Play Patterns for Path Prediction in Multiplayer Online Games

by

Jacob Agar

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science
in

Computer Science
Carleton University
Ottawa, Ontario, Canada
April 2012

Copyright ©
2012 - Jacob Agar

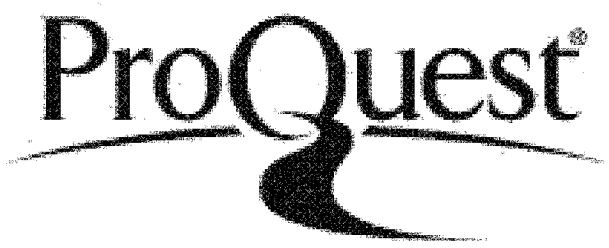
UMI Number: MR91493

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI MR91493
Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

Traditional dead reckoning schemes predict an entity's position by assuming that an entity move with constant force or velocity. However, because much of entity movement is rarely linear in nature, using linear prediction fails to produce an accurate result. Among existing dead reckoning methods, only few focus on improving prediction accuracy via genuinely non-traditional methods for predicting the path of a player. Multiplayer Online Games (MOGs), a recent new type of Distributed Interactive Simulation, have a large number of users. Providing accurate prediction results is crucial for quality of play, which is the ultimate goal to satisfy millions of online users. In this research, we propose a new prediction method based on *play patterns* involving the surrounding game state and objects of the player.

We implemented a 2D top-down multiplayer online game to act as a test harness that we used to collect play data from 44 experienced players. From the data for half of these players, we extracted play patterns, which we used to create our dead reckoning algorithm. A comparative evaluation proceeding from an extensive set of simulations (using the other half of our play data) suggests that all 3 versions of our *Experience Knows Best (EKB)* algorithm yield more accurate predictions than the IEEE standard dead reckoning algorithm and the recent "Interest Scheme" algorithm.

Acknowledgments

I would like to thank my supervisors Wei Shi and Jean-Pierre Corriveau for all their help, guidance and support. I would also like to thank Shannon Cressman for her support throughout the production of this thesis. My parents John and Celia also offered endless support throughout my research. The Lan Clan group offered much help in participating in play tests and play sessions and were invaluable to the completion of our research. The same goes for all the students at Carleton University (along with the visitors to the school) that participated in the play tests and play sessions. This research could not have been completed without their help. I would like to thank the financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC)

Table of Contents

List of Tables

List of Figures

List of Definitions

Term	Definition
Round Trip Time (RTT)	time it takes from sending a packet to a remote machine over the internet, to receiving an acknowledgement packet from that machine.
Lag/Delay/Latency	measures how long it takes for a bit of data to travel across the network from one node or endpoint to another.
Loss	the amount of packets that are not received at the remote machine due to signal degradation over a network medium, rejected packets and congestion at a given network node.
Path Prediction	predicting the current position of an entity in a distributed interactive simulation (DIS) application.
Dead Reckoning	the method through which the positional information of objects in a DIS or networked video game are predicted in order to minimize the appearance of lag and to minimize network traffic.

Playability	the point beyond which the player deems the game broken and cannot play as a result of high amounts of game-state inconsistency caused by adverse network conditions.

Chapter 1

Introduction

Consumers have spent 25.1 billion dollars on video games in 2010 [3]. 72% of American households play video games, and 19% of most frequent game players pay to play games online [3]. Multiplayer online games (MOGs) make up a huge portion of one of the largest entertainment industries on the planet. It is for this reason that it is useful to maximize a player's experience while playing such a video game. Since MOGs are a distributed system, they are inherently more difficult to design and produce than a traditional locally played video game. There are many architectural problems when dealing with a distributed system like an MOG that are non-existent in a traditional video game. Players playing in geographical locations thousands of kilometers away from each other need to have their actions appear to be executed in the same virtual space. This can be problematic because game messages are delayed or often lost when sending over large distances. This delay and loss causes discrepancies between the simulated environment from player to player, resulting in unfair scenarios and incorrect perception of events. This thesis has the goal of outlining a method to improve the playing experience and perception of lag of the player in the face of high amounts of network delay and loss.

One method of reducing the perception of adverse network conditions for the player is dead reckoning (DR). Dead reckoning is the method through which the

positional information of objects in a distributed interactive simulation (DIS) or networked video game are predicted in order to minimize the appearance of lag and to minimize network traffic.

1.1 Motivation

The main objective when designing the architecture of a networked video game is to maximize the user's playing experience by minimizing the appearances of the adverse effects of the network during play. In today's gaming world, a player expects the same play experience playing online over the internet as would be experienced from a locally played game. Users could be playing together with well over 30 other players simultaneously, on congested networks and at distances half way around the world. The reality of inter-continental network speeds and conditions are problematic, resulting in a lower quality of player experience. Physical improvements to the infrastructure of the internet can help, but are costly and might not fully solve the problem. In the real world, there will always be some degree of delay and loss in the internet. And so we turn to compensation methods that can reduce and mask the effects of delay and loss. The use of a *dead reckoning* (or path prediction) algorithm is one such compensation method, and can hide the effects of delay and loss considerably [5, 8, 18, 21, 26].

Late or lost packet transmission has the effect of objects in the scene being rendered at out of date or incorrect locations. If objects are simply rendered at their latest known position, their movement is, as a result, jittery and sporadic. This is because they are being drawn at a location where they actually are not, and this looks unnatural. Dead reckoning algorithms predict where an object should be based on past information. They can be utilized to estimate a rendering position that is

more accurate to the true path of the object. This ensures that once the player receives the true position of the object, the positional jump to the correct location is either non-existent or much smaller, creating the illusion that this object is behaving normally.

Applications of lag compensation are not reserved only for MOGs, but also for any distributed interactive simulation (DIS) application. DISs are used by military, space exploration and medicine organizations. When speaking of making improvements to the user's experience, it also speaks of improving the quality of such applications. By predicting the position of an object, it is not necessary to receive an update for that object's motion every time it moves, but only when there is a change in the motion. This allows for a greater degree of network lag and loss, as well lowers the amount of updates that are required to be sent over the network.

1.2 Statement of the Problem and Methodology

To maintain playability within video games that are played over the internet, there is a great need for prediction algorithms to maintain distributed consistency and accuracy at high levels of lag (reaching up to three seconds of packet latency). Traditional prediction schemes predict player position by assuming each player moves with a constant force or velocity. Because player movement is rarely linear in nature, using linear prediction cannot maintain an accurate result. Few dead reckoning methods that have been proposed focus on improving prediction accuracy by introducing new methods of predicting the path of a player. For example, the "Interest Scheme" detailed in [18] improves prediction accuracy in a 2D tank game. The "Interest Scheme" does so by assuming that a player's surrounding objects will have some anticipative effect on the player's path. Unfortunately, the success of the "Interest

Scheme" is not reproducible in a traditional team-based action game, where-in players can move in all directions freely, including not along the forward vector (eg. a tank cannot "strafe" to the left and right of the forward vector, but has to rotate to change direction).

To alleviate these problems, we propose a prediction scheme that takes user play patterns into account. We do so in the context of traditional and typical team-based action game, such as first-person, third-person or top-down shooters. After implementing a test harness/environment (a 2D top-down multiplayer online game titled "Ethereal"), we then conducted several play testing sessions. Our test harness allows us to record all data (including all raw input data) during each play testing session. From observing experienced players playing games, as well from analyzing the collected input data, we devised a series of simple player behaviors that can be used to improve prediction accuracy.

More specifically, we first gathered the keyboard and mouse input of experienced video game players playing our test harness. All world or environment variables are also recorded, such as game object and item positioning, world geometry information, and game events. We then play back this recorded information, simulating real world network conditions, allowing us to observe how different dead-reckoning algorithms behave given different network lag, loss and jitter. Each of 44 players played the game for 20 to 30 minutes each. We divided the players into two groups: one for developing and testing the algorithm, and the other for evaluating our proposed algorithm and several existing algorithms.

We chose to observe prediction algorithms in a 2D action based game because a player's movement is highly unpredictable, and is therefore highly prone to inaccuracies. As a result, an accurate prediction scheme maintaining the playing experience is required in these types of action games.

1.3 Contribution and Overview of Results

We propose a prediction scheme that takes user play patterns into account. We do so in the context of traditional and typical team-based action games, such as a first-person, third-person or top-down shooter games. We chose such a context because a player's movement is highly unpredictable in it, and is therefore highly prone to prediction inaccuracies, thus emphasizing the need for a better prediction scheme. Our work can be summarized as follows:

1. We first implemented a 2D top-down multiplayer online game entitled "Ethereal" to act as our test harness. This test harness allows us to record all data (including raw input data from all participants) during each play testing session. Beyond keyboard and mouse input, all world or environment variables are also recorded, such as game object and item positioning, world geometry information, and game events.

2. We then collected data by having 44 experienced players play Ethereal over three play testing sessions. We arbitrarily divided our players into two sets of equal size:

   - group A: those used to create our proposed dead reckoning algorithm
   - group B: those used to evaluate this algorithm against other algorithms

   More specifically, by observing players of group A play and from the subsequent analysis of the data collected from the games of these players, we identified a series of typical player behaviors, which we call *play patterns*. We then used these play patterns to create the 3 version of our *EKB* (Experience Knows Best) algorithm for dead reckoning.

3. Using the ability of our test harness to play back player inputs from the collected data, we played back the games of group B, varying network latency. We repeated each experiment with each of the 5 algorithms we considered, namely: the 3 versions of our EKB algorithm, the IEEE standard dead reckoning algorithm [5] and the "Interest Scheme" (IS) [7, 18] algorithm. The IEEE standard dead reckoning algorithm will be referred to as the traditional dead reckoning method (TDM).

4. In the end, this allowed us to develop a detailed comparative evaluation of these three algorithms. Our results are reported in section 4.

We compared the 3 versions of our algorithm against the other methods at different lag intervals between 300 ms of lag and 3000 ms of lag. Our comparative evaluation shows that the all 3 versions of the EKB algorithm improve overall prediction accuracy by predicting player position closer to the actual position than the TDM and the IS does. Each version of EKB improves overall prediction accuracy especially at high amounts of lag. Furthermore, the hybrid improved EKB results in a larger amount of accurate prediction hits than the other methods. Also, the hybrid improved EKB does not increase the number of packets transmitted over the other dead reckoning techniques.

1.4 Organization of Thesis

The remainder of our thesis is organized as follows: chapter 2 outlines the background of network simulations and discusses related works; chapter 3 describes the proposed algorithm; chapter 4 lays out the experimental procedures and examines the experimental results; chapter 5 is the conclusion section.

Chapter 2

Background and Related Work

In this chapter, the effects that negative network conditions have on the MOG experience are explained. Then, the common and best methods of increasing the accuracy and consistency of network game play and minimizing the appearance of adverse network conditions will be discussed. Please note that all cited figures in this and the following chapter are used without permission.

2.1 Effects of Delay, Jitter and Loss on Users

Three major issues facing distributed online simulations are the delay, jitter and loss of network data. Delay (or network latency) refers to the time it takes for packets of network data to travel from the sender to the receiver. This delay is usually said to be caused by the time it takes for a signal to propagate through a given medium, plus the time it takes to route the signal through routers. Jitter is a term used as a measure of the variability over time of delay across the network [10]. Loss (often higher when delay is higher), refers to lost network packets and data as a result of the following: signal degradation over a network medium, rejected packets, and congestion at a given network node. These 3 factors cause the MOG to suffer from a

lack of consistency between players, jittery movement of game objects and a general loss of accuracy in the simulation.

In [29], qualitative studies were conducted in determining the effects of adverse network states on the player. Participants were asked to comment on the quality of play at different levels of lag and jitter. Figures 1 and 2 show the mean opinion score (MOS) verses the amount of lag (ping) and jitter respectively. Their findings clearly show that higher quantities of lag and jitter are correlated with a lower player experience.

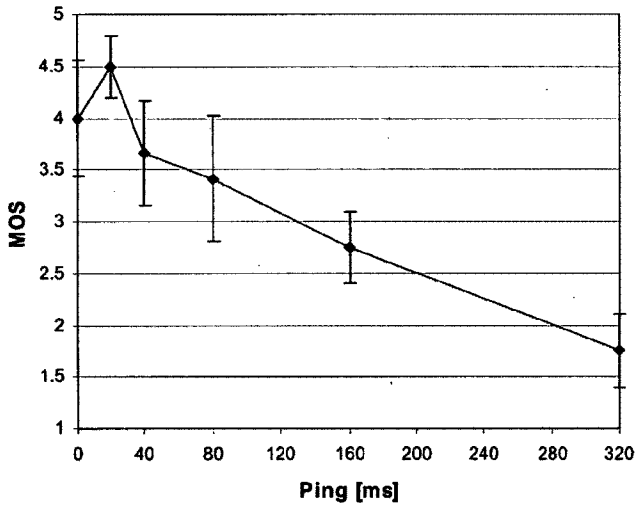


Figure 1: Mean Opinion Score and Lag [29]

In [24], the scores of players (the players' performance based on kills made and deaths suffered) was studied in "Unreal Tournament 2003" (a typical first person shooter video game). Through a series of 20 different scenarios of lag, with different players experiencing different amounts of lag, it was shown that high lag has a substantial negative effect on player performance. These findings are outlined in figure 3.

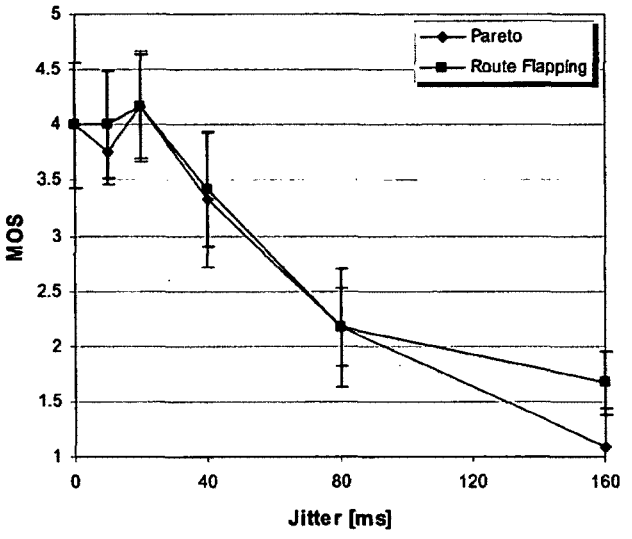


Figure 2: Mean Opinion Score and Jitter [29]

A player's score in a shooting based game is a common metric used when measuring the effects of unfavorable network conditions on the player experience. Findings have been consistent that a higher degree of network loss, delay and/or jitter result in fewer successful shots or kills made, and a lower player score [1, 19, 24, 29]. [1] ran tests in a fast-paced tank shooting game called "BZFlag", [29] gathered data from the popular shooting game "Quake IV", and [16] developed a distributed version of "Quake III" to test in. The performance of a player in these types of video games is based highly on reflexes and instant user input, and as a result, even fraction of a second delays in the network can affect player performance. However, these metrics should only be considered for a select genre or type of game. In [9], running tests in the popular Real-Time Strategy (RTS) PC Game "Warcraft III", it was found that latency of up to 3 seconds has only marginal effects on the performances of players. This is a result of the strategic nature of the RTS genre, where-in strategic planning (as opposed to split second decision making) is more important to good performance. However, under high network lag or loss scenarios, a player's perception of the quality of the game can be hindered. As shown in [29], the adverse effects of the network

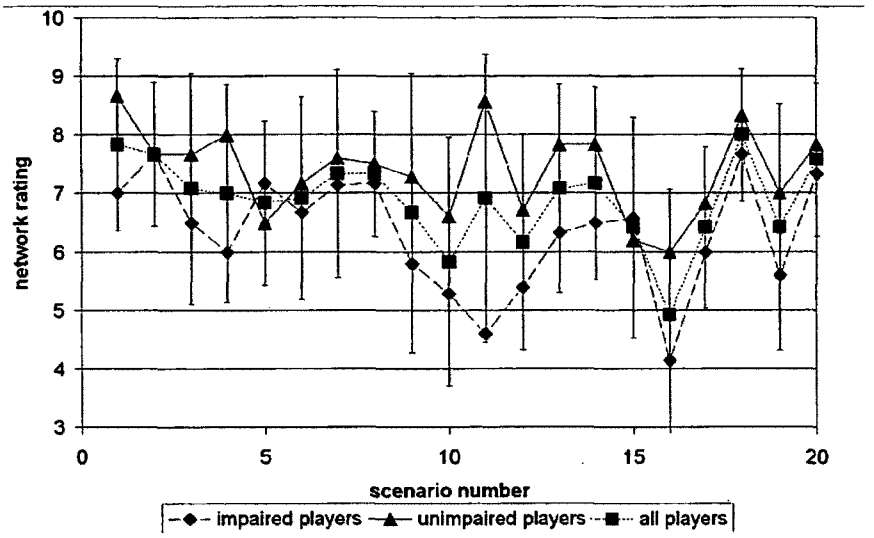


Figure 3: Player scoring based on network conditions [24]

will yield a perception of poor gaming quality. This is a result of either sporadic or jumpy positioning of game objects, or of a delay between the issuing a command for an action and the execution of that action.

