Delay Compensation in Networked Computer Games

Robert F. Buchheit
January, 2004
Master's Project

Table of Contents

Abstract

The Web is becoming increasingly popular and ubiquitous due to the value that near instant communication can add to many applications. One of the largest and most important of these applications, in both economic and cultural terms, is computer gaming. The addition of networking to gaming allows for interaction between many players which can greatly enhance entertainment value both through more interesting and relevant competition and also by creating a dynamic, interactive environment that simply would not be possible otherwise. As a result, the majority of new games incorporate at least some online, multi-player play mode. Others are entirely online without any sort of offline mode.

Many of the most popular games model physically realistic scenarios such as piloting a vehicle or interacting with and moving around in a virtual environment. In order to be enjoyable and engrossing it is necessary for game interaction to be as natural as possible, most similar to real life. In practice, this tends to require a very high level of network service, primarily in the area of latency but also, to a lesser extent, in bandwidth, jitter, and packet loss. Playing over a LAN is a commonly used solution but is much more limiting in the number of potential players than play over a WAN.

Thus, the challenge is to enable play of these sorts of games over WANs even when the quality of network service is less than ideal. This project seeks to explore the topic in detail through the development of a demonstrative networked game and implementation of an application-layer delay compensation strategy. The game will then be tested under different latency situations and the effectiveness of the chosen strategy will be evaluated.

1. Introduction

One of the fastest growing segments of the computer gaming market is that of online gaming. Contributing to this growth (or perhaps responding to demand) is the widespread support of networking, both in PCs and consoles such as Xbox and Playstation2. That fact, combined with the strong growth of the web in general, the presence of online play options in most newly released games, the popularity of a number of massively multiplayer games, and the generally enhanced entertainment value of social game play indicate that this trend is likely to continue [15]. Everquest for instance, currently the world's largest massively multiplayer online game, boasts over 430,000 subscribers [1,7].

Many of these games incorporate a high degree of realism whether it be piloting a vehicle or interacting with other players in a virtual environment. As a result, games such as these are highly interactive and tend to require high levels of network service in terms of low latency, jitter and packet loss. As a result, some users have confined their game play to LANs where network service is generally excellent. However, LANs can typically support only a limited number of users over a small geographic area. One solution, used by some, has been to hold a 'LAN party', where users physically gather together with their computers to form a temporary LAN and play networked games. However, such a strategy is not feasible in the general case (for a variety of reasons) and thus the challenge is to better enable play of such games more broadly, such as over WANs. Even with the recent growth of residential broadband service in the US, only 21 percent of online users currently subscribe [6]. Thus, with such a large number of users still using dialup connections and the potentially large separation of users on the web, situations of higher latency are largely unavoidable.

Although logical architectures other than the standard client-server model exist for networked games, peer-to-peer for instance, the majority of online games utilize this architecture [13]. In the client-server model either a machine is dedicated to coordinate a game or one of the player's machines is promoted to the status of server. It is then the responsibility of the server to coordinate the game, managing entry and exit of players, as well as the game itself, accepting user input, integrating to a new game state, and then

distributing the appropriate information to all users. As this is the predominate structure in use today, the games developed for this project will also utilize this architecture.

Given the realism of many games and their basis in Newtonian physics strong parallels can be drawn from the remote interaction with the virtual reality of a game and the remote interaction with physical reality (teleoperation). As such, work done in the area of compensation for games could well be applied to this field as well.

This topic, as well as others, will be discussed in this paper. Section 2 will review relevant background information on the issues of IP networks, physically realistic games, network games, and previous work done on the topic of latency compensation. Section 3 will discuss in detail the design and implementation of the demonstrative games as well as our approach to delay compensation. Section 4 will discuss the testing performed with these games and the results will be presented and analyzed. Section 5 will conclude the paper.

2. Background

As physically realistic games and their counterparts in the real world, teleoperation of physical systems, are the primary interest of this project the basics of such will be briefly described. Physical systems evolve according to the fundamental laws of physics which, for most everyday situations, the basic Newtonian equations describe adequately. Thus, physically realistic games derive their realism from the Newtonian characteristics that the sets of linear differential equations they are based upon have. A linear system is described by the system state x(t), the input function u(t), and the output function y(t). The rate of change of the system state and game output are then described by:

$$\dot{x}(t) = Ax(t) + Bu(t) \quad (1)$$

$$y(t) = Cx(t) \quad (2)$$

The game output is expressed as a function of the game state. This system which is based on continuous time is then sampled at regular intervals and the relevant portions of the state are collapsed into a packet and transmitted by the game server to the remote client.

The client server model is the most commonly used in computer games due to its relative simplicity and effectiveness. Games based on other models such as peer-to-peer connections and distributed or replicated game states have been developed but are much less common due to generally increased complexity with minimal explicit benefits. Massively multiplayer games do depend on multiple physical machines to run an instance of a game world and they likely run on a peer-to-peer type system but external clients interact with them as if there were a single server machine. All subsequent discussion will assume the client server model.

As the focus of this project is to study methods of improving playability of network games over wide area networks such as the web it is necessary first to examine the basic properties of IP networks. Because IP networks such as the web are shared by many users the service that will be available at any particular time is unpredictable. Furthermore, because IP is only a best effort protocol, there are no bounds on the length of time a packet will take to propagate across the network or even that it will arrive. Consequently, successful data transfer has to be verified and directed either in the transport layer (TCP) or in the application layer if UDP is being used.

Additionally, because computers across the web generally do not have synchronized clocks it is not possible to determine with any precision how much time has passed while a particular packet was enroute. If the clocks of two computers have been synchronized or have a known skew a local timestamp on each packet can be used to calculate its age by the remote side of the connection. However, methods of online clock synchronization such as NTP are only approximate and more accurate methods using. for instance.

the telephone system are less practical [3]. As a result, any client/server pair generally has to be able to operate without synchronized clocks and, as a result, neither is able to determine beyond a simple estimate how long a particular packet was enroute.

