time for a single packet per keypress.* Check how one-way response time is affected by a technique (Section 6.1.1).

M5: *Two-way response time.* Shows how a technique affects two-way response time (Section 6.2).

M6: *Rendering update method.* Measure how a technique influences the processing in the *update()* method (Section 6.3).

M7: *Packet sending.* Report the average packet sending per second (Section 6.4.2).

M8: *Packet size.* How does a technique affect the average packet size (Section 6.4.1).

8.1 Game Measurement Results for 3 clients with a 90% Reliable Server

This section describes the measurement results for the game when using a 90% reliable server. First, I briefly explain the test techniques and their combinations. Then I describe the measures which include one-way response time, two-way response time, and frequency of packet sending. The results are presented in tables together with results analyses.

8.1.1 Measurement of One-way Response Time for Multiple Packets per Keypress (M1)

This measurement focuses on the response time of a moving or rotating remote avatar which generates multiple packets from one keypress. The packets in this category are MOVE, ROTATE, and KEY_RELEASE (Section 7.1.2).

Techniques that directly affect this measurement

1. "DR and smoothing" (T2) which predicts movement.
2. "Visual field" (T3) reduces the response time of processing load.
3. "Packets grouping" (T6) delays the response time.

Results

	M1 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	75.17	36677.18	2363
DR (1,10) (T2a)	**55.71 (1st)**	26429.29	2660
DR (2,10) (T2b)	60.07	29814.22	2947
Visual field (T3)	69.51	35523.65	2856
Grouping (2) (T6a)	79.25	32320.07	2845

	M1 Average mean (ms)	Variance mean (ms)	Total samples
Grouping (3) (T6b)	96.77	36598.53	2559
All (T8)	69.99	28759.27	3030
All-Dy-Group (T10)	65.70	32625.42	2876
All-Dy-Group-Visual (T11)	**63.05 (2nd)**	31424.84	2798

Table 8.1**:** One-way Response Time for Multiple Packets per Keypress

In the table, the bold rows are the best and the runner up results, labeled as (1st) and (2nd) respectively.

Analysis

Tech 1	T1	T1	T1	T1	T1	T1	T1	T1	T2a	T6a
Tech 2	T2a	T2b	T3	T6a	T6b	T8	T10	T11	T2b	T6b
Z-scores	**-3.86**	**-2.98**	-1.07	0.79	**3.96**	-0.38	-1.00	**-2.71**	0.97	**3.46**

Table 8.2: The z-scores of One-way Multiple

In the table, the bold z-scores mean the differences are significant.

T2a is significantly better than T1 because it predicts moves (which count as response time) when packets are lost, and is the best for this measurement. T2b is better than T1 because it also predicts moves, but less than T2a since it needs to wait on 2 frames.

T3 is better than T1 because it does not update the unseen remote avatars, but the improvement is not significant. This means that T3 does not help much for this measurement.

T6a is worse than T1 since it waits to group 2 packets before sending them, but the slow-down is not significant. However, T6b is significantly worse than T1 because it waits for 3 packets, and T6b has the worst effect of this measurement.

T8 is better than T1 because of T2a, but the difference is not significant due to the slowdown effect of T6a.

T10 is better than T1 and T8 because T6a is disabled, but the difference is not significant.

T11 is significantly better than T1 by disabling T6a and T3. This contrasts with enabling T3 individually. This shows that enabling multiple techniques has a processing time overhead that can ‘eat up’ any improvement offered by a technique used on its own.

In summary, dead reckoning (T2a) is the best single technique for improving one-way response time for multiple packets per keypress. If a combination of techniques are required (to improve other measurements), then T11 is the best choice.

8.1.2 Measurement of Interval Update Time of a Remote Avatar Movement or Rotation (M2)

A remote avatar is updated when MOVE or ROTATE packets arrive. Normally, the update time should be equal to the game frame rate, and any reduction is caused by MOVE or ROTATE packets not arriving (Section 7.1.3).

Techniques that directly affect this measurement

1. "DR and smoothing" predict movements which trigger updates.
2. "Packets grouping" reduces response time and the number of updates.

Results

	M2 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	73.50	3176.95	2066
DR (1,10) (T2a)	**58.79 (1st)**	2119.90	2373
DR (2,10) (T2b)	61.62	1901.27	2471
Grouping (2) (T6a)	70.57	2317.31	2592
Grouping (3) (T6b)	71.98	5991.12	2528
All (T8)	**68.23 (2nd)**	2467.44	2528
All-Dy-Group (T10)	70.07	1849.31	2601

Table 8.3**:** Interval Update Time of a Remote Avatar Movement or Rotation

Analysis

Tech 1	T1	T1	T1	T1	T1	T1	T2a	T6a
Tech 2	T2a	T2b	T6a	T6b	T8	T10	T2b	T6b
Z-scores	**-9.43**	**-7.82**	**-1.88**	-0.76	**-2.64**	-1.47	**2.19**	0.78

Table 8.4: The z-scores of Interval Update

T2a is significantly better than T1 because it adds extra moves when packets are lost, and is the best technique for this measurement. T2b is significantly better than T1, but less so than T2a because it waits 2 frames before activating.

T6a is significantly better than T1 since when packets arrive at the destination, it guarantees 2 updates. T6b is better than T1, but not significantly, and is less beneficial than T6a since it waits for 3 packets.

T8 is significantly better than T1 due to the effect of T2a and T6a, but less good than T2a alone because of the overhead multiple techniques have on processing time.

T10 is better than T1, but not significantly, and is less than T8 because T6a is disabled.

In summary, T2a (DR (1,10)) is the best single technique for improve the interval update. T8 is the best combination techniques.

8.1.3 Measurement of Average % of DR Moving, Rotating Prediction Error Rate (M3)

The formulas are:
% of DR Moving Prediction Error Rate = ((predicted move distance - real move distance) / real move distance)*100
% of DR Rotating Prediction Error rate = ((predicted rotate distance - real rotate distance) / real rotate distance)*100

This measurement, from the game tests, shows the % error in DR, compared to the real update position or angle.

Results

	M3 Average mean of Moving Prediction Error Rate (%)	Variance mean (%)	M3 Average mean of Rotating Prediction Error rate (%)	Variance mean (%)	Total samples
DR(1,10) (T2a)	**1.77 (1st)**	0.02	**14.43 (1st)**	1.10	188
DR(2,10) (T2b)	2.31	0.03	16.40	1.70	157

Table 8.5**:** Average % of DR Moving, Rotating Prediction Error Rate

Analysis

	Moving Prediction	Rotating Prediction
Tech 1	T2a	T2a
Tech 2	T2b	T2b
Z-scores	**31.31**	**15.25**

Table 8.6: The z-scores of DR Prediction Error Rate

T2a is significantly more accurate than T2b for both move error and rotate errors. This shows that predictions are more accurate when performed on each frame rather than every 2 frames.

8.1.4 Measurement of One-way Response Time for a Single Packet per Keypress (M4)

This measurement is of one-way response time when one packet is generated per keypress, as when the user moves onto a life spot or onto a bullets area (Section 7.1.1).

Techniques that directly affect this measurement

1. "Visual field" may reduce the processing time of a response.
2. "Duplicate/triplicate packets" reduce the chance of packet loss, which improves the response time.

Results

	M4 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	89.58	45385.98	50
Visual Field (T3)	82.98	32340.01	49
Duplicate (T7a)	20.40	1359.73	50
Triplicate (T7b)	14.28	20.51	52
All (T8)	**14.14 (1st)**	65.69	51
All-Dy-Group-Visual (T11)	**15.42 (2nd)**	33.55	55

Table 8.7**:** One-way Response Time for a Single Packet per Keypress

Analysis

Tech 1	T1	T1	T1	T1	T1	T7a	T8
Tech 2	T3	T7a	T7b	T8	T11	T7b	T11
Z-scores	-0.17	**-2.26**	**-2.50**	**-2.50**	**-2.46**	-1.17	0.93

Table 8.8: The z-scores of One-way Single

T3 is better than T1 but not significantly, so reducing the updating load does not much help response time.

T7a is significantly better than T1 because sending the same packet twice reduces the chance of packet lost. T7b is also significantly better than T1 because it sends three packets at once. Although T7b is better than T7a, the difference is not significant, and sending 3 packets at once is best avoided since it increases the load on the network In that case, it is better to choose T7a rather than T7b.

T8 is significantly better than T1 because it includes T7a and T3. T8 is the best technique according to this measurement.

T11 is significantly better than T1 but its difference from T8 is not significant. This means that disabling T3 does not much affect response time.

In summary, T8 is the best way to improve one-way response time for a single packet per keypress, and T11 is the runner up.

8.1.5 Measurement of Two-way Response Time (M5)

Two-way response time is the time during a local player doing something to a remote avatar and seeing the change in his/her local game world. An example is shooting at a remote avatar and seeing an explosion (Section 7.2).

Techniques that directly affect this measurement

1. "Avatar blinking" makes the remote avatar blink, which counts towards the response time.
2. "Duplicate/triplicate packets" reduces the chance of packet loss, which improves response time.