2.2 Client-Server Architecture

The 2D action game used to gather and analyze player data in this research employs a client-server architecture. The client-server model is a network architecture where-in each client connect only to a single remote entity, being the server. Action games traditionally use a client server model because it suits their requirements. Action games use a client-server architecture because it keeps client-side computation to a minimum, and allows for a scalable amount of players within the game world. It also minimizes cheating by maintaining the game server as the single authority on all game events. In a client-server model action game, the server is the authority on game events and sends the clients game events, including the actions of other players [4, 28]. Client machines take input from the user and send this information to the server.

2.3 Consistency

2.3.1 Synchronization

A MOG must achieve the illusion that players are playing in the same game world, when in reality they are geographically in different locations. In fact, in a MOG, players do not technically play in the same game space, and are all only receiving a best-guess approximation of the game world. Players only need to see a "close enough" representation of what the server sees for play to continue, but it is not 100% identical. This is a result of network conditions, as well different simulation times and intervals between different parties. One can imagine for a fast paced game, that without some form of time synchronization, a game could easily be unplayable with any amount of lag. A player with higher latency to the server would suffer by seeing all objects later than they actually appear

Traditionally, the server is the authority on all game events, and thus the clients must do some form of time synchronization to the server. There are many ways to do this, each method with their own strengths and weaknesses. A discussion of these methods lies outside the focus of this thesis. In summary, these distributed time synchronization techniques involve timestamps and/or estimating lag times between machines. The time synchronization method used in this simulation is similar to the method described by Simpson in [27]. Periodically, to ensure time is still synchronized, a client sends a packet to the server time-stamped with the current time of the client. The server, immediately on reception of this, time stamps the packet with its own current time and sends it back to the client. The client, from this, can determine how long it took the packet to get to the server and back again (round trip time or

RTT) because of the time stamp. The client can also determine the exact time of the server, assuming it takes the same amount of time to get to and from the server (RTT/2). From here, the client will adjust its time delta between simulation ticks until its time matches the server. The client adjusts its time over several updates of the simulation because a large time jump all at once would cause objects in the scene to jump as well. If the time discrepancy is smoothed out over several frames, then there is no time-jump and the movements of the player are perceived as normal.

2.3.2 Local Lag and Time warp

Local lag as proposed in [19] refers to a network lag hiding technique wherein a delay is introduced between when an input is given and when its execution takes place. This hides lag and improves simulation accuracy because it effectively allows some time for the input packet to reach the server and subsequently to reach the other clients. Local lag allows all parties involved to receive a given event before its execution. Without local lag, in order to stay perfectly time-synchronized, a client or server would have to simulate the object or event forward to the current time, as it will have been received after its execution was supposed to take place. While this method is very effective at ensuring the time-synchronization of distributed events, it introduces a delay between when a player issues a command and when it is executed. This can be a problem, depending on what type of input the player gives. A player is more likely to notice delay regarding player movement or mouse movement input than delay regarding firing or shooting. The common method in most popular action games, for this reason, is to introduce a delay when dealing with firing or shooting, but to have no delay in regards to player movement or mouse movement input. For this reason, in our research simulation, we decided to employ no local lag for player movement, but to introduce a small amount of delay for weapons firing. This allows us to benefit from local lag, without causing the annoyance of having player movement delay.

In [19], Liang et. al. go on to propose a method to further reduce the negative effects of lag on the simulation, namely accuracy. Since a packet, due to jitter and different lag between players, can arrive at odd intervals, and even out of order, events need to be sorted in such a way to maintain temporal accuracy. To account for this, the common method used in the video game industry is a system called time warp. Time warp refers to the "rewinding" of the simulation to execute events at the appropriate time. This ensures that events happen the way they are supposed to, as well ensuring consistency between different parties geographically.

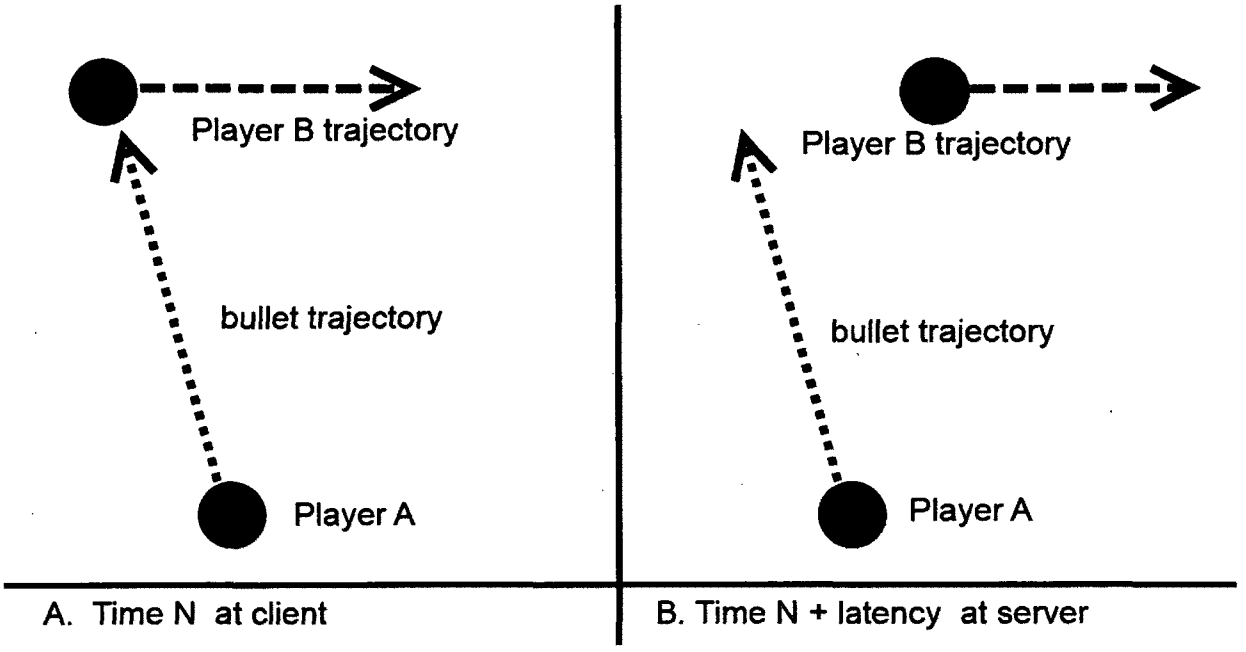


Figure 4: Inconsistency without time warping.

Referring to Figure 4, assume player A has a network delay of 150ms to the server. Player A shoots a bullet at player B. According to player A's view (left side of figure 4), it looks like the bullet hit player B. But since the network message took $150ms$ to reach the server, according to the lag of player B, where the hit is not registered because player B has moved out of the way in the time it took for the message to arrive at the server (right side of figure 4). Time warp ensures this does

not occur, by rewinding player A to its position $150ms$ ago when the bullet was fired to check for the collision (to make the situation server side look like the left side of figure 4). The simulation testbed used for this research includes this time warping technique, to help secure against the ill effects of lag.

2.4 Interactivity

2.4.1 Object Interpolation

An issue with regards to MOGs is that of obtaining smooth, seamless motion of network objects. If network conditions were perfect, then packets would never be lost, would be sent and received with no delay and in perfect temporal order. If this were the case, then it would be feasible to draw game objects only according to newly arriving information. Unfortunately this is not the case. When a packet arrives late, it means while the client is waiting for the late packet, it does not know where to draw that object, and thus it will draw it incorrectly. Without receiving position information, the object is drawn to be not moving. After the receiver has been waiting for the newest packet to arrive, it may then receive 2 or more data packets representing 2 or more time steps, which immediately get executed in a single time step. This results in the jittery movement of rendered objects. A common solution, as outlined by the popular video game development company Valve, is to draw game objects in the past, allowing the receiver to smoothly interpolate positional data between two recently received packets [4,28]. The method works because time is rewound for that object, allowing current or past information about an object represent the future information of that object. Then to draw the object, a position is interpolated from information ahead of and behind the new current position (which is actually in the

past), as shown in figure 5. Without information ahead of when rendering occurs, we can at best draw it at its current known position, which as mentioned before, yields jittery rendering. As long as the rewind time (or interpolation time) is greater than lag, the position of the object should be able to be interpolated. Interpolation times are traditionally a constant value, or equal to the amount of lag to the server at any given time. The method used for this research is a constant interpolation value of 100ms, as is used in the popular action game Half-Life 2 [4,28]. It is a value sufficient enough to cover a large percentage of the lag or loss that will occur.

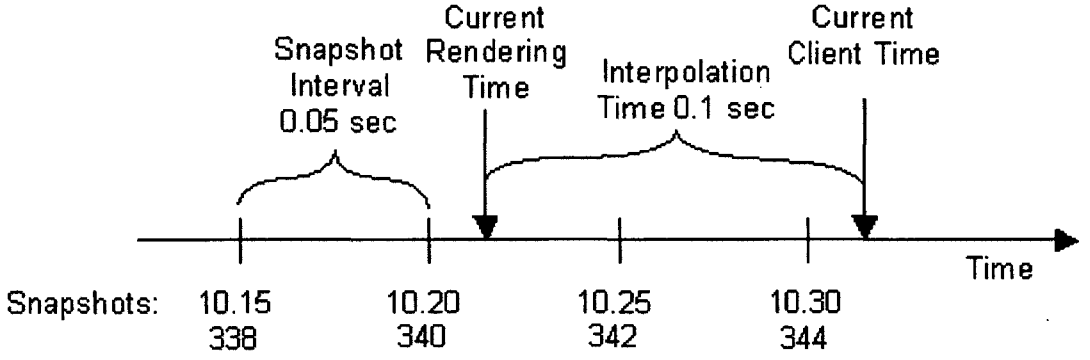


Figure 5: Object Interpolation [4,28]

2.5 Dead Reckoning

In a video game context, the user's playing experience is the most important factor. The main goal of network compensation methods is to minimize the perceived influence of adverse network conditions on the player. Dead reckoning is a widely used method to achieve this goal. Dead reckoning will be defined as any process to deduce the approximate current position of an object based on past information. This is done in an MOG context so that during high lag and loss conditions in the network, the client can approximate an object's position more accurately. When data is late

or lost beyond what interpolation can solve, the current position of an object needs to be predicted. Without prediction, an object would only be rendered at the latest known position, causing discrepancies in simulation state and great jitter in object motion. An example of the discrepancies in simulation is illustrated in figure 6. Each player A and player B see the other player's car at an old position, and thus interpret who is winning the race differently.

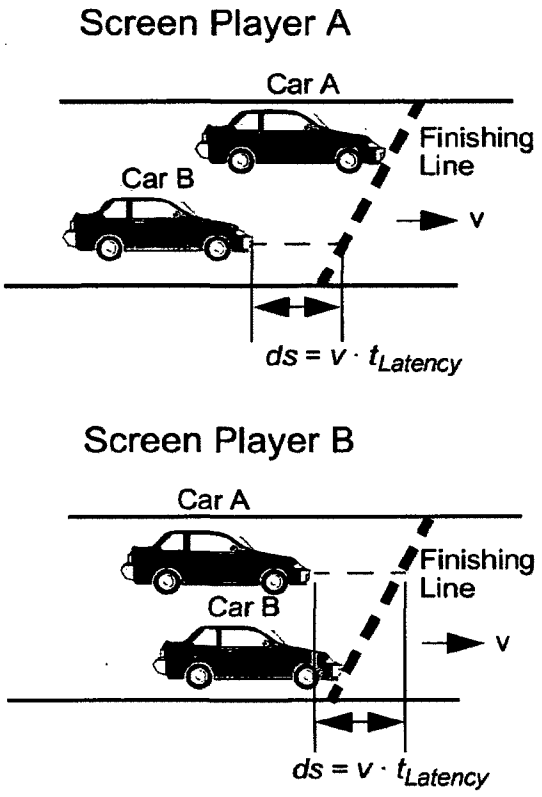


Figure 6: Inconsistency without Dead Reckoning [11].

With the use of dead reckoning, an assumption is made about the nature of game objects, such as adherence to certain forces or likelihoods. In the case of figure 6, from the point of view of player A, player B's car would be predicted to be ahead of its last received position to account for delay introduced by the network. At small amounts of lag and loss, dead reckoning does a great job concealing the fact that there was missing information about an object.

The most common and traditional method of dead reckoning involves doing a linear projection of information received from the server about this object. An IEEE standard dead reckoning formula [5] is described by the following equation:

$$P = P_0 + V_0\Delta t + \frac{1}{2}A_0\Delta t^2 \quad (1)$$

where P, Δt, P_0, and V_0 represent newly predicted, elapsed time, original position, and original velocity, respectively. This equation works well to accurately predict an object, assuming the object does not change direction. This method of prediction can become inaccurate after a time, especially for a player object, whose movement is very non-deterministic.

When the receiver finally receives the actual current position from the server, after having predicted its position up until now, there will be some deviation between the predicted position and the actual position. This error difference is known as the *export error* [1, 2, 7, 18]. Minimizing the *export error* has the effect of lowering the appearance of lag in the simulation.

In [1], the difference between time-stamped and non-time-stamped dead reckoning packet is explored. Without time-stamping a dead reckoning packet, the receiver cannot be entirely sure when the packet was generated, and as a result, discrepancies between the sender and receiver's view of the world will exist. Time synchronization between players and time-stamping dead reckoning packets means that the receiving user can execute a received packet in proper order, and in the exact same conditions as there were when generated. It is beyond dispute that time synchronization, while adding network traffic, greatly improves simulation accuracy and reduces export error [1, 14, 17, 19, 20, 30].

2.5.1 Minimize Traffic

One of the important functions of dead reckoning player position in a network game is to minimize how many packets of data need to be sent over the network. This works by allowing a client to accurately predict the current position of an object, as long as it follows an anticipative path. In fact, since the server is running the same prediction scheme as the client, the client only needs to receive data when the trajectory of an object strays from the dead reckoned path. Since both sides are running the algorithm used to predict the position of the object, the server only needs to send movement information to the clients when an object's movement strays from the predicted path by some threshold of error. For this reason, a dead reckoning algorithm that successfully improves path prediction does not only minimize the appearance of lag, but minimizes network traffic as well.

2.5.2 Predicting Objects

Some objects move in a deterministic fashion, and can actually be predicted remotely for the entire life of the object. A bullet is a good example of this. A bullet may move with a given initial velocity, then slow down due to air resistance, and perhaps start moving down based on a gravity force etc.. An object whose movement is only influenced by predictable and deterministic forces, only require network transmitted information in order to announce their initial parameters, after which time they can move based on a shared understanding of how that object behaves in the world.

2.5.3 Predicting Players

While all the prediction methods previously mentioned are capable of predicting player movement accurately to a point, more elaborate and contrived methods should be considered to handle high amounts of lag. This is because of the non-deterministic

manner in which players move in any given video game. Once there is a high amount of network delay, traditional methods of dead reckoning become too inaccurate, and *export error* starts to become too great, and players start to notice a loss of consistency [4,11,23,24,29]. Some work has been done in this regard, such as the Interest Scheme (IS) outlined by Li et. al. [7]. The IS method of dead reckoning will be explained in greater detail later in this section.

There are several ways to hide and lessen the effects of delay and loss in modern MOGs. Interpolation is a good scheme to hide a majority of the network conditions the average player faces in today's online gaming world. As well, factors such as time warping and time synchronization have major positive effects as latency compensation methods. Dead reckoning is an important and necessary method to hide the adverse effects of the network in an MOG, especially when reaching up to real world levels of delay and loss. But more needs to be done by dead reckoning to increase its useful threshold and to minimize the appearance of lag.

2.6 Threshold Adjustment and Auto Adaptive Dead Reckoning

Traditionally, dead reckoning algorithms dictate that the server should send a positional update to clients when the object strays from its predicted path by some threshold of distance. Duncan et al. propose a method called the Pre-Reckoning scheme, which sends an update just before it is anticipated that an object will pass a threshold [13]. To anticipate a threshold change, the angle between the current movement and the last movement is analyzed. If this angle is large enough, it is assumed that the threshold will be crossed very soon, and a dead reckoning packet is sent. The Pre-Reckoning algorithm effectively lowers the amount of packets transmitted over

the network. The Pre-Reckoning algorithm yields greater results when there was less variability in player movement.

Cai et al. [6] presented an auto-adaptive dead reckoning algorithm that uses dynamic threshold to control the extrapolation errors in order to achieve a reduction in transmitted packets. The results illustrate a considerable reduction in the number of transmitted packets without sacrificing accuracy in extrapolation. The threshold is adjusted based on the distance between the object and the viewer. A dynamic threshold is effective because each entity may not be fully interested in a given object, and therefore can have different amounts of prediction accuracy applied. When two entities are close to each other, each of them are likely more interested in each other's movement and state. When two entities are far away from each other, the opposite is true. A smaller threshold is used when objects are close, and a larger threshold is used when objects are far apart. A larger threshold allows the server to send less dead reckoning packets to clients for game entities for when slightly less accurate position data will not result in a loss of playability. In [8], a viewing angle is introduced to further reduce how much data needs to be sent over the network (because though an object may be close, if it is not inside the viewing cone of the player, it does not need to be dead reckoned).