Thus, the most common measurement of latency between two nodes is that of round trip time (RTT) which is the measure of time it take for a packet to reach a node and then return to its sender. This gives a rough approximation of latency between any two nodes but does not guarantee in any way that the out and back travel times are equal or that latency overall will not change in the very near future. Frequent, unpredictable changes in latency over time are referred to as jitter. This phenomenon makes simple latency estimate techniques less effective. Packet loss occurs when a data packet is dropped by the network before being delivered to its destination. Most often this occurs due to network congestion which causes router queues to fill faster than they are being emptied and subsequently, when the queue is full, the router must drop packets. As far as networked games are concerned, latency, jitter, and packet loss are the most critical areas and thus most prone to causing problems when they degrade. The majority of networked games are designed to saturate the narrow last mile link, such as a 56k modem, and thus require only moderate bandwidth [2].

In a networked game the primary goal is to present to the user an accurate state of the remote environment and at the same time allow the user to interact with the environment naturally. The challenge that network latency poses is that a time-varying delay of unknown length is inserted between the user and the server (for simplicity ½ RTT, in actuality, there is no guarantee on the symmetry of the client-to-server and server-to-client latencies). When the delay is very small it poses no problem (the system still behaves very similarly to real interactions). However, when the additional delay is non-negligible it causes three main problems for interactive environments:

1. The world perceived by the user is aged by the amount of time it took for the information to propagate across the network, ½ RTT. This age limits the user's ability to make the appropriate input choice.
2. When the user makes an interactive input an additional ½ RTT passes before the input actually reaches the server and is applied. Thus, the appropriateness of the user's input, already based on an aged system, is further limited.
3. The user must wait 1 full RTT to receive feedback regarding the outcome of their input. For systems where feedback is very important to control this can make successful interaction nearly impossible.

These are the primary problems that exist when a user interacts with a remote environment over an IP network. In the case of a game with multiple, interacting players two additional problems arise. The first is in the maintenance of a consistent game state among all the players. Because each player's connection to the central game server is subject to the limitations described above each player is likely to be experiencing different latency conditions. Additionally, the individual clients may be performing local delay compensations such as dead reckoning (described below). As a result, each player's view of the game state, for instance the position of their vehicle in a race, can be very different. As an example, players in such a racing game may each believe that they have the lead when, in actuality, only one has the lead [10]. This can lead to confusion on the part of the players and a less enjoyable experience overall. The second is fair interaction amongst the players. Because each player's input commands will likely require a different amount of time to reach the server there are fairness issues about how the server should apply each input. If the server applies each input in the order they arrive then players with higher latency connections are placed at an additional disadvantage in the game. On the other hand, at the risk of creating additional inconsistencies in the various game states, the server can attempt to estimate the latency of the client's connection and take that into account when applying control inputs.

These are the basic problems posed by the lower levels of network service which can be encountered on WANs. Thus, to better enable networked play under these conditions it is necessary to somehow compensate for these fundamental problems.

The issue of latency and the problems it poses is not confined to the realm of computing but rather, is the norm for human interaction. When a person interacts with the local, physical world they generally receive continuous input (light, sound, etc). The only limiting factor on how quickly they can react is the speed at

which their nervous system can process the incoming information, generate a plan of action, and implement it via the muscles. However, in many cases, these delays are non-trivial. For example, visual motion integration requires on the order of 100ms to complete [8]. This, along with delays for additional planning and reaction can quickly become a significant delay. There is evidence in fact, in the form of the flash-lag effect and related phenomenon that the neural pathways of humans and lower animals include built in compensation for processing delays in the form of an internal forward or motion-extrapolation model [9]. While it would seem that humans and other primates are likely not dependent on this due to their ability to learn and compensate, lower animals such as frogs would likely fail in critical behaviors such as the tongue-snap response and, as a result, likely starve [8].

These basic facts concerning human perception have two important implications. First, in order for a person's interaction with a virtual environment to be as natural and fluid as possible they should be able to view and interact with the virtual game world in the same way they do everyday in the real world. This means that the game state displayed to them should be as similar as possible the game state they are actually interacting with and, when they do interact, they should be able to see the resulting changes in the game state as they would in the real world (in a timely manner). While, in many circumstances, the user would be able to learn compensation techniques by which to interact with an inaccurate display of the virtual world, it would make the interaction less natural and distract from the game itself. Second, the same mechanism by which the internal biological systems compensate for latency can be used to help compensate for latency in the virtual environment of the game.

A delay compensation technique similar in nature to the one described above has already been developed and is used by some popular games. The technique, known as dead reckoning, is based on the forward extrapolation of game state [12]. As described above, games are based on sets of equations and evolve according to these. Consequently, a game client which is aware of the basis equations used by the game server to compute the state of the game can use the state information it has to predict forward the future state of the game. It can then display this extrapolated state to the user with the hope that it will be more relevant than the previous state information. For instance, the game client can maintain an estimate of the RTT between itself and the server. Then, it uses the state data sent from the server, plus the estimated RTT to extrapolate the game state forward in time by 1 RTT. In the event of a perfect extrapolation (no missing information) and perfectly symmetric delay between client and server, the extrapolation would precisely represent the game state at the server when the user's current inputs arrive. Thus, this technique should allow the user to interact with the game in a more normal fashion, despite the delay.

In reality the extrapolation is usually not perfect because the client does not have perfect information and unforeseeable events which occur at the server will not be included in the extrapolated state (for instance, a random perturbation in the system or the input of another player in the game). Additionally, the latencies between client and server are likely not symmetric. Thus, the extrapolated state will contain flaws. When errors do occur it is necessary to reconcile the extrapolated state shown to the user back with the actual game state which has arrived from the server. This is known as convergence [12]. A zero order convergence simply replaces the extrapolated state with the new state. In the case where the differences are significant enough to be noticeable this can lead to visual deterioration of the game. Higher order convergence can also be used to merge the inconsistent game states more smoothly over time but that can require additional computation by the client and also forces the user to wait longer for accurate information, a situation which is potentially problematic when immediate action by the user is required.

In general though, dead reckoning effectively addresses problems 1 and 2 as described on the previous page (although the user now interacts with a future state of the game). However, as in problem 3, the user is still forced to wait 1 full RTT before seeing any feedback about their control inputs coming from the server. To address this, another technique, also based on extrapolation, is used. This is generally referred to as short circuiting [13]. With this method, user input is not only sent to the game server but is also fed back into the local state extrapolation such that the user can immediately see feedback on the effect their inputs will likely have when they reach the server. If done properly, this can be very effective.

However, as mentioned above, the extrapolation used in both of these techniques will cause some anomalies with regards to what the user believes he is seeing and what actually happens and leads to greater inconsistency among the players of the game.