Results

	M5 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	230.50	227159.00	52
Blinking (T4)	**68.53 (2nd)**	635018.90	502

	M5 Average mean (ms)	Variance mean (ms)	Total samples
Duplicate (T7a)	80.16	35595.38	60
Triplicate (T7b)	48.18	10894.69	60
All (T8)	**18.75 (1st)**	5714.69	154

Table 8.9**:** Two-way Response Time Measurement

Analysis

Tech 1	T1	T1	T1	T1	T7a
Tech 2	T4	T7a	T7b	T8	T7b
Z-scores	**-2.16**	**-2.13**	**-2.70**	**-3.15**	-1.15

Table 8.10: The z-scores of Two-way

T4 is significantly better than T1 because it makes the remote avatar blink while waiting for a shooting response.

T7a is significantly better than T1 because it sends the same packet twice which reduces the chance of packet lost. T7b is also significantly better than T1 by sending the same packet three times. The difference between T7a and T7b is not significant so T7a is preferable since it reduces network's load.

T8 is significantly better than T1 since it includes T4 and T7a, and this offers the best overall improvement.

In summary, T8 is the best way to improve two-way response time, and T4 is the runner up.

8.1.6 Measurement of Rendering Update method (M6)

This measurement considers *update()* and *paint()* to see if the techniques affect their processing times (Section 7.3).

Techniques that directly affect this measurement

T1 to T5 are measured, but not network techniques (T6 and T7) since they are processed in *ClientHandler*.

Results

	M6 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	**13.34 (1st)**	60.75	15895
DR(1,10) (T2a)	15.52	49.90	21781
DR(2,10) (T2b)	15.70	30.64	18452
Visual Field (T3)	15.74	31.99	18244
Blinking (T4)	15.02	49.70	17416
Dying (T5)	15.22	44.69	16834
All (T8)	16.07	35.20	17458
All-Dy (T9)	15.85	48.94	16483
All-Dy-Group-Visual (T11)	**15.24 (2nd)**	54.06	14667

Table 8.11**:** Rendering Update Method Measurement

Analysis

Tech 1	T1	T1	T1	T1	T1	T1	T1	T1
Tech 2	T2a	T2b	T3	T4	T5	T8	T9	T11
Z-scores	**-27.98**	**-31.83**	**-32.14**	**-20.56**	**-23.36**	**-26.49**	**-31.12**	**-18.55**

Table 8.12: The z-scores of Rendering Update Method

All the techniques considered here increase the processing time of *update()* and *paint()*. However, the slowdowns are only a few milliseconds (up to 3 ms) which are hard for a player to notice.

8.1.7 Packets Sending (M7)

This measurement shows the frequency of packets sending (Section 7.4.2).

Techniques that directly affect this measurement

1. “Packet grouping” groups multiple packets before sending them out.
2. “Duplicate/triplicate packets” sends the same packet twice/three times.

Results

	M7 Average Packets Sending (packets/sec)	Variance of Packets Sending (packets/sec)	Total Samples
No techniques (T1)	7.43	51.15	3389
Grouping (2) (T6a)	**4.99 (2nd)**	23.32	2298
Grouping (3) (T6b)	3.67	13.18	1733
Duplicate (T7a)	7.31	49.75	3613
Triplicate (T7b)	7.86	56.87	4023
All (T8)	**4.83 (1st)**	24.18	2609
All-Dy-Group (T10)	7.55	52.03	3738

Table 8.13**:** Packets Sending Measurement

Analysis

Tech 1	T1	T1	T1	T1	T1	T1	T6a	T7a
Tech 2	T6a	T6b	T7a	T7b	T8	T10	T6b	T7b
Z-scores	**-15.36**	**-24.96**	-0.71	**2.52**	**-16.66**	0.70	**-9.91**	**3.29**

Table 8.14 The z-scores of Packets Sending

T6a is significantly better than T1 because it waits for 2 frames before sending a packet. T6b is significantly better than T1 and T6a because it waits for 3 frames. T6b performs best for this measurement, but it has the drawback of slowing down response time. For this reason, T6a is chosen instead.

The difference between T7a and T1 is not significant although the packets are sent twice. This is due to the fact that single packet per keypress packets occur much less frequently than multiple packets per keypress packets. T7b is significantly worse than T1 because it sends three packets at once.

T8 is significantly better than T1 due to the effect of T6a. Although T7a is enabled which could increase the packets sending, T6a has more effect than T7a because multiple packets per keypress typed packets are sent much more than the single packet per keypress.

T10 does not help because it disables T6a (it is worse than T1, but not significantly).

In summary, T8 is the best technique for reducing packets sending. T6a is the runner up.

8.1.8 Packets Size (M8)

This measurement judges how the techniques affect packet size (Section 7.4.1).

Techniques that directly affect this measurement

Packet grouping.

Results

	M8 Average Packets Size (Bytes)	Variance of Packets Size (Bytes)	Total Samples
No techniques (T1)	**43.09 (1st)**	97.51	3389
Grouping (2) (T6a)	82.98	775.37	2298
Grouping (3) (T6b)	101.21	2323.84	1733
All (T8)	80.34	807.04	2609
All-Dy-Group (T10)	**42.92 (2nd)**	96.43	3738

Table 8.15**:** Packets Size Measurement

Analysis

Tech 1	T1	T1	T1	T1	T6a
Tech 2	T6a	T6b	T8	T10	T6b
Z-scores	**67.57**	**50.34**	**64.06**	-0.73	**13.92**

Table 8.16: The z-scores of Packets Size

T6a and T6b are significantly worse than T1 because they group packets before sending.

T8 is significantly worse than T1 because it includes T6a.

T10 does not have any effect on packet size because it disables T6a.

In summary, packets grouping (T6a and T6b) adversely affect this measurement because they make the packet size bigger.

8.1.9 Summary of a 90% Reliable Server

The best and the runner up techniques for each measurement from M1 to M8 for a 90% reliable server are shown in Table 8.17.

	Recommendation	Runner up
M1	**T2a (2nd)**	T11
M2	**T2a (2nd)**	**T8 (1st)**
M3	T2a	T2b
M4	**T8 (1st)**	T7a
M5	**T8 (1st)**	T4
M6	T1	T11
M7	T8	T6a
M8	T1	T10

Table 8.17**:** Summary Detail of a 90% Reliable Server

To decide which technique is the best from the eight measurements, we must consider the measurements that affect response time, which are M1, M2, M4, and M5. M3 is not included because it considers DR.

From Table 8.17, T8 is the best technique to improve response time (it is best for M4 (Table 8.7) and M5 (Table 8.9), and runner up for M2 (Table 8.3)). This means that enabling all techniques can improve the response time better than enabling individual techniques. Although enabling all techniques leads to an increase in the processing time of *update()* and *paint()*, the slowdown is not noticeable for a player.

The runner up technique is DR (T2a) with two recommendations from M1 (Table 8.1) and M2 (Table 8.3). This means that in comparisons among individual techniques, DR is the best for improving response time.

8.2 Game Measurement Results for 3 clients with a 75% Reliable Server

This section gives the measurement results for a game using a 75% reliable server. The same techniques (T) and measurements (M) as those listed at the start of this chapter are used again, with the exclusive of M3 since it was already considered in section 8.1.3.

The other difference is for the multiple techniques (T8 to T11). T7b (triplicate packets) is chosen instead of T7a (duplicate packets) because T7b performs significantly better at this level of reliability.

8.2.1 Measurement of One-way Response Time for Multiple Packets per Keypress (M1)

The measurement is the response time of a moving or rotating remote avatar which generates multiple packets from one keypress (Section 7.1.2).

Techniques that directly affect this measurement

1. "DR and smoothing" (T2) which predicts movements.
2. "Visual field" (T3) reduces the response time of processing load.
3. "Packets grouping" (T6) delays the response time.

Results

	M1 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	180.35	82106.63	2351
DR (1,10) (T2a)	**133.92 (1st)**	68612.63	3067
Visual field (T3)	166.39	79989.16	2384
Grouping (2) (T6a)	182.95	82623.80	2247
All (T8)	162.71	76718.38	2453
All-Dy-Group (T10)	157.85	73580.44	2367
All-Dy-Group-Visual (T11)	**143.57 (2nd)**	73596.05	2982

Table 8.18**:** One-way Response Time for Multiple Packets per Keypress

Analysis

Tech 1	T1	T1	T1	T1	T1	T1
Tech 2	T2a	T3	T6a	T8	T10	T11
Z-scores	**-6.13**	**-1.69**	0.31	**-2.16**	**-2.77**	**-4.76**

Table 8.19: The z-scores of One-way Multiple

T2a is significantly better than T1 since it predicts moves when packets are lost, and this is the best for this measurement. T2b is better than T1 because it also predicts moves, but less than T2a since it needs to wait on 2 frames.

T3 is significantly better than T1 because it does not update the unseen remote avatars.

T6a is worse than T1 since it waits to group two packets before sending them, but the slow down is not significant.

T8 is significantly better than T1 because of T2a.

T10 is significantly better than T1 and also T8 because T6a is disabled.

T11 is significantly better than T1 by disabling T6a and T3. This contrasts with enabling T3 individually. This shows that enabling multiple techniques can consume a processing time that overrides any improvements offered by its technique.

In summary, dead reckoning (T2a) is the best single technique to improve one-way response time for multiple packets per keypress, and T11 is the best combination technique.