While using a dynamic threshold for predicting objects does result in fewer transmitted packets over the network, it does not eliminate the requirement of highly accurate prediction schemes. Though a dynamic threshold allows more distance objects to require a lower degree of accuracy, closer object still need to be predicted accurately. Furthermore, the method outlined in [6] assumes a perspective view on the world, such that farther away objects are smaller and less visible. In a 2D video game, in which an orthographic view is utilized, all objects in view are of normal size, and therefore almost all of the objects are of interest to the user.

2.7 Neural Networks, Neuro-based Fuzzy and a Hybrid Model using A*

Work has been done utilizing neural networks to enhance the accuracy of dead reckoning [15, 22]. In [22], McCoy et. al. propose an approach that involves each client and server employing a bank of neural network predictors trained to predict future changes in an object's velocity. McCoy et. al. in [22] achieve a reduction to the spatial error associated with object prediction, as well as a reduction in network traffic.The approach proposed by Hakiri et. al. in [15] is based on a fuzzy inference system trained by the learning algorithm derived from the neural networks. This method was also shown to reduce network loads. While these methods are shown to improve performance of dead reckoning, we do not consider any approach that requires training and imposes extra computation on each computer prior to the launching of a game.

Delaney et. al. utilize a statistically based model to predict the path a player is likely to assume [12]. By observing players race to the same goal location multiple times, they select a path that seems to encompass the most common path used to arrive at the goal location. This path is then used to help predict the player's path in reaching the same goal location. Further work was done by running under the assumption that players will move similarly to the path finding algorithm A* [21]. Instead of having the researcher observe players and manually choose a path, this method allows A* to predict the path the player might take, and use this to improve prediction accuracy. Computing the path with A* allows a player's path to be predicted in real-time for any given path, as opposed to a manually chosen one from observing play test data. The results of the work done in [21] are outlined in figure 7. Figure 7 compares the number of packets transmitted over the network using a traditional dead reckoning method, the method including path finding, and a

method that includes a visibility calculation based on the wall objects in the scene. It was found that using a path from point A to point B calculated by A* can reduce the number of packets sent over the network.

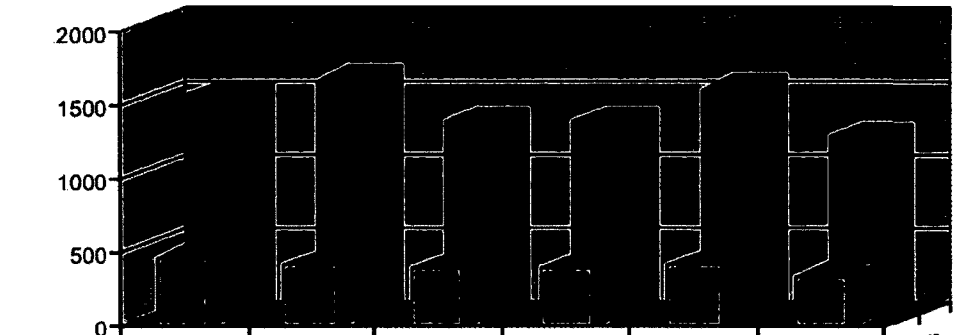
	User 1			User 2		
Seq	DR	Hybrid		DR	Hybrid	
		Visual	Path finding		Visual	Path finding
1	35	42	36	52	44	51
2	38	32	34	25	23	24
3	22	6	9	34	23	26
4	18	3	3	21	4	10
5	16	6	3	24	20	21
6	20	3	3	18	7	11

Figure 7: Path-finding in relation to Hybrid Strategy-based Models: Packets sent [21].

2.8 On the Suitability of Dead Reckoning Schemes for Games

Pantel et. al. explore and discuss the suitability of different prediction methods within the context of different types of video games [11]. What they found was that some prediction schemes are better suited to different games. They looked at essentially 5 different prediction schemes: constant velocity, constant acceleration, constant input position, constant input velocity, and constant input acceleration. Each prediction scheme was compared to each other in the context of a sports game, a racing game, and an action game. As a result of the evaluations of the different prediction methods in each type of game, Pantel et. al. demonstrated that different prediction schemes are better suited to different types of games. It was shown that predicting a constant input velocity is best suited to sports games, a constant input

acceleration is best for action games, and predicting constant acceleration is best suited to racing games [11]. The different prediction methods for action games are shown in figure 8, where prediction scheme 1/4, 2/3, 5, 6, 7 are predictions based on predicting constant velocity, acceleration, constant joystick position, constant joystick velocity, and constant joystick acceleration respectively.



	Prediction1	Prediction 2/3	Prediction 4	Prediction 5	Prediction 6	Prediction 7
100ms	471,2	422,98	398,37	397,45	414,9	344,32
200ms	1494,58	1590,54	1310,41	1307,19	1530,15	1213,24

Figure 8: Suitability of DR in action games [11].

Although it was shown in [11] that the prediction of constant joystick acceleration is best for action games, many action games do not employ joysticks, but buttons. As a result, the concept of input acceleration is not available. It was shown also in [11] that constant velocity and constant input position predictions perform relatively well. However, the evaluation was at a much lower level of lag than is considered in our research. Pantel et. al. only consider lag amounting up to 200 milliseconds (ms) [11], while we consider lag times up to 3000 ms. 3000 ms is a sufficiently high enough lag value to encompass almost all cases of adverse network conditions in a real world setting. 200 ms, while encompassing the majority of network conditions, is not sufficient if all network conditions should be considered. We find from observing the test results from our studies that even the best traditional methods of prediction result in relatively high prediction error at 3000 ms of lag.

Regardless, it was demonstrated in [11] that some methods of prediction are better suited to some games than others. In order to achieve lower levels of prediction error, a prediction scheme should be specifically designed for the game type in question.

2.9 Interest Scheme and Artificial Potential Fields

Li et. al. propose a method for predicting a player-controlled object's location they call the "Interest Scheme" [7]. It shows an increased accuracy of path prediction beyond a traditional dead reckoning models in a 2D tank game, with levels of lag up to 3000 ms. The Interest Scheme's strength lies in the way it uses the surrounding entities of a given player-controlled entity to better predict what actions the user will take. The method works on the assumption that a player's directional input is affected by his surroundings, such as items and enemy players.

In "Interest Scheme" (IS), the effect that other objects have on a player is described as a force of either attraction or repulsion. If the player would want to go towards the object (such as a health pack), then it would have an attraction force, while a repulsion force exists if the player would tend to move away from the object (such as an enemy). The intensity of these forces are affected by two factors: interest and distance. Interest refers to a degree of correlation between the player and the nearby entity, and distance is the amount of space between the player and the nearby entity. Interest can be either positive or negative, representing an attraction or repulsion respectively.

At low levels of network delay and loss, network conditions are adequate for play, and IS introduces an extra computational burden. To alleviate the extra computational burden introduced by IS during adequate network conditions, a hybrid method

is introduced into IS [18]. The hybrid method involves using traditional dead reckoning methods up until a fixed threshold of prediction time. Without a dead reckoning packet beyond this threshold, the "Interest Scheme" is then used until the reception of more data. This ensures saving on computation overhead, while losing only a negligible accuracy (at low levels of network delay, traditional dead reckoning methods are relatively accurate) [7].

The "Interest Scheme" improves prediction accuracy above traditional velocity extrapolation in a 2D tank game. Figure 9 shows the improvement the "Interest Scheme" has made over the traditional dead reckoning method.

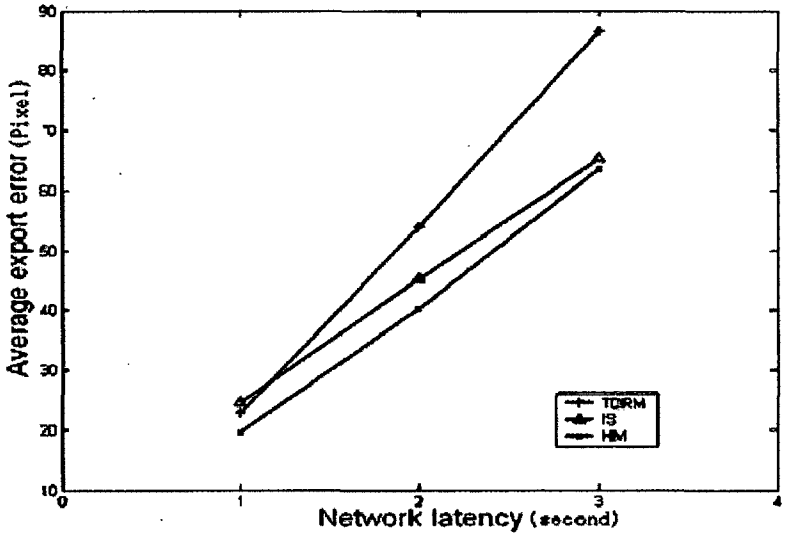


Figure 9: Interest Scheme: Average export error at different network latencies [18]

Though showing improvement in a 2D tank game, the "Interest Scheme" does not perform as well in a typical team-based action game involving bipedal, non-vehicular characters who are able to change direction quickly. This is because the "Interest Scheme" assumes a player must move straight toward or straight away from a player to attack or avoid being attacked. In a traditional action game, a player will tend to maintain a comfortable distance (often one with strategic advantage) between them

and an enemy to attack it, and take small bursts of evasive movement to avoid attack. Having said this, the "Interest Scheme" did improve prediction accuracy by taking into account the state and position of surrounding objects in the game. The "Interest Scheme" is designed to one very specific type of game. To achieve success in a general action game, a different scheme should be considered.

Shi et. al. present a similar method to improve the accuracy of dead reckoning in a 3D tank game by considering the effect that surrounding objects have on the player [26]. Different objects exhibit a different force onto the player, that can be either attraction or repulsion. Collision objects (walls) and enemy players exhibit a repulsion force, while mission objective objects (such as capture flags or locations) exhibit an attraction force. The strength of each force is determined by the distance to player. Each of these forces are given a weight, and are then used to predict the direction the player is likely to take. This method was shown to exhibit a prediction path more closely resembling the actual path the player took (see figure 10). In figure 10, the APF is the proposed potential fields method, and TDR is the traditional dead reckoning method. It was also shown to reduce the amount of packets that needed to be transmitted over the network (see figure 11). However, the method proposed considers the actions of the player based on mission objective (like capture flags or locations). Many games do not contain such motivations for the player, and therefore this method would not be suitable.

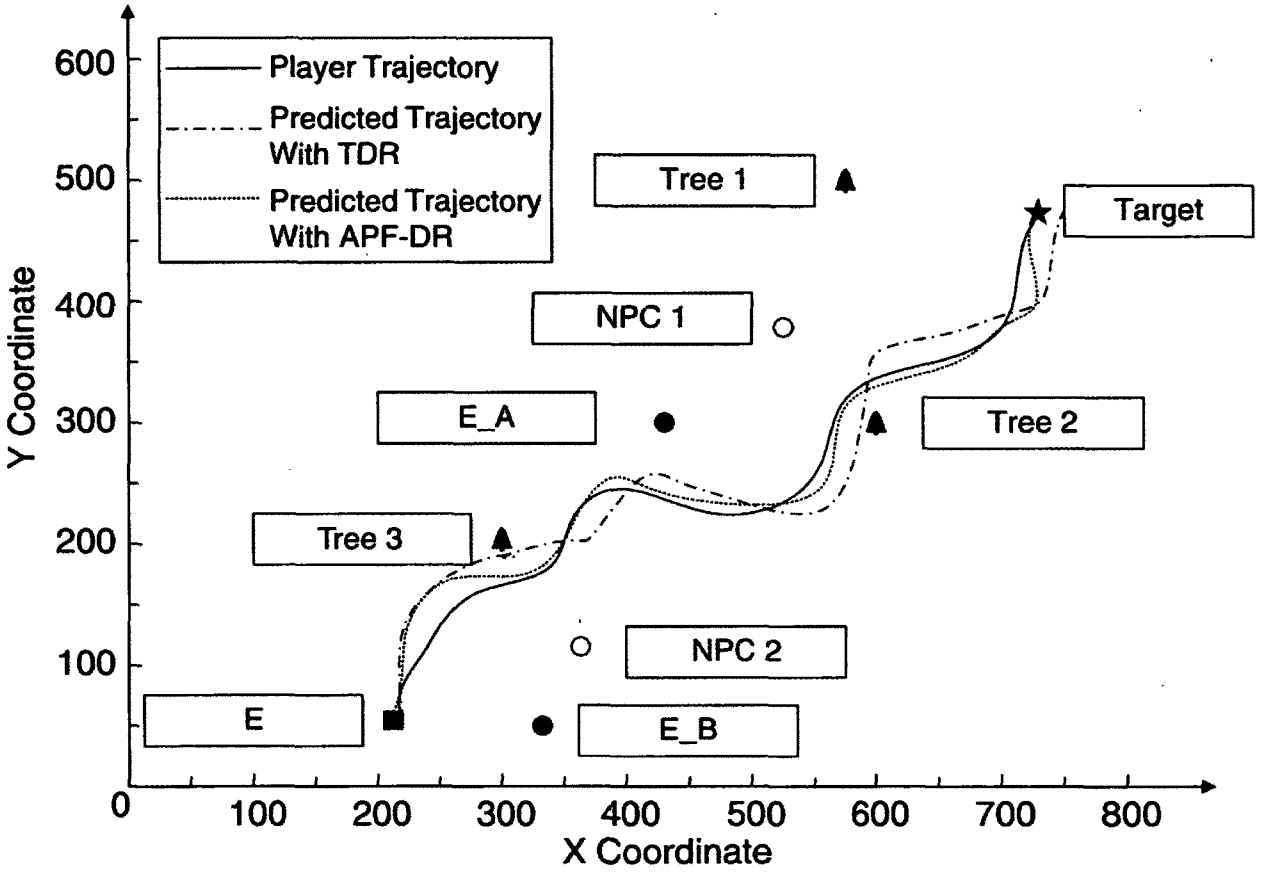


Figure 10: Artificial potential field DR: path [26]

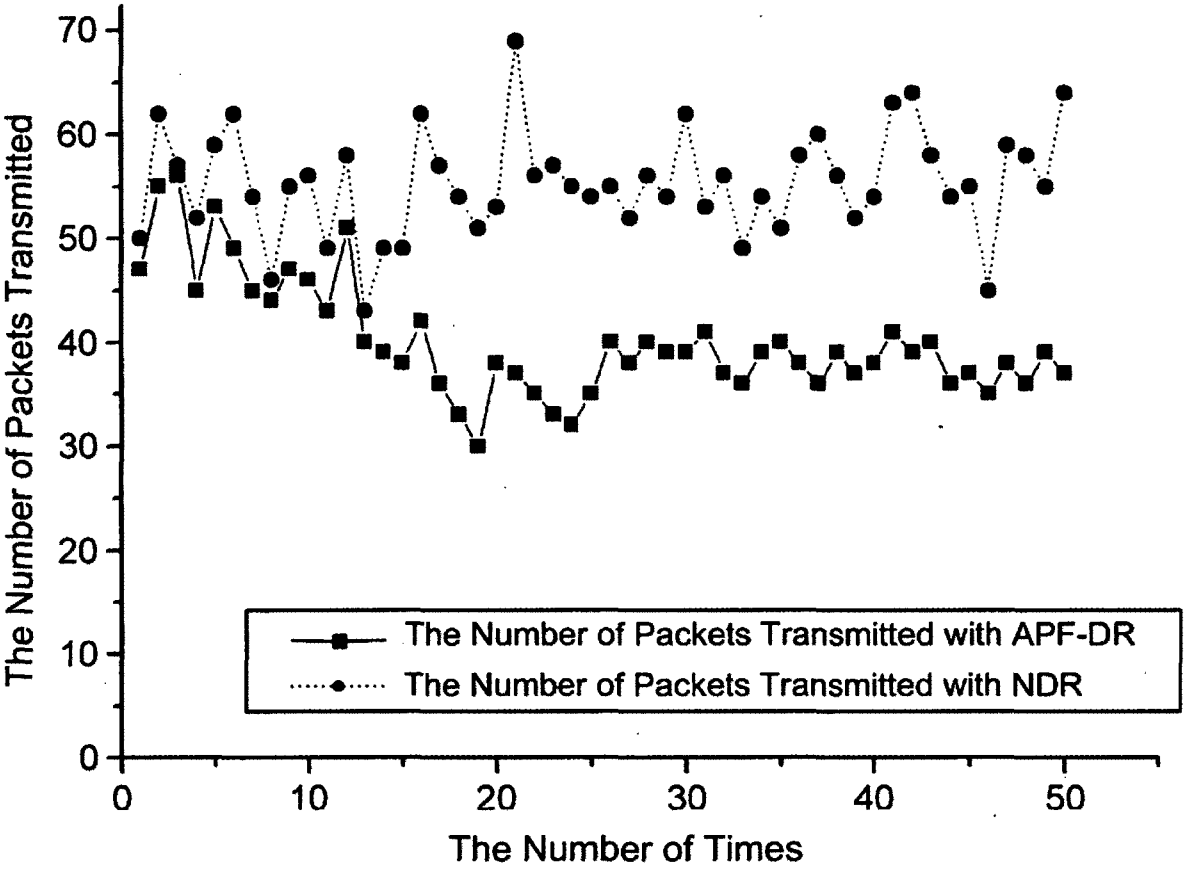


Figure 11: Artificial potential field DR: packets sent [26]

Chapter 3

Description of Algorithmic Designs

In this section, we will explain our proposed method of prediction algorithm: *Experience Knows Best* (EKB). We start by describing the different forces that compose the basic movement prediction. We then introduce a path finding algorithm into the method. We then describe our hybrid method, followed by the discussion of the parameter space. In order to clarify the algorithm description, in the rest of this chapter, we use *TPlayer* to refer to the target player for prediction. The *last known position* is the latest position data that was received over the network. The *last known position time (LKPT)* refers to the time stamp associated with the *last known position.* The *last known velocity* is the velocity associated with the LKPT.

