2005

Accuracy and Fairness in Dead Reckoning Based Distributed Multiplayer Games

Hemant Banavar

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

ACCURACY AND FAIRNESS IN DEAD RECKONING BASED DISTRIBUTED MULTIPLAYER GAMES

By

HEMANT BANAVAR

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2005

The members of the Committee approve the Thesis of Hemant Banavar defended on 1 June 2005.

Sudhir Aggarwal
Professor Directing Thesis

Sarit Mukherjee
Outside Committee Member

Sampath Rangarajan
Outside Committee Member

Kartik Gopalan
Committee Member

Zhenhai Duan
Committee Member

Approved:

Sudhir Aggarwal, Chair, Department of Computer Science

The Office of Graduate Studies has verified and approved the above named committee members.

This thesis is dedicated to my mother.

ACKNOWLEDGEMENTS

I owe my thesis to my advisor Dr. Sudhir Aggarwal whose constant support and guidance led the thesis to its current form. I like to thank Sarit Mukherjee and Sampath Rangarajan for their valuable and timely suggestions and remarks that helped me maintain a high standard of work. Their help is appreciated. I would also like to thank my committee members Dr. Kartik Gopalan and Dr. Zhenhai Duan for their confidence in me and motivating me in doing my work. A special thanks to Dr. Ted Baker whose exceptional insights greatly enriched my knowledge.

I would like to thank my friends Satish and Prithviraj with whom I spent the longest and the best time while at FSU.

Finally, I would like to express my deepest gratitude to my parents, my sister and brother-in-law for their unconditional love and confidence in me.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Distributed real-time multiplayer games are played over a network among a set of players competing against each other and/or against Artificial Intelligence (AI). A latency hiding and bandwidth reduction technique known as *'Dead Reckoning'* [6] and [24] is often used in these games. The games use dead reckoning vectors to inform other (at a distance) participating players about the movement of any entity by a controlling player. The dead reckoning vectors contain information about the current state of the entity and additional information that defines the projected path the entity is going to take. The state of the entity is made up of one or more of the following – position, velocity, direction, acceleration and other kinematics.

When a participating player receives a dead reckoning vector, games traditionally put the entity at the position specified by the dead-reckoning vector and start projecting the path of the entity from that point based on the type of the dead reckoning algorithm, using the local clock of the receiver.

This thesis shows that this traditional method of usage of dead reckoning vector brings in inaccuracy in the receivers' rendering of the entity. This inaccuracy can be substantial even with low network delay between the sender-receiver pairs and increases with network delay. We propose an *accuracy model* to overcome this inaccuracy problem. In the *accuracy model* we propose using globally synchronized clocks and timestamp augmented dead-reckoning vectors which allow the receiver to estimate the latency between sender and receiver and apply correction for the delay, thus rendering the entity *accurately*. An open-source game '*BZFlag*'[4] was modified to use timestamp augmented dead-reckoning vectors and globally synchronized clocks. The modified version showed significant quantitative improvement in the accuracy even for 100ms delay between the sender-receiver pairs and appreciable qualitative improvement in game playing experience.

The thesis also explores the issue of *fairness* among the participating players in real-time multi-player games. In a sense, the inaccuracy would be tolerable if it is

consistent among all players; that is, at the same physical time, all players see inaccurate (with respect to the real position of the object) but identical trajectory for an object. But due to *varying* network delays between the sender and different receivers, the inaccuracy is different at different players as well. This leads to unfairness in game playing.

We first considered two scheduling approaches to achieve fairness. It was observed that, even though they achieve a high degree of fairness, they do it at the cost of increasing the mean inaccuracy of the system. This led to overall performance degradation of the game as compared to the original case.

Next, we proposed a budget based probabilistic algorithm based on the intuition that using a fixed budget (same as in the base case) of DR vectors, by increasing the frequency of sending the dead reckoning information to receivers which are more inaccurate and decreasing the frequency of sending the dead reckoning information to receivers which are less inaccurate as compared to the sender, we can achieve fairness and at the same time improve the accuracy of the more inaccurate receivers in the system. The probabilistic nature of the algorithm led to a higher inaccuracy at the receivers when they did not get the dead reckoning information for a long time and hence increased the mean inaccuracy of the system.

Hence, a budget based deterministic algorithm on similar lines was proposed and it was shown that it achieves a higher degree of fairness and at the same time maintains the mean inaccuracy of the system same as in the base case. The deterministic algorithm also succeeded in increasing the accuracy of more inaccurate receivers in the system. At the same time it achieved fairness with the same overall budget of dead reckoning vectors as in the base case and hence at no extra cost. BZFlag was modified to use the *deterministic model* to achieve fairness. It was shown that it achieved a considerably higher degree of fairness as compared to the base case and led to appreciable qualitative improvement of the game play.

The last part of the thesis was to generalize the above work by extending an open source Game Networking Library – Open TNL [25] to support *dead reckonable game objects* (DRObject). Game developers using Open TNL can make use of the DRObject class and create dead reckonable objects without worrying about the sending and receiving of dead reckoning vectors. Once the distance and the time threshold are specified, for the triggering of the dead-reckoning vector, the replication of the DRObject to all the receivers every time the threshold is crossed happens automatically. The accuracy model discussed above was also extended to Open TNL so that the DR Objects take into account the delay between the sender and the receiver and project the object at the receiver accurately. The deterministic model for fairness is to be integrated to Open TNL in the future.

CHAPTER 1
INTRODUCTION

Many distributed multiplayer games are increasingly of the type where human players compete against each other in a virtual environment. Each player controls an entity in the *virtual world* and competes with entities controlled by other players and/or controlled by AI. The humans experience the virtual world in the first person and hence many of these games are referred to as *First Person Shooter* Games. The human players can be present anywhere geographically but they exchange information about their position and actions in the virtual world over the network, typically over the Internet and compete against each other in the same virtual environment. The thesis mainly focuses on the networking aspect of the game and deals with issues arising due to network latency.

1.1 Approaches to Game Networking:

Distributed games are designed to network in one of the following ways.

1.1.1 Client-Server Approach:

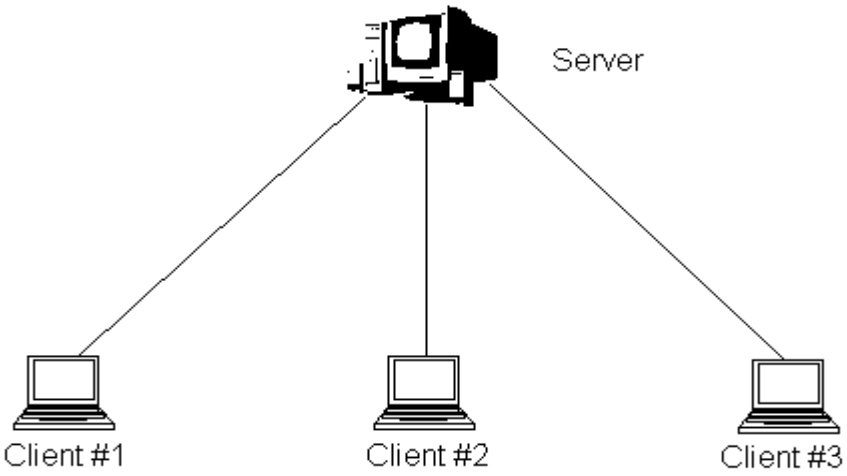


Figure 1.1 Client Server Games

In this approach each player's machine communicates only with a central game server. The server is responsible for tracking the activities of each player and sending necessary information to each player's machine so that it can display the game state to its player. Figure 1.1 shows a client-server setup.

The advantages of this approach are: the server which knows the whole game state at any instant of time can ensure that the players cannot get access to more information than they need to know and hence can protect the game against hacks; second, the player's machine is connected only to the server and sending and receiving only the information it needs to communicate, hence even if the player is on a dial-up connection, his bandwidth will not become overloaded and third, with a speedy server or a clustered server, it is possible to run Massively Multiplayer games involving tens of thousands of players in the game simultaneously.

The only disadvantage with this approach is that someone has to pay for the server and the bandwidth it uses. Some games like EverQuest charge the users for the service and some like Quake let the game owners to run their server anywhere on the Internet.

1.1.2 Peer-to-Peer:

In peer-to-peer approach, each player's machine is connected to every other player's machine and there is no central entity in such games. Every player knows the whole state of the game all the time. Figure 1.2 shows a peer-to-peer setup.

The biggest advantage of this approach is that the players themselves supply the hardware and bandwidth to play the game and thus removes the need for a third party to host the game.

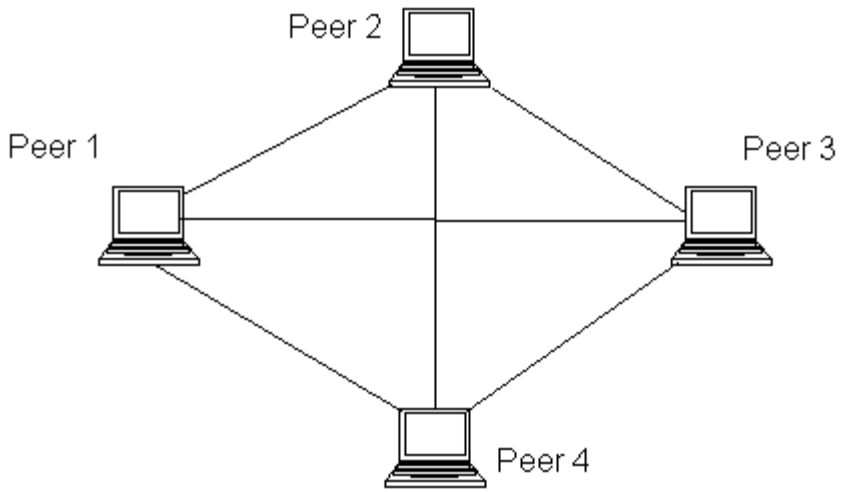


Figure 1.2 Peer-to-Peer Games

The drawbacks of this approach are: as there is no central entity to maintain the game state, each player has full information about the game making it susceptible to client hacks; second, increasing the number of players in the game increases the network traffic considerably resulting in exhausting the bandwidth for a player using a dial-up connection; third, this approach is not scalable because of lack of multicast support and hence peer-to-peer games used to be limited to 8 or fewer players, with clever compression schemes some games support up to 32 players.

1.1.3 Hybrid Approach:

In the hybrid approach, the games make use of a central server as an adjunct. The hybrid approach tries to combine the best of both client-server and peer-to-peer designs.

The server can be used for the sake of discovering other players in the game, or for relaying and validation, or as a fallback channel for communication if the peer-to-peer channel cannot be used. For example, BZFlag uses the server as a relay and the server does simple ordering of update messages and discards out of order updates.

The inherent issue that comes with exchanging information over the network is the network latency. Simply put, *network latency is the time required for a packet to travel from one computer to another across the network*. Games are played across the Internet and consequently the players can experience round trip delays in the range of 50-250 milliseconds. Even this amount of latency poses big problems for developers of fast action games. As the players on different machines need to be kept in sync, it would be unacceptable for the action to freeze for some time.

Latency is not a problem for playing slow moving, turn-based games such as *Chess* or *Go* across a network. On the other hand, games that require high levels of fast-paced interaction between players can be severely degraded by typical WAN or even LAN latencies. This becomes problematic, as players expect the level of performance of distributed games to approximate that of single computer, single player games. The effect of latency is the most prominent in these games due to their real-time characteristics. Our work concentrates on the effect of latency in dead reckoning based games from the perspective of accuracy and fairness. The work is applicable to games designed using any of the approaches mentioned above. *'Dead Reckoning'* is the most commonly used latency hiding and bandwidth reduction approach by these games.

1.2 Dead Reckoning:

Distributed Interactive Simulation (DIS) [17] contains a technique for latency hiding and bandwidth reduction known as *Dead reckoning* [24] that has been adapted by real-time multiplayer games. The concept of *Dead reckoning* originally comes from navigation and was used by sailors around the end of the 15th century for deducing their position in the ocean.

When a vehicle or entity is created, the computer that owns the entity sends out an entity state message to all other computers on the network. The entity state message contains information that uniquely identifies the entity; information that

describes the current kinematical state of the entity, including position, velocity, acceleration and orientation; and other information, such as entity's damage level.

Last, the entity state message contains an identifier that tells all other nodes on the net which dead reckoning algorithm to use for this entity. When other computers participating in the distributed simulation receive this message, they create local copies of the specified type of entity. Thus, every node on the net begins to see this entity.

With dead reckoning, after receipt of the first entity state Protocol Data Unit (PDU) for an entity, each receiver node on the net begins moving the entity by applying the agreed-upon dead reckoning algorithm. As long as the entity continues to move in a predictable fashion, it is in a consistent, synchronized way to all receiver nodes on the net with no further network traffic required.

The instant a player controlling an entity moves the control stick, the entity deviates from a smooth, algorithmically definable path. This is immediately detected and handled by the computer that owns the entity.

The owning simulation compares the dead reckoning values to the true state of the entity as controlled by the player. If the dead reckoning and true state values differ by an amount that exceeds the agreed-upon dead reckoning threshold, a new entity state message is sent out to update the other nodes on the net. All nodes update their copies of the entity to reflect the new entity state message values, and dead reckoning begins again with the new data point.

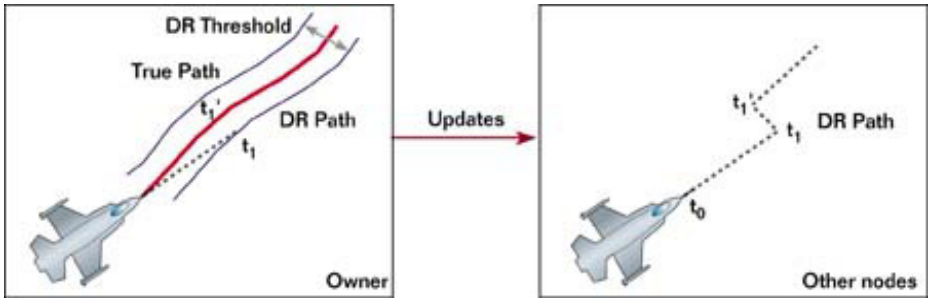


Figure 1.3 Dead Reckoning

In figure 1.3, it can be seen that the owner sends out an entity state message at t_0 to all the nodes that are dead reckoning the owner's aircraft. The owner takes input from a keyboard or a joystick and updates the state of the aircraft on its screen. At every input, the owner also checks if the dead-reckoning threshold has been crossed. At t_1 when the threshold is crossed, the owner sends out a new entity state message to other nodes indicating the new entity state. All the other nodes keep projecting the entity on the path indicated by the dotted line till they get the new entity state update. As soon as they get the new update, they position the entity at the position defined in the update and start dead reckoning the entity from that point ahead again. Dead reckoning then begins again, as shown at t_1' and beyond.

1.3 Dead-Reckoning Issues:

While the advantages of dead reckoning are visible – latency hiding and bandwidth reduction, it comes with some limitations. The thesis primarily focuses on two limitations of placement that result in a network environment: inaccuracy and unfairness across different players. It suggests mechanisms to curtail the effect of these limitations on the games. Before we discuss the mechanisms to tackle with the problems, let us first take a more detailed look at what exactly are these limitations and how are they caused.

1.3.1 Inaccuracy:

Consider the same example mentioned above and notice that whenever a state update is sent to the other nodes, it takes sometime for the update to reach each of the nodes. These nodes, after receiving the update put the object at the specified location and start using the specified algorithm for moving the aircraft on their screens. Due to the latency in the network, the receiver nodes will always place and see the aircraft in the past as they received the update message after a while after its generation. This is termed as

the *after export error*. Also, in the time interval between the generation of the update and reception at the receiver nodes, the receiver nodes render the aircraft inaccurately based on old information. This is termed as the *before export error*. The sum of the before and the after export error caused by each update from the sender is the total inaccuracy at the receivers. The amount of inaccuracy depends on the latency between the rendering node and the owner node.

1.3.2 Unfairness:

In the above scenario, observe that the latencies between the owner node and every other node might be different. Consequently, at the same physical time each nodes view of the owners object is lagged by different amounts. This leads to each receiver viewing the object at a different position at the same physical time. This difference in the position of the sender's object at each of the receiver's screen at the same physical time is termed as unfairness among the players. Unfairness is caused due to the difference in the before export error due to an update message at each receiver. The game would be fair if all the receivers have the same before export error.

1.4 Background and Related Work:

Online multi-player games are increasingly becoming popular due to advances in game design and the availability of broadband Internet access to the end-user. With players being distributed across the Internet the issue of accurate rendering of the entities becomes important. Modifications to dead reckoning have been suggested by [5] and [6] but not from the accuracy perspective. There has also been work that focuses on how to either reduce or mask player experienced response time. In [9, 10] synchronized message delivery techniques for client-server game architectures are discussed that aim to compensate for variable message delays from different players to the game server.

Also, with players being distributed across the Internet, the issue of fairness among the players with varying message delays from other players (in the case of peer-to-peer games) or from a centralized server (in the case of client-server games) becomes an important issue to study. For network games, to provide fairness, the concept of *local lag* has been used where each player delays every local operation for a certain amount of time so that remote players can receive information about the local operation (for example in the form of a DR vector) and execute the same operation at the about same time, thus reducing state inconsistencies [2]. The online multi-player game MiMaze [1], [7], for example, takes a static bucket synchronization approach to compensate for the unfairness introduced by variable network delays. In MiMaze, each player delays all events by 100 ms regardless of whether they are generated locally or remotely. Players with a network delay larger than 100 ms simply cannot participate in the game. In general, techniques based on bucket synchronization depend on imposing a worst-case delay on all the players. Much of the focus on improving real-time, online multi-player games is on how to reduce player experienced response time. For timely state updates at player consoles, a technique referred to as *dead reckoning* is commonly used to compensate for packet delay and loss [1], [2] and [3]. For client-server based first person shooter games, [8] discusses a number of latency compensating methods at the application level, which are proprietary to each game. These methods are aimed at making large delays and message losses tolerable for players but they does not consider the problems that may be introduced by varying delays from the server to different players or from the players to one another. There have been a few papers, which have studied the problem of fairness in a distributed game by more sophisticated message delivery mechanisms. But these works [9], [10] assume the existence of a global view of the game where a game server maintains a view (or state) of the game. Players can introduce objects into the game or delete objects that are already part of the game (for example, in a first-person shooter game, by shooting down the object). These additions and deletions are communicated to the

game server using “action” messages. Based on these action messages, the state of the game is changed at the game server and these changes are communicated to the players using “update” messages. Fairness is achieved by ordering the delivery of action and update messages at the game server and players respectively based on the notion of a “fair-order” which takes into account the delays between the game server and the different players. Objects that are part of the game may move but how this information is communicated to the players (for example using a DR vector or some other mechanism) seems to be beyond the scope of these works. It appears that the movements of the objects are determined by the server and are communicated to the players using mechanisms not described. In this sense, these works are very limited in scope and may be applicable only to first-person shooter games and that too to only games where players are not part of the game.

1.5 Objective:

The goal of the thesis is to provide accuracy and fairness in dead reckoning based games.

1.5.1 Reference Model:

Assume a multiplayer game where each player controls his tank and earns points by shooting other tanks in the game. Imagine a scenario where three players are playing the game against each other. Each of them has a controller attached to a computer and has a screen where he has the view of the virtual environment from his tanks perspective. Here assume that everyone is sitting in the same room and all the controllers and screens are connected to a single computer, which is running the game. As it can be seen that all the players are connected to the same computer and there is no network delay except for the negligible delay from the computer to the screens and the controllers, everyone gets the accurate view of the other entities in the game.

Now consider a similar setup where there are three players controlling their tanks and playing against each other. This time assume one player is in the same room as the

computer running the game and the other two players are on different continents with different and variable delays to the computer running the game.

The thesis uses the above, same room setup as the base reference model and aims at achieving a similar game playing experience in the second scenario where the multi player games played across a network with different and variable delays.

The thesis focuses on mainly two areas where the performance of dead reckoning based games can be improved. First, to make the dead reckoning accurate with respect to physical time so that all the participants get a more accurate view of the current state of the game. Second, give them a *fairer* view of the game so that everyone's experience of the game is relatively same regardless of their latencies to each other or to the server.

The final goal of the thesis is to generalize the work by extending an open source game-networking library to support dead reckonable objects, which can be configured to be *accurate* and *fairer* as described earlier. In this way, making it easier for game developers to build dead reckoning based games without worrying about the dead reckoning issues of *accuracy* and *fairness*.

1.6 BZFlag:

BZFlag [4] was chosen for the purpose of demonstrating the results, as it is a popular open source game. BZFlag (Battle Zone Flag) is a *first-person shooter* game where the players in teams drive tanks and move within a battlefield. The aims of the players is to navigate and capture flags belonging to the other team and bring them back to their own area. The players shoot each other's tanks using "shooting bullets". The movements of the tanks (players) as well as that of the shots (entities) are exchanged among the players using dead reckoning vectors.

1.7 Outline of Chapters:

The thesis is divided into four chapters detailing about the different stages of work done and a final chapter summarizing the results. The appendix contains some important code snippets from the implementation.

Chapter 2 discusses the Accuracy Model. Chapter 3 goes further and discusses the different approaches explored for fairness viz. two scheduling approaches and probabilistic and deterministic budget based approaches. Chapter 4 discusses the work done to generalize the idea of dead reckonable objects and implement the accuracy model in Open TNL. Chapter 5 contains some final thoughts based on the results and ideas of what can be done in the future. Appendix A describes the implementation details of the accuracy model in BZFlag. Appendix B describes some important details of implementation of different approaches to fairness in BZFlag. Appendix C contains the implementation details of the extensions to Open TNL.

CHAPTER 2
ACCURACY IN DEAD RECKONING BASED DISTRIBUTED MULTI PLAYER GAMES

2.1 Motivation

In distributed multi-player games involving a large number of players across the Internet, message delays between players are not negligible. When network delay is non-negligible, the trajectory of an entity that the sender expects receivers to follow in physical time may not be followed in physical time *before* and even *after* the receivers receive the dead reckoning vectors (henceforth referred to as DR vectors). This means, there is a deviation in physical time between the *placed path* at the receiver and the path that follows the DR vector exported by the sender (which we refer to as the *exported path*). This deviation is referred as the ***export error***.

The thesis explores this problem by considering the following path trajectories: (a) the trajectory of the player/entity at the receiver before the DR vector is received, and (b) the trajectory of the player/entity after the DR vector is received.

This work shows that by synchronizing the clocks at all the players and by using a technique based on time-stamping messages that carry the DR vectors, it can be guaranteed that the *placed* and the *exported* paths match *after* the DR vector is received; i.e. the export error after the DR vector is received is zero (referred to as the *after export error*). The game BZFlag (Battle Zone Flag) was instrumented with this technique and an experimental setup was used to perform (a) quantitative experiments to show the reduction in export error, and (b) qualitative experiments to show the improvement in game playing experience.

The idea that is conveyed through this work is that the game playing accuracy of any distributed multi-player game can be significantly improved using synchronized clocks among players and using a global time to project the trajectory of entities.

The next section illustrates the *before* and the *after* export errors. The third part presents quantitative results with modified implementation of BZFlag to show the reduction in export error compared to the current implementation of BZFlag.

2.2 Export Errors

In current implementations of multi-player games the clocks at the players are ***not*** synchronized and the DR vectors are generated and used as follows. Each DR vector sent from one player to another specifies the trajectory of exactly one player/entity. In the description below it is assumed that the use of a *linear DR vector* in that the information contained in the DR vector used by the receiving player is enough to compute the trajectory and render the entity in a straight-line path. Such a DR vector contains information about the starting position and velocity of the entity where the velocity is considered to be constant. Thus, the DR vectors sent by a player specifies the current position of the player/entity in terms of the x, y, z coordinates and the velocity vector in the direction of x, y and z coordinates.

In the following discussion when the export error is considered, a sequence of DR vectors sent by only one player and for only one entity is considered. DR_i is used to denote the *ith* such DR vector. This DR vector will denote the tuple (x_i; y_i; z_i; vx_i; vy_i; vz_i). When this DR vector is received, the receiver will use the x_i; y_i; z_i values to project the entity on the local console and then use the velocity vectors vx_i, vy_i and vz_i to continuously project and render the trajectory of the entity. This trajectory will be followed until a new DR vector is received which changes the position and/or velocity of the entity.

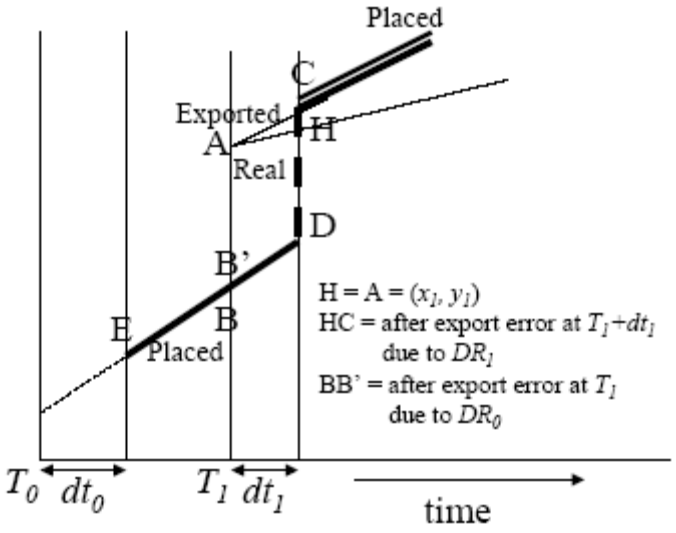


Figure 2.1 Export Errors

Based on this model, Figure 2.1 illustrates the sending and receiving of DR vectors and the different errors due to the deviation in the trajectory of the entity at the same physical time between the sender and the receiver. For ease of description, the DR vectors shown in the figure illustrate only two dimensions. The figure shows the reception of DR vectors at a player (henceforth called the *receiver*). The horizontal axis shows the time as per the sender's clock. Also, the sender's clock is used as the reference physical time for the description below. The vertical axis tries to conceptually capture the two-dimensional position of an entity. Assume a traditional game environment where the player clocks are not synchronized. Assume that at time T_0, a DR vector $DR_0 = (x_0; y_0; vx_0; vy_0)$, is computed by the sender and immediately sent to the receiver. Assume that DR_0 is received at the receiver after a delay of dt_0 time units. The receiver will place this entity at $(x_0; y_0)$ which are the coordinates specified in DR_0 (note that because of delay dt_0, this entity will be at a different position at the sender at the same physical time). This will now become the placed path at the receiver for that entity and this path *ED* is the trajectory of the entity at the receiver. Assume that at time T_1 a new DR vector $DR_1 = (x_1; y_1; vx_1; vy_1)$ is computed for the same entity and

immediately sent to the receiver. At this time, at the sender, the entity is at location (x_1; y_1). Assume that DR_1 is received at the receiver after a delay of dt_1 time units. When this DR vector is received, assume that the entity is at point D at the receiver. The entity is now moved at the receiver to (x_1; y_1) which is the coordinate specified in the DR. This is shown as point H in the figure. Note that the coordinates of point A and H are the same, as the figure shows the progression in time as the X-axis (and **not** the coordinate axes). At this physical time when the entity is at point H (x_1; y_1) at the receiver, the trajectory of the entity at the sender is following the same path except that the entity is at a different position (shown as point C). Point C is *ahead* in the trajectory because the entity was at point A (x_1; y_1) at time T_1 at the sender and only at time T_1 + dt_1 has it been put at the same coordinate position at the receiver. This means, the placed path at the sender *leads* the placed path at the receiver although they follow the same trajectory. This error between the position of the entity at the sender (point C) and the position of the entity at the receiver (point H) is the *after export error*. Note that the trajectory of the entity due to DR_0 also has a similar after export error where the receiver lags the sender. At time T_1, the entity at the sender is at point B (from which it is moved to point A due to the computation of DR_1) and the entity at the receiver lags the sender and is at point B'.