8.2.2 Measurement of Interval Update Time of a Remote Avatar Movement or Rotation (M2)

A remote avatar is updated when MOVE or ROTATE packets arrive. The update time should be equal to the game frame rate, but the reduction of update time may be caused by MOVE or ROTATE packets not arriving (Section 7.1.3).

Techniques on this measurement

1. “DR and smoothing” predict movements and make the updates.
2. “Packets grouping” reduces response time and the number of updates.

Results

	M2 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	111.29	11337.14	1558
DR (1,10) (T2a)	**96.40 (2nd)**	12435.63	1655
Grouping (2) (T6a)	101.12	12587.64	1672
All (T8)	**91.04 (1st)**	10353.63	1649
All-Dy-Group (T10)	102.75	18034.75	1400

Table 8.20**:** Interval Update Time of a Remote Avatar Movement or Rotation

Analysis

Tech 1	T1	T1	T1	T1
Tech 2	T2a	T6a	T8	T10
Z-scores	**-3.87**	**-2.64**	**-5.50**	**-1.90**

Table 8.21: The z-scores of Interval Update

T2a is significantly better than T1 because it adds extra moves when packets are lost.

T6a is significantly better than T1 since when packets arrive at the destination, it guarantees two updates.

T8 is significantly better than T1 due to the effect of T2a and T6a, and is also the best technique on this measurement.

T10 is significantly better than T1, but less than T8 because T6a is disabled.

In summary, T8 is the best combination techniques for improve the interval update and T2a (DR (1,10)) is the best single technique.

8.2.3 Measurement of One-way Response Time for a Single Packet per Keypress (M4)

This measures one-way response time when one packet is generated per keypress, as when the user moves onto a life spot or onto a bullets area (Section 7.1.1).

Techniques on this measurement

1. “Visual field” may reduce the process time of a response.
2. “Duplicate/triplicate packets” reduces the chance of packets loss.

Results

	M4 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	227.44	95258.55	34
Visual Field (T3)	171.39	88881.93	49
Duplicate (T7a)	69.88	29576.56	48
Triplicate (T7b)	**41.04 (1st)**	10543.08	46
All (T8)	**52.11 (2nd)**	22898.86	45
All-Dy-Group-Visual (T11)	98.13	40922.14	51

Table 8.22**:** One-way Response Time for Single Packet per Keypress

Analysis

Tech 1	T1	T1	T1	T1	T1	T8
Tech 2	T3	T7a	T7b	T8	T11	T11
Z-scores	-0.83	**-2.70**	**-3.39**	**-3.05**	**-2.15**	1.27

Table 8.23: The z-scores of One-way Single

T3 is better than T1 but not significantly, so reducing the updating load does not much help response time.

T7a is significantly better than T1 because it sends the same packet twice, which reduces the chance of packet lost. T7b is also significantly better than T1 because it sends three packets at once. T7b is significantly better than T7a. Because the difference is significant, T7b is the best technique of this measurement.

T8 is significantly better than T1 because it includes T7b and T3.

T11 is significantly better than T1 but its difference from T8 is not significant. This means disabling T3 does not much affect response time.

In summary, T7b (triplicate packets) is the best way to improve one-way response time for a single packets per keypress, and T8 is the runner up.

8.2.4 Measurement of Two-way Response Time (M5)

Two-way response time is the time between a local player doing something to a remote avatar and seeing the change in his/her local game world (Section 7.2).

Techniques on this measurement

1. "Avatar blinking" makes the remote avatar blink, which counts towards the response time.
2. "Duplicate/triplicate packets" reduces the chance of packet loss.

Results

	M5 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	469.66	304675.60	32
Blinking (T4)	**21.99 (1st)**	33334.34	797
Duplicate (T7a)	371.43	426347.80	51
Triplicate (T7b)	232.57	192906.70	60
All (T8)	**44.23 (2nd)**	56307.80	302

Table 8.24**:** Two-way Response Time Measurement

Analysis

Tech 1	T1	T1	T1	T1
Tech 2	T4	T7a	T7b	T8
Z-scores	**-4.58**	-0.73	**-2.10**	**-4.32**

Table 8.25: The z-scores of Two-way

T4 is significantly better than T1 because it makes the remote avatar blink while waiting for a shooting response, and it is the best technique for this measurement.

T7a is better than T1 because it sends the same packet twice which reduces the chance of packet lost, but the difference is not significant. This means T7a does not help much for a 75% reliable server. T7b is also significantly better than T1 by sending the same packet three times.

T8 is significantly better than T1 since it includes T4 and T7b, but less than T4. This means that it suffers from processing time overhead for enabling multiple techniques.

In summary, T4 (avatar blinking) is the best way to improve two-way response time, and T8 is the runner up.

8.2.5 Measurement of Rendering Update method (M6)

This measurement considers *update()* and *paint()* to see if the techniques affect their processing times (Section 7.3).

Techniques on this measurement

T1 to T5 are measured.

Results

	M6 Average mean (ms)	Variance mean (ms)	Total samples
No techniques (T1)	**13.51 (1st)**	53.92	16736
DR(1,10) (T2a)	14.59	62.93	16931
Visual Field (T3)	15.45	44.87	16453
Blinking (T4)	15.15	42.76	15809
Dying (T5)	14.79	47.67	18068
All (T8)	15.42	62.77	16464
All-Dy (T9)	15.41	54.71	15688
All-Dy-Group-Visual (T11)	**14.03 (2nd)**	63.70	15983

Table 8.26**:** Rendering Update Method Measurement

Analysis

Tech 1	T1	T1	T1	T1	T1	T1	T1
Tech 2	T2a	T3	T4	T5	T8	T9	T11
Z-scores	**12.97**	**25.15**	**21.30**	**16.72**	**22.77**	**23.20**	**6.13**

Table 8.27: The z-scores of Rendering Update Method

All the techniques in Table 8.26 increase the processing time of *update()* and *paint()*. However, the slowdowns are hard for a player to notice.

8.2.6 Packets Sending (M7)

This measurement shows how the techniques affect the frequency of packets sending (Section 7.4.2).

Techniques that directly affect this measurement

1. "Packet grouping" groups multiple packets before sending them out.
2. "Duplicate/triplicate packets" sends the same packet twice/three times.

Results

	M7 Average Packets Sending (packets/sec)	Variance of Packets Sending (packets/sec)	Total Samples
No techniques (T1)	7.16	47.65	3606
Grouping (2) (T6a)	**5.11 (2nd)**	23.83	2387
Duplicate (T7a)	7.48	52.23	3549
Triplicate (T7b)	7.60	52.75	3912
All (T8)	**4.77 (1st)**	23.69	2493
All-Dy-Group (T10)	7.51	53.12	3635

Table 8.28**:** Packets Sending Measurement

Analysis

Tech 1	T1	T1	T1	T1	T1	T7a	T6a
Tech 2	T6a	T7a	T7b	T8	T10	T7b	T8
Z-scores	**-13.46**	**1.91**	**2.69**	**-15.86**	**2.10**	0.71	**2.44**

Table 8.29: The z-scores of Packets Sending

T6a is significantly better than T1 because it waits for two frames before sending a packet.

T7a is significantly worse than T1 because the packets are sent twice. T7b is significantly worse than T1 and T7a because it sends three packets at once.

T8 is significantly better than T1 due to the effect of T6a, and is also the best for this measurement. Although T7b could increase packets sending, T6a has more effect because multiple packets per keypress typed packets are sent much more often than single packet per keypress.

T10 is significantly worse than T1 because T6a is disabled.

In summary, T8 is the best technique to reduce packets sending, and T6a is the runner up.

8.2.7 Packets Size (M8)

This measurement shows how the techniques affect packet size (Section 7.4.1).

Techniques that directly affect this measurement

Packet grouping.

Results

	M8 Average Packets Size (Bytes)	Variance of Packets Size (Bytes)	Total Samples
No techniques (T1)	**42.45 (1st)**	97.74	3606
Grouping (2) (T6a)	83.30	770.23	2387
All (T8)	78.26	816.93	2493
All-Dy-Group (T10)	**44.16 (2nd)**	97.23	3635

Table 8.30**:** Packets Sending Measurement

Analysis

Tech 1	T1	T1	T1
Tech 2	T6a	T8	T10
Z-scores	**69.07**	**60.12**	**7.37**

Table 8.31: The z-scores of Packets Sending

The z-scores here are big because of the huge samples.

T6a is significantly worse than T1 because it groups packets before sending them.

T8 is significantly worse than T1 because of T6a.

T10 does not have effect on the packets size because it disables T6a.

In summary, packets grouping (T6a) affects this measurement the most by making packet sizes bigger.

8.2.8 Summary of a 75% Reliable Server

The recommendation and the runner up techniques on each measurement from M1 to M8 of a 75% reliable server are shown in Table 8.32.

	Recommendation	Runner up
M1	**T2a (2nd)**	T11
M2	**T8 (1st)**	**T2a (2nd)**
M4	T7b	**T8 (1st)**
M5	T4	**T8 (1st)**
M6	T1	T11
M7	T8	T6a
M8	T1	T10

Table 8.32**:** Summary Detail of a 75% Reliable Server

To decide which technique is the best from the eight measurements, we must consider the measurements that affect response time, which are M1, M2, M4, and M5.