Another method of compensating for delay is through the use of a presentation delay [10]. This technique is similar in some ways to the buffering that is used to improve playback on streaming media. The basic concept of the technique is that when the game client receives an update from the server it delays displaying it to the user for a fixed period of time, say 100ms. In this way all users whose connections with the server have a latency of at most 100ms will be able to play the game on the same level as all other users. A client experiencing a delay of 50ms will delay display of the state for 50ms while another client experiencing a delay of 100ms will display the data immediately. In the event that all the players in a particular game have latencies below the presentation delay threshold, the global game consistency among the players (in regards to the game state they see) should be excellent (in the case of fixed, known delays it should be perfect). However, this method also has serious drawbacks. Aside from the fact that the entire game is played in the past, the primary problem is that the user is still delayed from seeing input feedback. This largely disqualifies the technique for physically realistic games that have fairly fast dynamics. However, recent work has shown some promise for the use of a modified presentation delay in teleoperation systems [5]. In this approach an active technique is applied to approximate the variable time delay of an IP network with a fixed delay (with 95% of delays falling within the fixed delay) thus allowing the use of a standard scattering transformation to avoid stability problems.

The three previously described techniques summarize the commonly discussed techniques in the current literature on the topic. Some hybrid techniques have been tested (combining dead reckoning and short circuiting with a small presentation delay) with some success [10]. Some testing has been done on the effectiveness of dead reckoning and short circuiting techniques and found them to be quite practical in certain categories of games such as simulation and action type games (first person shooters) [11]. However, as these techniques are only implemented on the game client they can not reduce actual latency, only its impact. Given near perfect state extrapolation the user will still be limited in his ability to react to unpredictable events. Specifically, for an event that is not predicted by the forward extrapolation, the player will not be able to respond to it until 1 RTT later, at the earliest. This is the amount of time it takes for notification of the event to arrive at the client and then for an immediately issued client input to make the return trip to the server.

For teleoperation and games whose physics behave in a similar, irreversible manner this fundamental limit will remain. The only way a user can give response sooner than this limit would be if the user had previously sent a contingency control input to the server which was then run locally at the server at the time of the event. For example, in the case of teleoperation of a robotic vehicle a contingency control signal might direct the remote controller to do an immediate, emergency stop in the case that the vehicle detects a steep drop in terrain (cliff).
However, in the physics of a game server this fundamental limitation need not necessarily apply. As mentioned above, a game server could use an estimate of a user's latency to place their input back in time and recalculate what would have happened had the control signal arrived at that time. This technique is in fact used in one of the most popular multiplayer games presently played on the web. The Half-life net code and rendering engine on which such popular Mods as Counter-Strike and Team Fortress are based have an 'un-lag' server option that, when enabled, back-calculates approximately where all the players were when they fired their bullets [4,14]. This is combined with the use of dead reckoning and short circuiting compensation mechanism on the game client. The combination of these techniques makes the games based on the Half-life engine relatively playable even from a rather high latency dialup connection. The games based on this engine are among the most played on the internet, likely because of this fact [4].

For the purpose of this project a delay compensation strategy consisting of a combination of the dead reckoning and short circuiting mechanisms just described was chosen for implementation. The systems will be described in detail in the next section. Additionally, in the interest of studying more thoroughly the teleoperation aspect of the concept the servers implemented in both the prototype and the final games only

run forward in time and consequently do not allow for any sort of direct latency reduction for user control inputs.

3. Game Development

Prototype Game

The first game was developed using Python, a fairly new, interpreted language.

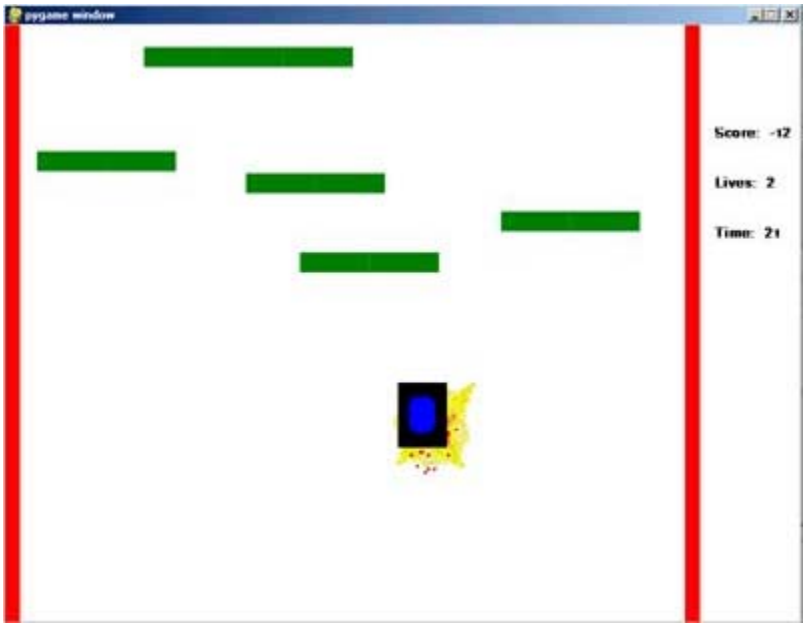


Figure 1. Prototype Screenshot – Collision with an obstruction

Game Scenario:

In this game the player controls a "Jet-Car" which has front, rear, left, and right maneuvering thrusters. The object of the game is to avoid oncoming obstructions (the green objects in the above screenshot) that vary in size and travel at different speeds while coping with a random "wind". A collision with one of these obstructions causes the player to loose four points while receiving one point for each one successfully avoided (passes out of view). Additionally, the red borders on either side of the screen represent impenetrable barriers that destroy the vehicle if it collides with either of them. The scenario may be thought of as flying through a side bounded debris field with a stable forward velocity such that the vehicle is progressing through the field. Thus, the player's goal is to guide the vehicle through the debris field, keeping the vehicle intact and obtaining a maximal score.

Physics:

The physics of this game were deliberately kept very simple. The equations of motion are first order, as they model velocity to determine position but acceleration is performed in a simply additive manner. As such, the equations on which the game is based are as follows:

(3) $V(t+1) = V(t) + U(t)/10 + W(t)$ — V(t) is the vehicle velocity, U(t) is the player's sampled controller input, and W(t) is the effect of the random wind.