3.1 Forces

Our approach involves predicting a player's position by predicting the potential behaviours that a player may adopt. To do so, we first collected data from having 44 experienced players play *Ethereal* over three play testing sessions. We partition the data collected from these 44 players into two groups arbitrarily. We use the first group to develop algorithm EKB and use the second group to verify our result (detail can

be found in chapter 4). From studying and analyzing the first group of collected data, we identified several behaviors that are deemed to affect the player's next movement. These behaviours each take the form of a force that is exerted on the players, affecting where they will be located next. These behaviour forces are applied at different strength levels depending on what is occurring in the game from the point of view of the player at hand. These forces are based on the positions and states of other objects in the scene. Forces are applied as either an attraction or repulsion force towards or away from a given position in space. The strength of these forces depends on several factors such as the distance to the object and the strength or weakness of the player. We first separate a player's behaviour into two categories: *in battle* and *out of battle*. We also say a player is in *in battle* state or *out of battle* state. We then exert different forces based on a player's current state. The following are the forces employed in our work: the *follow* force, the *bravery* force, and the *align* force.

Each force takes into account other players in the game world in order to determine direction and magnitude. They do so only if a given player is within a specified static distance threshold. In our current experiments, we set this distance to the size of the screen. Any player outside of such a region of interest is not considered in the computing of a force.

3.1.1 Follow

The follow force arises from our observation that a player tends to move towards and group up with friendly players (e.g., other teammates). It is computed by taking the average position of all friendly players within a specified radius, and having the player move towards that location. Furthermore, the speed of differentiation of this force does not depend on the distance of other players but is instead always set to the maximum speed of the *TPlayer*. From our observations, the follow force is the most important force to exert on a player. This is because, in multiplayer online games, a

player's actions generally proceed from a team-based strategy.

$$\vec{E}_f = \frac{\sum_{i=1}^{i=n} \vec{P}_f}{n} \tag{2}$$

$$\vec{F}_{follow} = \frac{\vec{E}_f - \vec{C}}{|\vec{E}_f - \vec{C}|} \times S \tag{3}$$

Equation 2 calculates the averaged position of all friendly (f) players within a given threshold, where $\vec{P}_f$ is the position of a friend, and n is the number of players of that type within the predefined region of interest (ROI). Equation 3 represents the force from the *TPlayer* current position $\vec{C}$ to the average position of all friendly players. S is the maximum speed of the *TPlayer*. The follow force is illustrated in Figure 12.

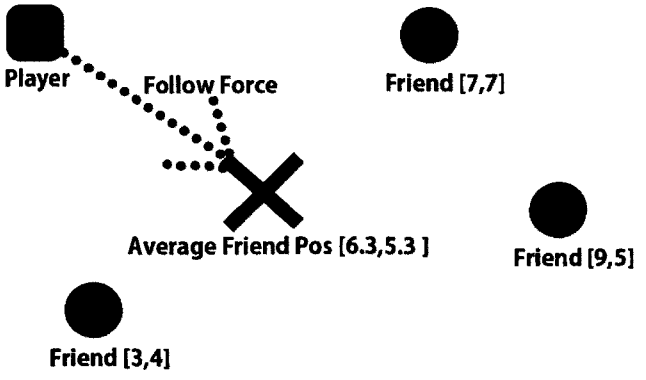


Figure 12: Follow force

To gather friendly and enemy player data, we impose a maximum distance that a player can be before it is not considered in any calculations. The game window is 1280 pixels wide and 960 pixels tall, and so the maximum distance for considering players is 1280 pixels in the x, and 960 in the y.

3.1.2 Align

The align force arises from a player's tendency to travel in a group along with friendly players. The align force takes into account all friendly players' velocities within a specified radius. The closer a friendly player is, the more weight it carries in affecting the direction of the align force. The align force's magnitude is affected by how many friendly players are within the predefined ROI, and how close they actually are.

$$D_f = \begin{cases} D_{fMAX} & if \quad D_f \geq D_{fMAX} \\ D_{fMIN} & if \quad D_f \leq D_{fMIN} \end{cases}$$

$$\vec{F}_{align} = \sum_{i=1}^{i=n} (\vec{V}_f \times (1 - \frac{D_f - D_{fMIN}}{D_{fMAX} - D_{fMIN}})) \tag{4}$$

$$|\vec{F}_{align}| \leq S$$

Equation 4 outlines the align force. D_f, $\vec{V}_f$, D_{fMIN}, D_{fMAX} represent the distance to the friendly player, the velocity of the friendly player, the minimum distance to consider for friendly players, and the maximum distance to consider for friendly players respectively. D_{fMIN}, D_{fMAX} are each a predefined threshold. $D_{fMIN} = 60$ pixels , because a player's diameter is approximately this amount, and a player within this distance is likely not be considered differently by the *TPlayer*. $D_{fMAX} = 1000$ pixels because the friendly player in question is certainly visible at this distance. Though a player may be visible up to 1600 pixels (the diagonal distance of the screen space), an object outside of 1000 pixels is unlikely to affect the alignment of the *TPlayer*.

3.1.3 Bravery

The bravery force arises from the observed behaviour of a player's tendency to fall back when outnumbered by the enemy and the tendency to advance on the enemy while winning. To obtain the bravery force, the total strength of all nearby friends and the total strength of all nearby enemies is calculated. The strength of each team is calculated by adding up the health and ammunition (ammo) values of each player. The relative strength of the friendly army versus the enemy army determines the direction of the resulting force. If the friendly army is stronger, the bravery force is positive, namely, towards the enemy. Otherwise it is negative, consequently, away from the enemy forces. The higher the magnitude of the force, the farther the *TPlayer* will move away or towards the enemy.

$$\vec{E}_e = \frac{\sum_{i=1}^{i=n} \vec{P}_e}{n} \tag{5}$$

$$I_{f,e,c} = \frac{H_{f,e,c}}{MH_{f,e,c}} + \frac{A_{f,e,c}}{MA_{f,e,c}} \tag{6}$$

$$Z_{f,e} = \sum_{i=1}^{i=n} I_{f,e} \tag{7}$$

$$\vec{V}_{bravery} = (\vec{E}_f + (\frac{(\vec{E}_e - \vec{E}_f)}{|(\vec{E}_e - \vec{E}_f)|} \times (u \times \frac{(kZ_f + I_c) - Z_e}{max((kZ_f + I_c), Z_e)}))) - \vec{C} \tag{8}$$

$$\vec{F}_{bravery} = \frac{\vec{V}_b}{|\vec{V}_b|} \times S \tag{9}$$

In equation 6, $I_{f,e,c}$ is the influence of a friendly, enemy or the current player in terms of its strength. H, MH A and MA are the health, maximum health, ammo value and maximum ammo of the player respectively. This influence value is then

combined into either the enemy or friendly team influence value, depending on which team the *TPlayer* is, represented by $Z_{f,e}$ in equation 7. $Z_{f,e}$ is made up of all the players on the given team that are within a predefined threshold. $\vec{V}_{bravery}$ in equation 8 is the direction vector for which is used for the bravery force. u is a coefficient that is the maximum distance that a player will run away or towards the enemy, and k is a coefficient that modifies the strength of the friendly influence. This is to model the fact that a player will consider their own strength over their allies' strength in a combat situation. The *TPlayer* is either moving towards or away from the enemy, in relation to the averaged friend position. This is illustrated in figure 13. Equation 9 is the actual force used for bravery.

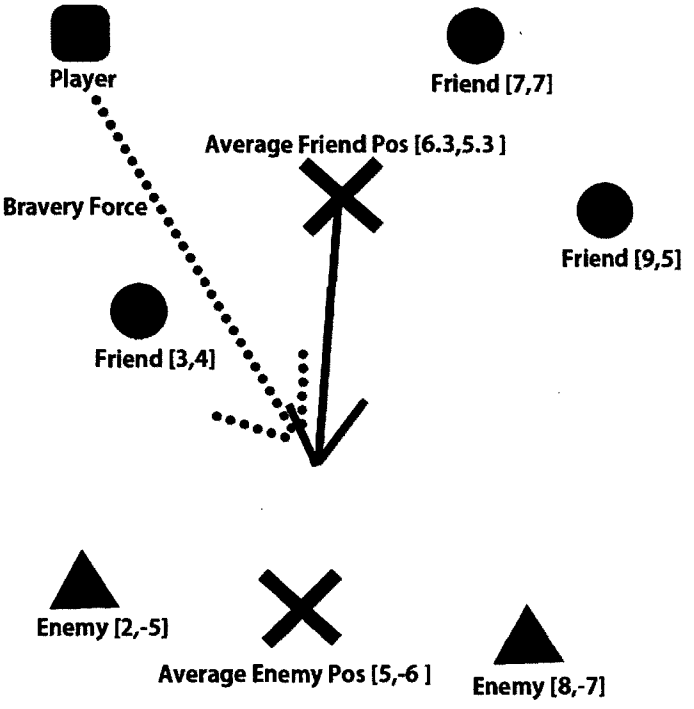


Figure 13: Bravery force when friendly team is stronger

k is a coefficient used in equation 8 to determine how much smaller we scale the strength of the friendly team. This is done for two reasons. First, a player will tend to consider their own strength when in combat, and won't adjust their behaviour if

their friends are strong or weak. Second, since enemy players are more often far away than friendly players, they often fall outside the maximum distance for considering players, many enemy players that the player is aware of are not considered in the strength calculations because they are simply too far away. This is easily adjusted for using k. In our studies, we set $k = 0.4$. We found through trial and error that this yielded the best results, and behaviour that best reflected reality. u is the maximum distance a player will aim to run towards or away from the friend epicenter depending on how strong each team is. We set u to 200 pixels. We chose this because this is what resulted in the best prediction accuracy from a trial and error approach.

3.2 Combination of Forces

As previously mentioned, in order to simplify how the forces interact with each other, we separate player behaviour into two categories: *in battle* behaviours and *out of battle* behaviours. When the *TPlayer* is *in battle*, the *TPlayer's* position is calculated by combining the follow and the bravery forces. When *out of battle*, the *TPlayer* position is calculated by combining the follow and align forces.

Whether the *TPlayer* is in battle or not is decided according to the result of a simple distance check to the closest enemy (equation 12). If the distance between the *TPlayer* and its closest enemy is smaller than a predefined threshold W, then the *TPlayer* is said to be *in battle* (the value of W is explained at the end of this section). Equation 10 calculates the distance from the current player position to a given enemy player.

$$D_e = |\vec{P}_e - \vec{C}| \tag{10}$$

$$closestEnemyDist = min\{D_{e1}, D_{e2}....D_{en}\} \tag{11}$$

$$InBattle = \begin{cases} true & if \quad closestEnemyDist \leq W \\ \\ false & otherwise \end{cases} \tag{12}$$

If the *TPlayer* is in battle, it will use the follow force and the fear force to predict a position. If the player is out of battle, it will use the follow force and the align force to predict a position. Algorithm 1 shows how the forces are handled. Coefficient q and r are static values that are less than 1, greater than 0 (the values of q and r are explained at the end of this section). They dictate how much of each force is used.

Algorithm 1 Apply Forces

1: **if** the player is in battle **then**
2: $\quad \vec{F}_r = (\vec{F}_{follow} \times q) + (\vec{F}_{bravery} \times (1 - q))$
3: **else**
4: $\quad \vec{F}_r = (\vec{F}_{follow} \times r) + (\vec{F}_{align} \times (1 - r))$
5: **end if**

To differentiate between a player *in battle* or *out of battle*, we use a static distance threshold W to the closest enemy player. If the distance to the closest enemy is less than this threshold, then the player is said to be *in battle*. For our results, we used a threshold of $W = 800$. From experimentation, it appears this value accurately represents when a player was engaged in combat or not in the context of our testing environment game.

When combining all the forces together, we use q and r to determine how much of each force is used to create the resultant movement force. We use $q = 0.5$ and $r = 0.6$. This is the result of trial and error test to see what works best.

3.3 Algorithm Enhancement: Smooth Transition

In algorithm 1, $\vec{F_r}$ is the final resultant force that is used to predict the player's position. After extensive experiments, we notice that if the player's velocity is immediately set to $\vec{F_r}$, the result is most often an inaccurate account of the player's movement. This is due to ignoring the player's *last known velocity.* The *last known velocity* is the last velocity information received in the last received position packet. We use position packet synonymously with dead reckoning packet, which refers to the information received from the server describing player information. To alleviate this, we perform a smooth transition of the player from the *last known velocity* to the one determined by the combined forces (see Equation 13). $\vec{F}j$ is the force that should be used as the velocity of the player and j is the number of updates that has occurred since the last known velocity $\vec{V_0}$. The more updates that have passed since the last known velocity was received (ie. the larger the value of j), the larger the value of $\vec{F_r}$ is and the smaller the value $\vec{V_0}$ is. Once j reaches the size of m, then we exclusively use $\vec{F_r}$ to determine $\vec{F_j}$.

$$\vec{F_j} = \begin{cases} \frac{m-j}{m}\vec{V_0} + \frac{j}{m}(\vec{F_r}) & if \quad j \leq m \\ \vec{F_r} & otherwise \end{cases} \tag{13}$$

$$m = \begin{cases} l \min\{closestEnemyDist, closestFriendDist\} & if \quad m \leq R \\ R & otherwise \end{cases} \tag{14}$$

The calculations for m are shown in equation 14. m is proportional to the distance between the player and the closer of the closest friendly or enemy player. This is due to the following observation: a player is more likely to react to a player that is close

to it, and is less likely to continue at the current velocity. l is a coefficient used to modify m so that it is in the right scale (we found that $l = 0.1$ works best). R is the upper bounds on m.

We used $l = 0.1$ because it allows for the best transition from the old velocity $\vec{V_0}$ to the new velocity calculated by EKB calculations. We used $R = 60$. This value represents a maximum allowable time to still consider the old velocity $\vec{V_0}$ in the calculations. An R of 60 corresponds to 1 second.

3.4 Algorithm Enhancement: A*

We employ the A* path finding algorithm [25] to further improve the accuracy of the above-mentioned prediction method. The use of the A* algorithm proceeds from observing a player's tendency to avoid walls and to find an efficient path through space to a desired location. This ensures that the *TPlayer's* predicted path avoids wall objects and looks more realistic. The implementation of the A* path finding algorithm in our scheme involves modification to the *Follow* and *Bravery* force, whereas the Align force remains the same. $\vec{F}_{follow}$ and $\vec{F}_{bravery}$ now point towards a desired position that is along the A* path, rather than pointing towards only the final destination. For $\vec{F}_{follow}$, this desired location is the average position of all nearby friendly players. For $\vec{F}_{bravery}$, this desired location is $\vec{V}_{bravery} + \vec{C}$. A shortest path to this desired location avoiding all obstacles is then calculated. Algorithm 2 outlines how A* is incorporated into our prediction scheme.

Algorithm 2 Improved EKB using A*

1: **if** the desired A* end destination location has changed since the last update **then**
2: recalculate an A*path to the desired location for $\vec{F}_{follow}$ and $\vec{F}_{bravery}$.
3: **end if**
4: **if** the next A* node in the path is reached **then**
5: increment the desired node location to the next node in the A* path.
6: **end if**
7: Use the next desired node location to calculate the vectors for $\vec{F}_{follow}$ and $\vec{F}_{bravery}$.
8: **if** the player is in battle **then**
9: $\vec{F}_r = (\vec{F}_{follow} \times q) + (\vec{F}_{bravery} \times (1 - q))$
10: **else**
11: $\vec{F}_r = (\vec{F}_{follow} \times r) + (\vec{F}_{align} \times (1 - r))$
12: **end if**

3.5 Algorithm Enhancements: Hybrid Approach

In order to further improve the prediction accuracy and reduce the number of packets transmitted across the network, we adopt a hybrid scheme. The hybrid method is as follows:

- below x ms of lag, EKB is always used. This is because according to our experiment results EKB performs best under this lag range (see figure 24 in chapter 4, section 4.3.2).
- if a player has been moving in the same direction for less than or equal to the same amount of time as the network delay, then we assume the player will continue to move in this direction and thus we use TDM for the prediction,
- otherwise, EKB is used.