From Table 8.32, T8 is the best technique to improve response time (it is the best for M2 (Table 8.20), and runner up for M4 (Table 8.22) and M5 (Table 8.24)). This means that enabling all the techniques can improve the response time better than enabling an individual technique, a finding similar to the 90% case.

The runner up of the summary is DR (T2a) with one recommendation from M1 (Table 8.18) and a runner up position for M2 (Table 8.20). This means that in comparisons among individual techniques, DR is the best for improving response time. This result is similar to what we found in the 90%.

8.3 Additional Tests

This section investigates the effects of increasing the number of players, and delay on game play, and how the techniques handle these more severe situations.

The tests were carried out on localhost since having many clients on one PC increases the game load more severly than having one client per PC. The machine specification is:

- Pentium 4 2.4 GHz
- 1.25 GB of RAM
- ATI Radeon 9550
- Windows XP
- WTK 2.5

Three players and five players with a 75% of reliable server are considered (five clients is the maximum number supported by the test PC). The comparison is between no techniques (T1) and all techniques except avatar dying (T9). It was dropped to stop remote avatars disappearing before taking their turn.

Triplicate packets are utilized rather than duplicate packets because 75% reliability triggers loss severe packet.

8.3.1 Three players with 75% reliable server

One-way Response Time for Multiple Packets per Keypress (M1)

Techniques that directly affect this measurement:
DR and smoothing, Visual field, Packets grouping.

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	218.53	96595.85	2535
All-Dy (T9)	**166.69 (1st)**	69839.66	3725

Table 8.33: One-way Response Time for Multiple Packets per Keypress

The z-score between T1 and T9 is **-6.88**. (The bold number means z-score is significant.)

T9 is significantly better than T1 because DR predicts moves when packets are lost and the visual field does not update the many unseen remote avatars.

Interval Update Time of a Remote Avatar Movement or Rotation (M2)

Techniques that directly affect this measurement:

DR and smoothing**,** Packets grouping**.**

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	129.52	14255.11	1784
All-Dy (T9)	**92.21 (1st)**	9975.46	2824

Table 8.34: Interval Update Time of a Remote Avatar Movement or Rotation

The z-score between T1 and T9 is **-10.99**.

T9 is significantly better than T1 because of DR**.**

One-way Response Time for a Single Packet per Keypress (M4)

Techniques that directly affect this measurement:

Visual field**,** Triplicate packets**.**

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	223.48	68481.95	46
All-Dy (T9)	**59.19 (1st)**	5170.08	42

Table 8.35: One-way Response Time for a Single Packet per Keypress

The z-scores between T1 and T9 is **-4.09**.

T9 is significantly better than T1 because triplicate packets overcome the problems with high packet loss.

Two-way Response Time (M5)

Techniques that directly affect this measurement:

Avatar blinking**,** Triplicate packets.

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	632.82	484716.6	22
All-Dy (T9)	**51.98 (1st)**	45927.81	171

Table 8.36: Two-way Response Time

The z-scores between T1 and T9 is **-3.89**.

T9 is significantly better than T1 because avatar blinking makes the remote avatar blink while waiting for a shooting response and, triplicate packets reduces the chances of packets lost.

8.3.2 Five players with 75% reliable server

One-way Response Time for Multiple Packets per Keypress (M1)

Techniques that directly affect this measurement:
DR and smoothing, Visual field, Packets grouping

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	259.56	83865.25	1324
All-Dy (T9)	251.75	93582.68	1559

Table 8.37: One-way Response Time for Multiple Packets per Keypress

The z-scores between T1 and T9 is -0.70.

None of the techniques help because the client machine has reached its processing limit with 5 players, and the techniques only add more work. The small improvement probably is DR.

Interval Update Time of a Remote Avatar Movement or Rotation (M2)

Techniques that directly affect this measurement:

DR and smoothing**,** Packets grouping**.**

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	226.27	50144.03	1101
All-Dy (T9)	**157.94 (1st)**	27546.11	1449

Table 8.38: Interval Update Time of a Remote Avatar Movement or Rotation

The z-scores between T1 and T9 is **-8.50**.

T9 is significantly better than T1 because of DR. Although one-way multiple is not much affected by DR because of processing time constraints, at least DR can maintain responsiveness by creating extra moves between the updates.

One-way Response Time for a Single Packet per Keypress (M4)

Techniques that directly affect this measurement:

Visual field**,** Triplicate packets**.**

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	283.76	86847.82	17
All-Dy (T9)	191.88	30374.28	25

Table 8.39: One-way Response Time for a Single Packet per Keypress

The z-score between T1 and T9 is -1.16.

T9 is not significantly better than T1. The techniques are affected by processing limitation and can't work as well as in the one-way multiple case. The minor speedups are probably to the triplicate packets.

Two-way Response Time (M5)

Techniques that directly affect this measurement:

Avatar blinking**,** Triplicate packets.

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	995.35	670961.5	17
All-Dy (T9)	**118.91 (1st)**	99551.24	95

Table 8.40: Two-way Response Time

The z-score between T1 and T9 is **-4.35**.

T9 is significantly better than T1 because of avatar blinking and triplicate packets. This means that two-way response time is more affected by network issues than processing limits.

8.3.3 Increasing the random delay of packets for three players using a 75% reliable server

These tests increase the random delay of packets from between 30 ms - 2 seconds to between 2 - 5 seconds. The tests were carried out on localhost with three players and a 75% reliable server. We compared the results with no techniques enable (T1) with the results when using all the techniques except for avatar dying (T9).

One-way Response Time for Multiple Packets per Keypress (M1)

Techniques that directly affect this measurement:
DR and smoothing, Visual field, Packets grouping.

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	518.23	558133.50	3192
All-Dy (T9)	**409.28 (1st)**	483568.60	4936

Table 8.41: One-way Response Time for Multiple Packets per Keypress

The z-score between T1 and T9 is **-6.60**.

T9 is significantly better than T1 because of DR predictions and visual field updating. This shoes that DR is still useful even when packet delay is increased drastically. Also, the massive amount of packet sending for this type of response time, means that packet delay tends to have less effect since many packets are still delivered successfully.

Interval Update Time of a Remote Avatar Movement or Rotation (M2)

Techniques that directly affect this measurement:

DR and smoothing**,** Packets grouping**.**

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	147.81	57131.58	2140
All-Dy (T9)	**88.71 (1st)**	32156.77	3674

Table 8.42: Interval Update Time of a Remote Avatar Movement or Rotation

The z-score between T1 and T9 is **-9.93**.

T9 is significantly better than T1 because of DR**.** Also, a comparison with the interval update times for packets delayed by the usual amount (Table 8.3.4) shows that there is very little change (130 ms compared to 148 ms). This shows that interval update time is not much affected by packet delay because of the massive numbers of packets which are delivered.

One-way Response Time for a Single Packet per Keypress (M4)

Techniques that directly affect this measurement:

Visual field**,** Triplicate packets**.**

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	545.07	494527.20	42
All-Dy (T9)	**212.90 (1st)**	186402.10	57

Table 8.43: One-way Response Time for a Single Packet per Keypress

The z-score between T1 and T9 is **-2.71**.

T9 is significantly better than T1 because of triplicate packets. This type of response time is affected by increased packet delay, but triplicate packets compensate, and are an effective form of help.

Two-way Response Time (M5)

Techniques that directly affect this measurement:

Avatar blinking**,** Triplicate packets.

	Average means (ms)	Variance mean (ms)	Total samples
No techniques (T1)	2035.33	5293069	42
All-Dy (T9)	**75.77 (1st)**	178234.40	418

Table 8.44: Two-way Response Time

The z-score between T1 and T9 is **-5.51**.

T9 is significantly better than T1 because of avatar blinking and triplicate packets. Two-way response time is the most affected by packet delay because the packets need to travel in two directions and their sending frequency sending is quite low.

8.3.4 Summary

This section's tests shows how the techniques help when the number of players is increased, and packet delay times are lengthened. Increasing the players and limiting processing affect the techniques that require extra processing time such as DR. The techniques do not improve one-way response time because the system is already consuming most of processing time, even in the one-way single case with triplicate packets (Table 8.39). However, avatar blinking and triplicate packets do help with two-way response time, since it is affected more by the networking state than limited processing.

When packet delays are increased, one-way multiple and interval updates tend to have less affect because of massive packets sending, but DR remains effective. Two-way response time is

affected the most by increasing packet delays, but avatar blinking and triplicate packets help improve matters.

8.4 Test Summaries

This chapter provided the game measurement results. The test were carried out on LAN with three WTK clients, and done once for a 90% reliable server and again for a 75% reliable server. The techniques were enabled one-by-one, or in combinations, and compared with the results when no techniques were used. Z-tests were utilized to judge the significance of the comparisons.

For both 90% and 75% reliable servers, enabling all the techniques improves the response times the most. For some individual tests, a single technique has the best effect. For example, in the 90% case, DR is the best for one-way multiple and interval updates; in the 75% case, triplicate packets helps the most for one-way single, and avatar blinking for two-way response time.

When there are more players or limited processing time, time consuming techniques such as DR has less effect on one-way response time. Two-way response time seems to be less affected by increased loads. One-way multiple and interval updates are less affected by increased packet delay times because of massive packets sending. Two-way response time is affected the most, but avatar blinking and triplicate packets help.