(4) $P(t+1) = P(t) + V(t)*(\Delta t)$ — P(t) is the vehicle's position and V(t) is its velocity, and Δt is the amount of time that has passed since the last update of the game state.

Obstructions move from the top to the bottom of the screen with a constant velocity. When a new obstruction 'spawns' it is given a random assigned size (small, medium, large) and velocity (slow, moderate, fast). Their motion then proceeds as follows:

(5) $P(t+1) = P(t) + V(t)*(\Delta t)$ — P(t) is the obstruction's position and V(t) is its velocity, and Δt is the amount of time that has passed since the last update of the game state.

Collisions are modeled via simple polygon collision such that, if the vehicle's polygon is found to overlap that of an obstruction or a terminal wall then a collision event occurs and the appropriate results are applied. In the event of a collision with an obstruction the obstruction is destroyed, an explosion event is generated, and the score is deducted. If a terminal wall is collided with the vehicle is destroyed, an explosion event is generated, a life is subtracted, and a new vehicle spawns at the starting point. If no lives remain the game ends.

Network:
In this game, once the client initiates the game it waits for the server to respond with a regular stream of updates, approximately every 10ms. Each time it receives a server input, the client updates the display for the user and sends the latest client control inputs to the server. In the event that the server's regular update is delay the game client updates the display with an extrapolated game state and sends the latest control inputs.

Delay Compensation:
Aside from the control/update data stream there is also a ping stream between the client and server. This is used by the client to maintain a running estimate of the latency between itself and the server. This estimate is then used to extrapolate forward the game state by 1 RTT as is the strategy of dead reckoning. To provide control feedback to the user the short circuit mechanism is used. Thus, in addition to sending the control inputs to the server they are also stored locally. The inputs are then compiled into a position offset for the vehicle and applied to the incoming state data before it is displayed to the user.

While the prototype game provided invaluable experience in regards to the delay compensation and networking game systems it also has some limitations. The critical timing of the game and the central role of collisions make the study of certain characteristics of the delay compensation, such as its effectiveness at higher latencies, more difficult. Thus, it was decided that a new game would be developed where critically timed events, such as collisions, could be avoided as was necessary. The environment of a sailing boat on the open ocean provides just such environment. Additionally, the more complex dynamics of the sailboat provided an opportunity to study the effectiveness of delay compensation with a more complex environment than the prototype provided. Finally, the Java programming language was chosen for development of the final game over Python to gain better platform independency as well as the ability to package the game client as an applet which could be simply distributed for demonstration purposes.

Final Game

The final game was developed using the Java programming language. This is game is designed to behave similarly to if the user were remotely operating an actual sailboat on the open ocean. As such the game server is designed to apply the user's inputs as best it can and to regularly send state information to the user. However, it does not attempt to do any active latency compensation which would involve reversing the game state and applying the user's controls. In the situation of a real boat this would, of course, be impossible.

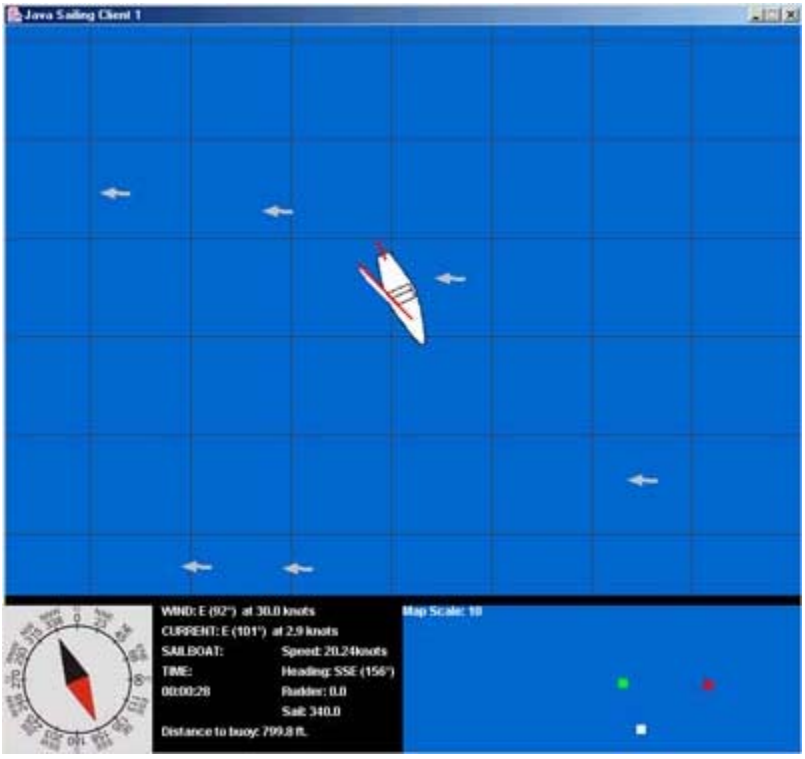


Figure 2. Sailing Game Screenshot

Game Scenario:

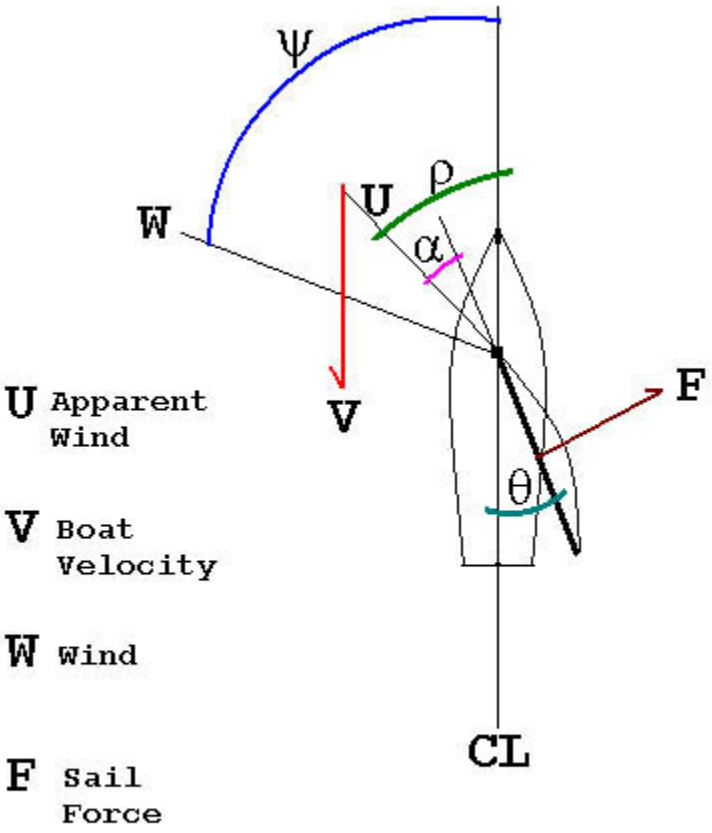
In this game the player controls a physically realistic sailboat on the open ocean by adjusting the maximum angle of the sail and the rudder. The sail's position is not rigidly fixed at the maximum angle but rather moves realistically in the wind. If the wind is behind the sail then it will move to its maximum angle. However, if the wind shifts to come from the other direction the sail will go limp and align with the wind. This behavior models the use of a rope to control the sail, either pulling it in or letting it out. The rudder angle is controlled directly and adjusted either left or right from center to yield the desired turn rate of the user. The speed at which the boat turns is based on both the speed of the boat and the angle of the rudder.