Again, consider Figure 2.1 but now assume that the clocks are synchronized at the players. In addition to maintaining synchronized clocks, the DR vectors sent includes the times T_0 and T_1 at which they are sent. When DR_0 = (T_0; x_0; y_0; vx_0; vy_0) is received, the receiver will compute the initial position of the entity as (x_0 + vx_0 X dt_0; y_0 + vy_0 X dt_0) (shown as point E). The receiver can figure out dt_0 as the time difference between when it received DR_0 and T_0 (which has been appended to DR_0). Again, the line *ED* represents the placed path at the receiver. When DR_1 = (T_1; x_1; y_1; vx_1; vy_1) is received, a new position for the entity is computed as (x_1 + vx_1 X dt_1; y_1 + vy_1 X dt_1) and the entity is moved to this position (point C). Again, the receiver can figure out dt_1 as the time difference between when it received DR_1 and T_1. The velocity components vx_1 and

vy_1 are used to project and render this entity further. Note that when the entity is moved to point C at the receiver at time $T_1 + dt_1$, the position of the entity at the sender is exactly the same (that is, points H and C are the same). This means, with synchronized clocks, there is no *after export error* and the placed paths at the sender and the receiver after the DR is received at the receiver is exactly the same.

Now consider the *before export error*. As mentioned earlier this error due to network delay is unavoidable even if the clocks are synchronized. Although DR_1 was computed at time T_1 and sent to the receiver, it did not reach the receiver until time T_1+dt_1. This means, although the exported path based on DR_1 at the sender at time T_1 is the trajectory AC, until time T_1+dt_1, at the receiver, this entity was being rendered based on DR_0 at trajectory BD in case of synchronized clocks and B'D in case the clocks are not synchronized. Only at time $T_1 + dt_1$ did the entity get moved to point C in case of synchronized clocks and to point H in case the clocks are not synchronized. This is the *before export error* due to DR_1 (that is, the error component due to the use of DR_0 to render the entity at the receiver before DR_1 is received). A way to represent this error is to compute the integral of the distance between the two trajectories (AC and BD in case of synchronized clocks and AC and B'D in case the clocks are not synchronized) over the time that they are out of sync. Note that there would have been a *before export error* created due to the reception of DR_0 at which time the placed path would have been based on a previous DR vector. This is not shown in the figure but it serves to remind the reader that the export error is cumulative when a sequence of DR vectors is received.

Although even with synchronized clocks there does exist a *before export error*, in the next section, using a modified version of BZFlag and using experimental measurements from the modified implementation, it is shown that the total export error (which includes the *before error* and the *after error*) is significantly reduced with our new implementation that uses synchronized clocks compared to the current implementation. The modifications to the BZFlag incorporate synchronized clocks and the time stamping

of the DR vector with the sending time of the DR vector as described above. The next chapter will go into the details of the work on equalizing the before export error and consequently making the game fairer.

2.3 Instrumentation of BZFlag and Numerical Results

The current implementation of BZFlag uses local clocks (i.e., not synchronized) for dead reckoning of the players on the screen. The implementation of BZFlag was modified to incorporate global clocks (i.e., synchronized) among the players and the server and exchange time-stamps with the DR vector. The instrumentation traces the real path traversed by a player/entity as a sender of the object. A receiver logs the reception of a DR vector and renders the player/entity to account for the delay between the sender and the receiver so that a single instance of game playing generates placed paths for both global and local clocks.

A test-bed with four player stations was setup. One station acts as a sender where the tank is moved; real path is traced and logged, and DR vectors are generated and sent. The other three stations act as receivers. NISTNet [17] was used to insert different but fixed amounts of delay, 100ms, 300ms and 800ms, between the three sender-receiver pairs. Each receiver gets the same DR vector from the sender, computes the placed path based on both local and global clocks, and logs the corresponding placed paths.

Three different sets of experiments were conducted, each differing in the frequency at which the DR vectors are generated. In the *Linear Motion* case, the tank is navigated (by human) in a straight line thereby g enerating very few DR vectors. The *Circular Motion* moves the tank in a circular path. This also generates a few DR vectors as BZFlag detects circular motion and dead-reckons using a circular motion DR vector. In the *Random Motion* case we move the tank in a random fashion, which generates moderate to large numbers of DR vectors. In each experiment, the real and the placed paths using both local and global clocks are logged. The logs are used to compute the instantaneous error between the two paths (i.e., the distance between them at any point

in time) using local and global clocks. In Figure 2.2 snapshots between 25 to 30 seconds (in the X-axis) of the entire game play are shown. The Y-axis shows the error between the real and the placed paths in terms of distance between BZFlag coordinate point units. The size of a BZFlag tank in these units is 2.8 units wide and 6 units long. The errors due to local and global clock are henceforth referred to as the local and global errors, respectively.

For the *Linear Motion* case, frequency of generation of DR vectors is low. Due to this, the exported DR vector is rendered *correctly* before the next DR vector is generated. This leads to mostly zero global error as the entity is placed exactly where it would be in the sender's screen. As the delay between the sender and the receiver increases, the possibility of the sender generating a new DR vector before the previous one reaching and getting rendered on the receiver's screen increases. Therefore, relatively higher peaks in the global error can be observed. In all cases the global error is much smaller than the local error. For example, with a delay of 300ms between the sender and the receiver (which is quite typical), the local error could be up to 2 tank units, whereas the global error is close to zero.

The *Circular Motion* case is similar to the Linear Motion case except that it captures the situation with a different type of DR vector (circular motion DR).

For the *Random Motion* case, the DR vectors are generated with moderate to high frequency. For low delay most of the DR vectors reach and get rendered accurately at the receiver and the global error reaches zero quite often. As the delay increases, the sender generates the next DR vector before the previous one reaches and gets rendered; this increases the global error. There are a few instances where global error is larger than the local error. This happens when a moving player becomes stationary (i.e., a tank stops). A new DR vector indicating that the tank is stationary is sent to all the receivers. Till it reaches the receiver, the receiver computes the placed path using the previous DR vector.

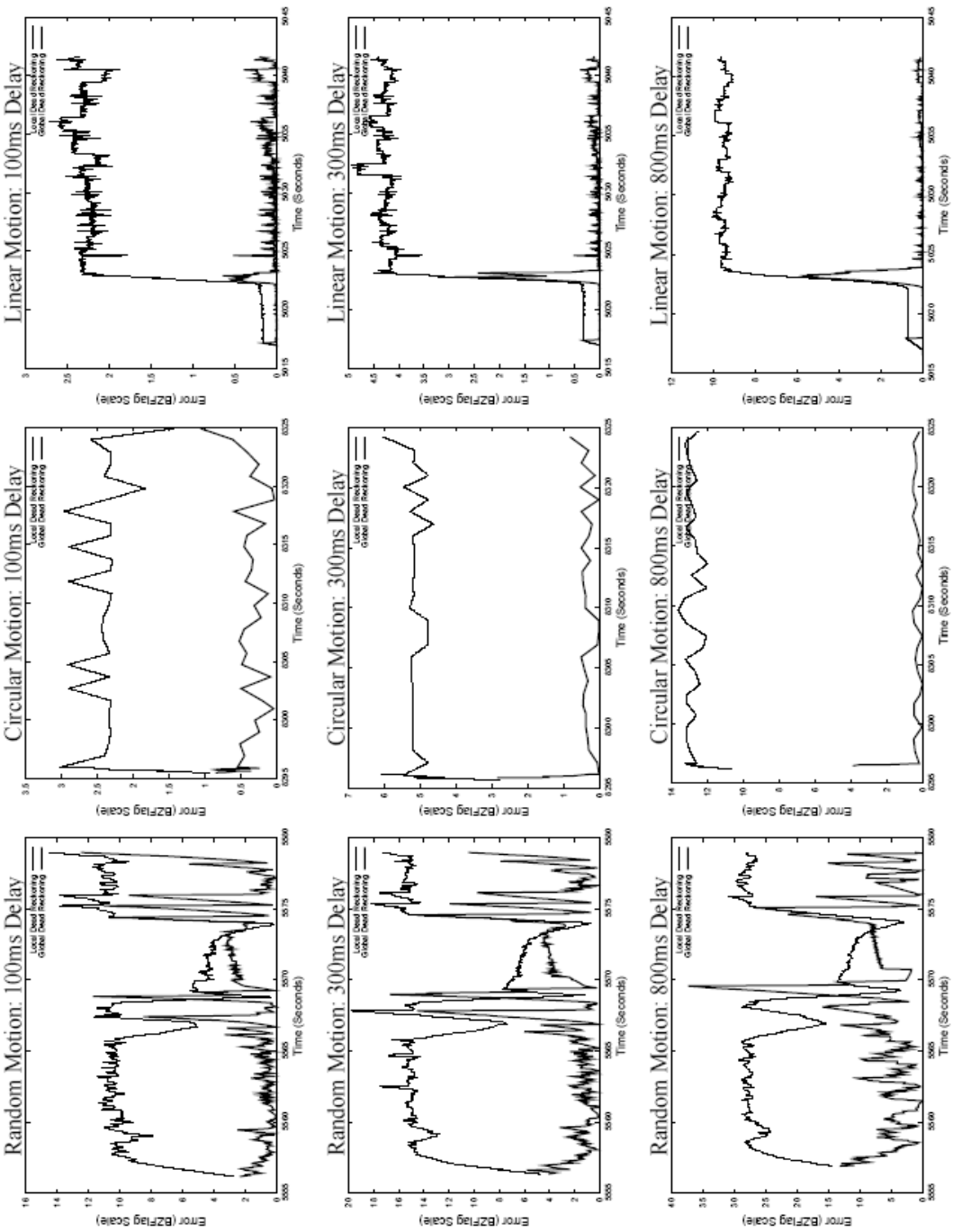


Figure 2.2 Accuracy Results

With global clock, since the player is placed in the exact position, the trajectory overshoots the stationary point by the time the DR vector arrives. With local clock, the placed path always (time-) lags by the amount of delay between the sender and the receiver. So the player's path does not overshoot but become stationary after a time lag. In general the average global error is substantially less than the average local error leading to the conclusion that with a mechanism to accurately estimate the lag between the sender and the receiver and predict the current position of the sender controlled entity on the receiver, it is possible to substantially increase the accuracy of a distributed multi-player game.

2.3.1 Qualitative Game Play

To evaluate the qualitative game playing experience, the game was played between two players with different delays. The game was played several times using both local and global clocks in order to qualitatively assess the differences in the two approaches. It was observed that the hit rate (due to the "shots") was better in case of global clock as compared to local clock particularly when the delay was high. This was because the player who was being dead reckoned was more accurately placed on the other players' screen and hence the other player was able to hit him where he could see him as compared to using local clock where the other player would miss him if he tried to hit him where he saw him.

2.4 Conclusion

We started out with the aim of reducing the after export error of a DR vector. By knowing the correct delay estimate, the receiver can accurately project the position of the sender's entity at the current time. We proposed a model using synchronized clocks and timestamp augmented DR vectors to estimate the delay accurately and using the estimate to project the current position of the object accurately and hence reducing the after export error to zero. BZFlag was instrumented with the accuracy model and it was

observed that the error in placement of the sender's entity at the receiver was considerably reduced using the accuracy model. From the results we conclude that the accuracy model considerably improves the game play quantitatively and qualitatively.

CHAPTER 3
FAIRNESS IN DEAD RECKONING BASED DISTRIBUTED MULTIPLAYER GAMES

3.1 Motivation:

In the previous chapter, it was shown that by synchronizing the clocks at all the players and by using a technique based on time-stamping messages that carry the DR vectors, it can be guaranteed that the *after export error* is zero. That is, the *placed* and the *exported* paths match *after* the DR vector is received. It was also shown that the *before export error* can never be eliminated since there is always a non-zero network delay, but can be significantly reduced using the *accuracy model*. Henceforth it is assumed in this chapter that the players use the *accuracy model*, which results in unavoidable, but small overall export error.

This chapter focuses on the issue of fairness among the receivers, which is caused by the difference in the export error over time at each receiver due to the different and varying network delays between each sender-receiver pair. Due to the difference in the export errors among the receivers, the same entity is rendered at a different position at the same physical time at different receivers. This brings in *unfairness* in game playing. For instance a player with a large delay would always see an entity *late* in physical time compared to the other players and, therefore, his action on the entity would be delayed (in physical time) even if he reacted instantaneously after the entity was rendered.

The goal of this work is to make the game *fairer* to all the players in spite of varying network delays by equalizing their export errors.

The first part of the work focuses on equalizing the export error of all the receivers over time by building up their export errors to the export error of the receiver with highest export error over time among all the receivers and in this way achieving

fairness among the receivers. Two scheduling algorithms are discussed under this section and compared against the base case for the degree of fairness achieved. The scheduling approaches achieve a higher degree of fairness among the receivers at the cost of increase in the mean of the export error of all the receivers.

The second part explores the possibility of an *optimal algorithm*, which can bring down the export error of the receiver, having the highest export error, to such a point where all the receivers can have the same export error over time. The intuition behind this work is that by using the same amount of DR vectors over time as in the base case, instead of sending the DR vectors to all the receivers at the same frequency as in the base case, if we can increase the frequency of sending the DR vectors to the receiver having the higher export error and decrease the frequency of sending the DR vectors to the receiver incurring lower export error, we can equalize the export error of all receivers over time and at the same time bring down the error of receivers incurring high accumulated export error in the base case. Two budget-based algorithms are explored in this section and observations made in both the probabilistic and deterministic approaches are presented here.

The game BZFlag (Battle Zone Flag) was instrumented with the scheduling algorithms and the budget based algorithms mentioned above and an experimental setup was used to perform (a) quantitative experiments to compare the standard deviation and the mean of the export errors in these approaches against the base case, and (b) qualitative experiments to compare the game playing experience in the all the approaches.

The next section explains the method to calculate the export error over time for a receiver. The third section explains the scheduling approaches in detail. The fourth section explains the budget based approaches.

3.2 Calculating Export Error over time:

This section explains the technique used to calculate the export error over time. Simply put, the difference between the exported trajectory and the placed trajectory at a receiver over time is called as the export error over time for that receiver.

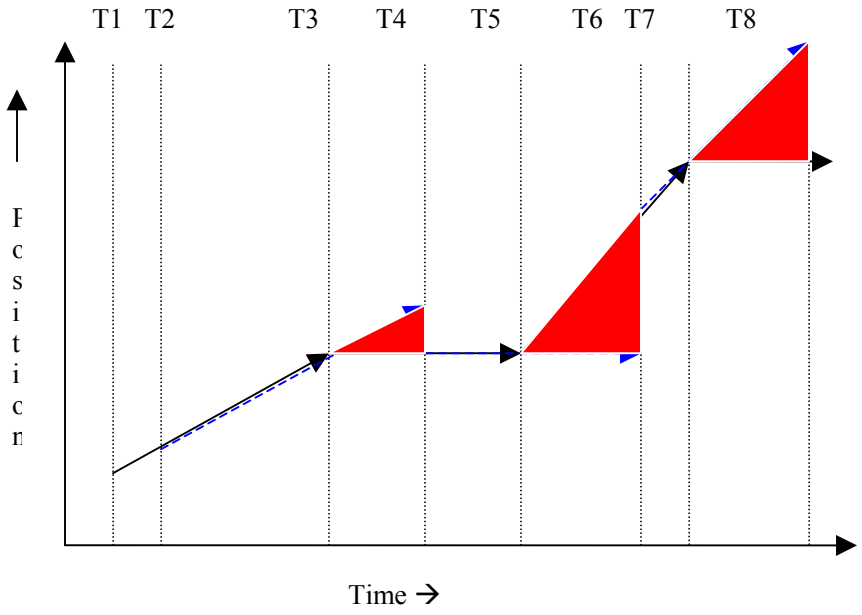


Figure 3.1 Cumulative Export Error

Consider an example in which there is a sender and a single receiver and the sender owns the entity, which is being dead reckoned by the receiver. The receiver incurs *before export error* continually due to the inherent delay in the network. The amount of the before export error depends on the delay.

Figure 3.1 shows the trajectories of the sender and the receiver in such a way that the receiver's view of the entities position over time is superimposed on the sender's view. As the clocks are synchronized, both the trajectories can be represented

on the same time frame. The Y-axis tries to conceptually capture the 2-dimensional position of the entity and the X-axis represents the time over which the entity was on the position shown. The solid lines indicate the trajectory of the entity at the sender and the dotted lines indicate the trajectory of the entity at the receiver.

T_1, T_3. T_5, T_7 represent the time at which DR_1, DR_2, DR_3 and DR_4 were generated and sent out by the sender respectively. T_2, T_4, T_6, T_8 represent the times at which DR_1, DR_2, DR_3 and DR_4 were received at the receiver respectively.

It should be noted that for the sake of explanation the figure considers a simple case where a DR is generated and after it reaches the receiver, a new DR is generated. There are complex cases where there might be many DR vectors generated and sent out by the receiver rapidly when the receiver is still using some old DR. The approach to calculate the export error over time is still the same for all the cases. It should also be noted that the explanation only discusses *linear motion* DR vectors for the ease of explanation. Other complex dead reckoning types can also be used to calculate the export error over time in a similar fashion.

Now, to calculate the error over time T_2 to T_8 (observe that the entity does not exist on the receiver till time T_2, hence we cannot calculate the error before T_2), the error due to the difference in the trajectories of the entity at the sender and the receiver over total time has to be calculated. To make it easier, we break it into time slices and calculate the error over each time slice.

Total Export Error(senders trajectory, receivers trajectory,T_2, T_8) =

$$
\begin{aligned}
& \text{Err}(DR_1, DR_1, T_2, T_3) \\
+ \; & \text{Err}(DR_2, DR_1, T_3, T_4) \\
+ \; & \text{Err}(DR_2, DR_2, T_4, T_5) \\
+ \; & \text{Err}(DR_3, DR_2, T_5, T_6) \\
+ \; & \text{Err}(DR_3, DR_3, T_6, T_7) \\
+ \; & \text{Err}(DR_4, DR_3, T_7, T_8)
\end{aligned}
$$

-------- Equation 3.1

Consider Err(DR_2, DR_1,T_3, T_4), The sender's trajectory can be represented as a function of time as ($X_2(t)$; $Y_2(t)$ = ($X_2 + vx_2 X t$; $Y_2 + vy_2 X t$), where <X_2, Y_2, vx_2, vy_2> is the state of the entity at T_3 when DR_2 was triggered. DR_2 contains the state information and communicates that to the receiver. The receiver's trajectory can be similarly represented as ($X_1(t)$; $Y_1(t)$ = ($X_1 + vx_1 X t$; $Y_1 + vy_1 X t$), where <X_1, Y_1, vx_1, vy_1> is the state of the entity that was contained in DR_1. It can be seen that in the figure that the last received DR by the receiver before T_3 is DR_1 and is using it till time T_4.
The distance between the two trajectories as a function of time then becomes,

$$dist(t) = \sqrt{((X_2(t) - X_1(t))^2 + (Y_2(t) - Y_1(t))^2}$$ ---------- Equation 3.2

$$= \sqrt{((X_2 - X_1) + (vx_2 - vx_1)t)^2 + ((Y_2 - Y_1) + (vy_2 - vy_1)t)^2}$$

$$= \sqrt{((vx_2 - vx_1)^2 + (vy_2 - vy_1)^2)t^2}$$
$$+ 2((X_2 - X_1)(vx_2 - vx_1) + (Y_2 - Y_1)(vy_2 - vy_1))t$$
$$+ (X_2 - X_1)^2 + (Y_2 - Y_1)^2$$

Let

$$a = 2((X_2 - X_1)(vx_2 - vx_1) + (Y_2 - Y_1)(vy_2 - vy_1))$$

$$b = ((vx_2 - vx_1)^2 + (vy_2 - vy_1)^2)$$

$$c = (X_2 - X_1)^2 + (Y_2 - Y_1)^2$$

The dist(t) can be written as,

$$dist(t) = \sqrt{(aXt^2 + bXt + c)}$$

Then *Err* (DR_2, DR_1,T_3, T_4) for time interval [t_3, t_4] becomes

$$\int_{t3}^{t4} dist(t)\, dt = \int_{t3}^{t4} \sqrt{(aXt^2 + bXt + c)}$$

A closed form of the integral is given by:

$(2at + b)(\sqrt{(aXt^2 + bXt + c)})/ 4a$

$+ ½ \ln(((1/2b + at)/ \sqrt{a}) + \sqrt{(aXt^2 + bXt + c)})c/\sqrt{a}$

$- 1/8 \ln(((1/2b + at)/ \sqrt{a}) + \sqrt{(aXt^2 + bXt + c)})b^2a^{-3/2}$

----- Equation 3.3

Applying the same method, all the other parts of the error can be calculated and hence the Total Export Error(T_2, T_8) can be calculated by adding all the errors as explained in equation 3.1. Refer Appendix B for implementation details.

It can be seen from the figure that the red areas where the trajectories of the sender and the receiver are off each other indicate the *before export error* and the areas where the dotted line overlaps the solid line indicates the *after export error* which is zero as a result of the accuracy model. The next section explains the details of the scheduling algorithm.

3.3 Scheduling Algorithms:

The goal of the DR vector scheduling algorithm is to achieve fairness by equalizing the cumulative export error at all receivers at every trigger. The algorithm achieves this by scheduling the DR vector to be sent to the receivers in such a way that the receiver having the highest cumulative export error gets the DR vector instantaneously and the other receivers get the DR vector at a schedule calculated based on the difference between the highest cumulative export error and the receivers cumulative export error.

The algorithm is running at the sender/owner of the entity being dead reckoned by all the receivers. The sender keeps track of the cumulative export error (Export error over time) for each receiver. At every trigger, the sender calculates a scheduling instant for each receiver such that the cumulative export error of the receiver equals the highest

cumulative export error at that point in the future when it will get the scheduled DR vector.

3.3.1 Calculating Cumulative Export Error:

The first step in the algorithm is to keep track of the cumulative export error each receiver is incurring. As this information has to be known at the sender side so that the sender can use this information while scheduling, a *feedback mechanism* from the receiver is required. At the reception of every DR vector the receiver sends an acknowledgment to the sender indicating the DR vector identifier (a sequence number) and the time (clocks are synchronized between the sender and the receiver) at which the DR was received. The sender gets this feedback and keeps track of the trajectory of the entity at each receiver. From section 3.2 it is clear that after knowing the sender's and the receiver's trajectory in a certain time interval, we can calculate the export error over that time interval using the closed form of the integral. At the receipt of every acknowledgment, the sender calculates the cumulative export error till the time indicated by the feedback as the reception time of the respective DR vector. At the next receipt of the feedback from the receiver, the sender has to only calculate the export error from the time when it had calculated the cumulative export error last time till the time indicated in the next feedback and add the amount to the existing cumulative export error (CEE).

The sender follows the same procedure for calculating the CEE for each receiver at the receipt of the feedback from the respective receiver. This way, the sender keeps track of the cumulative export error at each receiver.

3.3.2 Equalizing Errors: Calculating Scheduling Instants

At every trigger the sender tries to equalize the export errors over time of each receiver. The sender calculates the CEE of each receiver till a point in the past indicated as the time of receipt of the latest DR vector by that receiver. To compare the CEE's of

receivers, they should be between the same time intervals. Therefore, the sender has to estimate the export error from that point of time to the current time for each receiver, referred to as Estimated Export Error, and add the estimated export error (EEE) to CEE resulting in Accumulated Export Error (AEE) for the respective receiver.

The calculation of the estimated export error is same as the calculation of the CEE mentioned earlier but only differs in one aspect. As the sender has no knowledge of the receiver's trajectory for this calculation, the sender uses the best estimate of the receiver's trajectory by assuming that the receiver gets all the DR vectors it has generated in that time interval without any delay. If there is a mechanism for estimating the delay between the sender and the receiver, it can be used to better the estimate by estimating the receiver's trajectory more accurately in this time interval by using the delay estimate.

Once the AEE is known for each receiver till the current time, the aim is to equalize the AEE of all the receivers by sending the current DR vector to the receiver with the highest AEE instantly and schedule the DR vector to be sent to the remaining receivers in such a way that all the receiver's AEE equals the highest AEE after they receive the current DR vector.

Suppose there are two receivers R1 and R2 receiving DR vectors for dead reckoning an entity owned by the sender. If the AEE incurred by R1 is greater than R2 at the trigger of the current DR vector then the following condition holds,

$$\int_{0}^{\text{current Time}} \text{distR1-sender}(t)dt = AEE(R1) = AEE(R2) + \int_{\text{current time}}^{T} \text{distR2-sender}(t)dt \quad \text{----------Equation 3.4}$$

In the above equation, everything except T is known. The scheduling instant T has to be chosen such that the above equation is satisfied.

Generalizing the above case to n receivers, the algorithm calculates T_m for receiver m and schedules the DR to be sent to the receiver m at T_m, where m takes the values between 1 and n.

There is no closed form solution to find the value of T from the above equation. Consequently, a binary search is applied starting with the highest value possible. As dead reckoning uses a time threshold along with other thresholds, the value of the time threshold can be used as the highest possible value. If no other trigger happens within the specified time threshold, we know that there will be a trigger at the next time threshold. This is possible because at the next trigger all the schedules beyond that point are going to be recalculated. The binary search for T starts from the highest value and we keep on checking with the value if the above equation is satisfied till T reaches 0. At any point when the equation is satisfied, that value is chosen as the scheduling instant for the receiver. The receivers whose schedule is beyond T are tagged accordingly and the algorithm makes a decision of discarding (scheduling algorithm #1) /sending (scheduling algorithm #2) them at the next trigger.

3.3.3 Scheduling Algorithm #1

The previous sections discussed the method to calculate the Cumulative Export Error of individual receivers and also discussed the calculation of scheduling instants. This section puts it all together and discusses the scheduling algorithm #1 in steps. The steps are as follows:

1. At every trigger of a DR vector, the sender has knowledge of the CEE of each receiver till certain point of time in the past (the time at which the latest DR vector was received by the respective receiver). The sender estimates the EEE for each receiver from that point in the past till the current time. If the delay estimate of the receivers is known, then that information is used here in the calculation of EEE.