Enabling multiple techniques at once introduces additional processing overheads in the game, which reduce its response time (especially for one-way multiple and rendering), but the effects a few milliseconds are too small to be noticed by a player.

Enabling multiple techniques is preferable to employing individual techniques because they improve many types of response times at once, which is more effective than using one technique to improve one type of response.

From the player's point of view, penguin movement or rotation is quite responsive even with very bad packet delay or loss, due to the massive amount of packets sending. However, DR noticeably helps the penguin move more smoothly. Shooting is the most impacted by packet delay and loss, since shooting packets need to travel round-trip between two clients. In this case, avatar blinking improves responsiveness and triplicate packets increase the chance of successful packet delivery.

CHAPTER 9

SUMMARY

A series of experiments were carried out on a client/server 3D mobile first-person shooter (FPS) to determine the best techniques for improving client-side response times in the presence of severe network unreliability. Three measures of response time were utilized, to deal with the different types of communication employed among the clients. The response time techniques were grouped into three categories: general techniques, game-specific techniques, and packet-based techniques. A combination of all three types of technique – dead reckoning and smoothing, visual field updating, avatar blinking, avatar dying, packets grouping, and duplicate/triplicate packet – produce mean response times that are 20% to 90% less than the mean response time for the game with no techniques enabled.

9.1 Game Architecture

The game's client/server architecture is a typical multiplayer mobile game. The J2ME game clients each render a world of competing penguins; the goals of a player's penguin are to find "life spots", gather bullets, and shoot other penguins. The game's architecture is summarized in Figure 9.1.

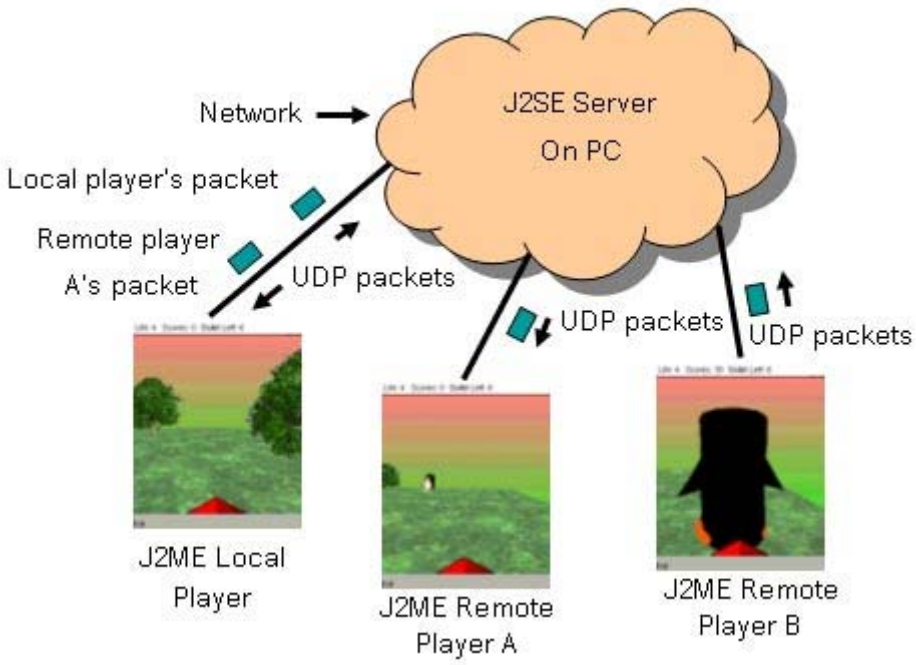


Figure 9.1: The Game's Architecture

The local player has a first-person view of a world, while the other penguins are remote avatars representing the other players. In Figure 9.1, the game currently has three users, so each player can see at most two other penguins (and its own penguin's red beak).

The rules of the game ensure that player behavior is fairly complicated, making it hard to predict a player's actions and the pattern of network activity. All the game's 3D assets (e.g. the penguins, the floor) are stored locally on the clients; no 3D models are transmitted via the network.

Game entry, inter-client communication, and game departure are controlled through a J2SE server which manages the delivery of data in the form of UDP packets. The server can be configured to delay packet delivery, and to lose a given percentage of datagrams, in order to test the game's responsiveness at different levels of network reliability. The system was run across a LAN. Therefore, real-world latency, bandwidth restrictions, and packet loss were not issues.

Various levels of reliability were investigated, including 90% reliability, which means that there was a 10% chance of a packet being delayed (i.e. one chance in ten) or lost. 75% reliability means that there is a 25% chance of packet delay or loss. A packet can be delayed between 30 ms and 2 seconds.

9.2 Measuring Response Time

A more accurate reflection of a game's responsiveness can be gained by measuring three slightly different forms of response time: one-way response time for *single* packet actions, one-way response time for *multiple* packet actions, and two-way response time.

One-way response time for an action is the time that a packet representing the action takes to travel from a remote player to the local player, and includes the time to update the remote player's avatar on the local device.

Some complicated types of action require multiple packets to be transmitted, typically for updating avatar position and orientation. However, most actions can be represented by single packets, such as when the player loses a life point or picks up a bullet. This distinction between multiple and single packets is important since it highlights the effectiveness of techniques which group, delete, or duplicate packets.

Two-way response time is the time for a packet to be sent from the local player to a remote device to be processed, and for a response packet to arrive back at the local player and update his game state. An example of two-way response time in the game is when a player shoots at a penguin. This requires that a message be sent to the remote client represented by the penguin, and for the local client to wait until the shot's outcome (e.g. penguin death) is returned.

There are other elements worth measuring to help judge the techniques: interval update time for remote avatar movement/rotation (the average time gap between updates of the avatar's position or rotation), rendering update speed (scene rendering), and various packet statistics (packet size and packet sending rate).

9.3 Techniques for Improving Response Times

A large number of techniques were tested to see which improved the game's response times. These techniques can be classified into three groups:

1. *General* techniques, which can be applied to any networked FPS. They include dead reckoning and smoothing, and visual field updating.

2. *Game-specific* techniques, which include avatar blinking and avatar dying (e.g. painting a translucent skull over a penguin to indicate its probable death).

3. *Packets-based* techniques, which include duplicate and triplicate packet sending, and packet grouping.

9.3.1 Dead Reckoning and Smoothing

Dead reckoning (DR) is used to 'guess' a penguin's translation or rotation when the packets holding that information have failed to arrive at the client. DR is activated after one movement packet is lost, and to keep it switched on for at most ten screen updates.

This approach requires packets to be time-stamped, and for a client to estimate how long it has to wait before a packet is judged to be lost. The code must also deal with a 'lost' packet turning up after a lengthy delay.

DR is switched on promptly, after only one packet has been lost, so a penguin will keep moving rather than appear unresponsive. DR is switched off after at most ten updates (500 ms in the game), since it becomes very difficult to predict movement accurately after multiple updates.

It is essential to pair DR with smoothing. When a movement packet eventually arrives, smoothing gradually adjusts the penguin's position to relocate and reorientate it to the correct spot. Smoothing is carried out over several screen updates, so a penguin does not 'jump' from one position to another.

9.3.2 Visual Field Updating

This technique uses the range of player's vision to update remote avatar. If the remote avatar is far beyond the range of view, e.g. behind the player, then there is no need to update its position or

angle if it moves. Related techniques, such as DR and smoothing, are deactivated if the remote avatar is not in the view range. When the remote avatar enters the vision area, it will be updated to its new position/angle.

9.3.3 Avatar Blinking

Avatar blinking is triggered when the local player shoots at a penguin, and the client has to wait for the shooting reply from the remote player. The uncertainty about a penguin's future is denoted by making it blink. This offers immediate feedback to the player, which is more reassuring than have nothing change on screen for perhaps several seconds.

After usability tests, we determined that players find blinking to be helpful for at most a few seconds, after which time it becomes rather irritating. Consequently, a penguin can blink for at most three seconds, which is enough time for a shooting response to arrive when the network is performing at 75% reliability.

9.3.4 Avatar Dying

Avatar dying makes a transparent skull image appear in front of the remote avatar when no packet is received from this remote player for 30 seconds. After this, the remote avatar will be removed from the local player's scene if there is no packet received from the remote player for another 30 seconds (a total of 1 minute of inactivity). This handles the situation when the remote client quits the game without informing the server.

9.3.5 Packets Grouping

This technique groups multiple packets into one packet and sends them out at once. The benefits of packets grouping are that it can reduce the frequency of packet sending and reduce the use of UDP headers. Packets grouping is used on packets which are most frequently sent, e.g. move or rotation messages. A group can consist of either two or three packets.

9.3.6 Duplicate/Triplicate Packet Sending

Duplicate/triplicate packet sending makes a client transmit the same packet two or three times to reduce the chance of it being lost. One drawback is that the receiver must be able to detect and ignore multiple packet copies. Also, indiscriminate multiple packet sending is a serious consumer of bandwidth. Consequently, this technique is used sparingly, only for important information whose loss would seriously impact the game. Such packets which are related to important avatar state changes, such as when a penguin loses life points, or shoots at another penguin. It also helps to correlate the amount of resending to the unreliability of the network.