The hybrid method is shown in algorithm 3. We use $x = 350$, because below this threshold the EKB method performs significantly better than TDM and IS. The amount of time between the *current time* and the *LKPT* is how long we have not

received a position packet from the server, and is how long we have not known the true position of the *TPlayer*. We call this Q time. Equation 15 describes its calculation.

$$Q = currentTime - LKPT \tag{15}$$

$LKPT$ is the time stamp of the last received positional information received with regards to the *TPlayer*. We use Q as the amount of time before the $LKPT$ to check to see if the player has changed direction. If the *TPlayer* has not changed direction since $LKPT - Q$, then it is assumed that the player will continue in this direction, and the TDM is used.

Algorithm 3 Hybrid Approach

1: **if** the amount of time since the last position packet was received (Q) is less than or equal to x ms **then**
2: predict the player's position using the EKB
3: **else if** the player has not changed direction since $LKPT - Q$ time **then**
4: predict the player's position using the TDM
5: **else**
6: predict the player's position using the EKB
7: **end if**

To determine whether a player has changed direction, we use the method outlined in algorithm 4. j is the amount of time that has passed since the last known position was received.

h is a threshold angle used to determine the angle above which the change in velocity needs to be before a change in direction is registered. We use $h = 40$ degrees, so that if a player changes direction by 45 degrees, it is detected as a change in direction. x is the minimum threshold of lag below which the EKB method is always used. We set $x = 350$ because after experimenting with different values, we determined this to result in the best performance.

Algorithm 4 Check Player Direction Change

1: initialize *count* to one
2: initialize *directionChange* to $false$
3: **while** *count* is less than j **do**
4: **if** the angle between the velocity at $t =$ *last received* and the *velocity at* $t =$ *last received minus count*'s is above a threshold angle h. **then**
5: set *directionChange* to $true$.
6: **end if**
7: increment *count* by dt.
8: **end while**

3.6 Discussion of the Parameter Space

In table 2, we list values and brief descriptions of all coefficients and parameters used in this chapter.

Table 2: Parameter Space

Parameter	Value	Description
W	800	static distance threshold to differentiate between a player *in battle* or *out of battle*
k	0.4	coefficient used in equation 8 to determine the how much smaller we scale the strength of the friendly team
u	200	maximum distance a player will aim to run towards or away from the friend epicenter depending on how strong each team is.
l	0.1	coefficient used to modify m so that it is in the right scale.
R	60	bound on m that ensures that there is always some transition that occurs from the old velocity $\vec{V_0}$ to the new velocity.
q	0.5	how much of each force (follow and align) is used to create the resultant movement force.
r	0.6	how much of each force (follow and bravery) is used to create the resultant movement force.
h	40	threshold angle used to determine the angle above which the change in velocity needs to be before a change in direction is registered.
x	350	minimum threshold of lag below which the EKB method is always used.

Chapter 4

Experimental Procedures, Results and Comparative Evaluation

4.1 Description of Experimental Procedures

4.1.1 The Test Environment

As mentioned earlier, we adopt the following methodology to conduct the proposed research:

1. collect data (all player input) from play testing sessions;
2. analyze the collected data;
3. extract play patterns from the collected data;
4. construct a new prediction algorithm;
5. improve the basic algorithm;
6. implement our algorithm and two other existing prediction algorithms;
7. compare the results and draw conclusion.

In order to collect players' input, replay the collected data (for data analysis and pattern extraction), and conduct empirical and comparative evaluations, we implemented an interactive distributed test environment. This test environment takes the form of a multiplayer online game named *Ethereal.* In this subsection, we introduce the major functionality of this test environment.

Playing the Game

We designed the test environment so that player activities would be similar to those that would be seen in any traditional action game. Players, each on a separate computer, can make a connection to the server machine to join the distributed interactive system, where-in players can interact with each other in real-time. Once a connection is made to the server, each player chooses a team to join, and can then start playing the game. In the game we implemented, players assume the role of a single entity, that can move in all directions on a 2D plane freely. They can also aim with the mouse to shoot. Gameplay is such that there are two teams, both pitted against each other in competition. Points are awarded to a team when a player from that team kills a player from the opposing team. A team wins when it reaches a certain amount of points before the other team does. A screenshot from the game can be seen in figure 14.

Recording a Session

To ensure adequate observations and depth of analysis, we implemented a replay system to record all events and inputs from play sessions and to conduct analysis based on this replay data. This allows us to do multiple predictions per update of the simulation on all players in the game without worrying about the analysis and collection of our results slowing down the simulation for the player during a play test. The replay system is a server side implementation that is designed to accurately

Figure 14: Screenshot of typical play in "Ethereal"

playback all game actions and events exactly as they were recorded. The replay system records all relevant information from a play session as it happens. It records player input, births time (player spawn time), death time, and state snapshots. Player input records consists of a time stamp, directional information, and mouse input at every update of the simulation. A player snapshot is taken every 5 seconds to ensure nothing becomes desynchronized. A player snapshot consists of all player states that are important to gameplay: player position, velocity, health and ammo.

Playback of a Recording

After recording all data necessary, the replay can then be played back. This is done by first spawning a given player at its recorded birth time. Then, as the simulation progresses, the time stamp of the input record that is next to execute is checked. If it is time to execute this input, it gets executed. The same is done for the snapshot

records. Once the death time of the player is reached, the player is killed. In this way, all recorded data is played back, such that an entire game play session can be observed after it has been recorded.

Our test environment also has the capability to run and evaluate different dead reckoning schemes, and to measure and record different metrics associated with the scheme. At each update of the playback simulation, the replay system can make a check to any number of criteria to see if a simulated dead reckoning prediction should be made. To accomplish measuring different dead reckoning schemes, the replay system first must know the positional history of all objects. It does this by recording player positional data at each update of the simulation. When a dead reckoning prediction is called upon, the replay system looks back in time an amount equal to the amount of prediction time requested to find the simulated *last known position.* It can then predict forward from this position to the current time, as well can measure how accurate this prediction is (by comparing it to the current position). One strength of our test environment is that it can make multiple predictions with varying amounts of prediction time in a single simulation update. This allows us to make exhaustive tests at any given time in the playback.

Since the replay system makes only simulated dead reckoning predictions, we can easily compare the prediction against the actual player data. We can see how far away the predicted position is from the actual position. We can also measure the number of packets that would be sent by the current dead reckoning scheme. This is accomplished by simulating how a server would normally schedule sending dead reckoning packets. More specifically, the replay system checks the accuracy of the dead reckoning prediction made at each update. Using the simulated last sent packet time to determine how long prediction is required, the replay system can then check to see how far away the predicted position is from the actual position. If this distance is beyond a certain threshold, a counter is incremented (counting sent packets), and

the last sent packet time is updated.

4.1.2 The Testing Methodology

Play Testing

To obtain our results, we conducted 3 play test sessions, each run for between 20-30 minutes, and with a total of 44 players (see details in table 3). The first play test ran for 20 minutes and had 10 participating players. The second play test ran for 30 minutes and had 19 participants. The third and last play ran for 30 minutes and had 15 participants. All participants were avid to professional video game players. All participants were male, between the ages of 17-30, and were either undergraduate students, graduate students, or graduates. We used just under of half of player data as the testing set, and the rest for evaluation of our 3 algorithms. More specifically, we used the second play test (19 participants for 30 minutes) and as the testing set, and the remaining session 1 and 3 (25 participants) for our evaluations. We chose this many players for our studies because it is an authentic representative set of the actual players that would be seen in an online game playing community. Furthermore, this size of sample follows suit with the sample sizes of similar works [18, 21, 26, 29]. In our tests, we had all the players play in our computer room with 20 independent computers connected by a local-area network (LAN). Teams within the game were organized by the configuration of the room, such that players sitting close to each other in real-life were on the same team, allowing them to discuss strategy.

Table 3: Play Test Outline

	Duration	# of Players	Usage
Session 1	20	10	evaluation
Session 2	30	19	algorithm development
Session 3	30	15	evaluation

Performance Testing and Metrics

We experimented with different prediction methods and analyzing them with the replay system. We could simulate any amount of delay into the simulation, and test the predicted position against the actual position of any player. At the time of making a prediction, we can then measure different metrics. We measured the *Average Export Error* (AEE), the number of hits, and the number of packets sent. We use these metrics to measure our EKB method, as well as 2 other dead reckoning schemes: the TDM [5] and the IS [7,18]. We feel that these metrics accurately test and contrast the accuracy of the dead reckoning schemes.

We use the metrics of *Average Export Error (AEE), number of hits (*hits*)* and *packets sent* to measure our algorithm and compare it with other dead reckoning schemes. AEE is the average distance from the predicted position and the actual position of the player for all predictions made. To calculate it, we take the median of all export errors at fixed intervals of time (e.g., $300ms$, $600ms$, etc.) to determine the general accuracy of an algorithm. The calculation of AEE is shown in equation 16. $\vec{P}_t$ and $\vec{E}_t$ is the actual position of the player and the predicted position of the player respectively at time t. n is the total number of predictions made throughout the lifetime of all replay data. The AEE is a measurement of how similar the estimated behaviour of a player is related to the true and actual movement of a player. The AEE is the best metric in determining the accuracy of any given prediction method.

$$AEE = \frac{\sum_{i=0}^{i=n} |\vec{P}_t - \vec{E}_t|}{n} \tag{16}$$

The next metric we adopt is hits. A *hit* is defined as when the predicted location is within a specific threshold (measured in pixels) of the actual position. It is taken at specific points in time. This metric measures how many times the prediction scheme has predicted a position correctly. Whenever the position of the player is accurately

predicted as a hit, this means that the play experience is improved for the player because it means that the estimated player position will not have to be corrected to the actual player position.

We also measure the number of packets that need to be transmitted over the network during each session of play. This is done by assuming that a packet only need be sent when the predicted position of the player is more than a certain static threshold h distance away from the actual position of the player. We use a threshold of $h = 45$, around half the width of a player. Measuring the number of packets sent is done because it is desirable to have less network traffic if it is possible. Network bandwidth is often a bottleneck of performance for DISs. When there is less packets that need to be sent per object, the game can then replicate more objects over the network. We present packets sent as a single integer, representing the total number of packets that have been sent throughout all 3 of the play test sessions that were conducted.

To ensure we test our prediction scheme against other prediction schemes in every situation of play, we make a prediction and measure its accuracy as often as possible. During playback of the recorded replay data, instead of simulating realistic lag onto the players, we test our prediction scheme at every update of the simulation. Furthermore, at each tick of the simulation, we test lag at varying degrees of network delay. At each tick, we simulate lag at ten different levels of lag, 300 ms apart. We test prediction at 300 ms of delay, 600 ms of delay, and so on up to 3000 ms of delay. We do 10 predictions per update of the simulation for each and every player in the game. We used non-random lag intervals to ensure that analysis and comparison is done absolutely fairly, such that nothing is left to chance and to ensure ease of interpreting the results. Combining testing prediction at every update of the simulation with an all-encompassing approach to lag simulation allows that every possible game scenario is tested at every level of lag, and with each dead reckoning scheme equally.

After having described our test environment and approach to testing and evaluating the gathered data, we will now report on our experimental results and findings, analyze the experimental results in detail. We will compare the results based on the 3 metrics (AEE, hits and packets transmitted). We will then give justification and explain the results.

4.2 Experimental Results and Their Analysis

4.2.1 Results and their Analysis

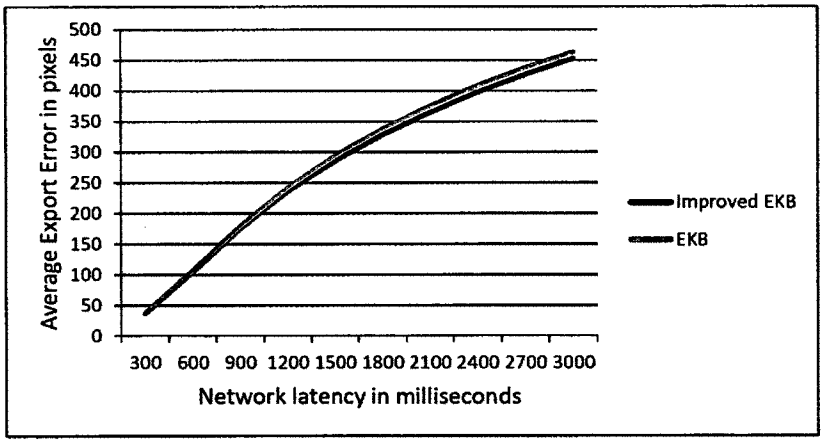
In this section, the results of our experiments regarding the 3 versions of EKB will be described. It is organized by discussions in regard to each metric. This discussion starts with AEE, followed by number of hits, then packets sent. The 3 versions of our algorithm will be labelled as follows: EKB, Improved EKB (with A* included), and Hybrid Improved EKB (with A* and the hybrid method included).

Average Export Error

The *Average Export Error (AEE)* is the discrepancy between the actual location of the player and the predicted location of the player (in pixels). Figure 15 and table 4 compare the EKB method against the improved EKB method (with A*). These figures show what effect introducing the A* algorithm has on AEE. For clarity, part a) and part b) of Figure 15 display the same data but as a line graph and a bar graph respectively. Given that the AEE varies significantly across the different levels of lag considered, table 4 displays AEE at high levels of lag above 1500ms. While the introduction of the path finding algorithm A* does not improve prediction accuracy drastically, it does result in more realistic predicted motion of the player, as a player controlled by a human user will tend to avoid obstacles to achieve their objectives.

AEE increases as prediction time increases. This is because as the prediction time increases, so does the time since the last player's true position was known. The way the line flattens out over time in figure 15 a) suggests that the EKB performs better under higher latency conditions.

a)



b)

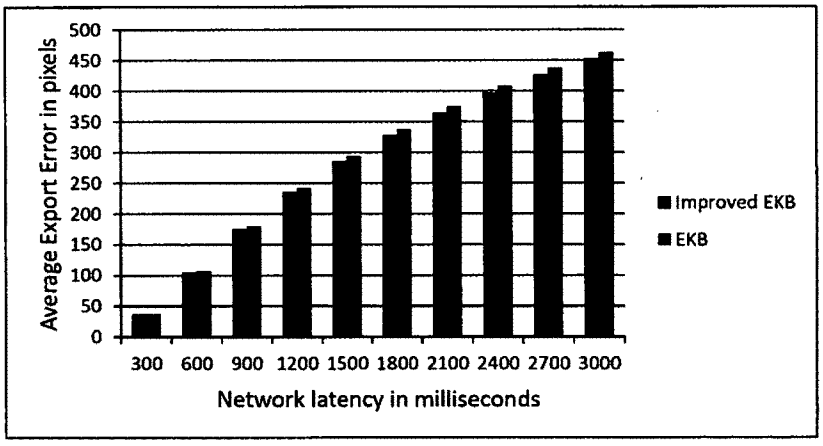


Figure 15: AEE of EKB and improved EKB

Table 4: AEE of EKB and improved EKB at High Latency

	1500ms	1800ms	2100ms	2400ms	2700ms	3000ms
EKB	293.6	337.3	374.9	408.2	437.5	463.1
improved EKB	285.7	327.8	364.5	397.2	426.4	452.5

Figure 16 shows a comparison between the improved EKB and hybrid improved EKB. The hybrid method combines moving with the TDM and moving with EKB, depending on how linear the players behaviour is observed to be in the past. We introduced the hybrid method to increase the number of hits and decrease the number of packets sent. While the hybrid method does this, it also has the effect of increasing the AEE.

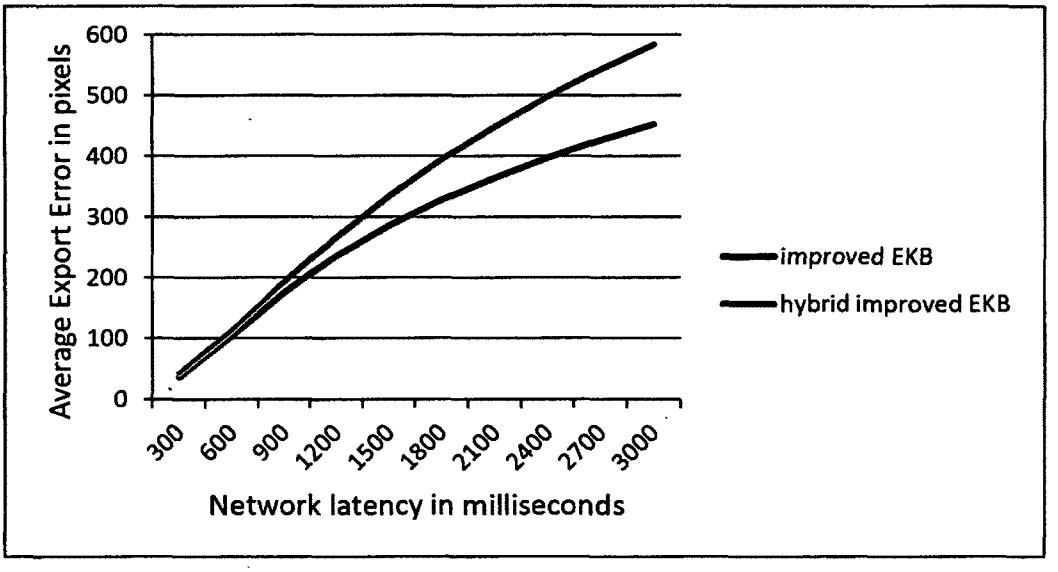


Figure 16: AEE of hybrid improved EKB

Table 5: AEE of improved EKB and hybrid improved EKB at High Latency

	1500ms	1800ms	2100ms	2400ms	2700ms	3000ms
hybrid improved EKB	334.0	394.1	448.5	498.2	543.5	584.2
improved EKB	285.7	327.8	364.5	397.2	426.4	452.5

Number of Hits

A *hit* is defined as when the predicted location is within a specific threshold (measured in pixels) of the actual position. This threshold should be small enough so that when correcting the predicted player location to the actual location, the correction is minimally visible. In our work, we use $h = 45$ pixels as the threshold, given it is less than the width of the player in our multiplayer online game. $h = 45$ is a small enough threshold that when correction is made to the player's predicted position, it will go unnoticed. We do not test against a larger value for h because it would result in a noticeable jump in position when correction is made. Furthermore, a larger value of h yields unrealistically positive performance results out of all 3 versions EKB, and would be a misrepresentation of the algorithms. We check for a *hit*at certain intervals of delay to obtain the accuracy of the dead reckoning scheme at precise moments in time. Figure 17 shows the number of hits at different levels of network latency for improved EKB and hybrid improved EKB. We consider latency values ranging from 300ms to 3000ms and report on the number of hits observed within this interval. Given that the number of hits varies significantly across our interval of latency, we break down our results into two figures: one for low latency (from 300ms to 1200ms), one from high latency (from 1500ms to 3000ms). Figures 18 and 19 show the number of hits organized into low and high amounts of latency respectively. To display the results more clearly, the same data is presented in table 6 and table 7.