This way, the sender calculates the AEE of each receiver by adding EEE to its respective CEE.

2. The sender now compares the AEE of each receiver with every other receiver to find out the receiver with the highest AEE, say receiver X.
3. The DR vector generated at the current trigger is immediately sent to the receiver having the highest AEE, receiver X.
4. For every other receiver the sender calculates a scheduling instant in the future such that, the AEE of the receiver equals the AEE of the receiver X at that point in the future.
5. *If a new DR vector is triggered before all the scheduled DR vectors are sent out, the DR vectors scheduled beyond the new trigger point are dropped and not sent to the remaining receivers and steps 1 to 5 are repeated.*
6. At the receipt of every feedback message from the receiver, the CEE is updated for the respective receiver.

3.3.3.1 Instrumentation with BZFlag

A test bed was setup with four players running modified version of BZFlag. The scheduling algorithm #1 and the base case where each DR vector was sent to every receiver at the every trigger were implemented in the same run by tagging the DR vectors according to the type of the algorithm used to send the DR vector. Both the base case and the scheduling case incorporate accuracy modifications mentioned in the previous chapter. NISTNet was used to introduce delays across the sender and the three receivers. Different runs with static and variable delays between the sender and the receivers were done. The receivers were at a mean delay of 800, 500 and 200ms from the sender respectively and three different runs with no delay variance, moderate delay variance (100ms +/- mean) and high delay variance (180ms +/- mean) were done.

The sender calculated the Cumulative Export Error of each receiver using a feedback mechanism from the receivers indicating the time at which they received a particular DR vector. The sender calculated export error over time for linear and circular motion DR vectors. The sender logged the CEE and AEE of each receiver in every iteration of the game loop for both the scheduling algorithm and the base case. The sender also kept track of the standard deviation and the mean of the AEEs of all the receivers in every iteration of the game loop.

3.3.3.1.1 Observations:

3.3.3.1.1.1 Standard Deviation:

The standard deviation of the AEE of all the receivers was used as the measure of fairness achieved by using a particular algorithm. The following graphs shows the standard deviation of the AEE of all the receivers in the scheduling case against the standard deviation of the AEE of all the receivers in the base case under different runs with no delay variance, moderate delay variance and high delay variance. It can be observed that the standard deviation due to the scheduling case is much lower as compared to the standard deviation in the base case. This means that the AEE's of the receivers in the scheduling case are closer to the AEE of the other receivers at the same time instant. This indicates that the scheduling approach achieves a higher degree of fairness among the receivers even if they are having different latencies to the sender.

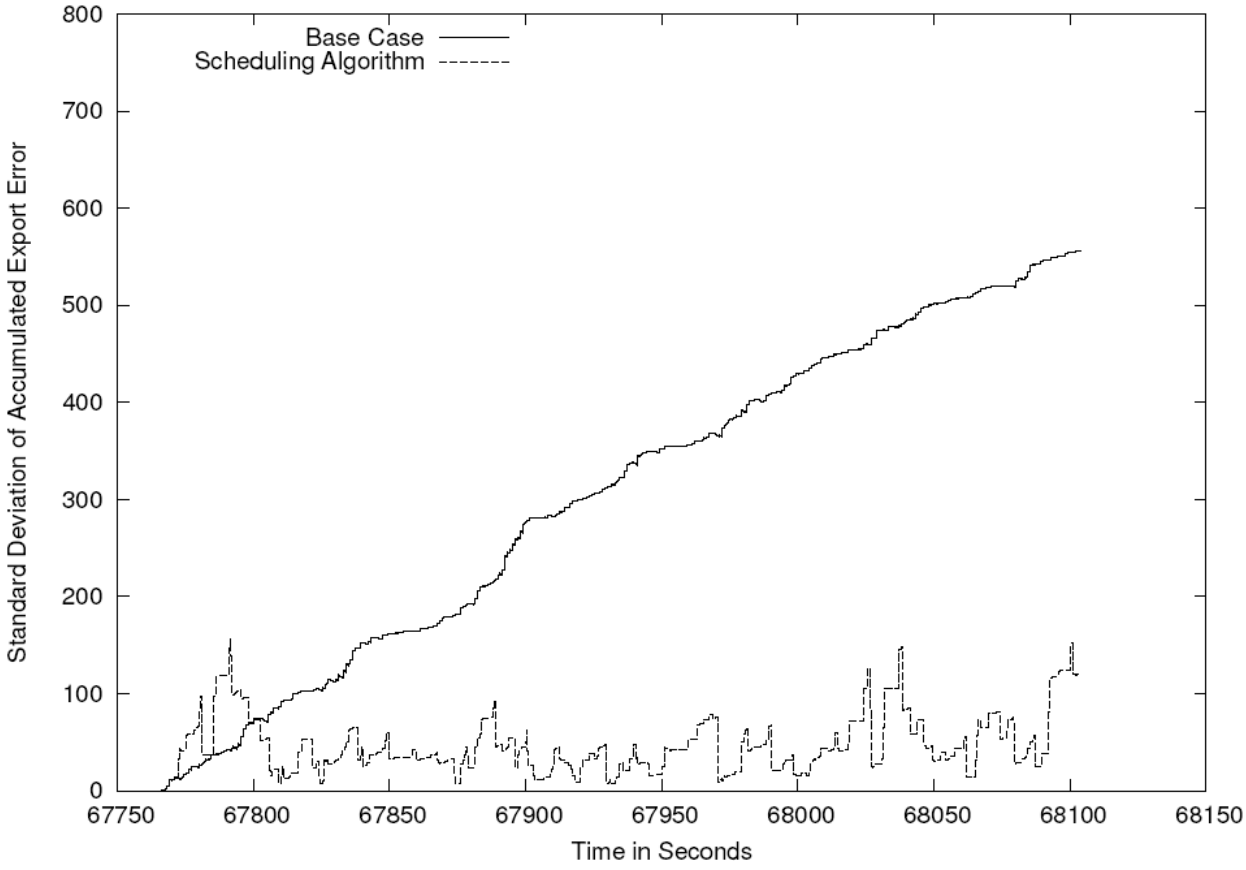


Figure 3.2 Standard Deviation - No Delay Variance (Scheduling Algorithm #1)

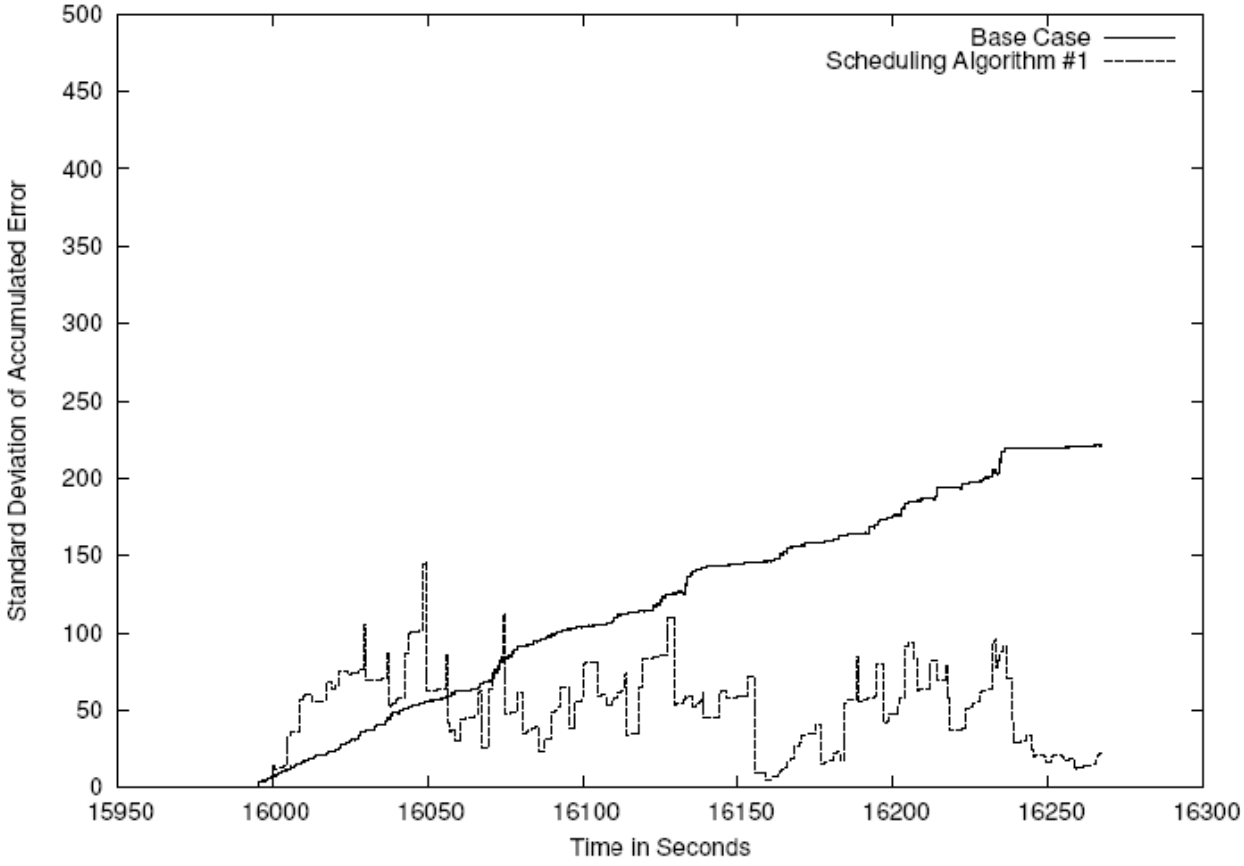


Figure 3.3 Standard Deviation - Moderate Delay Variance (Scheduling Algorithm #1)

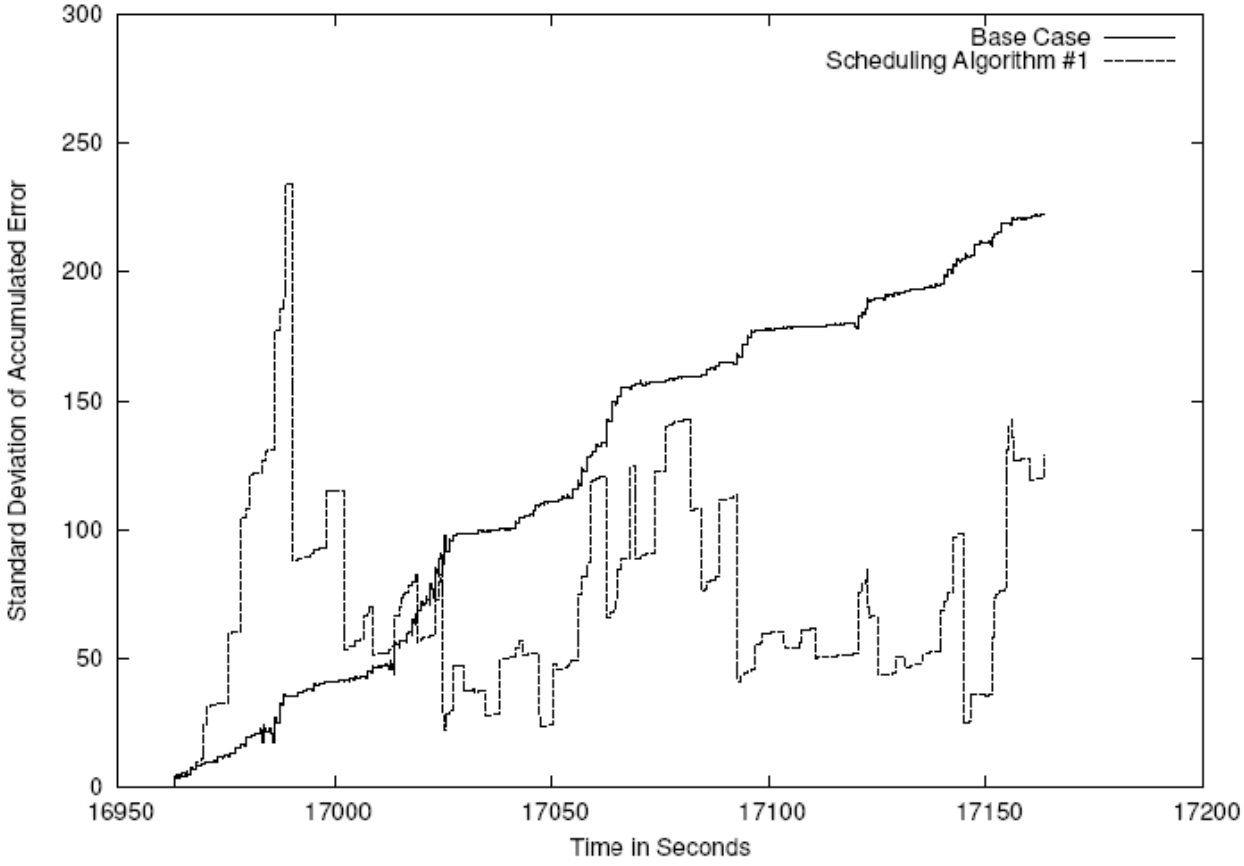


Figure 3.4 Standard Deviation - High Delay Variance (Scheduling Algorithm #1)

3.3.3.1.1.2 Mean of the Accumulated Export Error:

The mean of the accumulated export error was used as a measure of the overall inaccuracy of the system and hence a measure of qualitative degradation of game plays.

The graph of the mean of AEE's of all the receivers was plotted to compare the cost of using the base case against using the scheduling case #1. It can be observed from the graphs that the mean of the AEE increased multifold in the scheduling case #1 in

comparison to the mean of the AEE in the base case. This means that the overall inaccuracy of the system was much higher than the base case.

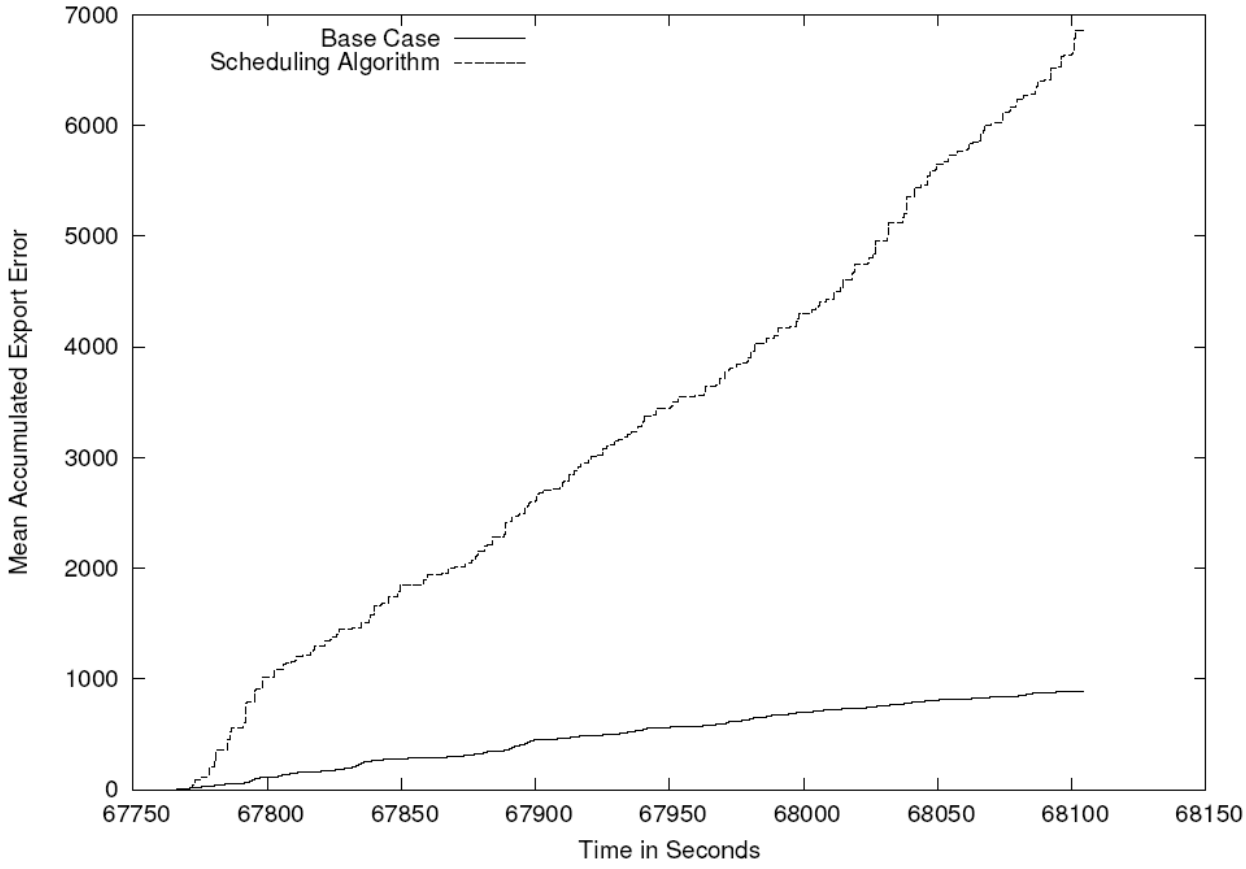


Figure 3.5 Mean - No delay variance (Scheduling Algorithm #1)

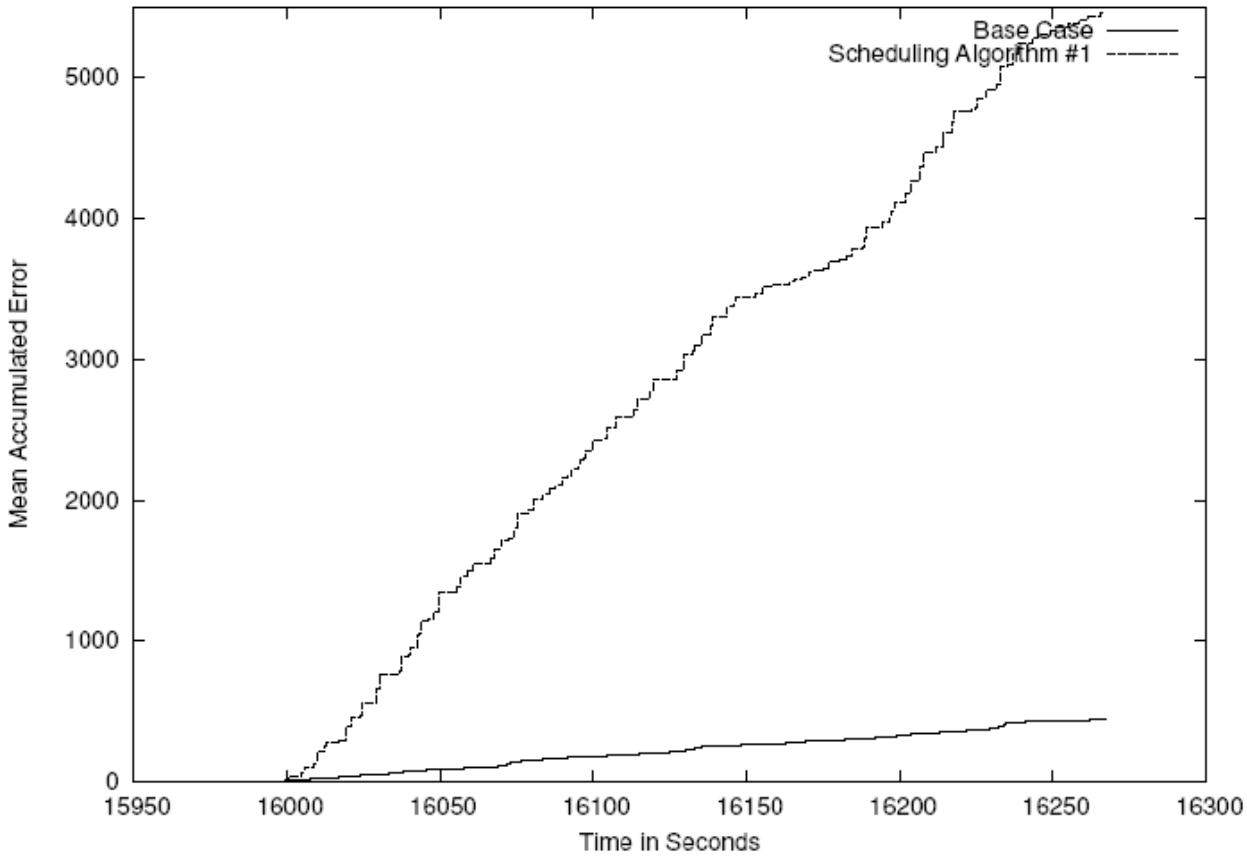


Figure 3.6 Mean - Moderate Delay Variance (Scheduling Algorithm #1)

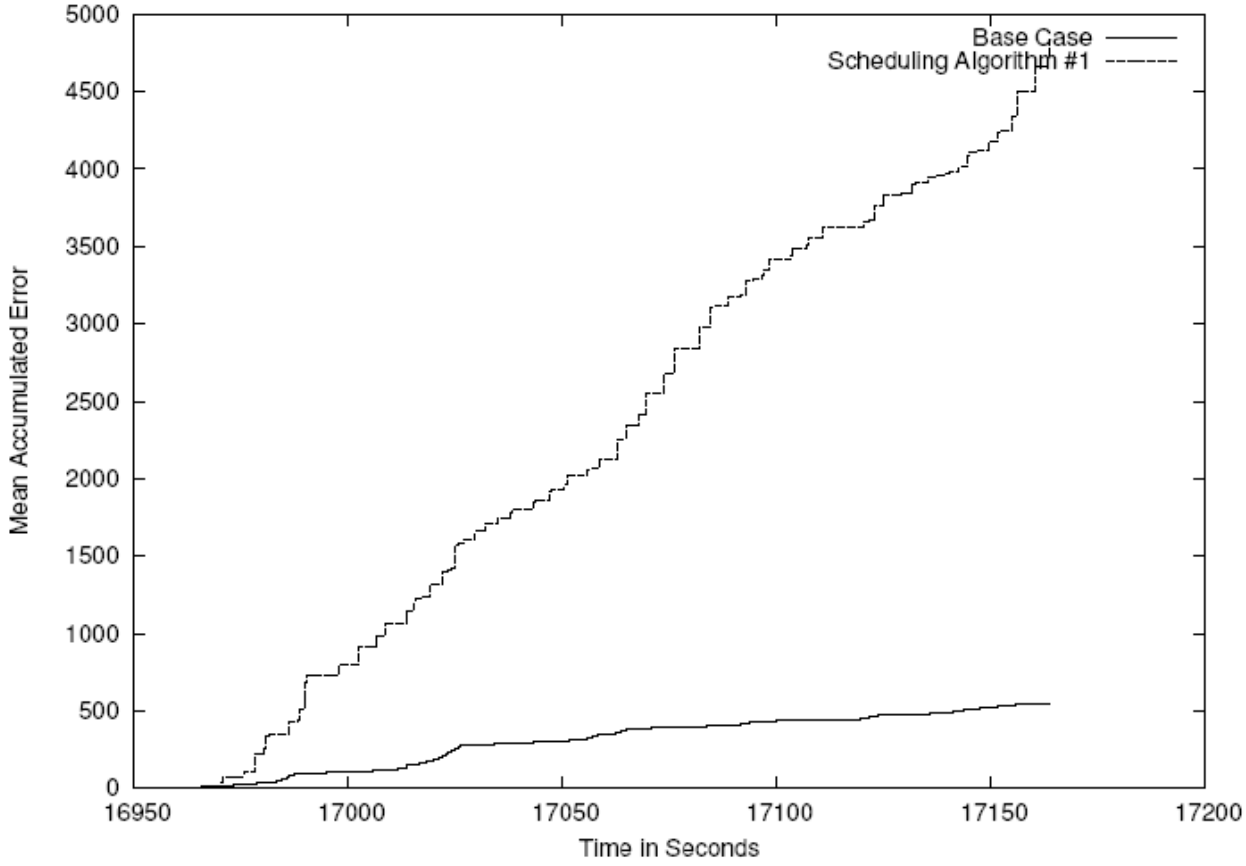


Figure 3.7 Mean - High Delay Variance (Scheduling Algorithm #1)

3.3.3.1.1.3 Accumulated Export Error at each receiver:

The chart of accumulated export error at each receiver over the whole run in both the cases lets us compare the actual error accumulated by each receiver. It can be clearly seen from the graphs that the accumulated export error of each player is greater in the scheduled case as compared to its accumulated error in the base case. This means that even though there is fairness in the game, it leads to degradation in the qualitative game play as each receiver is more off from the sender's trajectory than in the base case. The degradation would have been acceptable if it was limited to the accumulated export error of the highest AEE receiver in the base case. But it is observed that the

export errors of receivers are much greater than the highest export error in the base case.