9.4 Results

The game was run several hundred times with three clients, and results gathered over several minutes of typical game play in each game, and averaged. The tests reported here were carried out with the network set to be 90% and 75% reliable.

Three response times measurements were performed: one-way response time for multiple packets per keypress, one-way response time for a single packet per keypress, and two-way response time. The other measure elements are interval update time of a remote avatar movement or rotation, rendering update method, packet sending, and packet size.

The mean response times were calculated when no techniques were applied, and again when each of the techniques was switched on individually (e.g. DR and smoothing, avatar blinking, and duplicate/triplicate packets). Finally, the techniques were switched on together with various combinations.

The mean response times for the techniques were compared with the mean time when no techniques were enabled, using a standard one-tailed z-test with a 95% level of significance.

The tests consist of 11 combinations of different techniques (T1 to T11) with 8 measurements (M1 to M8). The definitions of each T and M were defined at the beginning of Chapter 8.

9.4.1 Summary of Game Measurement with a 90% Reliable Server

The best (recommendation) and the runner up techniques on each measurement of a 90% reliable server are shown in Table 9.1 (modified from Table 8.17). The best technique is the technique which has the most recommendation (at second columns) and runner up (at third columns), from the measurements that affect response time, which are M1, M2, M4, and M5 or in the bold rows.

	Recommendation	Runner up
M1: one-way multiple	**T2a: DR(1,10) + smoothing (2nd)**	T11: all - dying - grouping - visual
M2: interval update	**T2a: DR(1,10) + smoothing (2nd)**	**T8: all techniques (1st)**
M3: % DR error	T2a: DR(1,10) + smoothing	T2b: DR(2,10) + smoothing
M4: one-way single	**T8: all techniques (1st)**	T7a: duplicate packet
M5: two-way	**T8: all techniques (1st)**	T4: avatar blinking
M6: render update	T1: no techniques	T11: all - dying - grouping - visual
M7: packet sending	T8: all techniques	T6a: packets grouping (2)
M8: packet size	T1: no techniques	T10: all - dying - grouping

Table 9.1: Summary Detail of a 90% Reliable Server

T8 (enable all techniques) is the best technique that affect response time. It is the best for M4 (one-way response time for a single packet per keypress) and M5 (two-way response time), and runner up for M2 (interval update time of a remote avatar movement or rotation).

The runner up technique is T2a (DR and smoothing) with two recommendations from M1 (one-way response time for multiple packets per keypress) and M2 (interval update time of a remote avatar movement or rotation).

From the result, enabling all techniques can improve response time better than enabling individual techniques. And, in comparisons among individual techniques, DR and smoothing is the best for improving response time.

9.4.2 Summary of Game Measurement with a 75% Reliable Server

The recommendation and the runner up techniques on each measurement of a 75% reliable server are shown in Table 9.2 (modified from Table 8.32). The best technique is the technique which has the most recommendation (at second columns) and runner up (at third columns), from the measurements that affect response time, which are M1, M2, M4, and M5 or in the bold rows.

	Recommendation	Runner up
M1: one-way multiple	**T2a: DR(1,10) + smoothing (2nd)**	T11: all - dying - grouping - visual
M2: interval update	**T8: all techniques (1st)**	**T2a: DR(1,10) + smoothing (2nd)**
M4: one-way single	T7b: triplicate packet	**T8: all techniques (1st)**
M5: two-way	T4: avatar blinking	**T8: all techniques (1st)**
M6: render update	T1: no techniques	T11: all - dying - grouping - visual
M7: packet sending	T8: all techniques	T6a: packets grouping (2)
M8: packet size	T1: no techniques	T10: all - dying - grouping

Table 9.2: Summary Detail of a 75% Reliable Server

T8 (enable all techniques) is the best technique that affect response time. It is the best for M2 (interval update time of a remote avatar movement or rotation), and two runners up for M4 (one-way response time for a single packet per keypress) and M5 (two-way response time).

The runner up technique is T2a (DR and smoothing) with one recommendation from M1 (one-way response time for multiple packets per keypress) and runner up from M2 (interval update time of a remote avatar movement or rotation).

The result of 75% test is similar to the 90% test, enabling all techniques can improve response time better than enabling individual techniques. And, among individual techniques, DR and smoothing is the best for improving response time.

9.5 Results in Percentages for a 75% Reliable Server

Another type of results comparing between the best three techniques (DR and smoothing, avatar blinking and duplicate/triplicate packets), combined techniques (enable all techniques), with no enable techniques of the three types of response times (one-way response time for multiple packet per keypress, one-way response time for single packet per keypress, and two-way response time), with a 75% reliable server. The results are compared as percentages of the mean response time when no techniques are enabled (shown as the "No Techniques" bar). Consequently, a technique that reduces the time will have a percentage less than 100%, as shown in Figure 9.2 to Figure 9.4.

9.5.1 One-way Response Time, Multiple Packets Per Keypress

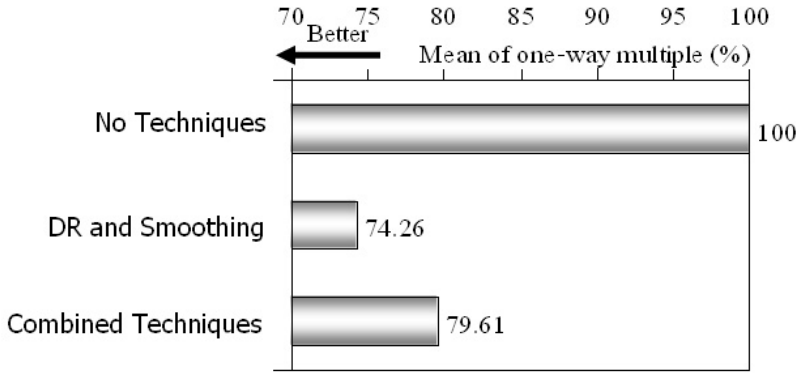


Figure 9.2: One-way response time, multiple packets

DR and smoothing reduce the mean response time by a tad over 25%, since the technique compensates for the loss of translation and rotation packets. Avatar blinking and duplicate/ triplicate packets sending have no significant effect on this type of responsiveness, and so are not listed in Figure 9.2.

9.5.2 One-way Response Time, Single Packet Per Keypress

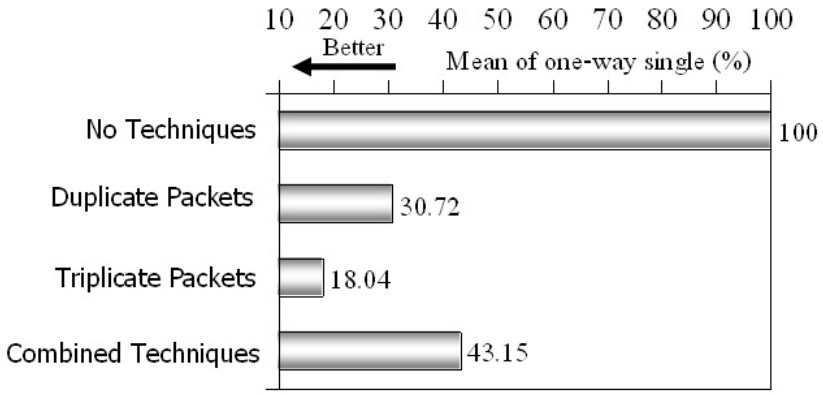


Figure 9.3: One-way response time, single packet

Duplicate and triplicate packet sending reduces the response time drastically: by over 80% for triplication which sends the same packet three times (see Figure 9.3). This reflects the impact that poor network reliability has on game play.

As the network becomes more reliable (e.g. moving from 75% to 90%), triplicate packet sending becomes slower, and duplicate packets becomes the better performer. The slowdown is caused by the cost of processing and ignoring so many multiple packets.

For this form of response time measurement, DR and smoothing and avatar blinking have no significant effect, so are not shown in Figure 9.3.

9.5.3 Two-way Response Time Measurements

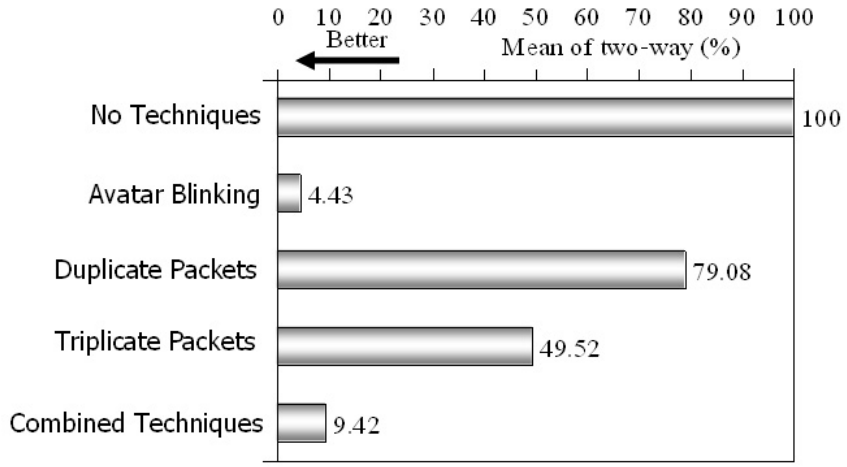


Figure 9.4: Two-way response time

Two-way response time is very susceptible to packet loss or delay since it depends on request and response packets both being successfully delivered. The loss of one or both of these packets will mean that the associated action cannot be completed.