The objective of the game is to sail towards the target buoy as quickly as possible given the environmental conditions (wind and current). At startup the user can choose from several different modes which control the wind and current conditions, along with whether or not the user would like to compete against a sailing AI run on the server.

Physics:

The physics of this game are more complex than those of the prototype game and are implemented by a separate class. This class is responsible for controlling the environmental conditions of the game as well as

for computing the position and velocity of the boat given its sail and rudder angles, the orientation and velocity of the boat, and the velocities of the wind and current. The system represented by this class uses scaling factors based on elapsed time since the last update in order to evolve the system realistically.



$$U^2 = V^2 + W^2 + 2\text{Cos}(\psi)VW$$

$$F = C\ \text{Sin}(\alpha)\ ½\ \rho V^2 A$$

Figure 3 Basic Sailboat Forces

The sail's position is handled by a specific function that moves it realistically with the wind as it is constrained by the sail control of the player. The sail attempts to align with the wind if it can to be in a lower energy state. When hanging loosely in the wind the sail flaps and generates no force. However, when the wind is behind the sail and the sail is tight against the control rope it generates force according to the equation:

(6) $F = C_0 (\sin \alpha) V^2$ — C_0 is a constant taking into account the sail's area, the density of air, and other factors, V is the velocity of the apparent wind and α is the sail's angle to the apparent wind

The force is directed perpendicular to the sail and thus the forward force on the boat is calculated with the equation:

(7) Forward Force = F sin(θ) — F is the sail force calculated in equation (4) and θ is the angle between the sail and the boat.

The leeway force is calculated similarly but using cosine instead. However, the centerboard is assumed perfect in preventing the boat from slipping sideways in the water and thus, no leeway is applied to the motion of the boat. The drag associated with moving through the water is calculated to be:

(8) Drag Force = $0.5\ V^2$ V is the velocity of the boat.

When calculating the new velocity of the boat the drag force is subtracted from the forward force being generated by the sail. If the number is positive the boat accelerates, otherwise it slows. The turn caused by the action of the rudder is given by:

(9) Turn = $C_1\ V \sin(\theta)$ C_1 is a scale factor, V is the boat's velocity, and θ is the angular distance from center of the rudder.

The boat's velocity is then redirected slightly according to the turn value from equation (9). The wind and current conditions of the game environment are handled differently depending on the options chosen by the user. If the user chooses 'God mode' then their commands are used to adjust the velocity of the wind. They also may also choose for the winds to be shifted randomly or shifted according to a fixed pattern which defines wind angle, speed, and length until next shift. The user can choose either no current for the game, a constant current which is randomly chosen at the beginning of the game, or a current which shifts randomly throughout the game, in a manner similar to the shifting of the wind.

Network:

The design of the network protocol for this game is focused on minimizing end to end latency as well as vulnerability to packet loss and reordering.

In order to make data transmission as fast as possible UDP is used to transmit data back and forth between the client and the server. Each packet generated by the client carries the user's desired sail and rudder controls as well as a client time stamp on the controls. When a client packet is received by the server during the course of the game the server replies with a collapsed game state (in string format) and an acknowledgement of the most recent client time stamp it has received in an input packet. It this way the client can know that a particular packet has arrived at the server and that packets previous to that either arrived already or will never "arrive".

This is because of the way the network class is designed. Each message carries an additional timestamp and processing flag attached to the data payload of the packet. When the timestamp of an arriving packet is older than the most recently arrived packet (indicating a reordering) the aged packet is thrown away. Additionally, the network class uses a buffer with only one slot. This way when either the client or the server checks the buffer for a message it only receives the freshest possible data. Because the game is time sensitive and the packets each carry full information only the most recently arrived packet is ever relevant. However, some messages, such as those which start or end the game are important and thus carry a special processing flag "P", standing for persistent, which indicates that they are not to be overwritten. Such messages persist in the buffer until the external client or server classes read them. However, because both the client and the server update very regularly, it is the case that the buffered message is generally read before another packet arrives and thus overwriting does not occur frequently. In the case of the client it is clear that only the most recent server data is ever relevant. However, in the case of the server it is more complex. Because command data is stored across client controls packets, such as with a rudder turn which requires multiple cycles over which to complete, the order of generation for control packets is relevant. The server, however, is only able to apply command inputs to the present game state. Thus, was a latency spike to cause a set of command packets representing say, a rudder turning sequence, to arrive after a newer command and the server collected all of the delay packets it would still need to apply them in order over a period of time to achieve the effect that the user originally desired. However, this would then delay all subsequent inputs by the user as the server attempted to faithfully apply them. Consequently, to avoid such

issues, the server is designed to simply drop delayed packets and allow the user to reissue any command sequence that might have been lost.

Additionally, each network connection implements its own listener thread that helps to guarantee that the packets are received as quickly as possible from the network

Delay Compensation:

The delay compensation system designed for the final sailing game follows the same general design as the prototype combining the dead reckoning and short circuiting techniques for delay compensation.

As the game is designed the client sends the client sail and rudder commands on every update it does. Along with every update it attaches a local timestamp. When the server receives the packet from the client it replies with the state of the system, collapsed into a string, and attaches the most recent client timestamp to the packet.