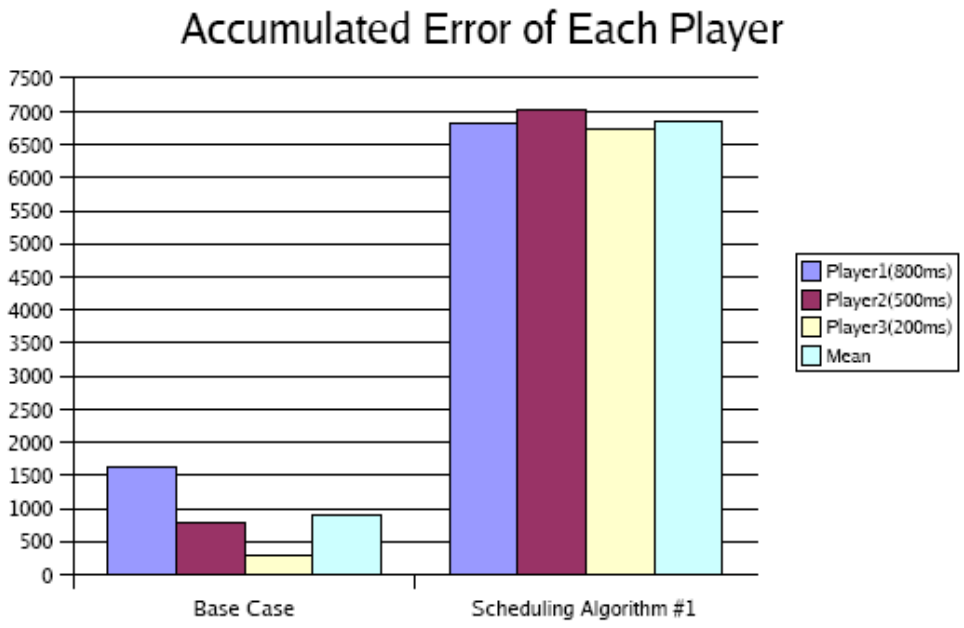


Figure 3.8 Accumulated Export Error - No delay variance (Scheduling Algorithm #1)

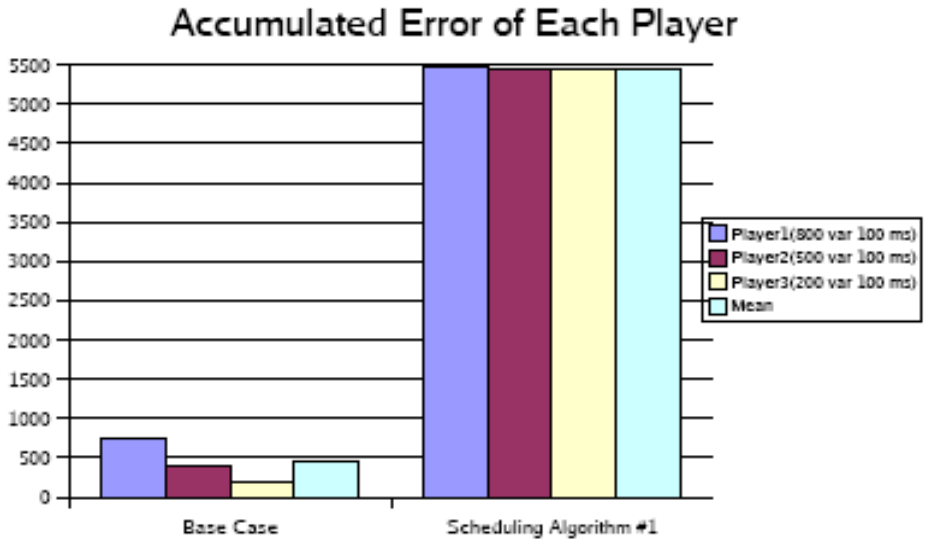


Figure 3.9 Accumulated Export Error - Moderate delay variance (Scheduling Algorithm #1)

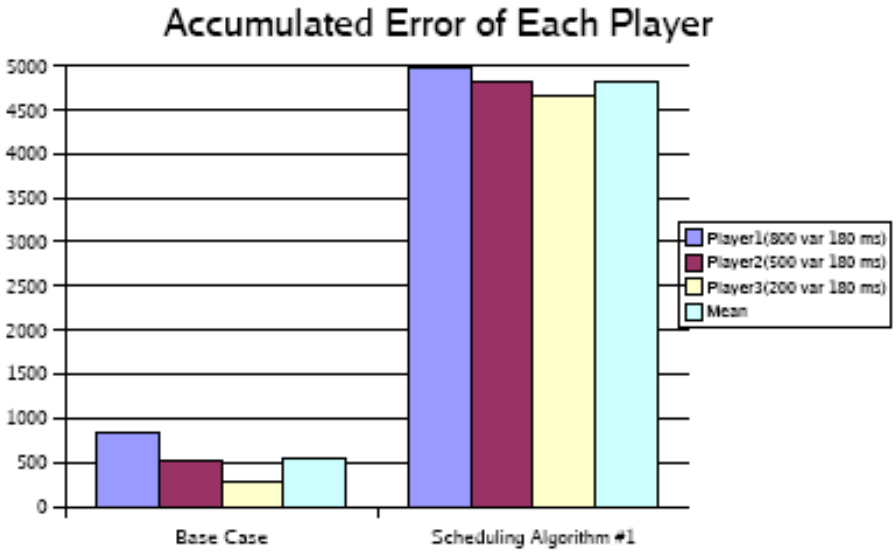


Figure 3.10 Accumulated Export Error - High delay variance (Scheduling Algorithm #1)

3.3.3.2 Conclusion:

The above observations show that the scheduling algorithm achieves fairness at the expense of increasing the accumulated export error for all the receivers. This happens due to the fact that when a schedule for a receiver is beyond the next trigger, the current DR vector is never sent to the receiver. Even though we cut down on the total number of DR vectors sent to all the receivers, this leads to a high error build up at the receivers. Also, when a DR vector is not sent for a long time to a receiver, the receiver's trajectory is way off of the sender's trajectory and hence the receiver starts building up the error at a very high rate. When a DR vector scheduled in the future is not sent due to a new trigger happening before the schedule time, it leads the receiver sometimes to build up error beyond the highest export error and hence making this receiver the one with the highest export error. Consequently, at the next trigger, the algorithm tries to equalize the errors of other receivers to this new highest error. This leads to a *hysteresis effect*, which increases the accumulated export error of receivers beyond their accumulated export error observed in the base case. This *hysteresis effect* keeps happening continually and leads to an unbounded increase in the export error of all the receivers.

This leads us to the conclusion that even though this algorithm achieves fairness among the receivers, it leads to a high increase in the accumulated error of the receivers scheduled to receive a DR vector beyond the next trigger by discarding the DR vectors scheduled beyond the next trigger. This, in turn, over a period of time, increases the accumulated error of each receiver due the *hysteresis effect* as explained above. The algorithm presented in the next section tries to overcome this high accumulated error buildup problem of the scheduling algorithm #1.

3.3.4 Scheduling Algorithm #2:

The scheduling algorithm #1 achieves a higher degree of fairness by trying to equalize the error in a single shot by scheduling the DR vectors in the future. When the DR vectors scheduled beyond the next trigger are discarded, it leads to the faster accumulated error build up at those receivers and leads to the *hysteresis effect* which leads to overall increase in the accumulated error of each receiver in comparison to the base case. Scheduling algorithm #2 proposes a modification to scheduling algorithm #1 so as to overcome the unbounded rate of error buildup caused in the earlier case and also to reduce the effect of the hysteresis effect happening in the scheduling algorithm #1.

It proposes that instead of discarding the DR vectors scheduled beyond the next trigger, if they are all sent at the next trigger, it will allow the receiver to buildup some error but less than what would be required to equalize it to the highest accumulated export error. This allows the standard deviation of error among the receivers to fall down in comparison with the base case and at the same bound the build up of error at all the receivers. The modified algorithm is presented below.

3.3.4.1 Scheduling Algorithm #2:

The scheduling algorithm #1 is modified in such a way that no DR vector is discarded at the next trigger. Instead, all the DR vectors scheduled beyond the next trigger are sent out at the next trigger to the respective receivers. The algorithm is as follows:

1. At every trigger of a DR vector, the sender has knowledge of the CEE of each receiver till certain point of time in the past (the time at which the latest DR vector was received by the respective receiver). The sender estimates the EEE for each receiver from that point in the past till the current time. If the delay estimate of the receivers is known, then that information is used here in the calculation of EEE

as explained previously. This way, the sender calculates the AEE of each receiver by adding EEE to its respective CEE.

2. The sender now compares the AEE of each receiver with every other receiver to find out the receiver with the highest AEE, say receiver X.
3. The DR vector generated at the current trigger is immediately sent to the receiver having the highest AEE, receiver X.
4. For every other receiver the sender calculates a scheduling instant in the future such that, the AEE of the receiver equals the AEE of the receiver X at that point in the future.
5. *If a new DR vector is triggered before all the scheduled DR vectors are sent out, the DR vectors scheduled beyond the trigger point are sent out at the trigger to the respective receivers and steps 1 to 5 are repeated.*
6. At the receipt of every feedback message from the receiver, the CEE is updated for the respective receiver.

3.3.4.2 Instrumentation with BZFlag – Scheduling Algorithm #2

A test bed was setup with 4 players running modified version of BZFlag. The scheduling algorithm #2 and the base case where each DR vector was sent to every receiver at the every trigger were implemented in the same run by tagging the DR vectors according to the type of the algorithm used to send the DR vector. Both the base case and the scheduling case #2 incorporate accuracy modifications mentioned in the previous chapter. NISTNet was used to introduce delays across the sender and the three receivers. Static delays of 800, 500 and 200 milliseconds were introduced between the sender and first, second and the third receiver respectively.

The sender logged the CEE and AEE of each receiver in each iteration of the game loop for both the scheduling algorithm #2 and the base case. The sender also kept track of the standard deviation and the mean of the AEEs of all the receivers in every iteration of the game loop.

3.3.4.2.1 Observations:

3.3.4.2.1.1 Standard Deviation:

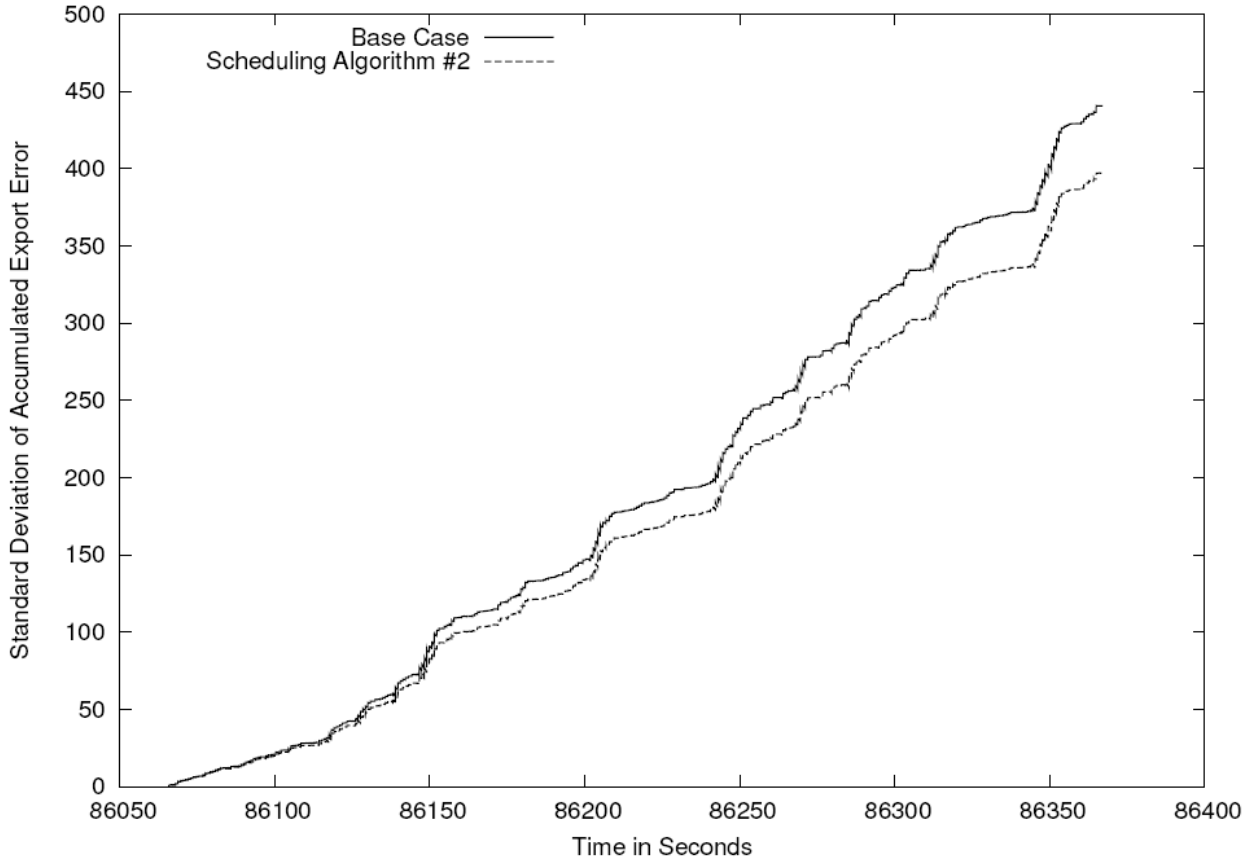


Figure 3.11 Standard Deviation – Scheduling Algorithm #2

It can be seen from the graph that the standard deviation of the accumulated export error at the receivers in the scheduling case #2 is lower than as compared the standard deviation in the base case. This is attributed to the scheduling of DR vectors in such a way that the receiver having the highest AEE gets it first and the others receive it according to their schedules or at the next trigger, whichever is earlier.

3.3.4.2.1.2 Mean of the accumulated export error:

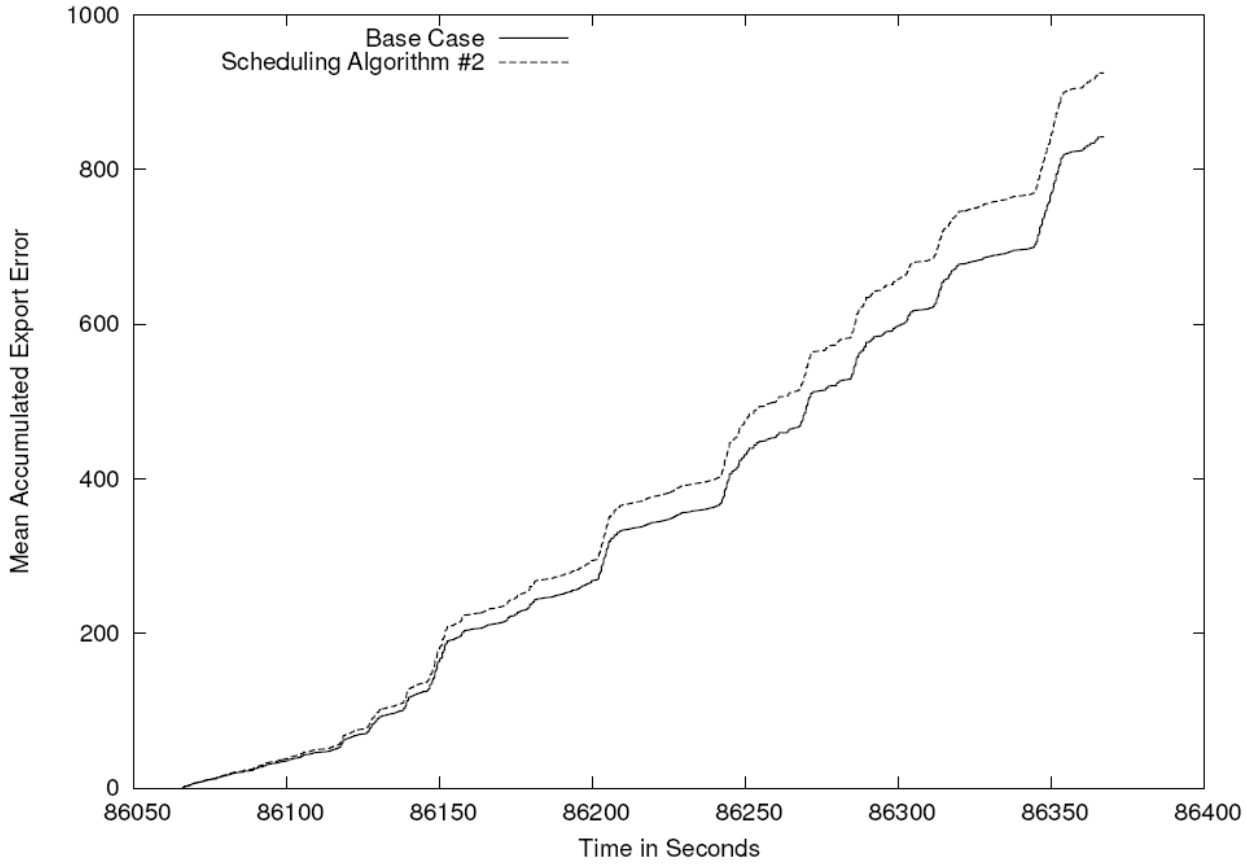


Figure 3.12 Mean – Scheduling Algorithm #2

The graph of the mean shows that there is a moderate increase in the mean of the accumulated export error due to scheduling algorithm #2 as compared to the base case. Recall that the increase in the mean in the scheduling case #1 was considerable in comparison to the base case. This is due to the fact that scheduling algorithm #2 overcomes the unbounded error build up problem and at the same time alleviates the hysteresis effect observed in scheduling algorithm #1. Even though it overcomes those problems, the increase in mean suggests that the overall inaccuracy of the system has increased.

3.3.4.2.1.3 Accumulated error of each receiver:

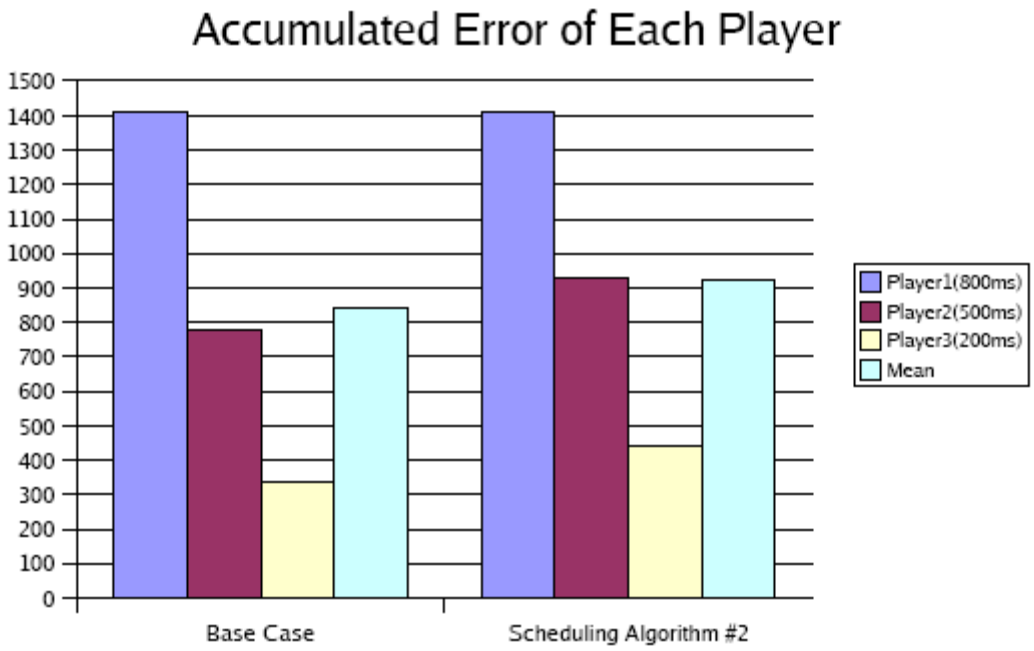


Figure 3.13 Accumulated Export Error – Scheduling Algorithm #2

From this chart, it can be seen that the player 3 – who is the farthest player is doing as good as in the base case and the other players have incurred more AEE and hence achieving fairness to an extent among all the receivers. It also visible that the mean of the accumulated export error has gone up in comparison with the base case indicating the degradation in the overall game play.

3.3.4.3 Conclusion:

We can conclude that scheduling algorithm #2 overcomes the unbounded rate of error build up problem of algorithm #1. It should be noted that, even though the *hysteresis effect* in algorithm #1 has been alleviated by algorithm #2, it will be evident in a variable delay environment. In a variable delay environment where the actual receipt of the DR vector might be different from what was expected at the time of calculating the schedule will cause the hysteresis effect, though the impact of the effect is negligible as

compared to the scheduling algorithm #1. What has happened is the rate of convergence of the accumulated export error of all the receivers has dropped down considerably as compared to algorithm #1. Also, the overall error of the system has increased indicating that the scheduling approach is degrading the qualitative game play for all the receivers.

This motivated us to explore algorithms where we can achieve fairness without degrading the overall performance of the game. We discuss some budget based algorithms on these lines in the next section.

3.4 Budget Based Algorithms:

The scheduling algorithms discussed in the previous section focus on the issue of fairness from the perspective of equalizing AEE of all receivers to the highest AEE in the system. Both the approaches lead to an increase in the mean of the accumulated error of all the receivers indicating a degradation of the qualitative game play for all the receivers. This led us to explore different ways to achieve fairness among the receivers without the drawbacks observed in the scheduling approaches.

The intuition behind the approaches discussed in this chapter is as follows:

The sender knows the AEE of each receiver continuously. If using the same budget of DR vectors as in the base case over time, we can send DR vectors more frequently to the receiver with a higher AEE and send DR vectors at a lower frequency to the receiver with a lower AEE, we can ensure fairness among the receivers. Intuitively, this will also lead to lowering the AEE of the receiver having high AEE in the base case. This is possible due to the fact that we can send more DR vectors, to the receivers with higher AEE, than in the base case at the cost of sending less DR vectors, to the receivers with lower AEE, than in the base case and still maintaining the same budget of DR vectors as in the base case.

The approaches discussed in this section are based on several factors and using probability functions over them.

The idea here is that at every trigger the sender knows certain information about every receiver like

- The export error caused by the last DR vector
- The last time a DR vector was sent to that respective receiver

The sender can compare this information of each receiver with every other receiver and find out receiver(s) who are the most deserving ones to get the current DR vector.

The following section discusses a probabilistic budget based algorithm and the next section discusses a deterministic budget based algorithm.

3.4.1 Probabilistic Budget Based Algorithm:

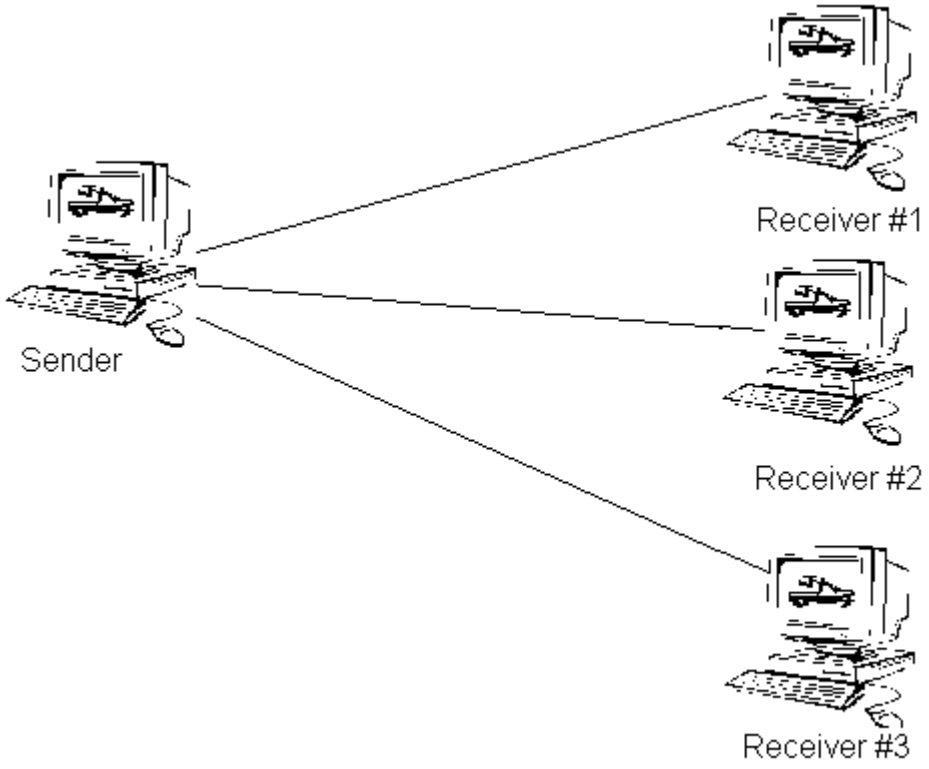


Figure 3.14 Probabilistic Budget Based Algorithm

The above figure shows four players in which the sender is controlling an entity and that entity is being dead reckoned by the three receivers. It is assumed that the setup uses the *accuracy model* as discussed in the second chapter. Consequently, the only error that is present is the *before export error*.

The probabilistic budget based algorithm can be broken down into three parts as follows:

1. Lowering the dead reckoning threshold
2. Calculating the relative probabilities of each player based on some factors.
3. Decision making of whom to send the DR vector at every trigger based on the probabilities calculated in step 2.

3.4.1.1 Lowering the dead reckoning threshold:

Consider the simple case where every time a threshold is crossed, the sender generates a DR vector and that DR vector is sent to all the receivers. The idea is to lower the threshold such that it results in more triggers. At each trigger, send out DR vector to fewer receivers as opposed to all the receivers in the base case, but send same number of DR vectors over time (as the threshold is smaller). By choosing the appropriate receiver(s) to send the DR vector so that all the receivers incur same AEE, fairness can be achieved among all the receivers.

The threshold is lowered such that more triggers are generated in the same time. If 'n' is the number of DR vectors sent at every trigger by the sender and 't' is the number of triggers caused by the threshold in the base case in a certain amount of time and n_1 and t_1 are the number of DR vectors sent at each trigger and number of triggers caused in the same amount of time respectively in the modified case, then to use the same budget as the base case in the modified base case, the following condition needs to be satisfied.

$nXt = n_1Xt_1$ ---------- Equation 3.5

It can be seen that $t_1 > t$ (the threshold was lowered, which leads to more triggers) hence, to satisfy equation 3.5, $n_1 < n$ which means that with the new threshold we have to send less than n DR vectors at every trigger. This in turn means that at every trigger we need not send the DR vector to all the receivers.

In the above figure it can be seen that in the base case n=3 and let us assume that there were in all 100 triggers over a fixed amount of time say X seconds. Hence, it makes nXt = 300 DR vectors generated over X seconds. Now, the threshold is lowered such that it leads to t_1 triggers. Then n_1 should be chosen such that the equation 3.5 holds true. Suppose we chose t_1 as 300, which mean that we lower the threshold such that instead of 100 triggers in X seconds in the base case, now there will be 300 triggers in X seconds. From the equation 3.5, we get the budget per trigger, n_1, to be 1. Hence, at every trigger we need to send DR vector to 1 in 3 receivers as opposed to sending it to all the receivers in the base case. *The lower the threshold, the better as we can have a higher frequency of sending DR vectors to the receiver with higher error.*

3.4.1.2 Calculating the relative probabilities of each player:

Now the next question that arises is that which receiver(s) the DR vector should be sent to if we know the number of DR vectors to be sent at the trigger. It is reminded that, the number of DR vectors is less than the number of receivers. The decision of whom to send the DR vector can take into account many factors. The factors are as follows:

1. Accumulated Export Error incurred by the receiver till the current time
2. Instantaneous Export Error
3. Last time when a DR vector was sent to the receiver.
4. A combination of the above factors by assigning different weights to each of them.

3.4.1.2.1 Accumulated Export Error:

The sender keeps track of the exported trajectory and at the same time also keeps track of the trajectory of the entity on all the receivers based on the feedback from them. The difference in the exported trajectory and the receiver's trajectory integrated over the time from which the dead reckoning for that entity was started till the current time is called as the accumulated export error. This can be used as a measure to decide which receiver(s) deserve to get the current DR vector. The more the accumulated export error the receiver(s) experience, the more it deserves to get the current DR vector. The probabilities are assigned accordingly. The probabilities are calculated as follows:

PA(Receiver X) = Accumulated Export Error of Receiver X/ Total Accumulated Error of all
the receivers

3.4.1.2.2 Instantaneous Export Error:

The instantaneous export error is most recent component of the accumulated export error. The integrated export error from the time the DR vector, which is currently used by the receiver, was generated at the sender to the current time is called as the instantaneous export error. The more the instantaneous export error the receiver(s) experience, the more the receiver(s) deserves to get the current DR vector. Probabilities are assigned in the similar way as done in case of accumulated error.

PI(Receiver X) = Instantaneous Export Error of Receiver X/ Total Instantaneous Error of all
the receivers

3.4.1.2.3 Last Time when a DR vector was sent to the receiver:

The sender keeps track of the last time it sent a DR to each receiver. This way at every trigger, the sender can send the DR vector to the most deserving receiver(s) who have not received a DR vector recently. The probabilities are assigned as follows:

Suppose,
TimeDiffX(Current Time – Last time) = Time difference between the current time and the last time the receiver X got a DR vector.