Avatar blinking does a great job of disguising the delay, which under 75% network reliability conditions may be as much as 2-3 seconds. Duplicate/triplicate packet sending is necessary to ensure that copies of the lost datagrams eventually arrive.

As with the one-way response times for single packet per keypress in Section 9.5.2, if the network's reliability is increased, then the overhead of triplicate packet sending becomes excessive, and duplicate packet sending becomes the better choice.

9.6 Conclusions

The experiments with a client/server 3D mobile game highlight several issues related to improving client-side response times.

Response time must be measured in multiple ways for a good understanding of how it is affected by varying network reliability and different techniques. One-way response time for single packet reflects how simple datagram transfer is affected by the network. One-way response time for multiple packet focuses on more complex data delivery. Two-way response time deals with communication that employs a query/response form.

We have classified the techniques for improving response time into three categories: general, game-specific, and packet-based. A mix of techniques from all these categories gives the best across-the-board improvements. Combined techniques (e.g. dead reckoning and smoothing, avatar blinking, and duplicate/triplicate packet sending) produce mean response times 20% to 90% less than the mean response time for the game with no techniques enabled.

Some response time techniques can be politely termed as ‘tricks’, since their aim is to distract the user from the delays inherit in networks with high latency, limited bandwidth, and unreliable packet delivery. Avatar blinking is a good example, but is nevertheless a valuable approach.

References

[1] Fishlabs Entertainment. (2008, February). Robot Alliance. Online. Available: http://www.fishlabs.net/en/games/shooter/robot_alliance_3d.php

[2] Sun Microsystems. (2005, March). J2ME Wireless Toolkit. Online. Available: http://java.sun.com/j2me

[3] Sun Microsystems. (2005, March). The Java 2 SDK, Standard Edition. Online. Available: http://java.sun.com/j2se

[4] S. Singhal, "Networked Virtual Environments, Design and Implementation," Addison-Wesley, 1999

[5] L. Pantel and L. C. Wolf, "On the Suitability of Dead Reckoning Schemes for Games," In *proceedings of the 1st Workshop on Network and System Support for Games*, 2002, pp. 79-84

[6] R. M. Fujimoto, "Parallel and Distributed Simulation Systems", John Wiley & Sons, 2000

[7] A. Davison. (2007, December). Java Games Programming Techniques: Networking and Mobile3D Chapters. Online. Available: https://fivedots.coe.psu.ac.th/~ad/jg/

[8] F. T. Mario, "Elementary Statistics", 7th Edition, Addison-Wesley, 1998

[9] S. K. Singhal, "Effective Remote Modeling in Large Scale Distributed Simulation and Visualization Environments," In *PhD thesis, Department of Computer Science, Stanford University*, 1996

[10] S. Lee and J. Knudsen , "Beginning J2ME: From Novice to Expert," 3rd Edition, Apress, 2005

[11] J. F. Kurose and K. W. Ross, "Computer Networking: A Top-Down Approach Featuring the Internet," 2nd Edition, Pearson Education, 2003

[12] Doctor Robe. (2006, July). Computing Angles of a Right Triangle. Online. Available: http://mathforum.org/library/drmath/view/51875.html

Appendix

Appendix A: Published Paper

Improving Response Time in a Client/Server 3D Mobile Game

Prapat Lonapalawong
Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla, 90110, Thailand
prapatz@yahoo.com

Andrew Davison
Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla, 90110, Thailand
ad@fivedots.coe.psu.ac.th

ABSTRACT
A series of experiments were carried out on a client/server 3D mobile first-person shooter (FPS) to determine the best techniques for improving client-side response times in the presence of severe network unreliability. We utilized three measures of response time, which closely parallel the different types of communication employed between the clients. The response time techniques were grouped into three categories: general, game-specific, packet-based. A combination of the best three – dead reckoning and smoothing, avatar blinking, and duplicate/triplicate packet sending – produce mean response times that are 20% to 90% less than the mean response time for the game with no techniques enabled.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *client/server.*

General Terms
Measurement, Performance, Design, Reliability.

Keywords
Client/server, 3D, mobile game, response time measurement, dead reckoning and smoothing, avatar blinking, duplicate/triplicate packet sending.

1. INTRODUCTION
Interest in multiplayer 3D gaming has never been higher, and is starting to gain traction on mobile devices, with the success of games such as *Robot Alliance* and *Need for Speed: Carbon*. However, underlying networking issues (e.g. high latency, limited bandwidth, and lossy/reordered packet delivery) make it difficult to implement FPS-type games that offer rapid player interaction [1, 2, 6]. As a result, many multiplayer mobile games are turn-based, and use the network primarily for messaging and accessing server-side databases.

This paper describes experiments carried out upon a client/server 3D mobile FPS. The game executes on a LAN, but the server can simulate varying degrees of communication reliability, thereby emulating WAN/Internet conditions. A range of techniques for improving the game's response time were tested, which fall into three broad groups: general (applicable across a wide-range of FPS games), game-specific (tailored to our game), and packet-based. The success (or otherwise) of the techniques was judged by gathering statistics related to three different measures of response time.

CyberGames 2007, September 10–11, 2007, Manchester, UK.

2. GAME ARCHITECTURE
Our game's client/server architecture is quite typical of many multiplayer mobile games. The Java ME (`http://java.sun.com/j2me/`, [5]) game clients each render a world of competing penguins; the goals of a player's penguin are to find "life spots", gather bullets, and shoot other penguins. The game's architecture is summarized in Figure 1.

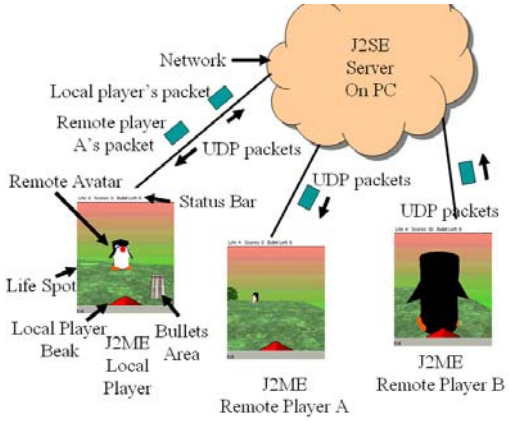


Figure 1. The client/server 3D mobile game.

The local player has a first-person view of a world, while the other penguins are remote avatars representing the other players. In Figure 1, the game currently has three users, so each player can see at most two other penguins (and its own penguin's red beak).

The rules of the game ensure that player behavior is fairly complicated, making it hard to predict a player's actions and the pattern of network activity. All the game's 3D assets (e.g. the penguins, the floor) are stored locally on the clients; no 3D models are transmitted via the network.

Game entry, inter-client communication, and game departure are controlled through a Java SE (`http://java.sun.com/j2se/`) server which manages the delivery of data in the form of UDP packets. The server can be configured to delay packet delivery, and to lose a given percentage of datagrams, in order to test the game's responsiveness at different levels of network reliability. The system was run across a LAN, so real-world latency, bandwidth restrictions, and packet loss were not issues.

Various levels of reliability were investigated, including 75% reliability, which means that there was a 25% chance of a packet being delayed (i.e. one chance in four), and a 25% chance that it would be lost. 90% reliability means that there is a 10% chance of packet delay, and 10% chance of packet loss. A packet can be delayed between 30 ms and 2 seconds.

2.1 Measuring Response Time

A more accurate reflection of a game's responsiveness can be gained by measuring three slightly different forms of response time: one-way response time for *single* packet actions, one-way response time for *multiple* packet actions, and two-way response time.

One-way response time for an action is the time that a packet representing the action takes to travel from a remote player to the local player, and includes the time to update the remote player's avatar on the local device.

Some complicated types of action require multiple packets to be transmitted, typically for updating avatar position and orientation. However, most actions can be represented by single packets, such as when the player loses a life point or picks up a bullet. This distinction between multiple and single packets is important since it highlights the effectiveness of techniques which group, delete, or duplicate packets.

Two-way response time is the time for a packet to be sent from the local player to a remote device to be processed, and for a response packet to arrive back at the local player and update his game state. An example of two-way response time in our game is when a player shoots at a penguin. This requires that a message be sent to the remote client represented by the penguin, and for the local client to wait until the shot's outcome (e.g. penguin death) is returned.

3. TECHNIQUES FOR IMPROVING RESPONSE TIMES

We experimented with a large number of techniques to improve the game's response times. We classify these techniques into three groups:

1. *General* techniques, which can be applied to any networked FPS. They include dead reckoning and smoothing, and selective visual field updating [3].

2. *Game-specific* techniques, which include avatar blinking and avatar dying (i.e. painting a translucent skull over a penguin to indicate its probable death).

3. *Packets-based* techniques, which include duplicate and triplicate packet sending, and packet grouping.

Due to space constraints in this paper, we will only discuss the best performing technique from each of these groups: dead reckoning and smoothing, avatar blinking, and duplicate/triplicate packet sending.