Thus, at every update the client side delay compensation system receives either the most recently arrived packet or nothing at all. Also, at each update, the local state of the sail and rudder controls are supplied to the delay compensation class (the same controls that are sent to the server). At startup, the delay compensation class creates a new buffer of fixed size (can be arbitrarily large but for most practical delay scenarios the default size of 250 is adequate). This buffer itself consists of three arrays: two integer arrays and one long array (see Figure 4 below). At each update of the client, the control inputs along with the timestamp for when they were entered are stored in the buffer at the current write index.

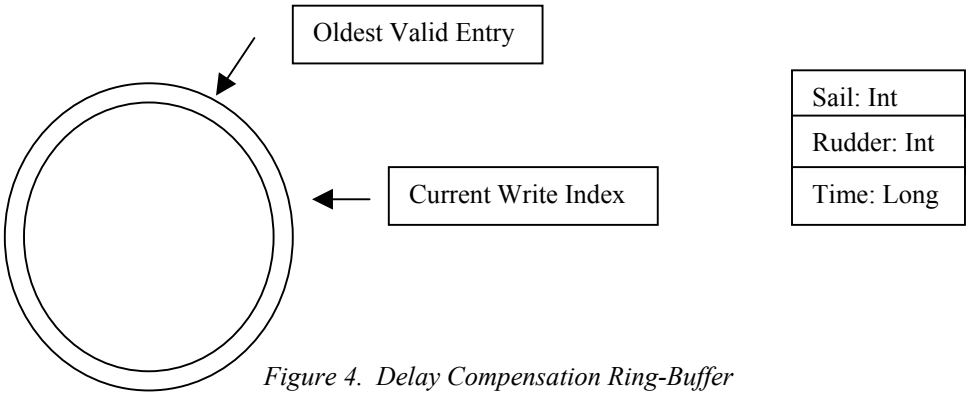


Figure 4. Delay Compensation Ring-Buffer

The two integer pointers, “Oldest Valid Entry” and “Current Write Index” define the currently used portion of the buffer. The first pointer indicates the index at which the oldest, not yet acknowledged, input resides. The second indicates the current write position index. As the game progresses the used portion rotates around the ring and grows and shrinks depending on the delay experienced by the client.

When there is a latency spike the used buffer automatically grows and increases the length of the extrapolation (the front continues to grow while the rear index remains the same) . If later, latency falls the used buffer decreases in size and the extrapolation gets shorter once again (the back index moves forward). In this way the delay compensation buffer automatically and instantly adjusts to changing network conditions. Additionally, because the acknowledgements are time based the strategy is minimally sensitive to packet loss as the rear pointer is skipped ahead as far as the acknowledgement goes.

When the delay compensation is activated the incoming state data is loaded into the same physics class as generated it at the server. The delay compensation class then uses the physics class to evolve the system forward applying all the buffered controlled inputs in order from the oldest valid entry up to the most

recently written. At each extrapolation step the player's boat controls are set to the values from the buffer and the system is evolved forward by the difference in time between the current input's timestamp and the next input's timestamp. This is repeated until all inputs have been applied. The evolved state is then passed out of the delay compensation class to be rendered.

.

Overall this delay compensation strategy has been found to work quite well. Having been tested with delays as high as 1500ms the system performs well with minimal rendering errors and a generally smooth display of new data.

Clocking/Timing:

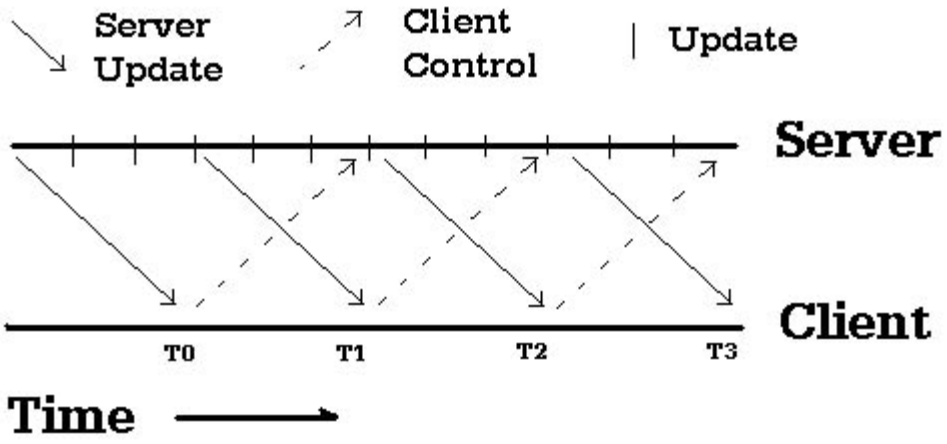


Figure 5. Game Timing

The issue of update clocking was one of the first problems encountered during development of this game. A variety of different interesting effects were noticed when thread.sleep() was tested on different operating systems (Windows: NT,XP,ME, Solaris). These differences seem to be based primarily on the different scheduling algorithms used by the operating systems as well as the processing power of the system hardware. The bottom line however, is that the code needs to function properly given uncertain update timing that it experiences on different systems.

Thus, the system is designed with the client using a timer thread to schedule update events which, although varying from system to system, was found to perform in a reasonably stable manner on any given system. The server on the other hand takes the strategy of using short thread sleeps and basing all the updates it does on the actual amount of time that has elapsed, as defined by the System.currentTimeMillis() function of Java.

Thus, when a game is in progress the server updates the system as often as it can with the relative change based on actual elapsed time between cycles (see figure 5). Then, when ever a control/clocking input arrives from the client the server sends out a copy of the game state to the client. In this way, the client automatically synchronizes itself with the server by clocking its own updates while the server updates as continuously as is possible and responds as quickly as it can when inputs arrive.

Additionally, the game server keeps track of the average rate at which the client is clocking it and, if the client does not clock it at the expected time, will send out packets on its own to keep the network 'pipeline' full of updates in the event that there was simply a packet loss or latency spike causing the missing client update(s). However, if the server doesn't hear from the client at all for a long period of time (four seconds) it assumes the client has quit or been disconnected and resets itself to accept a new client connection.