PT(Receiver X) = TimeDiffX(Current Time – Last Time)/ Sum of TimeDiff(Current Time – Last time) of all the receivers

3.4.1.2.4 A combination of the above factors by assigning different weights to each of them:

Each of the factors is assigned a weight between 0 and 1, such that all the weights add up to 1. The sum of the product of the weight and the probability value associated with the factor gives us the new probability value.
Suppose,
Probability based on Accumulated Error = PA,
Probability based on Instantaneous Error = PI,
Probability based on Last Time DR was sent = PI,
And Weight associated with PA is α1, PI is α2 and PT is α3
Where, α1 + α2 + α3 = 1
Then,
Probability based on the Combination PC = α1 X PA + α2 X PI + α3 X PT

Varying the weights assigned to each factor lets us compare the effect of each factor in the overall fairness of the game and hence compare the different approaches among them. Refer Appendix B for implementation details.

3.4.1.3 Decision making of whom to send the DR vectors at every trigger based on the probabilities calculated in step 2

The following steps are followed at each trigger except the first trigger:

1. Depending on the factor to be used, the sender calculates the relative probabilities of each receiver as explained in 3.4.1.2.
2. For each receiver, this value is multiplied with the budget available at each trigger (calculated in 3.4.1.1) to give the frequency of sending the DR vector to each receiver.
3. If any of the receiver's frequency after multiplying with the budget goes over 1, the receiver's frequency is set as 1 and the surplus amount is equally distributed to all the receivers by adding the amount to their existing frequencies. This process is repeated until all the receivers have a frequency of less than or equal to 1. This is due to the fact that at a trigger we must not send more than one DR vector to the respective receiver. That will be wastage of DR vectors by sending redundant information.
4. Then, for each receiver, it generates a random number between 0 and 1 and based on the fact that if the number is between 0 and the *frequency* calculated in step 3 it decides if to send or not to send the DR vector to the respective receiver. If the random number is in the interval [0-Frequency], the DR vector is sent, else it is not sent to the respective receiver. In this way ensuring that the budget of DR vectors over time is maintained the same as in the base case.

At the first trigger, the sender sends a DR vector to all the receivers.

3.4.1.4 Instrumentation with BZFlag

To compare the various approaches mentioned above, a test bed was setup with 4 players running modified version of BZFlag. NISTNet was used to introduce delays across the sender and the three receivers. Delays of 800, 500 and 200 milliseconds were introduced between the sender and first, second and the third receiver respectively. The sender tracked the following information about each receiver:

1. Accumulated Export Error (AE)
2. Instantaneous Export Error (IE)

3. Last Time a DR vector was sent to the respective receiver. (LT)

Five different cases with the following probabilities were implemented in BZFlag. The cases are as follows:

Case #	Probability Associated with Accumulated Export Error *(PA)*	Probability Associated with Instantaneous Export Error *(PI)*	Probability Associated with Last time a DR vector was sent *(PT)*
Case 1	1	0	0
Case 2	0	1	0
Case 3	0	0	1
Case 4	1/3	1/3	1/3
Case 5	½	½	0

BZFlag was modified in such a way that at every trigger caused by a threshold, all the five cases mentioned above got executed and chose the appropriate receiver to send the DR vector and tagged the DR vector and sent it. Each of the above cases works in the manner explained in section 3.4.1. As the dead reckoning threshold in the base case was already very fine, it was decided that instead of lowering the threshold, the approaches would be compared against a modified base case that would use a higher threshold. The base case was modified such that every third trigger would be actually used to send out a DR vector to all the players. This was called as the 1/3rd base case as it resulted in 1/3 number of DR vectors being sent as compared to the base case. The budget per trigger for the probability based approaches was calculated as one DR vector at each trigger as compared to 3 DR vectors at every 3rd trigger in the 1/3 base case, hence satisfying the equation 3.5.

3.4.1.4.1 Observations:

3.4.1.4.1.1 Standard Deviation of the Accumulated Export Error:

All the five approaches were compared against each other and 1/3 base case. The standard deviation of the accumulated export error of all the receivers in each case was a measure of the overall fairness achieved by the particular approach. The graph shows the standard deviation of the accumulated export error for each of the five cases and the 1/3 base case on the same run.

From the graph of standard deviation of accumulated export error given below it can be seen that the standard deviation of all the probabilistic approaches is lower than the 1/3 base case. It shows that all the approaches perform better than the 1/3 base case in terms of overall fairness among the receivers. The best approach among them being PI=1 where the Instantaneous Error was used as the factor to decide whom to send the DR vector at the trigger.

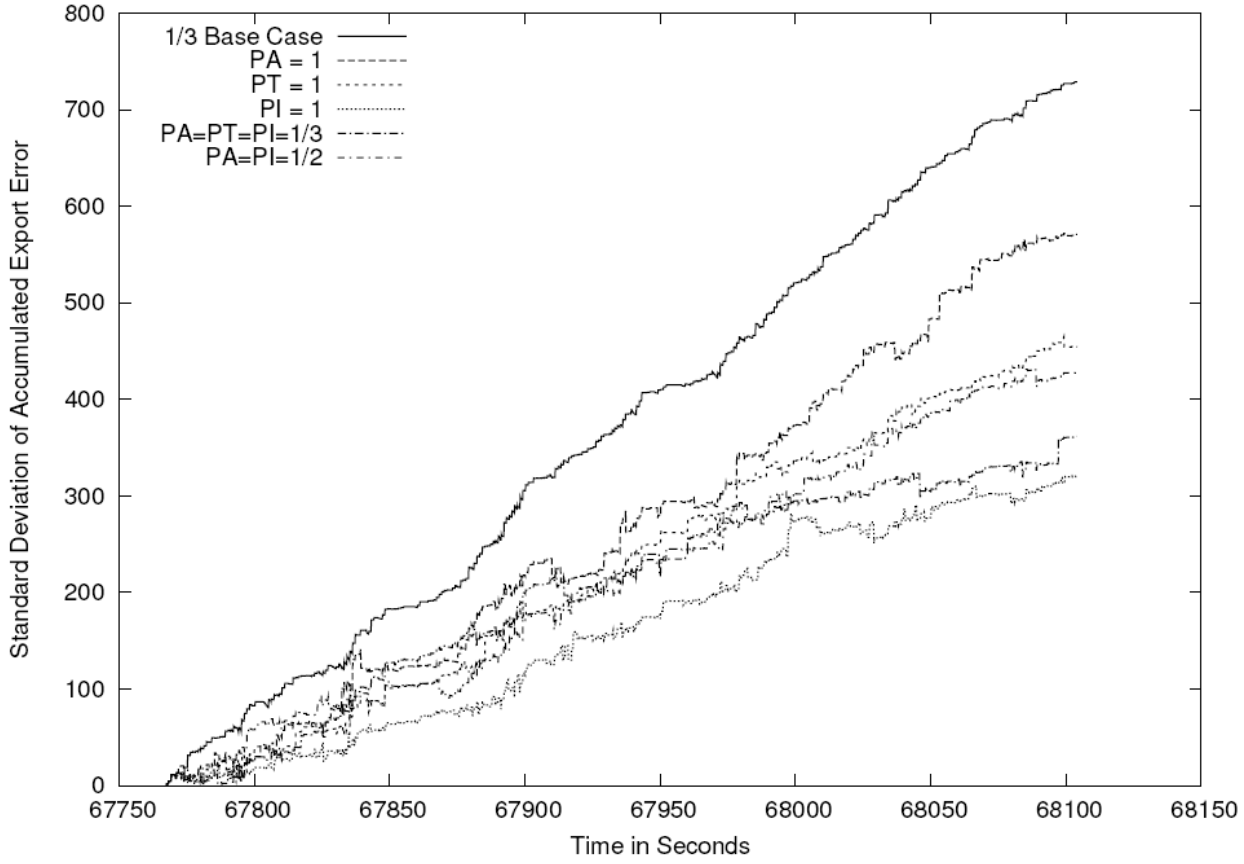


Figure 3.15 Standard Deviation – Probabilistic Algorithm

3.4.1.4.1.2 Mean of the Accumulated Export Error:

The graph is the mean of the accumulated export error in all the approaches. It can be observed that the mean of the accumulated export error in all the probabilistic approaches is higher than the mean in the 1/3 base case. This means that even the probabilistic approaches achieve fairness at the cost of increasing the overall error of the system indicating that the probabilistic approaches also lead to a degradation of the overall qualitative game play.

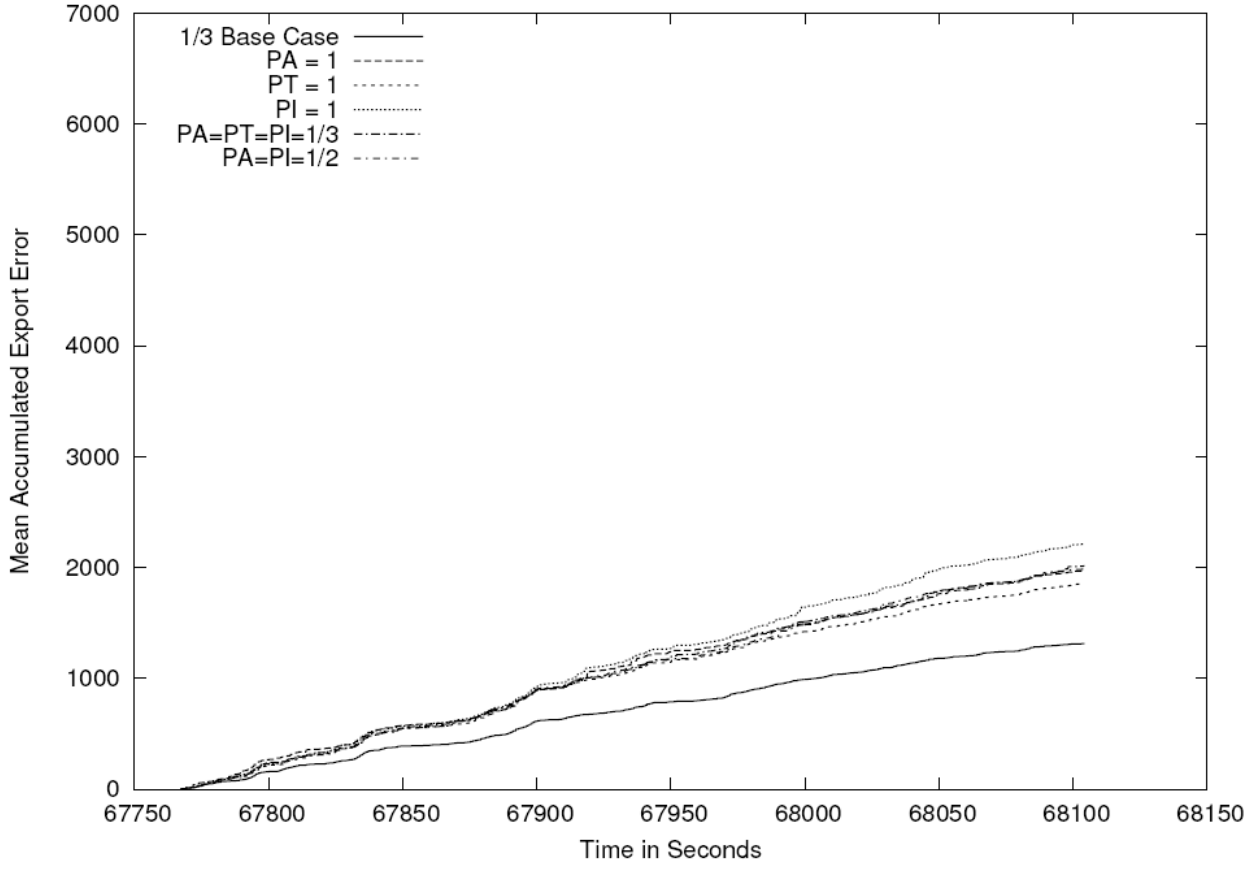
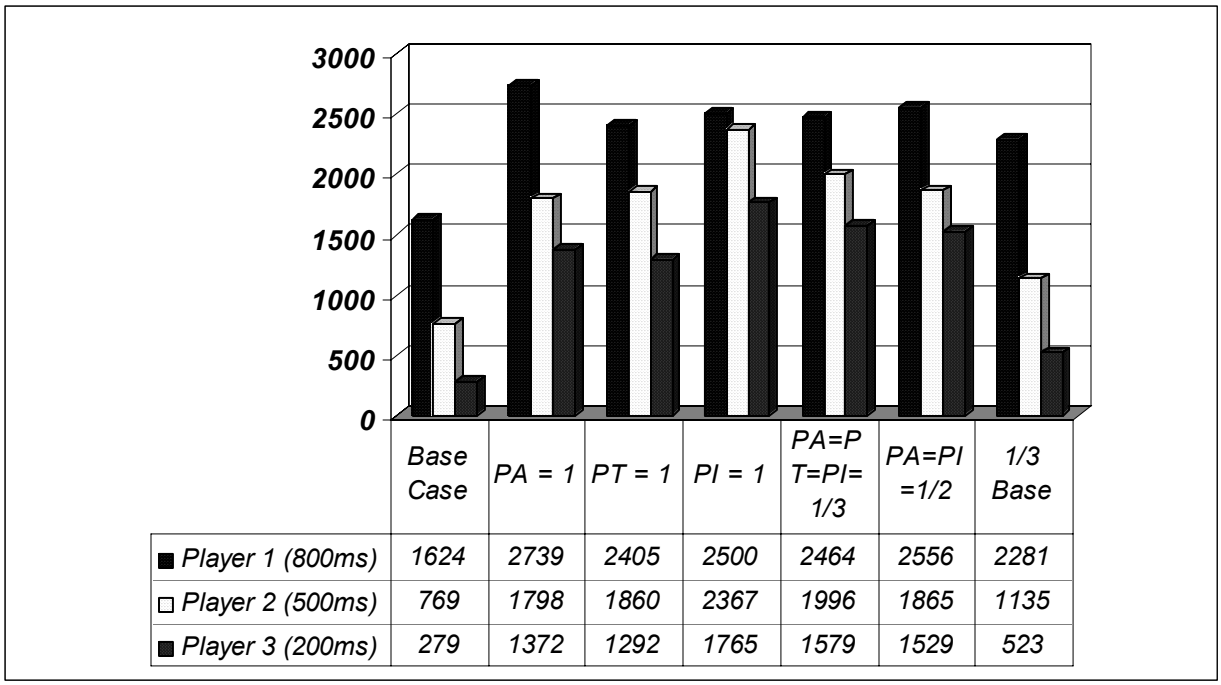


Figure 3.16 Mean – Probabilistic Algorithm

3.4.1.4.1.3 Accumulated Export Error of each player in all the cases:

The intuition behind the probabilistic approaches was that by choosing the appropriate receiver at each trigger, we can send DR vectors at higher frequency to receivers having higher accumulated export error and hence bring the AEE of those receivers below what is observed in the base case. But from figure 3.17 it can be seen that the accumulated export error of all the receivers are higher than the 1/3 base case. We further explored the inconsistency between the intuition and the results to find the reasons. The observations section discusses this issue.



	Base Case	PA = 1	PT = 1	PI = 1	PA=PT=PI=1/3	PA=PI=1/2	1/3 Base
Player 1 (800ms)	1624	2739	2405	2500	2464	2556	2281
Player 2 (500ms)	769	1798	1860	2367	1996	1865	1135
Player 3 (200ms)	279	1372	1292	1765	1579	1529	523

Figure 3.17 Accumulated Error – Probabilistic Algorithm

3.4.1.5 Observations:

We decided to run 1/3 base case against 1/3 probabilistic case and find answers to the questions raised in the previous section. By 1/3 probabilistic case, we mean that, every receiver has a fixed probability of 1/3 at each trigger of receiving a DR vector. At each trigger, for each player a random number between 0 and 1 is chosen, and if the number is less than or equal to 1/3, a DR vector is sent to the receiver. Everything else is the same as the probabilistic approaches discussed earlier.

The figure 3.18 shows the accumulated export error of each receiver over the run in the base case, 1/3 base case and 1/3 probabilistic case.

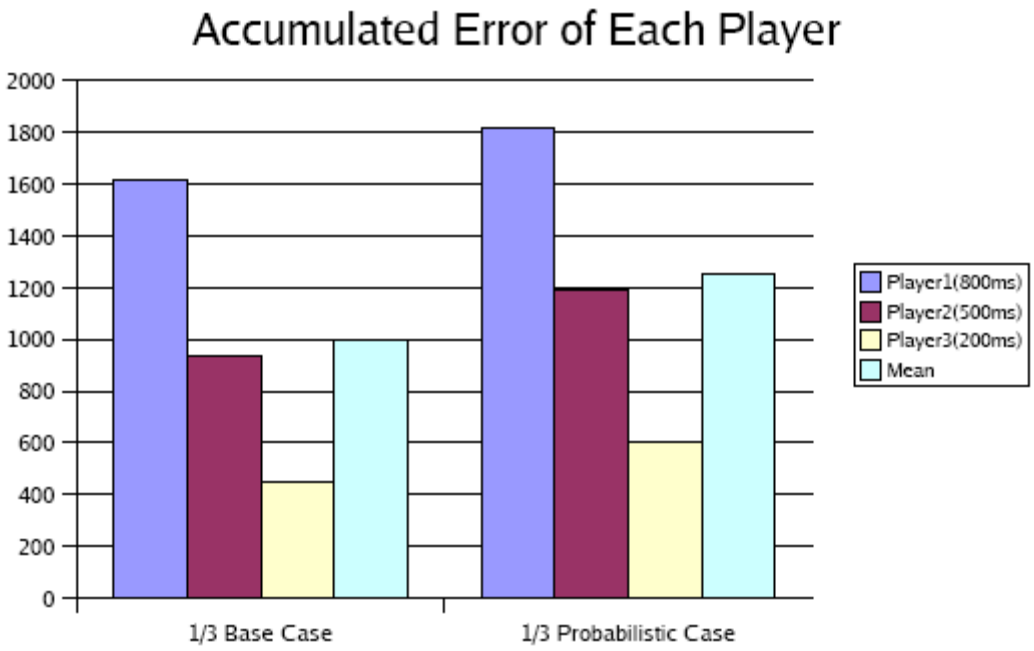


Figure 3.18 Accumulated Error – Probabilistic Algorithm vs Base Case

It is observed that even though 1/3 base case and 1/3 probabilistic case send 1/3 the total number of DR vectors sent in the base case, the AEE of each receiver is higher in the probabilistic case as compared to the 1/3 base case. This is attributed to the fact that the even though the probabilistic approach on average sends the same number of DR vectors as the 1/3 base case, it sometimes sends DR vectors to a receiver less frequently and sometimes more frequently than the 1/3 base case due to its probabilistic nature. When a receiver does not receive a DR vector for a long time, the receiver's trajectory is more and more off of the sender's trajectory and hence the rate of buildup of the error at the receiver is higher. At times when a receiver receives DR vectors more frequently, it builds up error at a lower rate but there is no way of reversing the error that was built up when it did not receive a DR vector for a long time. This leads the receivers to build up more error in 1/3 probabilistic case as compared to the 1/3 base case where the receivers receive a DR vector almost periodically.

3.4.1.6 Conclusion:

We started out with the probabilistic approaches with the intuition that by sending DR vectors at different frequencies to different receivers, we can achieve higher degree of fairness among them and at the same time we can lower the accumulated export error of the receiver having the highest accumulated export error by sending DR vectors more frequently to the receiver than the base case. In doing this, still maintaining the same budget of the DR vectors as in the base case. We explored various probabilistic approaches to achieve fairness. In the end we observed that the probabilistic approaches increased the mean accumulated export error of the system indicating degradation in overall game play.

This is due to the reason that when a receiver does not receive a DR vector for a long time, the receiver's trajectory is more and more off of the sender's trajectory and hence the rate of buildup of the error at the receiver is higher. At times when a receiver receives DR vectors more frequently, it builds up error at a lower rate but there is no way of reversing the error that was built up when it did not receive a DR vector for a long time. Hence, we conclude that no probabilistic algorithm will be able to achieve what we intended to and we need to have a deterministic algorithm, which will be able to achieve the desired results.

3.4.2 Deterministic Budget Based Algorithm

As it was concluded previously that the problem with the probabilistic approaches is the indeterministic nature of the approach in sending the DR vector to the receiver that shoots up the accumulated export error at the receiver, it was concluded that we need a deterministic algorithm rather than a probabilistic one. The following section discusses a deterministic algorithm where we provide a deterministic guarantee that a specific receiver receives DR vectors at a certain frequency.

The intuition behind the deterministic algorithm is the same as in the probabilistic case that by using the same budget of DR vectors as in the base case and varying the

frequency of sending DR vector to respective receivers we can achieve fairness among the receivers and at the same time do better for receivers with higher accumulated export error by sending them DR vectors at a higher frequency as compared to the base case.

The algorithm is divided into three parts:

1. Lowering the dead reckoning threshold
2. Calculating the relative probabilities of each player based on some factors.
3. Calculating deterministic schedules for sending the DR vectors based on the relative probabilities calculated in step 2.

The First two parts of the algorithm are exactly the same as explained in section 3.4.1 under the probabilistic budget based algorithm. The threshold is lowered and the budget per trigger is calculated using equation 3.5. The relative probabilities PA, PT, PI or PC are calculated in exactly the same way explained in section 3.4.1.2. The third part of calculating the deterministic schedules for sending the DR vector is explained below.

3.4.2.1 Calculating the deterministic schedules for the receivers

The following steps are followed at each trigger except the first trigger:

1. If there is any receiver(s) tagged to receive a DR vector at the current trigger, the sender sends out the DR vector to the respective receiver(s). If at least one receiver was sent a DR vector, the sender calculates the relative probabilities of each receiver as explained in 3.4.1.2 and follows steps 2 to 6, else it does not do anything.
2. For each receiver, this value is multiplied with the budget available at each trigger (calculated as mentioned in 3.4.1.1) to give the frequency of sending the DR vector to each receiver.

3. If any of the receiver's frequency after multiplying with the budget goes over 1, the receiver's frequency is set as 1 and the surplus amount is equally distributed to all the receivers by adding the amount to their existing frequencies. This process is repeated until all the receivers have a frequency of less than or equal to 1. This is due to the fact that at a trigger we cannot send more than one DR vector to the respective receiver. That will be wastage of DR vectors by sending redundant information.
4. (1/frequency) gives us the schedule at which the sender should send DR vectors to the respective receiver. Credit obtained previously (explained in step 5) if any is subtracted from the schedule. Observe that the resulting value of the schedule might not be an integer; hence, the value is rounded off by taking the ceiling of the schedule.
5. The difference between the schedule and the ceiling of the schedule is the credit that the receiver has obtained which is remembered for the future and used at the next time as explained in step 4.
6. *For each of those receivers who were sent a DR vector at the current trigger,* the receivers are tagged to receive the next DR vector at the trigger that happens exactly *schedule (the ceiling of the schedule)* number of times away from the current trigger. Observe that no other receiver's schedule is modified at this point as they all are running a schedule calculated at some previous point of time. Those schedules will be automatically modified at the trigger when they are scheduled to receive the next DR vector.

At the first trigger, the sender sends the DR vector to all the receivers and uses a relative probability of 1/n for each receiver and follows the steps 2 to 6 to calculate the next schedule for each receiver in the same way as mentioned for other triggers.

This algorithm ensures that every receiver has a guaranteed schedule of receiving DR vectors and hence there is no irregularity in sending the DR vector to any receiver as was observed in the budget based probabilistic algorithm.

3.4.2.2 Instrumentation with BZFlag

A test bed was setup with four players running modified version of BZFlag. The deterministic algorithm using the budget of 1 DR vector per trigger so as to use the same number of DR vectors as in 1/3 base case according to equation 3.5, the base case and the 1/3 base case were implemented in the same run by tagging the DR vectors according to the type of the algorithm used to send the DR vector. NISTNet was used to introduce delays across the sender and the three receivers. Delays of 800, 500 and 200 milliseconds were introduced between the sender and first, second and the third receiver respectively. All the factors mentioned in the probabilistic budget based algorithm were used to compare against the 1/3 base case. It was observed that the deterministic algorithm using accumulated export error alone as the factor performed the best in comparison to all the other cases. Hence, the following discussion only compares the deterministic algorithm, which uses accumulated export error as the factor, with the 1/3 base case.

3.4.2.2.1 Observations:

3.4.2.2.1.1 Standard Deviation:

The standard deviation of the accumulated export error on all the receivers was plotted to compare the degree of fairness in all the approaches. It can be observed from the figure 3.19 that the standard deviation in the deterministic algorithm is lower than the base and the 1/3 base case. This indicates that the deterministic approach achieves a higher degree of fairness as compared to the base case and the 1/3 base case.

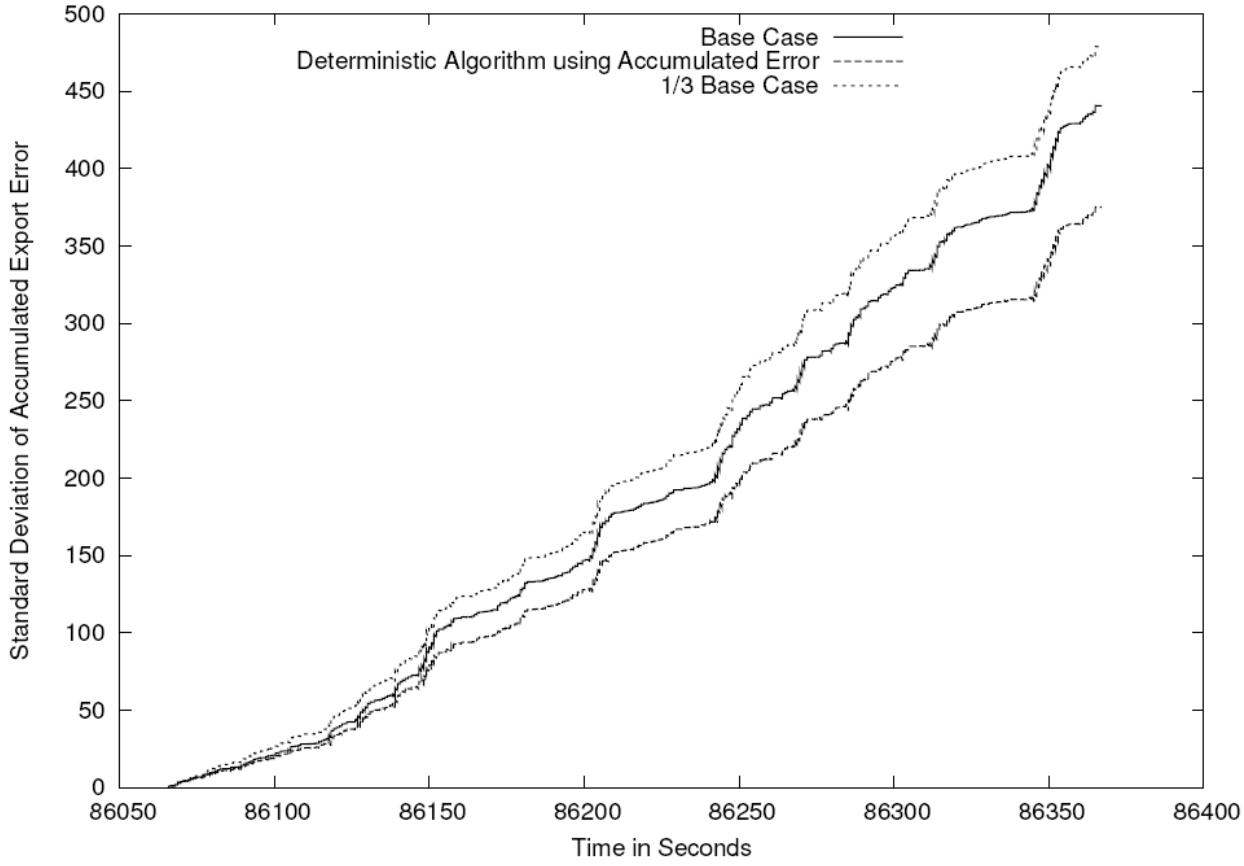


Figure 3.19 Standard Deviation – Deterministic Algorithm

3.4.2.2.1.2 Mean of the Accumulated Export Error:

The graph of the mean in figure 3.20 shows that the mean in the deterministic case is almost the same as in the 1/3 base case which allows us to conclude that the deterministic approach achieves fairness and at the same time does not let the accumulated export error of the system go up. This means that there have to be some receivers whose error has fallen down in comparison to the 1/3 base case and there have to be some receivers whose error must have increased so as to explain the observation of decrease in the standard deviation and almost no increase in the mean. This also indicates that the overall error of the system is the same and hence there is no overall degradation in the quality of the game play.