3.1 Dead Reckoning and Smoothing

Dead reckoning (DR) is used to 'guess' a penguin's translation or rotation when the packets holding that information have failed to arrive at the client [4]. We choose to activate DR after one movement packet is lost, and to keep it switched on for at most ten screen updates.

This approach requires packets to be time-stamped, and for a client to estimate how long to wait before a packet is deemed to be lost. The code must also deal with a 'lost' packet turning up after a lengthy delay.

DR is switched on promptly, after only one packet has been lost, so a penguin will keep moving rather than appear unresponsive. DR is switched off after at most ten updates (500 ms in our game), since it becomes very difficult to predict movement accurately after multiple updates.

It is essential to pair DR with smoothing. When a movement packet eventually arrives, smoothing gradually adjusts the penguin's position to relocate and reorientate it to the correct spot. Smoothing is carried out over several screen updates, so a penguin doesn't 'jump' from one position to another.

3.2 Avatar Blinking

Avatar blinking is game-specific: it is triggered when the local player shoots at a penguin, and the client has to wait for the shooting outcome from the remote player. The uncertainty about a penguin's future is denoted by making it blink. This offers immediate feedback to the player, which is more reassuring than have nothing change on screen for perhaps several seconds.

After usability tests, we determined that players find blinking to be helpful for at most a few seconds, after which time it becomes rather irritating. Consequently, a penguin can blink for at most three seconds, which is enough time for a shooting response to arrive when the network is performing at 75% reliability.

3.3 Duplicate/Triplicate Packet Sending

Duplicate/triplicate packet sending makes a client transmit the same packet two or three times to reduce the chance of it being lost en route. One drawback is that the receiver must be able to detect and ignore multiple packet copies. Also, indiscriminate multiple packet sending is a serious consumer of bandwidth. Consequently, we use the technique sparingly, only for important information whose loss would seriously impact the game. Such packets tend to be related to important avatar state changes, such as when a penguin loses life points, or shoots at another penguin. It also helps to correlate the amount of resending to the unreliability of the network.

4. RESULTS

The game was run many times with three clients, and results gathered over several minutes of typical gameplay in each game, and averaged. The tests reported here were carried out with the network set to be 75% reliable.

Three response times measurements were performed: one-way response time for multiple packet actions, one-way response time for single packet actions, and two-way response time.

The mean response times were calculated when no techniques were applied, and again when each of the techniques was switched on individually (i.e. DR and smoothing, avatar blinking, and duplicate/triplicate packets). Finally, all three techniques were switched on together.

The mean response times for the techniques were compared with the mean time when no techniques were enabled, using a standard one-tailed z-test with a 95% level of significance [7]. In the figures below, only the techniques that produced a significant reduction in the mean response time are reported.

4.1 One-way Response Time, Multi ple Packet Action

Figure 2 displays mean response times as percentages of the mean response time when no techniques are enabled (shown as the "No Techniques" bar). Consequently, a technique that reduces the time will have a percentage less than 100%. Data for the other response time measures in sections 4.2 and 4.3 are reported in a similar way (see Figures 3 and 4).

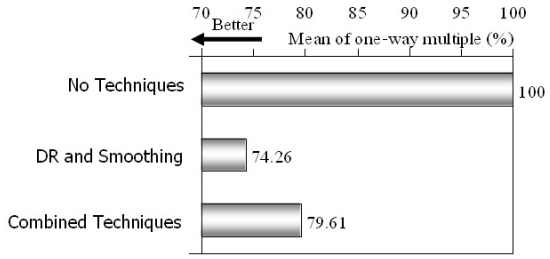


Figure 2. One-way response time, multiple packets.

One-way response times for multiple packet actions are mostly concerned with the processing of avatar movement (translations and rotations). This explains why DR and smoothing reduce the mean response time by a tad over 25% in Figure 2, since that technique compensates for the loss of translation and rotation packets.

Also of interest is that avatar blinking and duplicate/triplicate packets sending (the other two techniques tested here) have no significant effect on this type of responsiveness, and so aren't listed in Figure 2.

4.2 One-way Response Time, Single Packet Action

One-way response times for single packet actions cover the majority of the packets sent in the game, where an action can be codified as a single datagram.

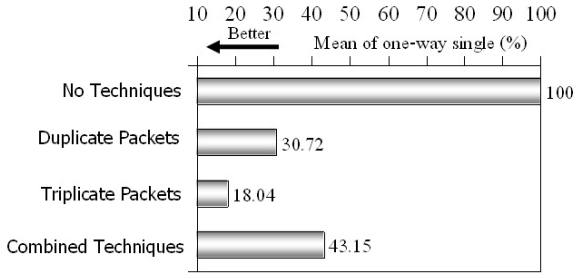


Figure 3. One-way response time, single packets.

Duplicate and triplicate packet sending reduces the response time drastically: by over 80% for triplication which sends the same packet three times (see Figure 3). This reflects the impact that poor network reliability has on game play – at 75% reliability, the "No Techniques" version of the game is almost unplayable.

As the network becomes more reliable (e.g. moving from 75% to 90%), triplicate packet sending becomes slower, and duplicate packets becomes the better performer. The slowdown is caused by the cost of processing and ignoring so many multiple packets.

For this form of response time measurement, DR and smoothing and avatar blinking have no significant effect, so are not shown in Figure 3.

4.3 Two-way Response Time Measurements

In our game, the most important two-way response time measurement is for a player shooting a penguin and waiting for the outcome. Figure 4 shows that avatar blinking is very important for maintaining a good response time, with duplicate/triplicate packet sending also playing a role.

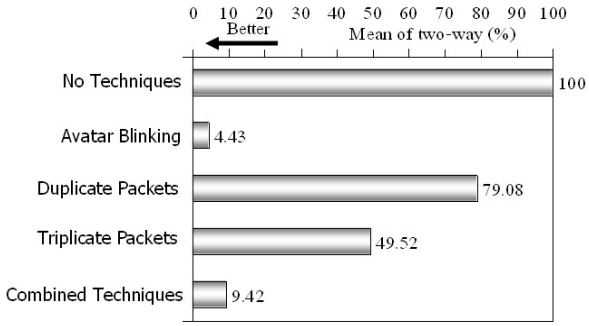


Figure 4: Two-way response time.

Two-way response time is very susceptible to packet loss or delay since it depends on request *and* response packets both being successfully delivered. The loss of one or both of these packets will mean that the associated action cannot be completed.

Avatar blinking does a great job of disguising the delay, which under 75% network reliability conditions may be as much as 2-3 seconds. Duplicate/triplicate packet sending is necessary to ensure that copies of the lost datagrams eventually arrive.

As with the one-way response times for single packet actions in section 4.2, if the network's reliability is increased, then the overhead of triplicate packet sending becomes excessive, and duplicate packet sending becomes the better choice.

5. CONCLUSIONS

Our experiments with a client/server 3D mobile game highlight several issues related to improving client-side response times.

Response time must be measured in multiple ways for a good understanding of how it is affected by varying network reliability and different techniques. One-way response time for *single* packet actions reflects how simple datagram transfer is affected by the network. One-way response time for *multiple* packet actions focuses on more complex data delivery. Two-way response time deals with communication that employs a query/response form.

We have classified the techniques for improving response time into three categories: general, game-specific, and packet-based. A mix of techniques from all these categories gives the best across-the-board improvements. Figures 2, 3, and 4 show that "Combined Techniques" (i.e. dead reckoning and smoothing, avatar blinking, and duplicate/triplicate packet sending) produce mean response times that are 20% to 90% less than the mean response time for the game with no techniques enabled.

Some response time techniques can be politely termed 'tricks', since their aim is to distract the user from the delays inherit in networks with high latency, limited bandwidth, and unreliable packet delivery. Avatar blinking is a good example, but is nevertheless a valuable approach.

6. ACKNOWLEDGMENTS

Our thanks to the Department of Computer Engineering, Faculty of Engineering, at Prince of Songkla University for generously supporting this research.

7. REFERENCES

[1] Fujimoto, R.M. 2000, *Parallel and Distributed Simulation Systems*, John Wiley.

[2] Singhal, S. 1999, *Networked Virtual Environments, Design and Implementation*, Addison-Wesley.

[3] Singhal, S.K. 1996, *Effective Remote Modeling in Large Scale Distributed Simulation and Visualization Environments*, PhD thesis, Dept. of Computer Science, Stanford University.

[4] Pantel, L., Wolf, L.C. 2002, "On the Suitability of Dead Reckoning Schemes for Games", *Proc. of the 1st Workshop on Network and System Support for Games*, 79-84.

[5] Li, S. and Knudsen, J. 2005, *Beginning J2ME: From Novice to Expert*, 3rd Ed., Apress.

[6] Kurose, J.F. and Ross, K.W. 2003, *Computer Networking: A Top-Down Approach Featuring the Internet*, 2nd Ed., Pearson Education.

[7] Mario, F.T. 1998, *Elementary Statistics*, 7th Ed., Addison-Wesley .

VITAE

Name Mr. Prapat Lonapalawong

Student ID 4712019

Educational Attainment

Degree	Name of Institution	Year of Graduation
Bachelor of Computer Engineering	Prince of Songkla University	2002

List of Publication

P. Lonapalawong, and A. Davison, "Improving Response Time in a Client/Server 3D Mobile Game," in *CyberGames 2007*, Manchester, UK, 2007.