The number of hits without using the hybrid method was much lower than with, and this is why we introduced the hybrid method. The fact that the hybrid method

has the effect of improving the number of total hits while at the same time lowering AEE demonstrates that although the position of the player is often accurately predicted, it does not mean that the position of the player is overall better approximated. The hybrid method improves the number of hits because it allows for frequent predictions of the player moving in a straight line, during which time it is exactly accurate in predicting the player (as long as the player is moving linearly). Without the hybrid scheme, the EKB does approximate the player relatively accurately, but does not produce as many exact predictions of player position.

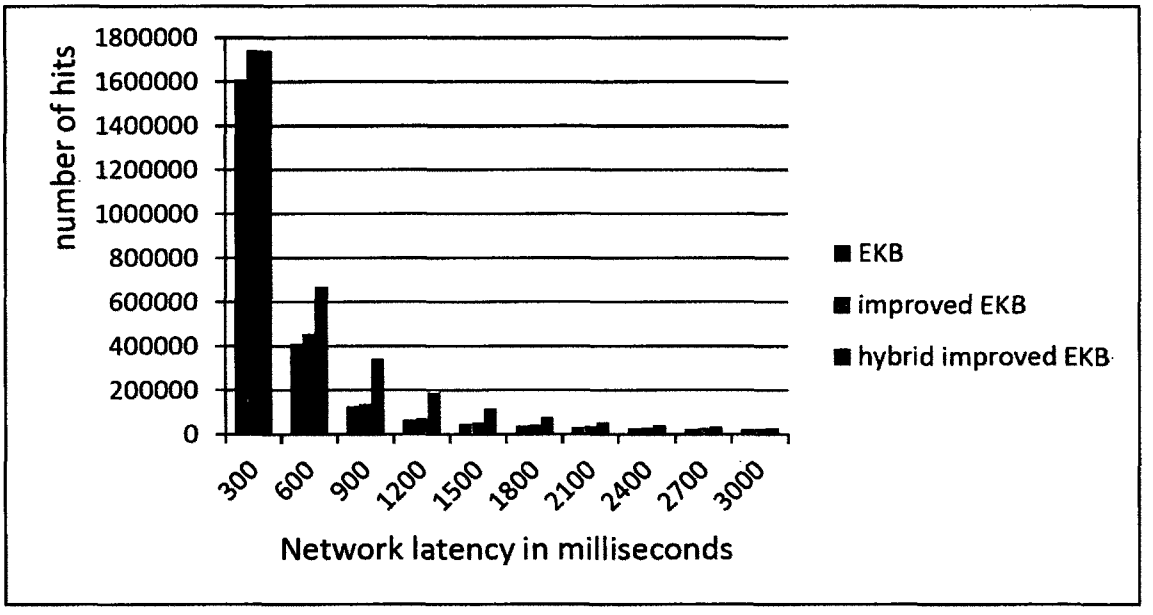


Figure 17: Number of hits, EKB at threshold $h = 45$

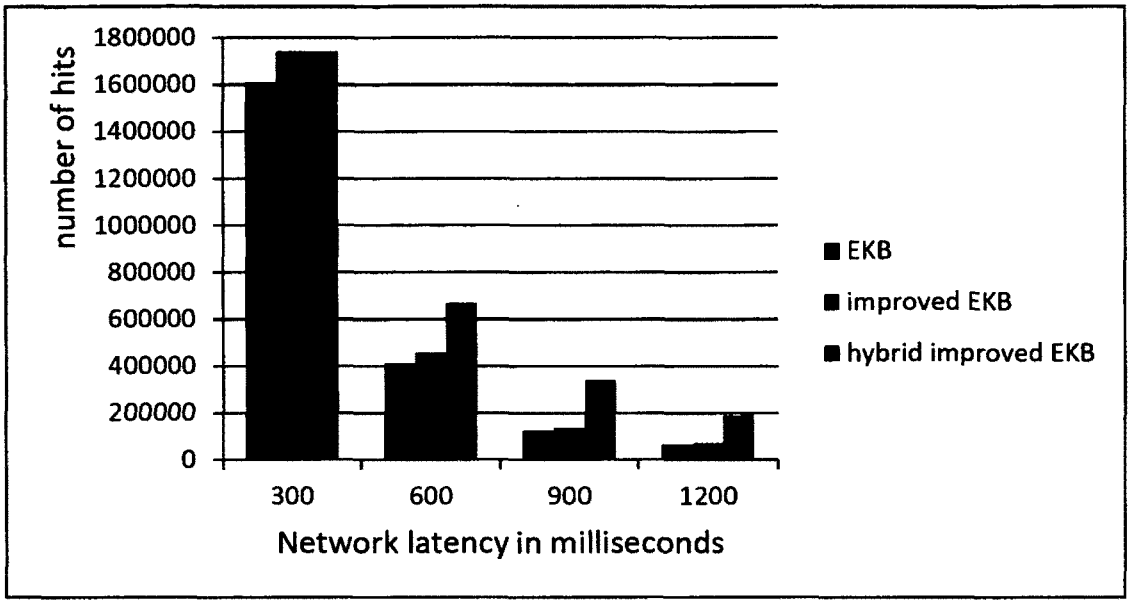


Figure 18: Number of hits, EKB at threshold $h = 45$, low latency

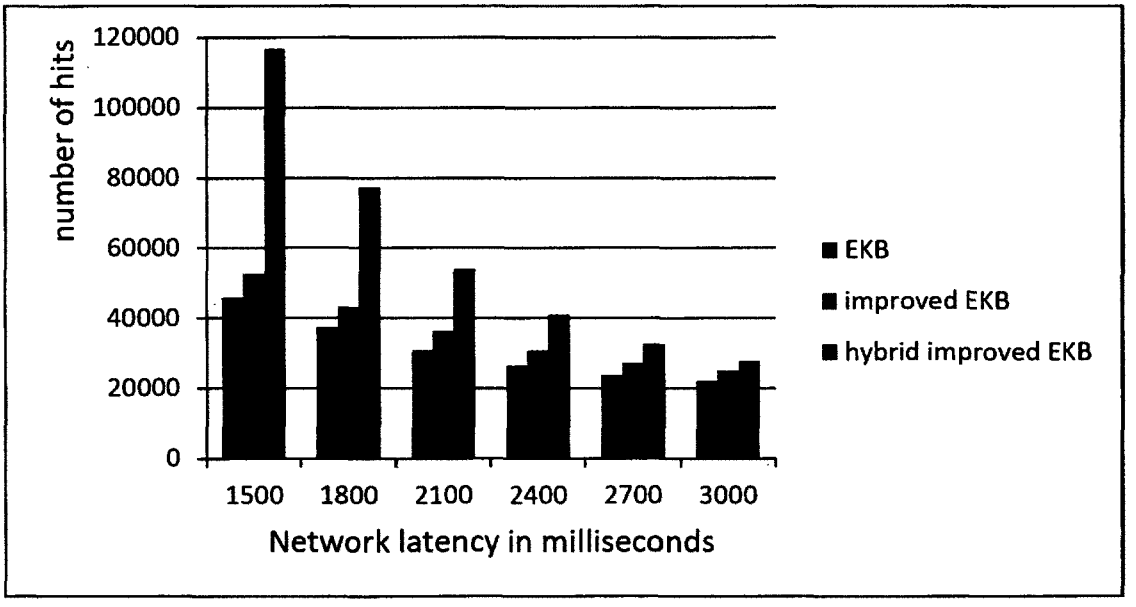


Figure 19: Number of hits, EKB at threshold $h = 45$, high latency

As can be seen clearly in figure 20, the hybrid improved EKB method improves upon the number of hits made by EKB greatly. It can also be seen that while

Table 6: Number of hits, EKB at threshold $h = 45$ with low latency

	300ms	600ms	900ms	1200ms
EKB	1610454	410739	125078	64352
improved EKB	1741841	456425	136576	72051
hybrid improved EKB	1741841	668684	341268	189671

Table 7: Number of hits, EKB at threshold $h = 45$ with high latency

	1500ms	1800ms	2100ms	2400ms	2700ms	3000ms
EKB	45859	37376	30881	26478	23743	22173
improved EKB	52670	43276	36264	30988	27295	25070
hybrid improved EKB	116821	77275	54083	41049	32684	27823

introducing A* into EKB did not provide entirely relevant improvements to AEE, it made a significant improvement to the total number of accurate predictions made (as shown in figure 15). This further provides evidence that A* allows the scheme to more accurately predict the movement of the player.

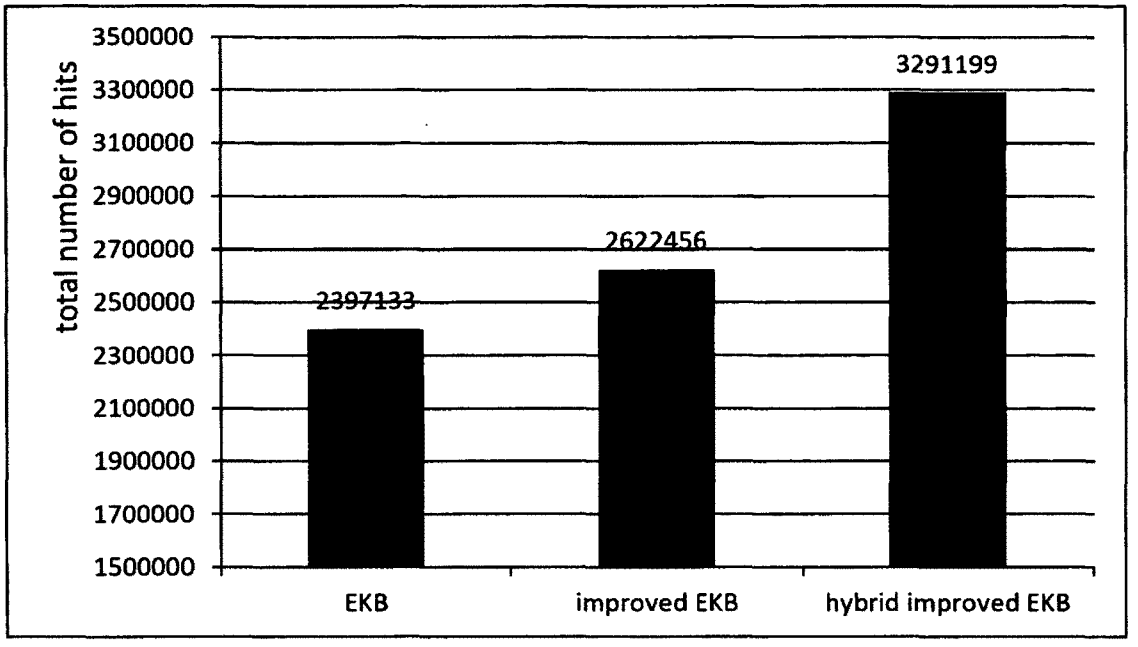


Figure 20: Total hits, EKB at threshold $h = 45$

Number of Packets Sent

We also measure the number of dead reckoning packets (or position packets) that need to be sent through the network. Dead reckoning packets are sent from the server when the predicted position of the player is more than a certain threshold distance h away from the actual position of the player. Figure 21 shows the number of packets required to be sent by EKB. The hybrid method greatly reduces the number of packets that need to be sent.

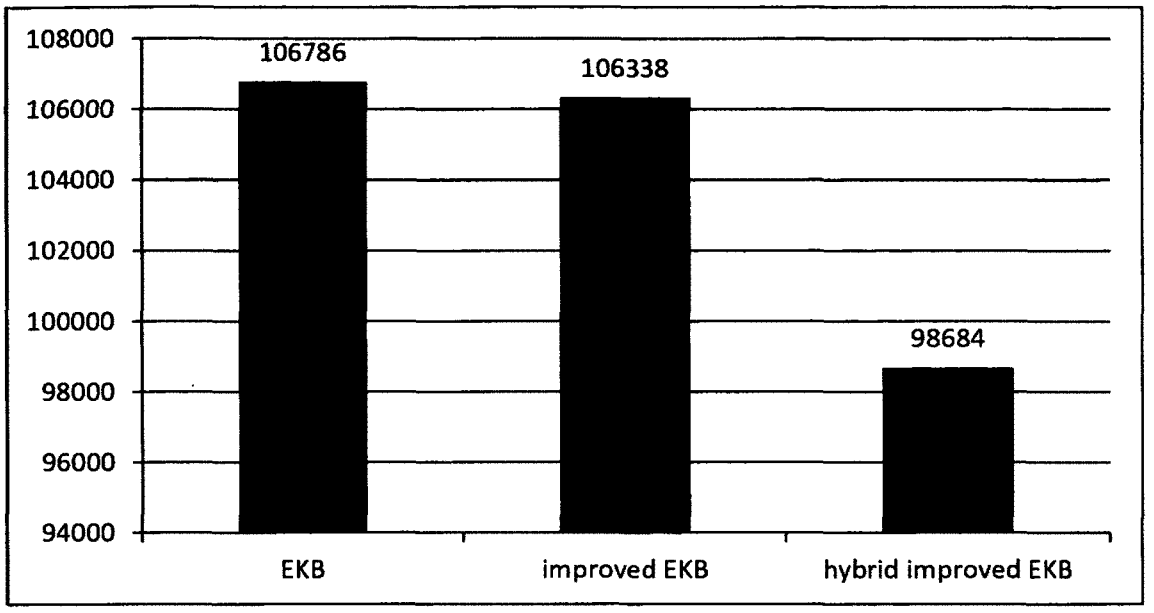


Figure 21: Number of Packets Sent, EKB at threshold $h = 45$

In this section, we analyzed and evaluated the experimental results of EKB. We have shown that the improved EKB is best suited to produce the lowest AEE, and the hybrid improved EKB method is best suited for increasing the number of hits and reducing network traffic. In the next section we will compare EKB with TDM [5] and IS [7, 18]. Improved EKB will be used when observing AEE. The hybrid improved EKB is used when reference is made to the EKB in regards to the number of hits and number of packets sent.

4.3 Comparative Evaluation

4.3.1 Average Export Error

Figure 22 shows the AEE introduced by each the TDM, IS and improved EKB algorithms. From this figure we can see that improved EKB greatly lowers the overall prediction error when predicting at large amounts of network delay. This is a result

of the improved EKB's strong ability to approximate the position of the player. It considers various factors that would affect the player in the context of the game, and uses these to predict the path of the player. To more clearly display the AEE values, table 8 shows the detailed comparison results between the three algorithms on AEE when the latency is between 1500ms to 3000ms. The improved EKB performs especially well at high levels of lag. This is demonstrated by the slope of improved EKB's AEE-prediction time relationship decreasing as high levels of prediction time are reached, while the TDM and IS seem to take a relatively linear increase in AEE as prediction time is increased.