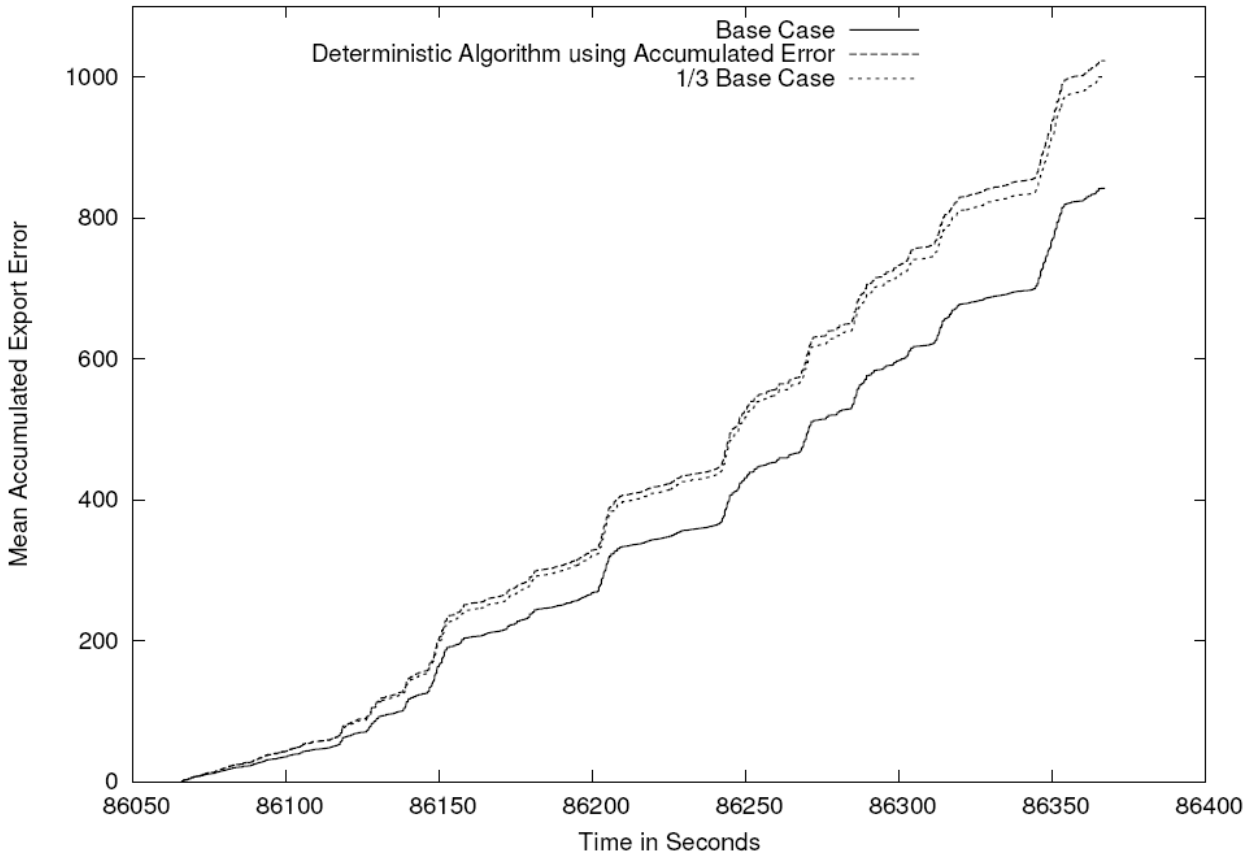


Figure 3.20 Mean – Deterministic Algorithm

3.4.2.2.1.3 Accumulated Export error of each player:

Figure 3.21 supports our supposition that there is a receiver whose error has fallen down as compared to the 1/3 base case and there is a receiver whose error has gone up. The figure clearly justifies our intuition that by changing the frequencies of sending DR vector to each player, we can manage to bring down the highest error and at the same time maintain the mean export error at the same point as in the 1/3 base case.

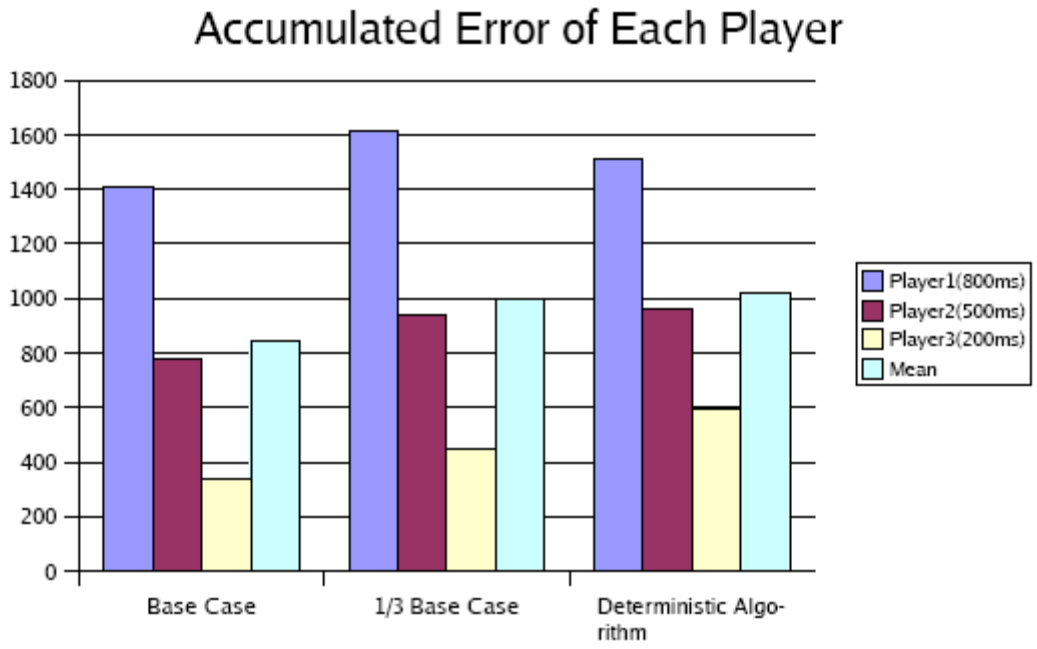


Figure 3.21 Accumulated Error – Deterministic Algorithm

3.4.3 Results: Budget Based Algorithms with Variable Delays

All the results mentioned above in the budget based probabilistic and budget based deterministic algorithms are in a static delay environment. We conducted more runs with both the probabilistic and the deterministic algorithms in a variable delay environment.

A test bed was setup with four players running modified version of BZFlag. The probabilistic and deterministic algorithm using the budget of 1 DR vector per trigger so as to use the same number of DR vectors as in 1/3 base case according to equation 3.5, the base case and the 1/3 base case were implemented in the same run by tagging the DR vectors according to the type of the algorithm used to send the DR vector. NISTNet was used to introduce variable delays across the sender and the three receivers. Mean delays of 800, 500 and 200 milliseconds were introduced between the sender and first, second and the third receiver respectively. Two different runs with

moderate delay variance (100ms +/- mean) and high delay variance (180ms +/- mean) were done. The following are the observations.

3.4.3.1 Standard Deviation of Accumulated Export Error

The figure 3.22 and 3.23 show that the standard deviation in the budget based deterministic algorithm and the budget based probabilistic algorithm is lower as compared to the 1/3 base case in a variable delay environment. This supports our observation in the static delay environment and hence we can conclude that the budget based algorithms achieves a higher degree of fairness in static and variable delay environments.

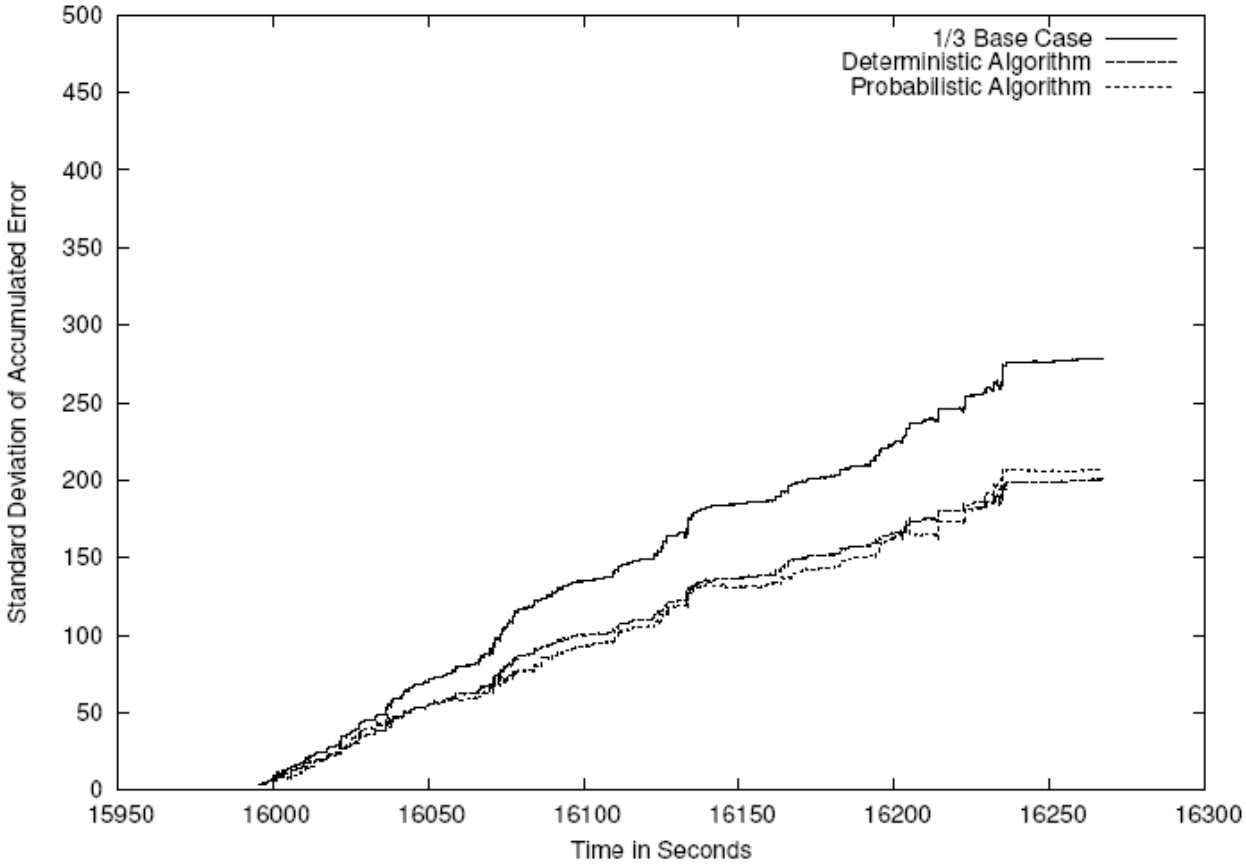


Figure 3.22 Standard Deviation – Moderate Delay Variance (Budget based Algorithm)

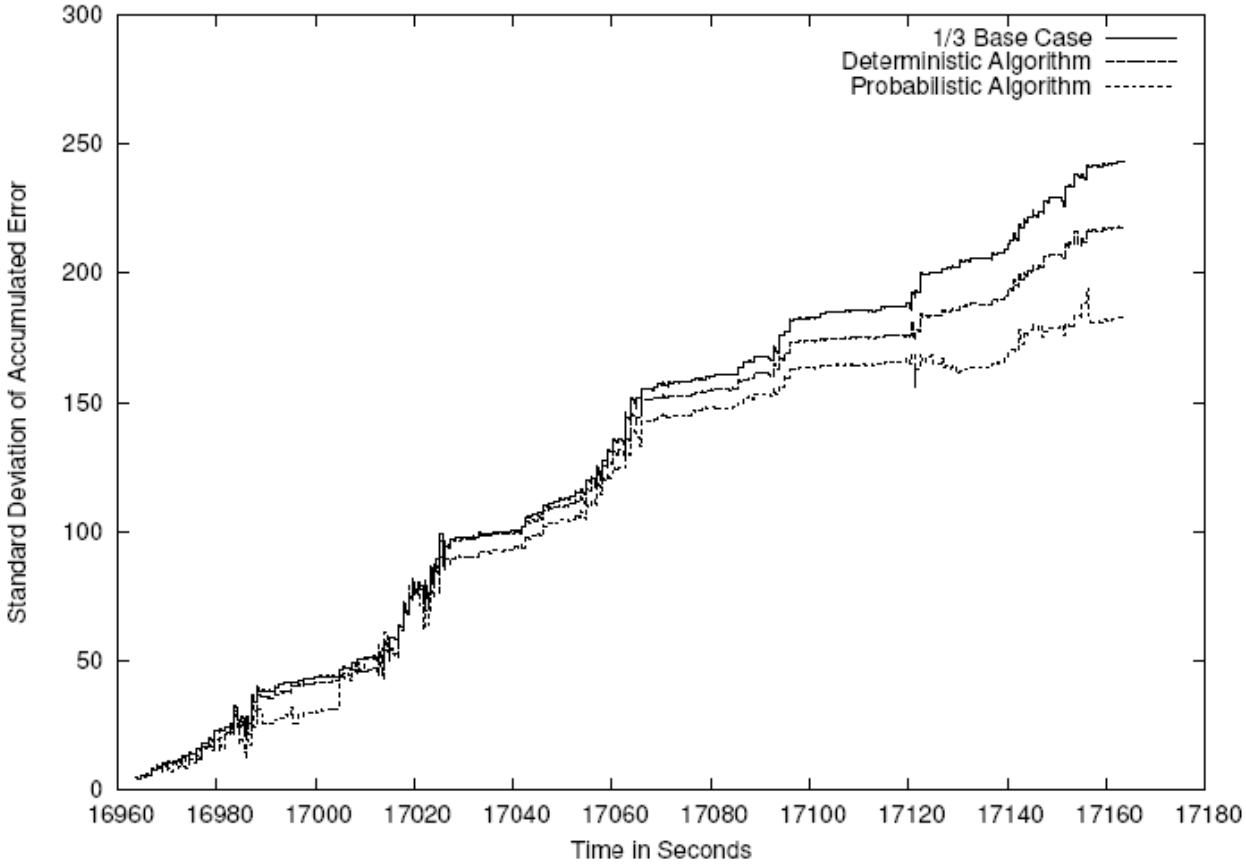


Figure 3.23 Standard Deviation– High Delay Variance (Budget based Algorithm)

3.4.3.2 Mean of the Accumulated Export Error

The figures 3.24 and 3.25 show the mean accumulated export error in the budget based probabilistic and deterministic algorithms as compared with the 1/3 base case in a variable delay environment. It can be seen that the mean accumulated export error in the budget based probabilistic algorithm is higher than the mean accumulated export error in the 1/3 base case. This conforms to the observation in the static delay case for the budget based probabilistic algorithm. It is observed that the mean in the budget based deterministic algorithm is almost the same as the mean accumulated export error in the 1/3 base case. This also conforms to the observation in the static delay case.

Hence, we can observe that the budget based deterministic algorithm achieves higher degree of fairness, without degrading the overall performance of the game, as compared to the 1/3 base case.

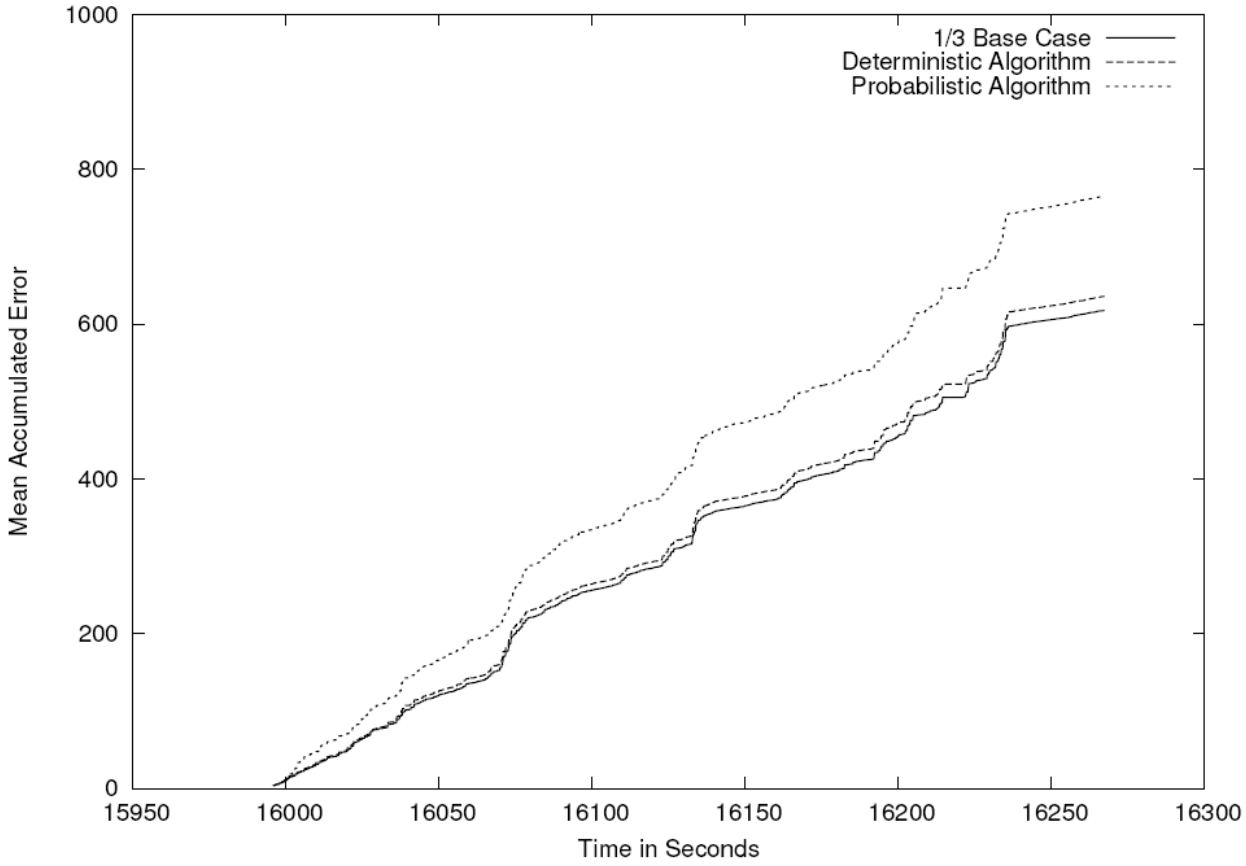


Figure 3.24 Mean Accumulated Export Error – Moderate Variance (Budget based Algorithm)

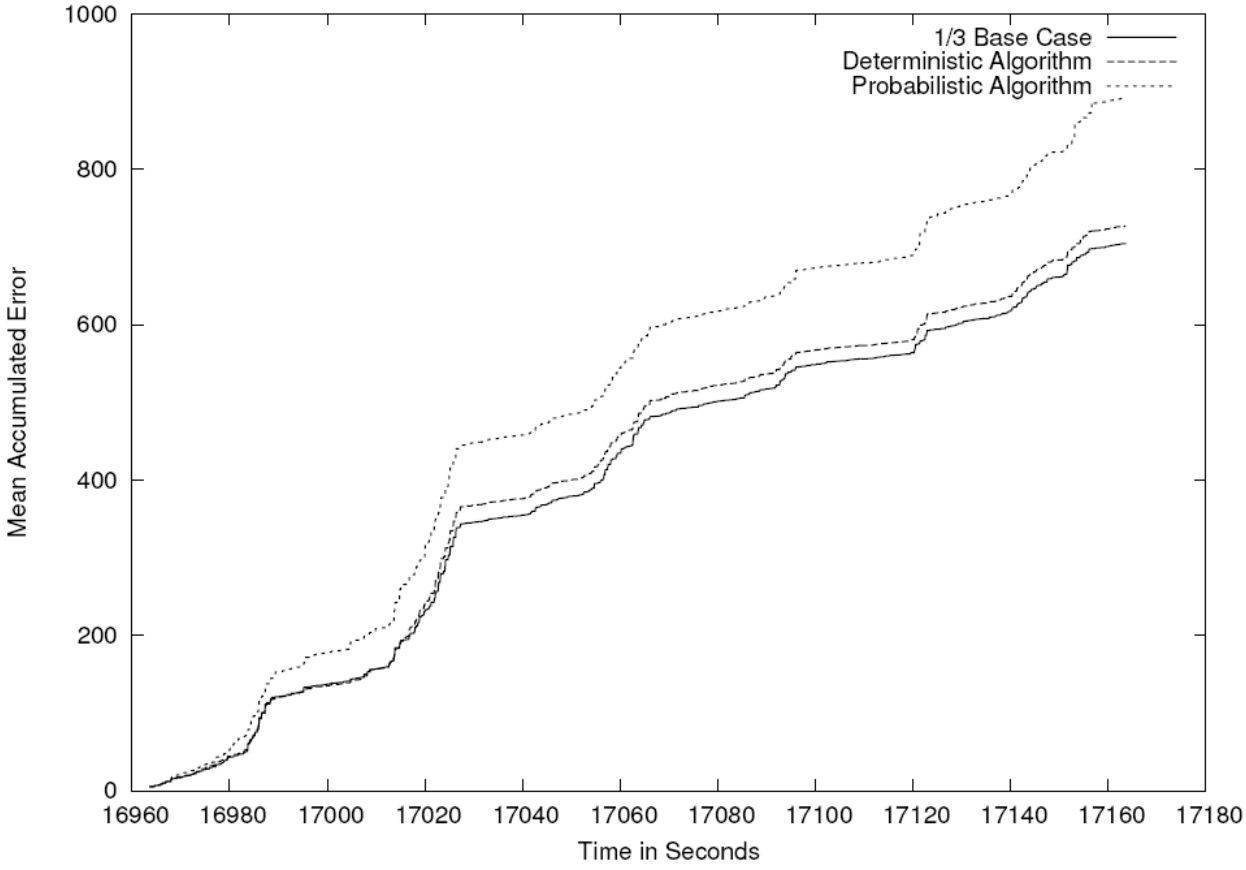


Figure 3.25 Mean Accumulated Error – High Delay Variance (Budget based Algorithm)

3.4.3.3 Accumulated Export Error at each player

The figures 3.26 and 3.27 show the accumulated export error at each player in the budget based algorithms as compared to the 1/3 base case in a variable delay environment. From the figures, it can be seen that the budget based deterministic algorithm results in a higher degree of fairness and at the same time maintain the mean export error of the system as the same as in the 1/3 base case.

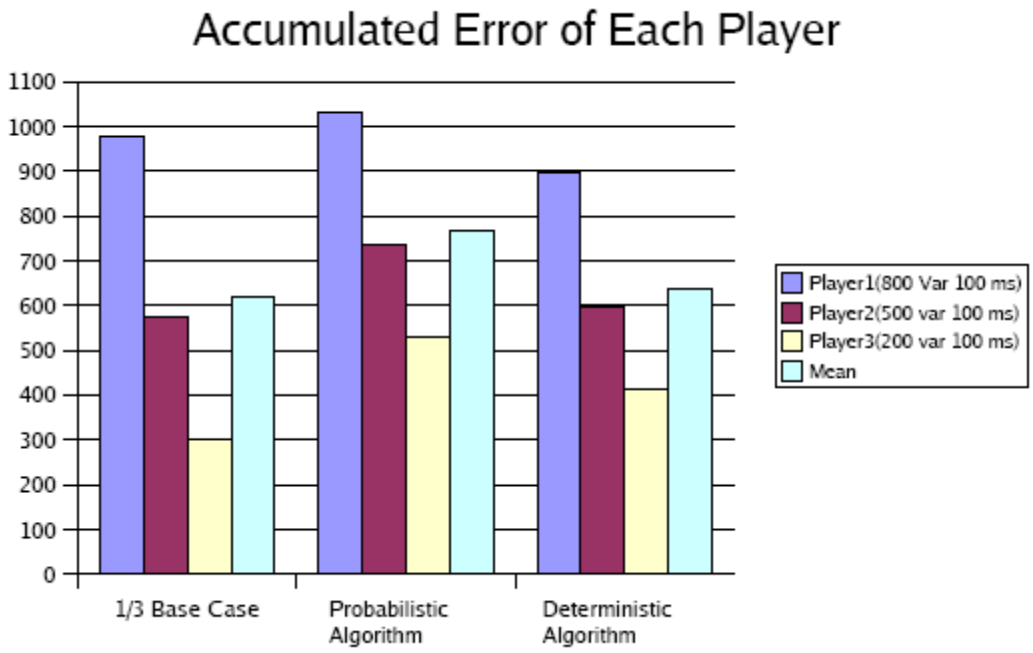


Figure 3.26 Accumulated Export Error – Moderate Delay Variance (Budget based Algorithm)

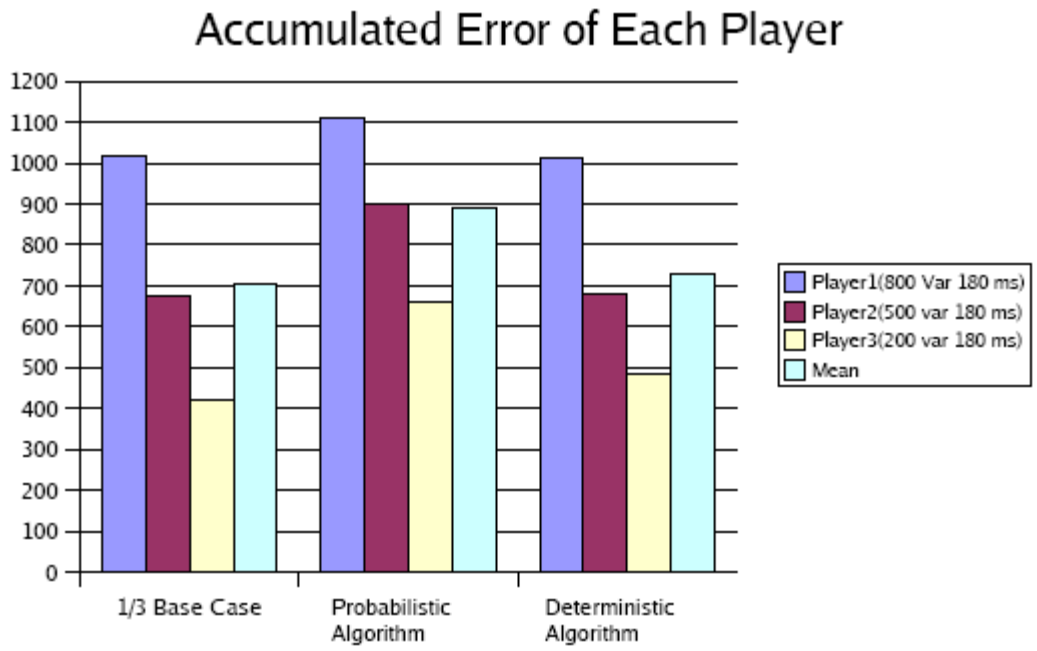


Figure 3.27 Accumulated Export Error – High Delay Variance (Budget based Algorithm)

3.5 Conclusion:

From the observations made in both the static and the variable delay environments, we can conclude that by using accumulated export error as a factor to change the frequency of sending DR vectors to receivers, we can achieve higher degree of fairness among the receivers without increasing the mean error of the system using the budget based deterministic algorithm. The scheduling algorithms and the budget based probabilistic algorithm suffer the drawback of increase in the mean accumulated export error of the system. The budget based deterministic algorithm achieves a higher degree of fairness among the receivers in the best possible way by improving the performance of the game for receivers with higher accumulated export errors and pushing the accumulated export error of all the receivers towards the mean export error of the system. It is worth noting that the lower the threshold, the better, as that will allow the

sender to increase the frequency of sending DR vectors to the receiver with the highest accumulated error.