Animation/Rendering:

This aspect of the game design relates back in part to process priority issue mentioned above. The Python prototype does all of its animation by drawing pre-made images to the screen at defined positions. It uses a double buffer in order to provided smooth animation and does so with no noticeable performance hit (updating at 10ms intervals). Consequently the first version of the Java game was designed to work in much that same way. Overall, the strategy worked quite well on the development system. However, when tested on other systems it was found to run quite differently. The rendering was taking longer than the basic update rate of the system (20ms) and consequently led to a generally poorer animation quality.

The reason for this seems to be that with the OS on the development system (Windows ME) the process that is in focus (top window) seems to be given generally higher priority and thus was rendered more rapidly. However, it is necessary for the game to run effectively on most all platforms and, as such, modifications needed to be made. Through testing it was determined that most of the rendering time was being spent drawing the images for the game. Thus, to help improve efficiency, the majority of the images were replaced with polygons that are drawn much more efficiently. Unfortunately, it was found that the biggest performance hit was being caused by the double buffering system itself.

The concept of double buffering is fairly simple. Because drawing is not instantaneous, doing so directly to the video buffer causes the actually drawing of the scene to be visible to the user, which is not desirable because it causes noticeable flicker. With double buffering, as the name implies, all the drawing is first done to an off-screen buffer and, when drawing is complete, the image is "blitted" to the screen (blit stands for bit block line transfer, a method of buffer copying).

In Java the standard method for doing such double buffering is to create an Image object the same size as the game screen and then to get Graphics on this image. Once drawing is complete the entire Image is drawn to the frame buffer using the drawImage() method. However, while drawing to the image through the Graphics object is very efficient the final drawImage() called to paint the buffer is very costly. (Java doesn't seem to have implemented a more efficient method than drawImage for blitting.)

Thus, in addition to the improved efficiency for actual drawing, it was necessary to add a second display mode with a much smaller screen size (1/4) to save on double buffering costs for slower systems or those that give the process a lower priority by default.

The actual task of rendering was assigned to a specific rendering class in which each aspect of the game animation was given its own function. This makes the game optimally adjustable and the code much easier to modify. The game is designed such that the player's boat always remains at a fixed center point on the screen. When the boat turns it rotates about this point. To indicate position and velocity a grid system is drawn over the ocean that represents an absolute coordinate system. Thus the boat's velocity is indicated by the movement of the grid lines. For long-range position awareness the user has a mini-map which scales up and down according to the player's distance from the target buoy. This way the player never sails out of "sight" of the buoy and can steer their boat towards it. Additionally, when the player gets close enough to either the buoy or the opponent the player can see them in the main view at their actual position. (The buoy being represented by a red square and the opponent's boat represented by a larger, green square)

Below are the class diagrams for game client and server. To view the full source code and documentation please view the accompanying CDROM.

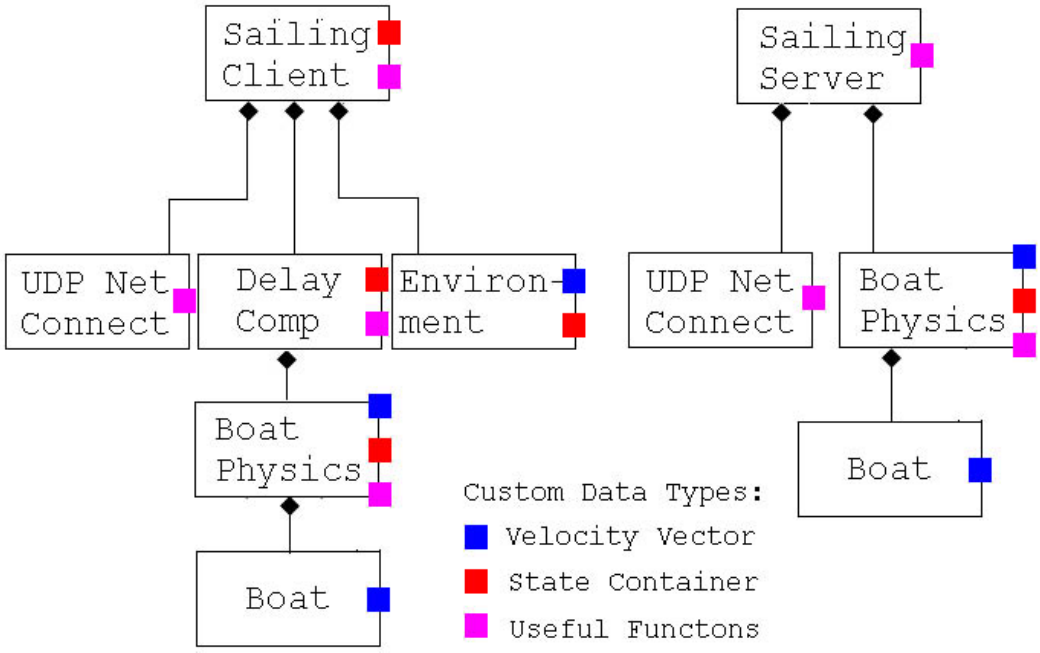


Figure 6. Software Class Diagrams

4. Testing and Results

For the purposes of these tests all references to latency refer to the amount of time required for a packet to be sent from the client to the server and back again, one full Round Trip Time (RTT).

All tests run to study the effects of delay on the game client and the effectiveness of the delay compensation mechanisms developed for this project utilized a 'Middleman' that sat in between the client and server to simulate network induced latency. This script was developed using Python and was found to consistently generate packet latencies within +/- 5ms of the target latency. Additionally, for the tests of the sailing game, a 100Mbps Ethernet LAN was used to connect the client and server machines and was found to generate negligible latency between client and server.

Prototype Testing:

These tests were performed on a 1.2Ghz Athlon with 256MB of memory, running the Windows ME operating system. The client, server, and middleman all run on this machine. Each of the data points below are based on the score after two minutes of play by an experienced human user. The data points each represent the average score of four consecutive games with the same testing scenario.

There were seven latencies tested - 20, 40, 100, 200, 400, 800, and 1600ms. Additionally, the first set of tests was run with the random wind disabled while the second was run with it enabled.