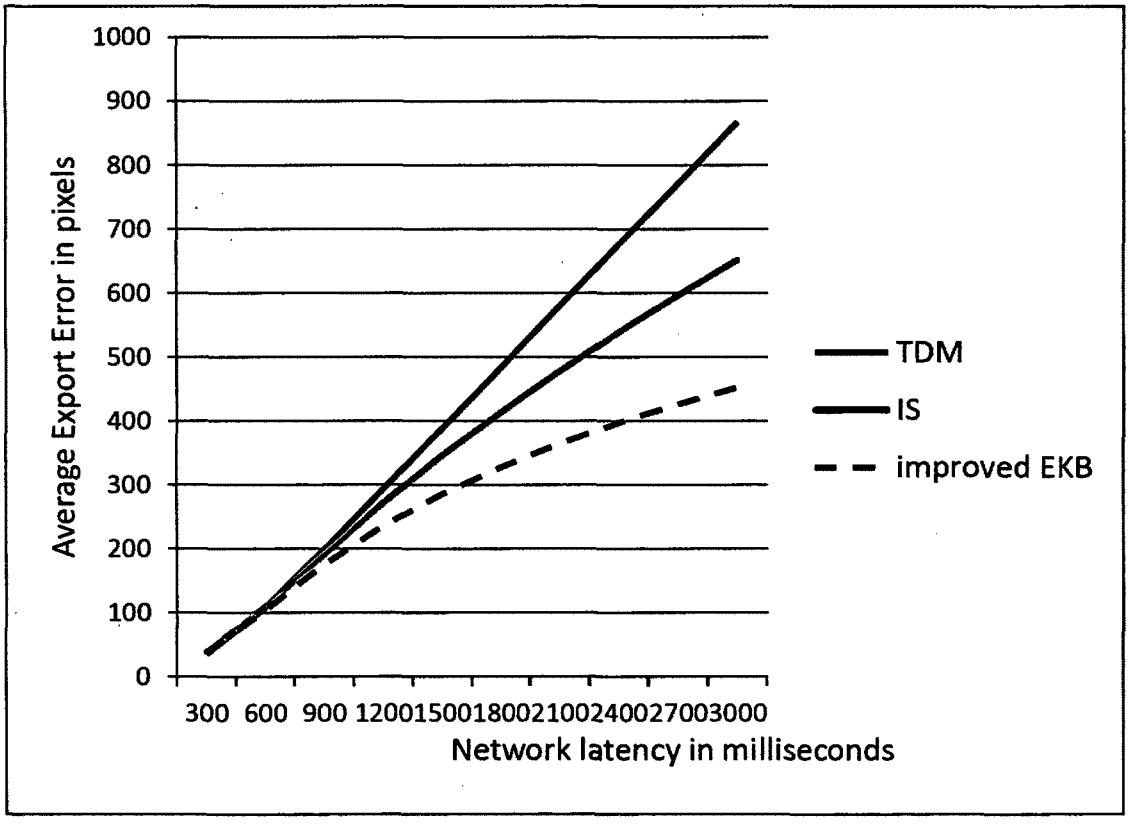


Figure 22: Average Export Error Comparison

Figure 23 shows a comparison of the dead reckoning algorithms TDM and IS against the hybrid improved EKB is included. While the hybrid improved EKB

Table 8: AEE Comparison with High Latency

	1500ms	1800ms	2100ms	2400ms	2700ms	3000ms
TDM [5]	388.8	484.2	579.5	675.4	771.7	876.5
IS [7,18]	345.7	413.8	477.7	538.5	596.6	651.4
improved EKB	285.7	327.8	364.5	397.2	426.4	452.5

yields significantly worse AEE over all compared to the improved EKB results of figure 22, it still outperforms the TDM and IS.

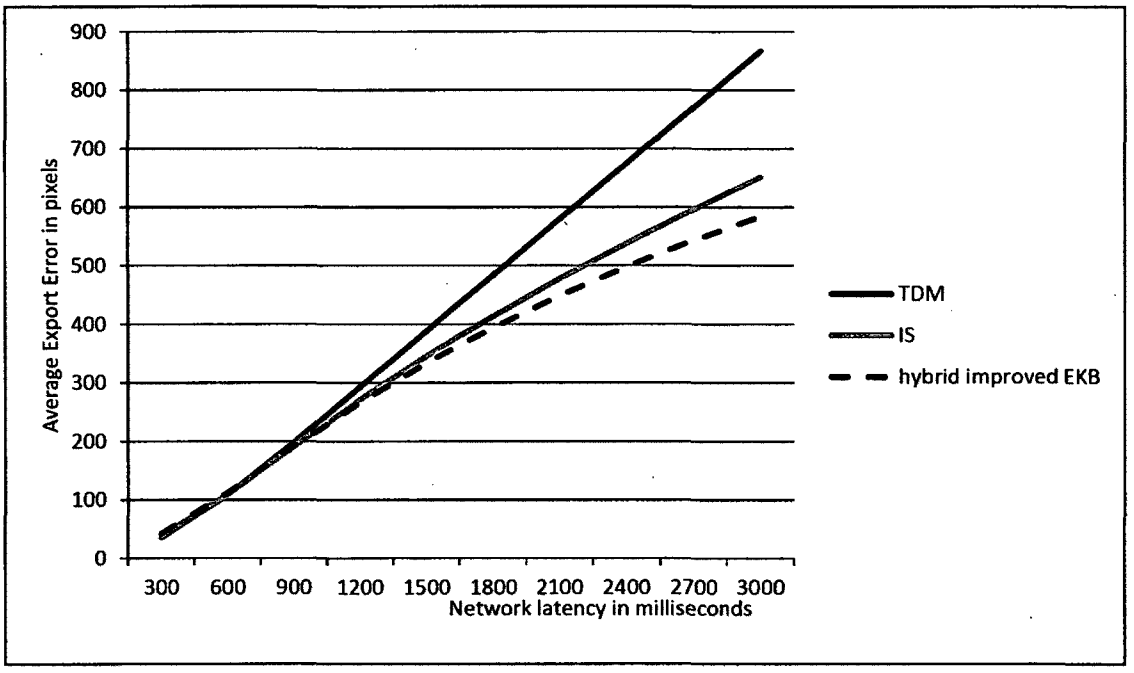


Figure 23: Average Export Error, Comparison with hybrid improved EKB

4.3.2 Number of Hits

We then measure the number of times each algorithm makes an accurate *hit*. Figure 24, figure 25 and figure 26 lay out the number of hits that were recorded at each given time interval. Given that the number of hits varies significantly across our interval of latency, we break down our results into two figures: one for low latency (from

300ms to 1200ms), one from high latency (from 1500ms to 3000ms). Figure 25 and figure 26 organize the data shown in figure 24 into high and low latencies. For added clarity, Table 9 and table 10 show the same data as in figure 25 and figure 26. Hybrid improved EKB performed relatively well at very low amounts of lag (300 ms), as well at very high amounts of lag (from 2400ms to 3000ms of lag). The TDM is a close second in terms of number of hits to the hybrid improved EKB. The strength of the TDM is its ability to predict an object moving in a linear direction. So while hybrid improved EKB can better predict a players overall behaviour, the TDM can better predict a players behaviour when moving in the same direction (which players will often do).

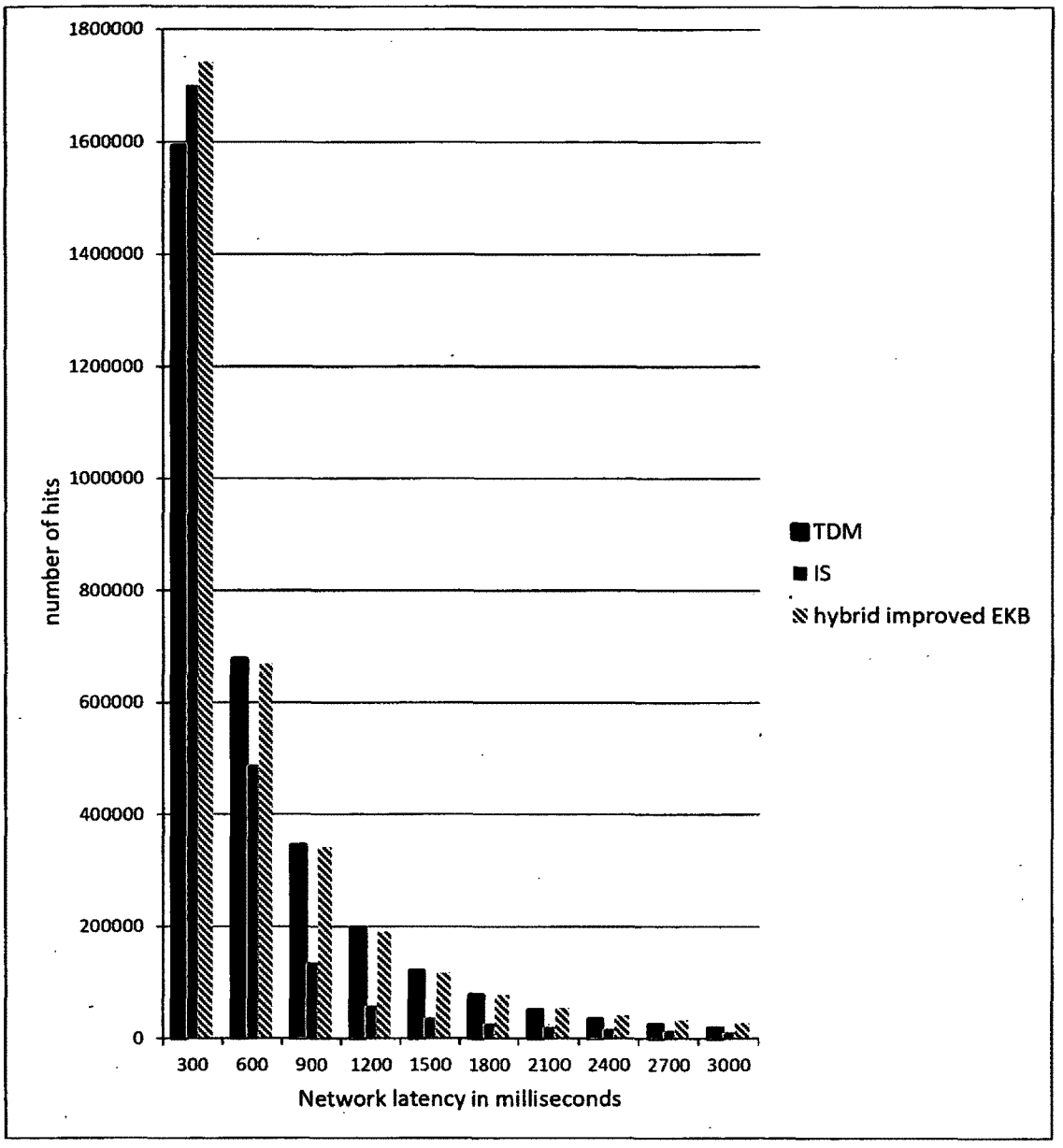


Figure 24: Number of hits, EKB, TDM and IS at threshold $h = 45$

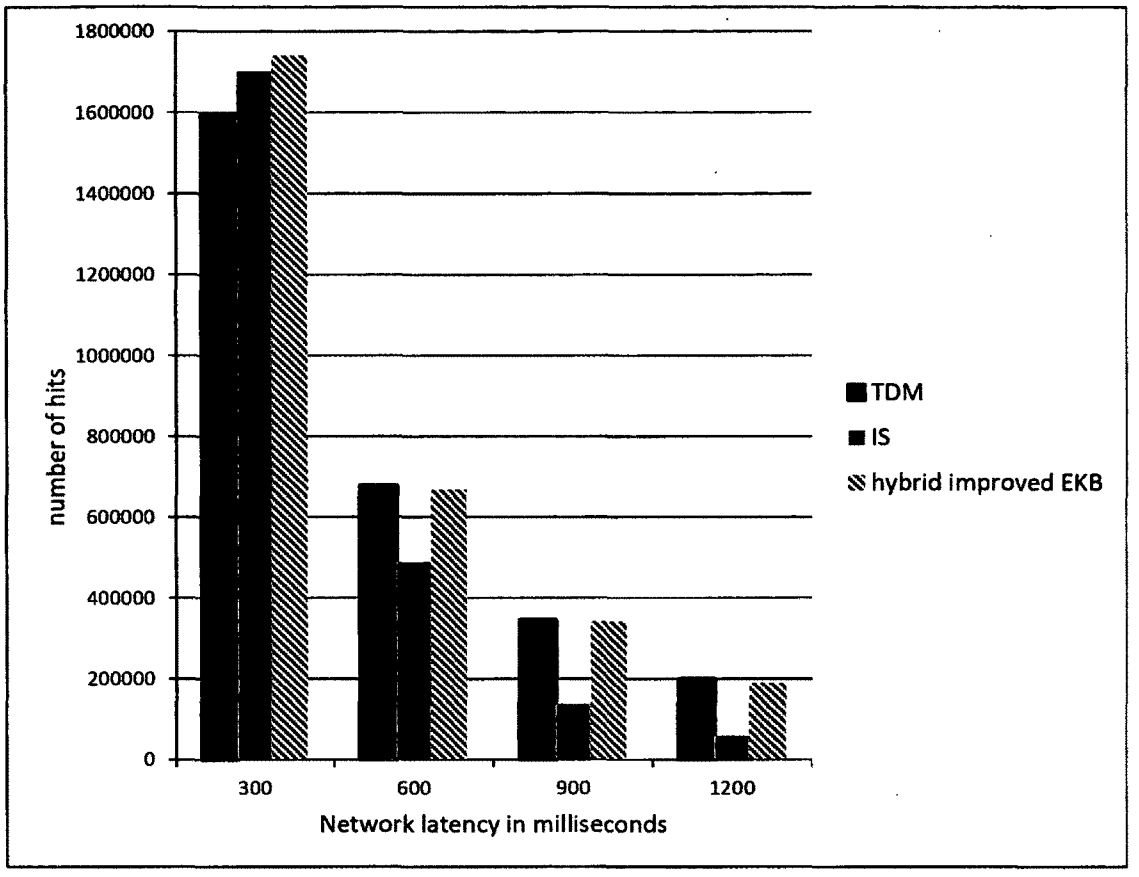


Figure 25: Number of hits, EKB, TDM and IS at threshold $h = 45$, low latency

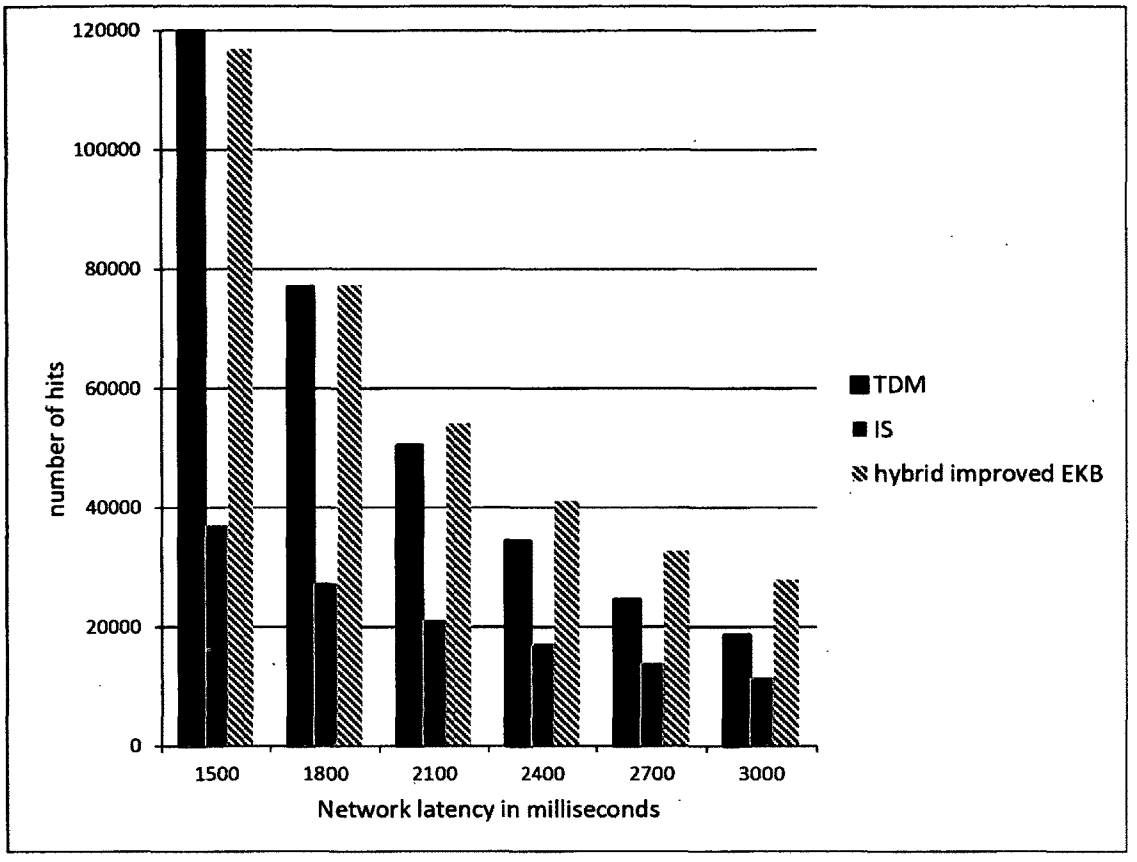


Figure 26: Number of hits, EKB, TDM and IS at threshold $h = 45$, high latency

Table 9: Number of hits at threshold $h = 45$ with low latency

	300ms	600ms	900ms	1200ms
TDM [5]	1592380	677044	344346	197604
IS [7,18]	1700786	489070	136548	59058
hybrid improved EKB	1741841	668684	341268	189671

The total number of hits counted for TDM, IS and hybrid improved EKB were 3135574 , 2513613 and 3291199 respectively (as shown in figure 27). Hybrid improved EKB performed best, followed very closely by the TDM. This is because the TDM predicts very accurately when a player moving in a single direction, which is often the case. The IS performed poorly because it failed to account for the case when a

Table 10: Number of hits at threshold $h = 45$ with high latency

	1500ms	1800ms	2100ms	2400ms	2700ms	3000ms
TDM [5]	119720	76907	50359	34238	24464	18513
IS [7,18]	37093	27391	21164	17138	13905	11460
hybrid improved EKB	116821	77275	54083	41049	32684	27823

player would move in a straight line for an extended period of time. The IS assumes that the player will assume the original velocity $\vec{V_0}$ for only a relatively short amount of time.