Hence, we can conclude that with the budget based deterministic algorithm using accumulated export error as a factor and a very low threshold it is possible to achieve a higher degree of fairness and push the accumulated export error of all the receivers towards the mean export error of the system.

CHAPTER 4
EXTENSIONS TO OPEN TNL

4.1 Motivation:

The accuracy and the fairness models implemented in BZFlag resulted in quantitative and qualitative improvement in the game play and motivated us to generalize the work to all the dead reckoning based games. The goal here is to extend or create a game library to support dead reckonable objects and implement the accuracy model for the dead reckonable objects thus allowing game developers to easily build dead reckoning based games without worrying about the dead reckoning issues. Implementation of the fairness model will be done is a part of the future work. Existing game networking libraries were reviewed for the scope of extending them for dead reckonable objects with the accuracy model. A short description of the libraries and the factors considered in evaluating them follows:

Three game network libraries (GNLs) were reviewed:

1. Raknet (http://www.rakkarsorft.com) [26]
2. Zoidcom (http://www.zoidcom.com) [27]
3. Open TNL (http://opentnl.sourceforge.net) [25]

4.1.1 Raknet

Raknet is a free networking API that provides reliable UDP and high level networking constructs on Windows, Linux, and UNIX. It is an advanced networking API that provides services for and over Berkeley Sockets, or on Windows systems "Winsock." It allows any application to communicate with other applications that also use Raknet, whether that be on the same computer, over a LAN, or over the internet.

4.1.1.1 Features:

1. Synchronize class instances allowing you to network game objects with a fair degree of control without writing a single packet.
2. Utilize remote procedure calls, allowing you to call functions on other computers with variable parameters.
3. Get statistics such as ping, packet loss, bytes sent, bytes received, packets sent, packets received, and more.
4. Optional per-packet time stamping so you know with a fair degree of accuracy how long ago an action was performed

4.1.1.2 Our interest in Raknet: DistributedNetworkObject Class

Objects that are instantiated on one system are instantiated on all systems. Objects that are destroyed on one system are destroyed on all systems. Tagged memory is matched between systems. When a new player connects, these objects are also created on his system. This is very useful conceptually because it has a direct analogy to game objects. For example, a one tank in a multiplayer game with twenty players actually has been instantiated twenty times. However, as far as the player is concerned there is only one tank. The player expects that the position, orientation, number of shells left, and amount of armor is the same on all systems. Traditionally, to maintain a tank you would have to craft a series of custom packets for the tank. One packet to describe position, another packet for the tank to shoot, and another to have the tank take damage. Using the distributed network object system, you can synchronize the tank, those 3 member variables, and everything matches automatically.

The DistributedNetworkObject can be adapted for dead reckonable objects and made to synchronize clients by sending selective updates (or different update frequency for each client) based on error and hence achieve *fairness*. The *accuracy* model can be implemented to make the object synchronization accurate even with different and varying network delays.

4.1.2 Zoidcom:

Zoidcom is easy, flexible and simple C++ API. Zoidcom is a high-level, UDP based networking library providing features for automatic replication of game objects and synchronization of their states over a network connection in a highly bandwidth efficient manner. This is achieved by multiplexing and de-multiplexing object information from and into bit streams, which make it easily possible to avoid sending redundant data.

4.1.2.1 Features:

Automatic Object State Synchronization

1. synchronize bools, ints, floats and strings
2. interpolate ints and floats
3. only send data that really changed

Full control

4. Minimum and maximum update frequency for each single data item
5. Intercept, manipulate and prevent data updates for debugging, monitoring and cheat protection

Efficient

6. Adjust the amount of relevant bits per replicated item
7. Default values to avoid transmission of redundant data
8. Interpolation with adjustable interpolation strength or totally custom interpolation

4.1.2.2 Our Interest:

Currently, Zoidcom does not handle dead reckoning directly, but can be added with an interceptor system. A call back is received when Zoidcom wants to update data to a certain client. Within the call back, the relevance of the node in question to the client (a float) can be changed which directly affects the update frequency of the node to this client. One can keep track of the client's prediction there and hold back the update when the error margin is not exceeded. Dead reckoning and fairness can be implemented here. On the client side, an interceptor call back gets called when an update arrives,

too. The accuracy model can be implemented here. The infrastructure is there but the implementation has to be done.

4.1.3 Open TNL:

The Torque Network Library is a cross-platform C++ networking API designed for high performance simulations and games. TNL is designed to overcome, as much as possible, the three fundamental limitations of network programming

1. High packet latency
2. Limited bandwidth and
3. Packet loss.

TNL will be discussed in detail in the next section.

4.1.3.1 Our Interest: Ghosting

Ghosting is the server object replication mechanism available in Open TNL. The GhostConnection and the NetObject classes are the two important classes that handle the *ghosting* in open TNL. The NetObjects can be replicated over the GhostConnections.

The NetObject class can be extended to support dead reckoning by triggering the updates when a threshold is crossed. The updates received at the server can then be replicated using the server object replication mechanism – ghosting. The Accuracy model can be implemented on the server and the 'ghosts' so as to render the object accurately once they receive the update. The fairness model can be implemented on the server and let the server decide which ghost to send the current update.

After reviewing the above three libraries, it was concluded that Open TNL was the most suitable platform to work on. Raknet is a fairly simple library, which is being used in a few games. Zoidcom is not open source. Open TNL is available under GPL and provides a lot of functionality to the user. At the same time it is being used in many multiplayer game projects. The user base of Open TNL is larger than the other two

libraries and hence extending Open TNL would lead to a greater impact than extending any of the other two libraries.

The next section gives a brief overview of open TNL's functionality. The third section discusses the network architecture in Open TNL. The fourth section details the object replication mechanism in open TNL. The fifth section discusses the work done in extending the object replication mechanism for dead reckonable objects.

4.2 Introduction to Open TNL

The Torque Network Library is a robust, secure, easy to use, cross-platform C++ networking API designed for high performance simulations and games. The network architecture in TNL has powered some of the best Internet multiplayer action games to date. TNL was released under GNU General Public License (GPL) in 2004 and is known as Open TNL.

4.2.1 Features:

The features of open TNL are as follows:

4.2.1.1 Multiple platform support

- Windows 98, ME, NT, XP
- Linux on x86
- Mac OS X

4.2.1.2 Robust connection architecture

- UDP based delivery notification connection protocol
- Two-phase connect for prevention of IP spoofing attacks
- Client-puzzle system for server CPU depletion Denial of Service protection
- AES symmetric encryption with SHA-256 message authentication
- Efficient packet streaming architecture for consistent bandwidth utilization

4.2.1.3 Multiple levels of data guarantee

- Reliable, ordered - data will arrive, and will be delivered in the order it was sent
- Reliable, unordered - data will arrive, and will be delivered as soon as it is received
- Unreliable - data is not guaranteed to arrive
- Most Recent State with prioritization - most recent state of an object will be reflected on a client, updated according to priority
- Guaranteed Quickest - data is sent with every packet until its delivery is confirmed

4.2.1.4 Bit-level compression using the BitStream class for optimal bandwidth utilization

- Specific bit length encoding for simple data types (floats, integers)
- 3D point compression based on maximum error tolerance from reference point
- Huffman encoded strings and common substring skipping

4.2.1.5 Server Object replication (ghosting) and management

- Objects can be cloned from server to client
- User code has complete control over scoping policy (which objects get replicated)
- Prioritization of updates can be specified on a per-object basis
- Support for up to 32 orthogonal state categories per object
- Automatic class registration and remote instantiation

4.2.1.6 Simple and efficient event and RPC (remote procedure call) framework

- RPC methods declared using simple macros
- RPC parameter lists can contain simple data types, as well as vectors

- RPC´s can be invoked on connection instances as well as ghosted objects
- Generic network event class for data not suited to RPCs

4.2.1.7 Extensible master server framework

- Supports filtering of game servers on several criteria
- Acts as a matchmaker, and can facilitate a direct connection between hosts behind firewalls and NATs
- Easily extended to particular game needs
- RPC versioning feature makes exposing new interfaces easy while maintaining backward compatibility

The network architecture in TNL has powered some of the best internet multiplayer action games, including Vivendis Award-Winning Starsiege: TRIBES and Tribes 2 products, as well as independently produced titles including Bravetrees ThinkTanks, 21-6 Productions Orbz, Mecha title Dark Horizons: Lore. The next section discusses the network architecture of Open TNL in detail.

4.3 Open TNL network architecture

TNL is built in layers, each adding more functionality to the layer below it.

4.3.1 The Platform Layer

At the lowest level, TNL provides a platform independent interface to operating system functions. The platform layer includes functions for querying system time, sleeping the current process and displaying alerts. The platform layer includes wrappers for the entire set of C standard library functions used in the TNL.

The platform layer also contains the Socket and Address classes, which provide a cross-platform, simplified interface to datagram sockets and network addresses.

4.3.2 The NetBase Layer

The NetBase layer is the foundation upon which most of the classes in TNL are based. At the root of the class hierarchy is the Object base class. Every subclass of Object is associated with an instance of NetClassRep through a set of macros, allowing for instances of that class to be constructed by a class name string or by an automatically assigned class id.

This id-based instantiation allows objects subclassed from NetEvent and NetObject to be serialized into data packets, transmitted, constructed on a remote host and deserialized.

Object also has two helper template classes, SafePtr, which provides safe object pointers that become NULL when the referenced object is deleted, and a reference pointer class, RefPtr, that automatically deletes a referenced object when all the references to it are destructed.

4.3.3 The BitStream and PacketStream classes

The BitStream class presents a stream interface on top of a buffer of bytes. BitStream provides standard read and write functions for the various TNL basic data types, including integers, characters and floating-point values. BitStream also allows fields to be written as bit-fields of specific size.

BitStream huffman encodes string data for additional space savings, and provides methods for compressing 3D points and normals, as routines for writing arbitrary buffers of bits.

The PacketStream subclass of BitStream is simply a thin interface that statically allocates space up to the maximum size of a UDP packet. A routine can declare a stack allocated PacketStream instance, write data into it and send it to a remote address with just a few lines of code.

4.3.4 The NetInterface and NetConnection Layer

The NetInterface class wraps a platform Socket instance and manages the set of NetConnection instances that are communicating through that socket. NetInterface is manages the two-phase connection initiation process, dispatch of protocol packets to NetConnection objects, and provides a generic interface for subclasses to define and process their own unconnected datagram packets.

The NetConnection class implements the connected Notify Protocol layered on UDP. NetConnection manages packet send rates, writes and decodes packet headers, and notifies subclasses when previously sent packets are known to be either received or dropped.

NetInterface instances can be set to use a public/private key pair for creating secure connections. In order to prevent attackers from depleting server CPU resources, the NetItnerface issues a cryptographically difficult *client puzzle* to each host attempting to connect to the server. Client puzzles have the property that they can be made arbitrarily difficult for the client to solve, but whose solutions the server can check in a trivial amount of time. This way, when a server is under attack from many connection attempts, it can increase the difficulty of the puzzle it issues to connecting clients.

4.3.5 The Event Layer - EventConnection, NetEvent and Remote Procedure Calls

The EventConnection class subclasses NetConnection to provide several different types of data transmission policies. EventConnection uses the NetEvent class to encapsulate event data to be sent to remote hosts. Subclasses of NetEvent are responsible for serializing and deserializing themselves into BitStreams, as well as processing event data in the proper sequence.

NetEvent subclasses can use one of three data guarantee policies. They can be declared as GuaranteedOrdered, for ordered, reliable message delivery; Guaranteed, for reliable but possibly out of order delivery, or Unguaranteed, for ordered but not guaranteed messaging. The EventConnection class uses the notify protocol to requeue

dropped events for retransmission, and orders the invocations of the events' process methods if ordered processing is requested.
Because declaring an individual NetEvent subclass for each type of message and payload to be sent over the network, with its corresponding pack, unpack and process routines, can be somewhat tedious, TNL provides a Remote Procedure Call (RPC) framework. Using some macro magic, argument list parsing and a little assembly language, methods in EventConnection subclasses can be declared as RPC methods. When called from C++, the methods construct an event containing the argument data passed to the function and send it to the remote host. When the event is unpacked and processed, the body of the RPC method implementation is executed.

Now that we have a brief idea of the network architecture of Open TNL let us go into the details of the object replication mechanism in Open TNL. The existing mechanism will act as a base for the extension for accuracy and fairness for dead reckonable objects.

4.4 Object replication in Open TNL

Open TNL provides a replication mechanism to replicate server objects over the network know as *Ghosting*. The two classes that are the crux of object replication mechanism are the NetObject and the GhostConnection classes.

The GhostConnection class subclasses EventConnection in order to provide the most-recent and partial object state data update policies. Instances of the NetObject class and its subclasses can be replicated over a connection from one host to another. The GhostConnection class manages the relationship between the original object, and its *ghost* on the client side of the connection.

In order to best utilize the available bandwidth, the GhostConnection attempts to determine which objects are *interesting* to each client - and among those objects, which

ones are most important. If an object is interesting to a client it is said to be *in scope* - for example, a visible enemy to a player in a first person shooter would be in scope.

Each GhostConnection object maintains a NetObject instance called the scope object - responsible for determining which objects are in scope for that client. Before the GhostConnection writes ghost update information into each packet, it calls the scope object's performScopeQuery method which must determine which objects are *in scope* for that client.

Each scoped object that needs to be updated is then prioritized based on the return value from the NetObject::getUpdatePriority() function, which by default returns a constant value. This function can be overridden by NetObject subclasses to take into account such factors as the object's distance from the camera, its velocity perpendicular to the view vector, its orientation relative to the view direction and more.

Rather than always sending the full state of the object each time it is updated across the network, the TNL supports only sending portions of the object's state that have changed. To facilitate this, each NetObject can specify up to 32 independent states that can be modified individually. For example, a player object might have a movement state, detailing its position and velocity, a damage state, detailing its damage level and hit locations, and an animation state, signifying what animation, if any, the player is performing.

A NetObject can notify the network system when one of its states has changed; allowing the GhostConnections that are ghosting that object to replicate the changed state to the clients for which that object is in scope.

In this way a NetObject can be replicated over the GhostConnections in an efficient manner. Open TNL provides a general mechanism for replication of objects over the network but does provide any mechanism for dead reckoning and replicating dead reckonable objects. The idea is to extend the replication mechanism of open TNL to provide a mechanism for replication of dead reckonable objects. Once this is in place, the accuracy model will also be implemented for the dead reckonable objects. This

extension can be then used by developers using open TNL to create dead reckonable game objects which will automatically be accurate.

4.5 Dead Reckoning Extenstions to open TNL

Open TNL supports replication of NetObjects over the GhostConnections where GhostConnections are responsible for replicating the NetObject to the *ghost*. This framework can be further extended to dead reckonable objects. This section discusses the work that was carried out in extending open TNL's framework to dead reckonable objects.

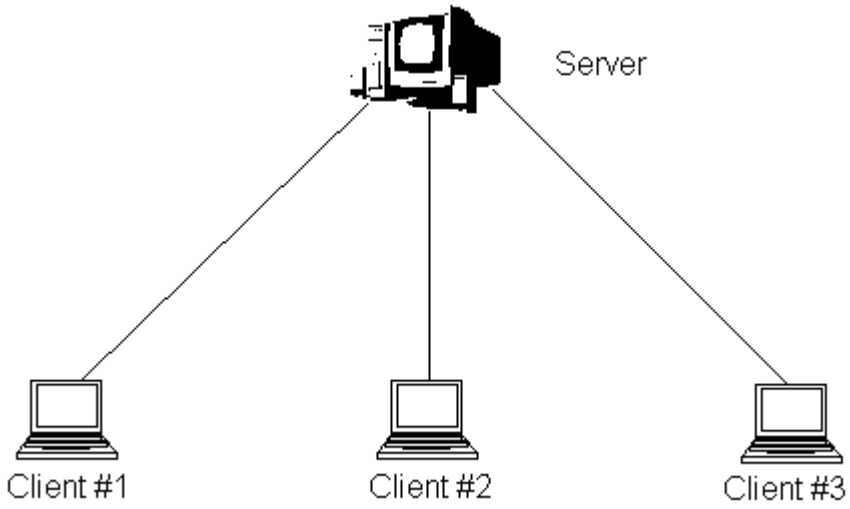


Figure 4.1 Dead reckoning extensions to Open TNL

A class called 'DRObject' was subclassed from NetObject and hence inheriting all the replication mechanism of NetObject.

Every dead reckonable object has a position attribute in the virtual world. We assume the virtual world to be 2 dimensional and hence only deal with dead reckoning objects in 2-D. 3-D dead reckoning can be done on the similar lines with a little more

work. In 2-D there can be many ways of dead reckoning – Linear, Circular and other curve fitting techniques. We implement the linear dead reckoning. Other dead reckoning ways can be added to the existing architecture as per requirements later.

The position attribute contains the information about the current state of the DRObject. The state consists of current location, velocity, acceleration, time and other kinematics associated with the DRObject.

Consider the above setup where three game clients are connected to a server. A DRObject is created such that it is owned by client 1 and the clients 2 and 3 want to dead reckon the object. The following description will explain the details of the dead reckoning extension and how it works.

The setThreshold function sets the distance and time threshold for the DRObject. The TestThresholdAndUpdate function which is called every time in the main loop of the game at the client, checks if any threshold has crossed by calling the isThresholdCrossed method and if it returns true, the client performs an RPC on the server for updating its own position information.

Once the server receives the update from client 1, it updates the position information about the DRObject owned by client 1 in the game and then sets an update flag indicating that the position information needs to be updated for the DRObject owned by client 1.

On the setting the update flag, the GhostConnection replicates the information from the server to all the clients' dead reckoning the object. In this way, the new object information is received at client 2 and client 3.

Now, these clients call the DRObject::DeadReckon method on the DRobject from within the main loop so as to update its position by dead reckoning the entity based on the last DR vector.

We implement linear dead reckoning and hence, every time there is a deviation from the linear path beyond the threshold at the owners screen a DR vector is generated and the information is updated at the server and in turn on to the clients. The

clients dead reckon the object on a linear path as indicated in the DR vector by calling DRObject::DeadReckon method every game loop and updating the clients position on the screen based on it. In this way, the DRObject provides a simple mechanism to dead reckon objects by setting the distance and the time thresholds. The current implementation only handles linear dead reckoning but other complex forms of dead reckoning can be added in a similar fashion. Refer appendix C for implementation details.

4.5.1 Accuracy

The aim here is the same as in chapter two, that is, once the receiver gets the DR vector, the trajectory of the object at the sender and the receiver should be exactly the same, which is, after export error should be zero. With the DRObject extension to Open TNL, implementing accuracy model for the dead reckonable objects is as follows:

Consider the figure 4.1 and assume that client 1 controls an entity, which is dead reckoned by client 2 and client 3. Every time any of the thresholds is crossed, it results in an RPC being executed on the server and the server updates the position information of the clients object. Just before the server updates the position, the server calculates the one-way delay between the client 1 and the server and based on the delay projects the current position of the object. TNL provides functionality for the calculation of the one-way delay of a connection via the function call getOneWayTime for the respective connection.

The server calculates the delay and does the projection is done as follows:

Current X = Received X + Received Vx * (one way delay between the server and client1)

Current Y = Received Y + Received Vy * (one way delay between the server and client1)

After this projection, the server knows the exact position of the object at the current time. Now, whenever the server sends the update to the other clients – client 2 and client 3, they estimate the one-way delay on the connection between themselves and the server and apply the same projection mentioned above. This leads them to know the accurate position of the client 1's object once they receiver the update from the server. Observe that there is no need to synchronize the clocks and send timestamps with the updates in this case as this is substituted by the one-way delay estimation function.

4.5.1.2 Instrumentation with a Sample Test Program

A sample test program was adapted to use the above dead reckoning model with accuracy. The Test application demonstrates some of the other features of the Torque Network Library also. The application presents a single window, representing an abstract simulation area. Red rectangles are used to represent buildings that small squares representing players move over.

Each instance of the Test program can run in one of four modes: as a server, able to host multiple clients; as a client, which searches for and then connects to a server for game data; a client pinging the localhost for connecting to a server on the local machine, and as a combined server and client.

The client owns a DRObject, in this case the player represented by a small square on the screen. The distance and time thresholds are set for the object and hence as the object moves on the owner player's screen and if a threshold is crossed, it leads the client to execute the RPC on the server to update the objects position on the server. The server updates the position accurately by projecting the position based on the one way delay from the client to the server. Then the server sends out the update messages to other clients in the game about this object depending on the relevance of the object to the other clients. Those clients at the reception of the update message again use the accuracy model to accurately position the object on their screens and

then use linear dead reckoning to dead reckon the object until a new update message is received for that object.

The test program was tested qualitatively by running the program with multiple clients over a period of time. No quantitative measurements were done. The test program demonstrated qualitative improvement in the game play.

4.6 Conclusion

We started off with the aim of generalizing the accuracy and fairness models to all the dead reckoning based games. Three existing game networking libraries were explored and after a detailed review of each one of them, Open TNL was chosen as the suitable platform for extending the accuracy and fairness work.

Open TNL was first extended to support dead reckonable objects. Once this was in place, the accuracy model was integrated into the DRObject framework. A test program using Open TNL was adapted to use the DRObject framework. The test program demonstrated qualitative improvement in the game play.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

We started out with the following goals:

1. Explore latency issues in dead reckoning based games.
2. Increase the accuracy in dead reckoning based games
3. Achieve higher degree of fairness among players in dead reckoning based games.
4. Generalize the work to all dead reckoning based games.

5.1 Accuracy in dead reckoning based games:

We started out with the observation that the network latency causes two types of errors in the dead reckoning of the entities in dead reckoning based games. The errors are the *before export error* and the *after export error* for a DR vector. The after export error is the difference in the trajectories after the receiver gets a DR vector because of the fact that the receiver places the senders object always in the past. The receiver does not take into account the fact that it took some amount of time for the DR vector to reach the receiver from the sender.

We proposed a mechanism that allows the receiver to take into account the delay between the sender and the receiver and hence project the correct position of the object based on the delay estimate and as a result render the object correctly. It was observed that using our proposed modification, *the after export error reduced to zero*. The results indicated that there was a quantitative improvement in the accuracy of rendering the object on the receiver's screen. At the same time, a qualitative improvement in the game play was observed.

5.2 Fairness in dead reckoning based games:

The other part of the error, which is the before export error, is the difference in the sender's and the receiver's trajectory in the time interval from the generation time of a DR at the sender till the DR vector is received at the receiver. This part of the export error can never be eliminated and is different for different receivers. Due to the difference in the export errors among the receivers, the same entity is rendered at the same position at different physical time at different receivers. This brings in *unfairness* in game playing. It would have been tolerable if all receivers rendered the object erroneously but at the same position at the same physical time. Hence, the goal is to equalize the export error at all the receivers.

We explored two scheduling algorithms which aim at equalizing the accumulated export error at all the receivers by making them equal to the highest accumulated export error in the system. This was achieved by scheduling a DR vector to be sent to a receiver such that, at the reception of the DR vector, the receiver's accumulated export error will be equal to the highest accumulated export error. The first algorithm suffered a serious drawback of the unbounded rate of error build up at receivers because the DR vectors scheduled beyond the next trigger were discarded. The second algorithm fixed this problem by sending out all the DR vectors, scheduled beyond the next trigger, at the next trigger to the respective receivers. Even though the second algorithm achieved higher degree of fairness than in the base case and worked fine in the static delay environment, it increased the overall export error of the system and also suffered from the *hysteresis effect* in a variable delay environment which led to increase in the accumulated export error of all the receivers in the system as compared to the base case.

This motivated us to explore the possibility of an algorithm which will achieve fairness among all the receivers and at the same time not increase the mean accumulated export error for all the receivers in the system and achieve this in the same budget of DR vectors as in the base case.

Intuitively it seemed that using the same budget of DR vectors as in the base case it is possible to send DR vectors more frequently to a receiver having higher accumulated export error, say receiver X and send DR vectors at a lower frequency to a receiver having a lower accumulated export error, say receiver Y. In this way we can bring down the error of receiver X and raise the error of receiver Y such that their accumulated export errors are equal. This will achieve fairness among the receivers and at the same time improve the performance of the game at receiver X as he gets DR vectors more frequently than the base case.

Budget based probabilistic algorithm was explored to implement the intuition discussed above. It was observed that the probabilistic algorithm did not conform with the intuition. This was due to the fact that there is no guarantee that a receiver gets a DR vector within a certain amount of time. Hence, we concluded that a deterministic algorithm is needed to achieve the goal discussed above.

We then proposed a budget based deterministic algorithm and showed that by using a very low threshold, it is possible to achieve a higher degree of fairness and at the same time push the export error of all the receivers towards the mean export error of the system. We showed that the budget based deterministic algorithm performed better than the 1/3 base case not only in the static delay environment but also in the variable delay environment.

5.3 Extensions to Open TNL:

Our final goal was to generalize the work we did to all dead reckoning based games. We reviewed three different *game-networking libraries* and decided to extend our work under Open TNL which has been used in creating many popular multiplayer games till date and is actively being used in the open source community. We extended Open TNL to support the concept of dead reckoning using DRObjects and thus allow developers to create and use dead reckonable game objects. The *accuracy model* was also implemented in the DRObject extension to Open TNL so as to make the dead reckoning

more accurate. The budget based deterministic model will be implemented in Open TNL in the future.

APPENDIX A
IMPLEMENTATION DETAILS – ACCURACY MODEL

The appendix contains implementation details of the accuracy model implemented in BZFlag. It shows how the receiver in dead reckoning the object uses the timestamp more accurately. It is assumed that the sender's and the receiver's clocks are synchronized.

```
/** unpack the update message */
void* Player::unpack(void* buf, double timestamp)
{
  buf = state.unpack(buf,timestamp);
  setDeadReckoning_ts();
  return buf;
}

/** setDeadReckoning_ts() sets all the information necessary to dead reckon the object
(including time) */
void Player::setDeadReckoning_ts()
{
  // save stuff for dead reckoning
  TimeKeeper ts(state.timestamp);

  inputTime = ts;
  inputStatus = state.status;
  inputPos[0] = state.pos[0];
  inputPos[1] = state.pos[1];
  inputPos[2] = state.pos[2];
  inputSpeed = hypotf(state.velocity[0], state.velocity[1]);
```

```
  if (cosf(state.azimuth) * state.velocity[0] + sinf(state.azimuth) * state.velocity[1] < 0.0f)
    inputSpeed = -inputSpeed;
  if (inputSpeed != 0.0f)
    inputSpeedAzimuth = atan2f(state.velocity[1], state.velocity[0]);
  else
    inputSpeedAzimuth = 0.0f;
  inputZSpeed = state.velocity[2];
  inputAzimuth = state.azimuth;
  inputAngVel = state.angVel;
}

/** Dead Reckon – Code snippet*/
{
    inputPrevTime = TimeKeeper::getTick();
    const float dt = inputPrevTime - inputTime;

    predictedVel[0] = fabsf(inputSpeed) * cosf(inputSpeedAzimuth);
    predictedVel[1] = fabsf(inputSpeed) * sinf(inputSpeedAzimuth);

    // follow a simple parabola
    predictedPos[0] = inputPos[0] + dt * predictedVel[0];
    predictedPos[1] = inputPos[1] + dt * predictedVel[1];

}
```

APPENDIX B

IMPLEMENTATION DETAILS – FAIRNESS MODEL

Appendix B contains implementation details of some of the important functions used in the fairness models discussed in chapter three.

/ CalcError – used to calculate the error over time between two different DR vectors in a given time interval */**

```
double calcError(DR old, DR lat, struct timeval start, struct timeval end)
{
   double starttime, endtime, olddrtime, curdrtime;    //Time related stuff
   double x, y, vx, vy, oldx, oldy, oldvx, oldvy, latx, laty, latvx, latvy; //X, Y and VX and
VY
   double sxnew, sxold, synew, syold;
   double t, a, b, c, error;

   if(old.state.order == lat.state.order)
        return 0;

   /** Project initial X and Y on basis of DR info */
   olddrtime = (double)old.ts.tv_sec + (double)((1.0e-6f)*(old.ts.tv_usec));
   curdrtime = (double)lat.ts.tv_sec + (double)((1.0e-6f)*(lat.ts.tv_usec));

   oldx = old.state.pos[0];
   oldy = old.state.pos[1];

   oldvx = old.state.velocity[0];
   oldvy = old.state.velocity[1];
```

```
latx = lat.state.pos[0];
laty = lat.state.pos[1];

latvx = lat.state.velocity[0];
latvy = lat.state.velocity[1];

starttime = (double)start.tv_sec + (double)((1.0e-6f)*(start.tv_usec));
endtime = (double)end.tv_sec + (double)((1.0e-6f)*(end.tv_usec));

/** Projected initial position at time t = 0 for old DR*/
sxold = oldx + (starttime - olddrtime) * oldvx;
syold = oldy + (starttime - olddrtime) * oldvy;

/** Projected intial position at time t = 0 for new DR */
sxnew = latx + (starttime - curdrtime) * latvx;
synew = laty + (starttime - curdrtime) * latvy;

/** Input Angular Velocity = 0 -> straight path, else, circular path */
if(old.state.angVel != 0.0f)
{
    // find current position on circle:
    // tank with constant angular and linear velocity moves in a circle
    // with radius = (linear velocity/angular velocity).  circle turns
    // to the left (counterclockwise) when the ratio is positive.

    float inputSpeed = hypotf(oldvx,oldvy);
    float radius = inputSpeed / old.state.angVel;
    float offAzimuth = old.state.azimuth - 0.5f * M_PI;
```

```
        float angle = offAzimuth + (starttime - olddrtime) * old.state.angVel;
        sxold = oldx + radius * (cosf(angle) - cosf(offAzimuth));
        syold = oldy + radius * (sinf(angle) - sinf(offAzimuth));

        angle = offAzimuth + (endtime - olddrtime) * old.state.angVel;
        float exold = oldx + radius * (cosf(angle) - cosf(offAzimuth));
        float eyold = oldy + radius * (sinf(angle) - sinf(offAzimuth));

        oldvx = (exold - sxold)/(endtime - starttime);
        oldvy = (eyold - syold)/(endtime - starttime);
    }

    if(lat.state.angVel != 0.0f)
     {
        // find current position on circle:
        // tank with constant angular and linear velocity moves in a circle
        // with radius = (linear velocity/angular velocity).  circle turns
        // to the left (counterclockwise) when the ratio is positive.
        float inputSpeed = hypotf(latvx, latvy);
        float radius = inputSpeed / lat.state.angVel;
        float offAzimuth = lat.state.azimuth - 0.5f * M_PI;
        float angle = offAzimuth + (starttime - curdrtime) * lat.state.angVel;
        sxnew = latx + radius * (cosf(angle) - cosf(offAzimuth));
        synew = laty + radius * (sinf(angle) - sinf(offAzimuth));

        angle = offAzimuth + (endtime - curdrtime) * lat.state.angVel;
        float exnew = latx + radius * (cosf(angle) - cosf(offAzimuth));
        float eynew = laty + radius * (sinf(angle) - sinf(offAzimuth));
```

```
        latvx = (exnew - sxnew)/(endtime - starttime);
        latvy = (eynew - synew)/(endtime - starttime);

    }

    /* Error Calculation for t */
    x = sxnew - sxold;
    y = synew - syold;

    vx = latvx - oldvx;
    vy = latvy - oldvy;

    /** The time interval */
    t = endtime - starttime;

    a = vx*vx + vy*vy;
    b = 2.0*x*vx + 2.0*y*vy;
    c = x*x + y*y;

    if(a == 0.0)
        error = sqrt(c)*t;
    else
      error = (1.0/4.0)*((2*a*t + b)*(sqrt(a*t*t +b*t +c))/a)
          + (1.0/2.0)*log((0.5*b + a*t)/sqrt(a) + sqrt(a*t*t +b*t +c))*c*(1.0/sqrt(a))
          - (1.0/8.0)*log((0.5*b + a*t)/sqrt(a) + sqrt(a*t*t + b*t + c))*b*b*pow(a,-1.5) - (
            (1.0/4.0)*((b)*(sqrt(c))/a) + (1.0/2.0)*log((0.5*b)/sqrt(a) + sqrt(c))*c*(1.0/sqrt(a)) -
(1.0/8.0)*log((0.5*b)/sqrt(a) + sqrt(c))*b*b*pow(a,-1.5)  );
```

```
   return error;
}

/** CalculateP – Calculate the relative probabilities based on the factors */
int calculateP(perPlayerInfo * playerInfo, int type)
{
  // Calculate the total error of all players
      double totError = 0.0;
      double totAccError = 0.0;
      double instTotError = 0.0;
      double var = 0.0;
      double mean = 0.0;
      double stdev = 0.0;
      int pl = 0;

      for(int i = 0; i < noOfPlayers; i++)
          {
            if(playerInfo[i].drCnt == 0 || playerInfo[i].drList.empty())
                continue;
            totAccError += playerInfo[i].accError;

            totError += playerInfo[i].TotalError;
            instTotError += (playerInfo[i].TotalError - playerInfo[i].accError);
            pl++;

          }
```

```
mean = totAccError/pl;

for(int i = 0; i < noOfPlayers; i++)
    {
      if(playerInfo[i].drCnt == 0 || playerInfo[i].drList.empty())
          continue;
      var += (playerInfo[i].accError - mean)*(playerInfo[i].accError - mean);
    }

var = var/pl;
stdev = sqrt(var);

// Calculate the total time since last DR for all the players
double totTime = 0.0;
   for(int i = 0; i < noOfPlayers; i++)
    {
      if(playerInfo[i].drCnt == 0 || playerInfo[i].drList.empty())
          continue;
      PlayerAck autoAck;
      autoAck = playerInfo[i].latestAck;
      totTime += (getWallTime(nowTime) - getWallTime(autoAck.ts));
    }

for(int i = 0; i < noOfPlayers; i++)
    {
      if(playerInfo[i].drCnt == 0 || playerInfo[i].drList.empty())
        {
```

```
            playerInfo[i].PAccError = 0.0;
            playerInfo[i].PTime = 0.0;
            playerInfo[i].PInstError = 0.0;
            continue;
          }
          playerInfo[i].PAccError = playerInfo[i].TotalError/totError; //Probability based on Total error till current time

          PlayerAck autoAck = playerInfo[i].latestAck;
          playerInfo[i].PTime = (getWallTime(nowTime) - getWallTime(autoAck.ts))/totTime;  //Probability based on time since last DR was sent to the player

          playerInfo[i].PInstError = (playerInfo[i].TotalError - playerInfo[i].accError) / instTotError; //Probability based on Inst Error
        }
    return 0;
}
```

APPENDIX C

IMPLEMENTATION DETAILS – OPEN TNL EXTENSIONS

This section contains the implementation details of the work done in extending Open TNL for supporting dead reckonable objects. The accuracy model discussed in chapter two is also incorporated in the dead reckoning extensions.

DRObject Header File –

```
/** DRObject Header File */
//DRObject derived from NetObject

#include "../tnl/tnl.h"
#include "../tnl/tnlNetBase.h"
#include "../tnl/tnlGhostConnection.h"
#include "../tnl/tnlNetInterface.h"
#include "../tnl/tnlNetObject.h"
#include "../tnl/tnlLog.h"
#include "../tnl/tnlRPC.h"
#include "../tnl/tnlString.h"
#include "../tnl/tnlBitStream.h"
#include "../tnl/tnlNetConnection.h"
#include "../tnl/tnlRandom.h"
#include "../tnl/tnlSymmetricCipher.h"
#include "../tnl/tnlAsymmetricKey.h"
#include <math.h>


namespace TNLTest
{

  // Position structure used for Dead Reckonable objects current state information
  struct Position
  {
    TNL::F32 x; //< X position of the object
    TNL::F32 y; //< Y position of the object

    TNL::F32 angle;//Angle to the X-Axis to indicate the direction of the object
    TNL::F32 speed;//Speed of the object
```

```
    TNL::F32 vx;// X Component of Velocity
    TNL::F32 vy;// Y Component of Velocity

    TNL::F32 angVel;// Angular Velocity (can be used to detect circular motion/ circular
dead reckoning)

    TNL::U32 timeNow;// Time at which the object holds the position.

    TNL::U32 lastSendTime; // Last time a DR vector was sent

    TNL::U32 genTime; //Time the DR was generated on owner or on server

  };


   /** DRObject class inherits from the TNL NetObject and adds dead reckoning
capability to the NetObject*/
   class DRObject: public TNL::NetObject
    {
      typedef TNL::NetObject Parent;

    protected:

      TNL::F32 epsilon;//Error Threshold
      TNL::U32 tDelta; //Time Threshold in MilliSeconds
      TNL::F32 calcAngle(TNL::F32 x1, TNL::F32 y1, TNL::F32 x2, TNL::F32 y2);//
Calculate the direction in which the DRObject is moving

    public:

      Position currPos;

      /** Default Constructor*/
      // Sets the position to initial value and sets the Ghostable flag indicating the object is
ghostable
      DRObject();

      /** Destructor */
      ~DRObject();

      /** set the error threshold */
      // Allows the error threshold and the time threshold to be set
```

```
    // The threshold is checked everytime the object state changes and if the threshold
is crossed an update is sent
    bool setThreshold(TNL::F32 eps, TNL::U32 tmD);

    /** isThresholdCrossed is called from within TestThresholdAndUpdate()*/
    // returns true if any of the threshold - error or time have been crossed
    // else returns false
    bool isThresholdCrossed(TNL::F32 x, TNL::F32 y);

    /** deadReckon is called at the reciever side to dead reckon the DRObject between
updates */
    //  Simple linear dead reckoning is implemented here
    //  A user can override this function in the derived class to do more complex dead
reckoning
    virtual void deadReckon();

    /** TestThresholdAndUpdate() will be called from within the game loop or every time
an event like object moving (Mouse Event )occurs */
    void TestThresholdAndUpdate(TNL::F32 x, TNL::F32 y);


    /** Remote function that client(Owner of the object) calls to set the position of the
player on the server. */
    TNL_DECLARE_RPC(rpcSetObjectPos, (TNL::F32 x, TNL::F32 y, TNL::F32 vx,
TNL::F32 vy, TNL::F32 angVel));

    /** sendPosUpdate will be the first call from packUpdate() from the derived class
object */
    //  sendPosUpdate checks if new update has arrived on the server from the owner of
the object
    //  If there is new update, send the update to all the clients replicating the object
    void sendPosUpdate(TNL::GhostConnection *connection,  TNL::U32 updateMask,
TNL::BitStream *stream);

    /** recvPosUpdate will be the first call from unPackUpdate() from the derived class
object */
    //  recvPosUpdate checks if DRObject update has been sent along with other
updates for the derived object
    //  if there is an update, it updates the position information for the object.
    void recvPosUpdate(TNL::GhostConnection *connection, TNL::BitStream *stream);

    // This macro invocation declares the Player class to be known
    // to the TNL class management system.
```

```
    TNL_DECLARE_CLASS(DRObject);

  };


};

using namespace TNLTest;
```

/** DRObject Implementation - Source File */**

```
/** The Implmentation of DRObject */

#include "DRObject.h"

namespace TNLTest
{
 /** Implement the DRObject */
 TNL_IMPLEMENT_NETOBJECT(DRObject);

 // Constructor for the DRObject
 // Initializes the Position variables
 DRObject::DRObject()
 {
  //Initialize the position
  currPos.x = 0.0;
  currPos.y = 0.0;

  currPos.angle = 0.0;
  currPos.speed = 0.0;

  currPos.vx = currPos.speed*cosf(currPos.angle);
  currPos.vy = currPos.speed*sinf(currPos.angle);

  currPos.angVel = 0.0;
  currPos.timeNow = TNL::Platform::getRealMilliseconds();
  currPos.lastSendTime = 0;

  // Set the ghostable flag so that the TNL knows this object is to be ghosted
  mNetFlags.set(Ghostable);
 }

 //Destructor
```

```
DRObject::~DRObject()
{
  //Nothing as yet
}

/** Set the error and time threshold values */
bool DRObject::setThreshold(TNL::F32 eps, TNL::U32 tmD)
{
  epsilon = eps; // error threshold - distance from the trajectory on which the object
currently is
  tDelta = tmD;//Time in milliseconds

}

/** Test if any of the threshold have crossed, if true, send the update to the server */
void DRObject::TestThresholdAndUpdate(TNL::F32 x, TNL::F32 y)
{
  //Check if the Error Threshold is crossed and if the threshold is crossed, generate a
DR to the Server
  if(isThresholdCrossed(x, y))
    {
      //TNL::logprintf("Threshold Crossed sending update %g, %g %u", x, y,
currPos.timeNow);

      //call the rpc to update Position to server
      rpcSetObjectPos(currPos.x, currPos.y, currPos.vx, currPos.vy, currPos.angVel);

    }
}

/** check if the position of the object currently is sufficient to generate an update*/
bool DRObject::isThresholdCrossed(TNL::F32 x, TNL::F32 y)
{
  Position tempPos = currPos;

  if(tempPos.timeNow < TNL::Platform::getRealMilliseconds()) // ensure that some time
has passed since the function was called last
    {
      currPos.timeNow = TNL::Platform::getRealMilliseconds();

      if(currPos.x != x && currPos.y != y ) // ensure that the object has moved
        {
```

```
            /** If above condition is satisfied, update all the information that has changed -
<X,Y,currTime> */

            currPos.x = x;
            currPos.y = y;

            /** Calculate the angle and speed based on the last information we have */
            currPos.angle = calcAngle(tempPos.x, tempPos.y, x, y);
            currPos.speed = (sqrt((x - tempPos.x)*(x - tempPos.x) + (y - tempPos.y)*(y -
tempPos.y)))/(currPos.timeNow - tempPos.timeNow);

            currPos.vx = currPos.speed*cos(currPos.angle);
            currPos.vy = currPos.speed*sin(currPos.angle);

          }
         else //position did not change, hence velocity should be zero
          {

            currPos.vx = 0.0;
            currPos.vy = 0.0;

          }

         if((tempPos.lastSendTime + tDelta) <= currPos.timeNow) // If time threshold has
elapsed
          {
            currPos.lastSendTime = currPos.timeNow;
            return true;
          }

         /*  Important */
                if(distanceFromLine(tempPos, x, y) > epsilon)
                  {
                    return true;
                  }

     }
   return false;
 }


 /** Calculate the direction in which the object is moving */
 TNL::F32 DRObject::calcAngle(TNL::F32 x1, TNL::F32 y1, TNL::F32 x2, TNL::F32 y2)
```

```
{
  TNL::F32 xdiff, ydiff, slope, angle;

  xdiff = x2-x1;
  ydiff = y2-y1;

  angle = atan2(ydiff, xdiff);

  return angle;

}

/** Linear dead reckoning */
// The derived class can override this function to have more complex deadreckoning */
void DRObject::deadReckon()
{
  Position tempPos;

  tempPos = currPos;

  currPos.timeNow = TNL::Platform::getRealMilliseconds();

  TNL::U32 timeDiff = currPos.timeNow - tempPos.timeNow;

  currPos.x = tempPos.x + (timeDiff)*tempPos.vx;
  currPos.y = tempPos.y + (timeDiff)*tempPos.vy;

}

/**  Send the position update */
void DRObject::sendPosUpdate(TNL::GhostConnection *connection,  TNL::U32
updateMask, TNL::BitStream *stream)
{

  if(stream->writeFlag(updateMask & BIT(0)))
    {
       //Write the position information to the clients
       stream->write(currPos.x);
       stream->write(currPos.y);
       stream->write(currPos.vx);
       stream->write(currPos.vy);
       stream->write(currPos.angVel);
```

```
    }

  }

  /** Receive the position update */
  //shud be the first call from unpackupdate as the bit zero has to be checked for position
information update
  void DRObject::recvPosUpdate(TNL::GhostConnection *connection, TNL::BitStream
*stream)
  {
    // see if the objects's position has been updated:
    //Note that the first bit of the mask is used for Position information hence, we should
call recvPosUpdate before any other call in unpackupdate

    if(stream->readFlag())
     {
         Position tempPos;

         //read the position information and update the ghost
         stream->read(&tempPos.x);
         stream->read(&tempPos.y);
         stream->read(&tempPos.vx);
         stream->read(&tempPos.vy);
         stream->read(&tempPos.angVel);

         // predict the current position based on velocity and lag - Accuracy ( Between
Server and all clients)
         currPos.x = tempPos.x + tempPos.vx*(connection->getOneWayTime());
         currPos.y = tempPos.y + tempPos.vy*(connection->getOneWayTime());

         currPos.vx = tempPos.vx;
         currPos.vy = tempPos.vy;

         currPos.angVel = tempPos.angVel;

         // update the current time
         currPos.timeNow = TNL::Platform::getRealMilliseconds();

     }

  }
```

```
  /** The owner of the object calls the RPC and updates the position on the server*/
  //RPC Function to be executed on the Server
  TNL_IMPLEMENT_NETOBJECT_RPC(DRObject, rpcSetObjectPos,
                              (TNL::F32 newx, TNL::F32 newy, TNL::F32 newvx, TNL::F32
newvy, TNL::F32 newangVel), (newx, newy, newvx, newvy, newangVel),
                              TNL::NetClassGroupGameMask,
TNL::RPCGuaranteedOrdered, TNL::RPCToGhostParent, 0)
  {

    // predict the current position based on velocity and lag - Accuracy (Between Owner
and Server)
    currPos.x = newx + newvx*(mRPCSourceConnection->getOneWayTime());
    currPos.y = newy + newvy*(mRPCSourceConnection->getOneWayTime());

    currPos.vx = newvx;
    currPos.vy = newvy;

    currPos.angVel = newangVel;

    // update the current time
    currPos.timeNow = TNL::Platform::getRealMilliseconds();
    currPos.genTime = TNL::Platform::getRealMilliseconds(); //the time the server got the
update

    // Bit zero is reserved for Dead-reckoning information
    // This bit if set indicates that the server has new information about the objects
position and the information needs to be sent
    // to all the clients. The packUpdate() function is called when any of the bits are set
and in turn calls the sendPosUpdate().
    setMaskBits(BIT(0));
  }
};
```

REFERENCES

[1] L. Gautier and C. Diot, “Design and Evaluation of MiMaze, a Multiplayer Game on the Internet,” in Proc. of IEEE Multimedia (ICMCS’98), 1998.

[2] M. Mauve, “Consistency in Replicated Continuous Interactive Media,” in Proc. of the ACM Conference on Computer Supported Cooperative Work (CSCW’00), 2000, pp. 181–190.

[3] S.K. Singhal and D.R. Cheriton, “Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality,” Presence: Teleoperators and Virtual Environments, vol. 4, no. 2, pp. 169–193, 1995.

[4] BZFlag Forum, “BZFlag Game,” URL: http://www.bzflag.org.

[5] Z. B. Simpson, “A Stream Based Time Synchronization Technique for Networked Computer Games,” URL:http://www.mine-control.com/zack/timesync/timesync.html.

[6] L. Pantel and L. Wolf, “On The Suitability of Dead Reckoning Schemes for Games,” in *Proc. of NetGames2002*, 2002.

[7] L. Pantel and L.C. Wolf, “On the Impact of Delay on Real-Time Multiplayer Games,” in Proc. of ACM NOSSDAV’02, May 2002.

[8] Y. W. Bernier, “Latency Compensation Methods in Client/Server In-game Protocol Design and Optimization,” in Proc. of Game Developers Conference’01, 2001, URL: http://www.gdconf.com/archives/proceedings/2001/prog_papers.html.

[9] Y. Lin, K. Guo, and S. Paul, “Sync-MS: Synchronized Messaging Service for Real-Time Multi-Player Distributed Games,” in Proc. of 10th IEEE International Conference on Network Protocols (ICNP), Nov 2002.

[10] K. Guo, S. Mukherjee, S. Rangarajan, and S. Paul, “A Fair Message Exchange Framework for Distributed Multi-Player Games,” in *Proc. of NetGames2003*, May 2003.

[11] Nation Institute of Standards and Technology, “NIST Net,” URL: http://snad.ncsl.nist.gov/nistnet/.

[12] M. Allman and V. Paxson, “On Estimating End-to-End Network Path Properties,” in Proc. of ACM SIGCOMM’99, Sept. 1999.

[13] “NetZ – Multiplayer architecture for online games”, url: http://www.proksim.com

[14] B. Kelly, S. Aggarwal, “A Framework for a fidelity based agent architecture for distributed interactive simulation”, Proc of 14th workshop on standards for Distributed Interactive Simulation.

[15] S. Aggarwal, C. Chraibi, “On the combined scheduling of Hyperperiodic, Periodic, and aperiodic tasks.

[16] Li Zou, Mostafa H. Ammar, C. Diot, „An Evaluation of Grouping Techniques for state dissemination in networked multi-user games“, Proc of Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01) August, 2001.

[17] “IEEE standard for distributed interactive simulation – application protocols”. IEEE Computer Society,1995.

[18] C. Diot, L. Gautier, “A Distributed Architecture for Multiplayer Interactive Applications on the Internet”, IEEE Network, August 1999.

[19] “The PARADISE (Performance Architecture for Advanced Distributed Interactive Simulation Environments) Project”, url: http://www-dsg.stanford.edu/paradise.html

[20]E. Frecon, M. Stenius, “DIVE: A Scaleable network architecture for distributed virtual environments”, Distributed Systems Engineering Journal, September 1998.

[21] D.L. Mills, “Network Time Protocol (Version 3) Specification, Implementation and Analysis”, RFC-1305, March 1992.

[22]Forum Nokia, ”Overview of Multiplayer Mobile Game Design”, December 2003.

[23] Forum Nokia, “Multiplayer MIDP Programming”, October 2003.

[24] J. Arnson, “Dead Reckoning: Latency Hiding for Networked Games”, September 1997. url: http://www.gamasutra.com/features/19970919/aronson_01.htm

[25] “Open TNL”, url: http://www.opentnl.org/

[26] “Raknet”, url: http://rakkarsoft.com/

[27] “Zoidcom“, url: http://www.zoidcom.com/

[28] Grenville Armitage. “An experimental estimation of latency sensitivity in multiplayer Quake 3”. In Proceedings of the 11th IEEE International Conference on Networks (ICON 2003), Sydney, Australia, September 2003

[29] Tristan Henderson and Saleem Bhatti. “Modelling user behaviour in networked games.” In Proceedings of ACM Multimedia 2001, pages 212–220, October 2001.

[30] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. “The effect of latency on user performance in Warcraft III.”. In Proceedings of the 2nd Workshop on Network and System Support for Games (NetGames 2003), pages 3–14, 2003.

[31] B. Knutsson, H. Lu, W. Xu, B. Hopkins, “Peer-to-Peer Support for Massively Multiplayer Games”, Proc of INFOCOM, July 2004.

BIOGRAPHICAL SKETCH

Hemant Banavar was born on 22nd March 1981 in the city of Bangalore, India. He was brought up in Kolhapur, India. He completed his 10th grade from St. Xavier's High school in 1996 and his 12th grade from Vivekanand College in 1998 with high grades. By the time he finished his 12th grade, he was deeply interested in pursuing his long time interest – Computers. He pursued his interest and graduated with a Bachelor's degree in Computer Science from Shivaji University, India. At this time, he decided to further enrich his knowledge by going for an advanced degree in Computer Science. He has now attained a Master's degree in Computer Science from Florida State University.