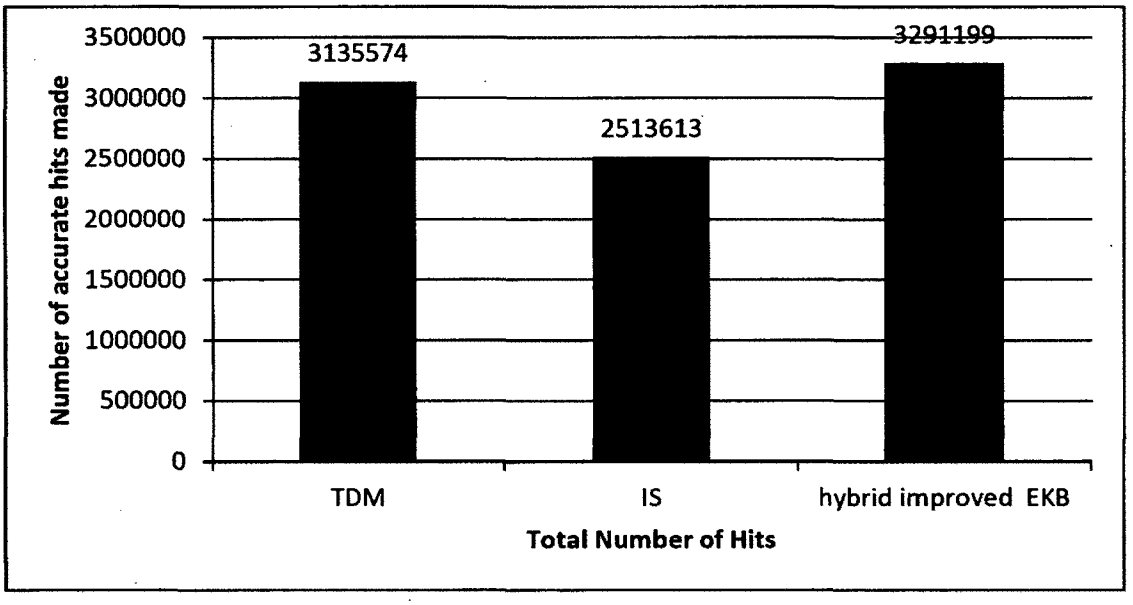


Figure 27: Total number of hits at threshold $h = 45$

4.3.3 Number of Packets Sent

We also measure the number of packets that need to be transmitted over the network during each time interval we monitor. Figure 28 shows hybrid improved EKB improves prediction accuracy over TDM and IS while sending as few packets as TDM. The difference in number of packets between TDM and hybrid improved EKB is

negligible, whereas IS needs to send a significantly higher number of packets.

It is worth noting that while, when compared with the TDM, the EKB scheme made great reduction to the AEE, it yielded relatively poor improvements/results when it came to the number of hits and the number of packets sent. This is because the AEE is a measure of overall accuracy, while hits and packets sent are concerned with making a binary observation of whether the prediction was within a small threshold h or not. The strength of the TDM is in its ability to perfectly predict the player, as long as it continues to move in a constant direction. The strength of the hybrid improved EKB scheme is that it selectively chooses to predict the player moving in a constant direction or to a defined behaviour. In this way, AEE can be improved without adversely affect the number of hits made or the number of packets sent.

We conclude that contrary to IS, hybrid improved EKB improves the prediction accuracy without increasing the network traffic. And while the number of packets sent is not improved over the TDM, the hybrid improved EKB does result in more realistic player movement on account of the great improvements made to the AEE.

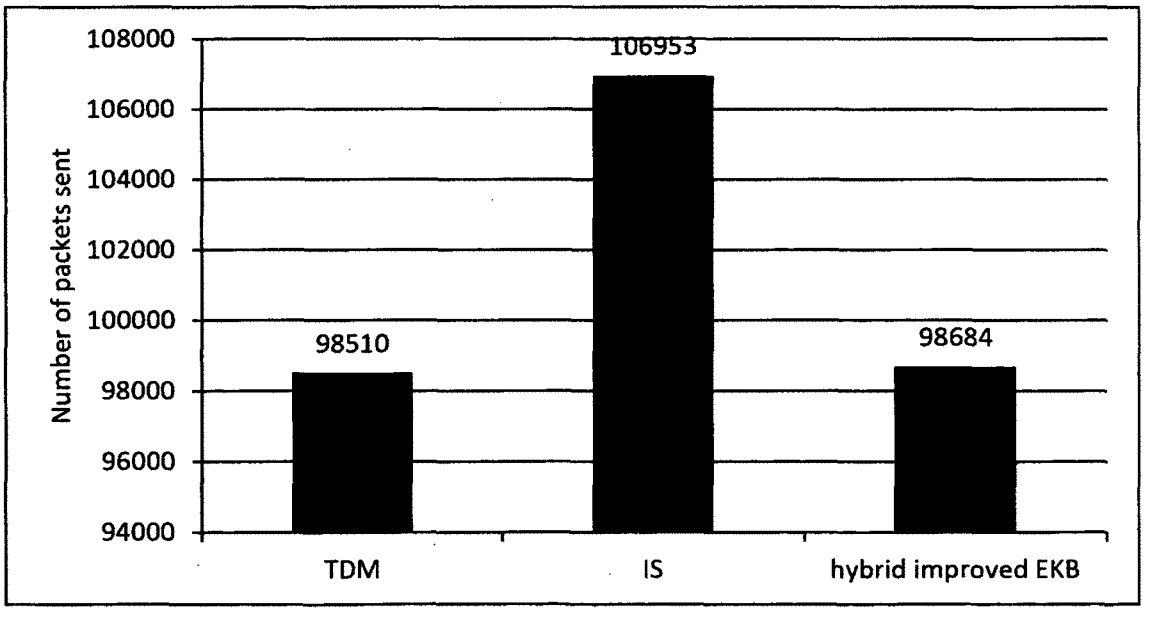


Figure 28: Packets Sent Comparison

In this chapter, we analyzed our algorithm and compared it to the TDM [5] and the IS [7, 18]. We have shown that while our method does not make improvements to the number of packets sent, it does improve the number of accurate predictions made and overall AEE. We have shown that while the hybrid method is best suited to improve hits and packets sent, its introduction also reduces the improvement to AEE, while still outperforming the other dead reckoning schemes. The next chapter will discuss our conclusions and future work.

Chapter 5

Conclusions and Future Work

5.1 Summary of Results

In light of the limitations observed in existing work on dead reckoning, we have proposed here a new prediction scheme that relies on user play patterns. Our research takes place in the context of a 2D top-down multiplayer online game we have developed. In such typical team-based action game, a player's movement is highly unpredictable and is therefore highly prone to prediction inaccuracies, thus emphasizing the need for a better prediction scheme. We started by implementing a test environment where-in we could record the game state and input information of skilled to professional players playing a traditional action game that we implemented. We then analyzed this recorded data to come up with a series of forces to exert on the player coinciding to observed player patterns. We integrated the A* algorithm to further improve performance, as well we introduced a hybrid scheme to further improve results. We have evaluated our algorithm against the IEEE standard dead reckoning algorithm [5] and the recent *"Interest Scheme"* (IS) algorithm [7, 18]. For AEE, when the network latency increases from 300ms to 3000ms, improved EKB varies from 36.9 to 452.5; TDM varies from 35.9 to 876.5; IS varies from 38.2 to 651.4. For *hits*, when the network latency increases from 300ms to 3000ms, hybrid

improved EKB makes a total of 3291199 hits; TDM makes a total of 3135574 hits; IS makes a total of 2513613 hits. For packets sent, hybrid improved EKB sends a total of 98684 hits; TDM sends a total of 98510 packets; IS sends a total of 106953 packets.

5.2 Conclusions

Our algorithms are game independent as long as the game involves a team scenario. Furthermore, our algorithms could be utilized in any given Distributed Intractive Simulation System with a team scenario, such as military training simulations. Our simulation results suggest that all 3 of our algorithms yield more accurate predictions than the IS and TDM. The overall AEE is greatly improved upon when comparing all 3 versions of our EKB scheme with the TDM and the IS. AEE is especially improved when using the improved EKB, because it best approximates the player's position. In terms of packets transmitted across the network, the hybrid improved EKB outperforms IS in all different network latency situation. And the hybrid improved EKB sends almost as few packets as TDM, while providing more realistic movement and more accurate predictions than TDM.

In conclusion, we have demonstrated that all 3 versions of our EKB improve prediction accuracy in a networked video game setting.

5.3 Future Work

In the future of our work, we would like to explore the potential for our method to take into account past decisions and play styles of the player to increase prediction accuracy. Our method assumes all players are interested in following teammates and

reacting to enemy players at the same rate. In reality, many players possess differing play styles and skill levels, and will react differently in any given situation. Combined with the A* path finding algorithm, this could yield even greater accuracy.

We would also like to experiment with taking into account the play styles associated with different weapons or character classes to increase prediction accuracy. Many games have different ways to play in the world such as different weapons to use and different characters to use that can greatly change how a player interacts with teammates and enemies. Building a framework that would allow knowledge of such factors could increase prediction accuracy even further.

We would like to improve upon how the desired position is calculated for the player. While A* made improvements to the method, the improvements were quite minimal. This may be caused by the fact that the position that is currently being used to predict where the player would like to be is fundamentally incorrect. More work should be done in finding exactly where the player would like to be at any given time.

Finally, work needs to be done in order to reduce the number of packets that are sent over the network. Though each of the 3 versions of the EKB results in improved AEE and hits over the other dead reckoning methods, it does so with no improvement to network traffic.

References

[1] S. Aggarwal, H. Banavar, A. Khandelwa, S. Mukherjee, and S. Rangarajan. Accuracy in dead-reckoning based distributed multi-player games. *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 161–165, 2004.

[2] S. Aggarwal, H. Banavar, S. Mukherjee, and S. Rangarajan. Fairness in dead-reckoning based distributed multi-player games. *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games (NetGames '05)*, pages 1–10, 2005.

[3] Entertainment Software Association. Essential facts about the computer and video game industry. 2011.

[4] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. http://developer.valvesoftware.com/wiki/Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization#Footnotes, August 2009.

[5] IEEE Standards Board. Ieee standard for distributed interactive simulation application protocols. pages 123–129, September 1995.

[6] W. Cai, F. Lee, and L. Chen. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. *Proceedings of the thirteenth workshop on Parallel and distributed simulation (PADS '99)*, pages 82–89, 1999.

[7] C. Chen and S. Li. Interest scheme: A new method for path prediction. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games (NetGames '06)*, page 41, 2006.

[8] L. Chen and G. Chen. A fuzzy dead reckoning algorithm for distributed interactive applications. *Proceedings of the 2nd international conference on Fuzzy Systems and Knowledge Discovery (FSKD'05)*, 2:961–971, 2005.

[9] M. Claypool. The effect of latency on user performance in real-time strategy games. *Proceedings of the 2nd workshop on Network and system support for games (NetGames '03)*, pages 3–14, 2003.

[10] D. E. Comer. Computer networks and internets. *Prentice Hall*, page 476, 2008.

[11] L. C.Wolf and L. Pantel. On the suitability of dead reckoning schemes for games. *Proceedings of the 1st workshop on Network and system support for games (NetGames '02)*, pages 79–84, 2002.

[12] D. Delaney, T. Ward, and S. McLoone. On reducing entity state update packets in distributed interactive simulations using a hybrid model. *Proceeding Applied Informatics*, pages 833–838, 2003.

[13] T. Duncan and D. Gracanin. Pre-reckoning algorithm for distributed virtual environments. *Simulation Conference, 2003. Proceedings of the 2003 Winter*, 2:1086–1093, 2003.

[14] S. Ferretti. Interactivity maintenance for event synchronization in massive multiplayer online games. *Bologna : Technical Report UBLCS*, 2005.

[15] A. Hakiri, P. Berthou, and T. Gayraud. Qos-enabled anfis dead reckoning algorithm for distributed interactive simulation. *Proceedings of the 2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications (DS-RT '10)*, pages 33–42, 2010.

[16] Y. Ishibashi, Y. Hashimoto, T. Ikedo, and S. Sugawara. Adaptive ?-causality control with adaptive dead-reckoning in networked games. *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games (NetGames '07)*, pages 75–80, 2007.

[17] F. Li, L. Li, and R. Lau. Supporting continuous consistency in multiplayer online games. *Proceedings of the 12th annual ACM international conference on Multimedia (MULTIMEDIA '04)*, pages 388–391, 2004.

[18] S. Li, C. Chen, and L. Li. A new method for path prediction in network games. *Computers in Entertainment*, 5(4):8:1–8:12, 2008.

[19] D. Liang and P. Boustead. Using local lag and time warp to improve performance for real life multi-player online games. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games (NetGames '06)*, pages 37–es, 2006.

[20] Y. J. Lin, K. Guo, and S. Paul. Sync-ms: Synchronized messaging service for real-time multi-player. *Proceedings of the* 10^{th} *IEEE International Conference on Network Protocol (ICNP '02)*, pages 1092–1648, 2002.

[21] D. Madden, D. Delaney, S. Mcloone, and T. Ward. Visibility path-finding in relation to hybrid strategy-based models in distributed interactive applications. *Proceedings of the* 8^{th} *IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '04)*, pages 91–97, 2004.

[22] A. McCoy, T. Ward, and S. McLoone. Multistep-ahead neural-network predictors for network traffic reduction in distributed interactive applications. *ACM Transactions on Modeling and Computer Simulation*, 17(4), 2007.

[23] W. Palant, C. Griwodz, and P. Halvorsen. Evaluating dead reckoning variations with a multiplayer game simulator. *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video (NOSSDAV '06)*, pages 4:1–4:6, 2006.

[24] P. Quax, P. Monsieurs, W. Lamotte, D. Vleeschauwer, and N. Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. *Proceedings of* 3^{rd} *ACM SIGCOMM workshop on Network and system support for games (NetGames '04)*, pages 152–156, 2004.

[25] S. Russell and P. Norvig. Artificial intelligence - a modern approach. *Prentice Hall*, 1995.

[26] X. Shi, X. Wang, J. Bi, F. Liu, D. Yang, and X. Liu. A dr algorithm based on artificial potential field method. *Multimedia Tools and Applications*, pages 247–261, 2009.

[27] Z. B. Simpson. A stream-based time synchronization technique for networked computer games. http://www.mine-control.com/zack/timesync/timesync.html, March 2010.

[28] Valve. Source multiplayer networking. valve developer community. http://developer.valvesoftware.com/wiki/Source Multiplayer Networking, October 2011.

[29] A. F. Wattimena, R. E. Kooij, J. M. Vugt, and O. K. Ahmed. Predicting the perceived quality of a first person shooter: the quake iv g-model. *Proceedings*

of 5^{th} *ACM SIGCOMM workshop on Network and system support for games (NetGames '06)*, pages 1–4, 2006.

[30] Y. Zhang, L. Chen, and G. Chen. Globally synchronized dead-reckoning with local lag for continuous distributed multiplayer games. *Proceedings of* 5^{th} *ACM SIGCOMM workshop on Network and system support for games (NetGames '06)*, page 7, 2006.

Chapter 6

Example Replay Raw Data

Following is an example of the raw gameplay data recorded from a play session. This raw recorded data is used to play back the original play session for viewing and analysis. Map, Us, U, UI, N, P, Pos, Sy, Alignment, Si, Is, I, T and B, X, Y represent the current map, the list of all users, information for a single user, the user's identification number, the user's name, starting information of a user's player, starting position, character type, team, spawn time of the player, list of input data, individual input entry, time of input, character representing directional and mouse input, the X position of the mouse cursor, and the Y position of the mouse cursor respectively respectively.

```
<Map>ProvingGrounds.xml</Map>
<Us>
    <U>
        <Ui>0</Ui>
        <N>Unnamed</N>
        <P>
            <Pos>
                <X>-2560</X>
                <Y>1024</Y>
```

```
        </Pos>
        <N>Unnamed</N>
        <Sy>4</Sy>
        <Alignment>0</Alignment>
        <Si>3621</Si>
    </P>
    <Is>
        <I>
            <T>3723</T>
            <B>0</B>
        </I>
        <I>
            <T>3740</T>
            <B>0</B>
        </I>
        <I>
            <T>3757</T>
            <B>0</B>
        </I>
        <I>
            <T>3774</T>
            <B>8</B>
            <X>-1827.08</X>
            <Y>1147.52</Y>
        </I>
        <I>
            <T>3791</T>
```

```
    <B>8</B>
</I>
<I>
    <T>3808</T>
    <B>8</B>
</I>
```