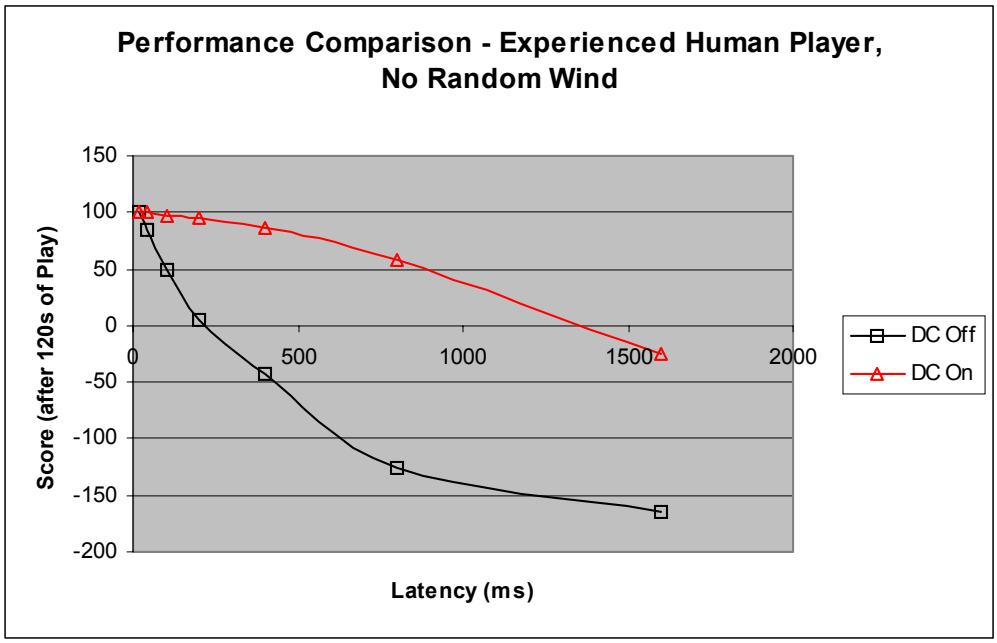


Figure 7. Comparison of Performance of Human Player with Delay Compensation and Without.

As described earlier, the rules of this game dictate that the player receives 1 point for each obstruction successfully avoided while loosing 4 points for each obstruction collided with. The score of 100 seen above in the low latency trials represents approximately the maximum score that a player could receive in the course of a two-minute game. The low score seen in the high latency trials (-165) is the result of colliding with 30-40% of the oncoming obstructions and is approximately equivalent to 'parking' the vehicle and allowing all obstructions to collide with it. As can be seen from the data the positive effect of the delay compensation is quite substantial. In order to perform well this game requires the user to maintain very precise control of the vehicle and, even an experienced player requires fairly rapid feedback in order to issue appropriate thruster commands. The delay compensation techniques used allow for control feedback which aids in control stability and helps to prevent oscillations of over control which occur commonly without compensation and frequently lead to destruction of the vehicle. Additionally, the forward extrapolation of obstruction position allows for more effective maneuvering. These two factors combine to make the compensated game almost as playable at high latencies as it is at lower latencies. The primary factor limiting this (and causing the deterioration in performance) in the compensated trials is simply the timing of the game. At 1600ms the faster moving obstructions appear to spawn at the very bottom of the screen (the position they are likely to be in when your control inputs reach the server). This makes avoiding them nearly impossible even if the player is wisely playing towards the bottom of the screen. If they are playing further up they will likely collide with the obstruction before they even see it. This highlights one of the fundamental limitations of any client-side delay compensation mechanism. Rapidly evolving events can occur before a distant client can respond or leave then barely enough time to respond.

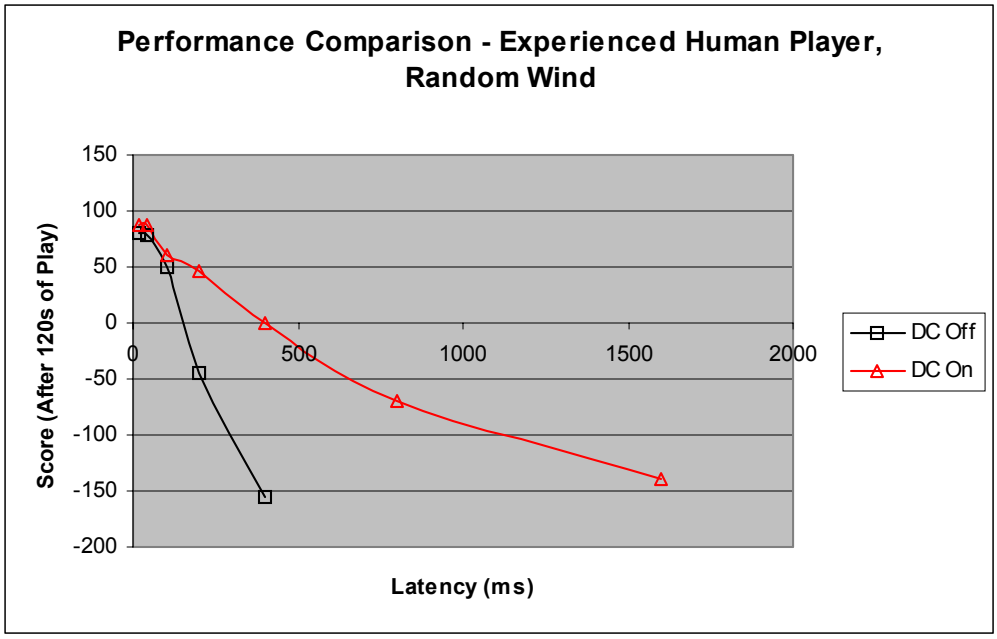


Figure 8. Comparison of Performance of Human Player with Delay Compensation and Without (Random Wind Activated)

As compared to the trial above the performance of the player deteriorates much more rapidly for both the compensated and uncompensated trials. In the case the uncompensated trials the 800 and 1600ms scenarios could not even be measured because the vehicle was destroyed more than three times in the two minutes allotted for the test. The addition of random wind adds a dimension to the game that is more critically timed and less predictable than already exists in the form of the oncoming obstructions. The random wind buffets the vehicle quickly knocking it into obstructions and frequently the terminal walls of the environment. Because the event is completely unpredictable there is nothing that the client side delay compensation mechanism can do in many circumstances as the event has occurred by the time the first news of it has reached the client or will occur before the client can respond. In the case of this trial the forward extrapolation of the DC has the effect of 'teleporting' the vehicle across the screen to its newly predicted position when first news of a wind perturbation arrives (the prototype uses a zero-order convergence scheme). This generally then allows the client to quickly fire thrusters and avoid collision with the wall but does nothing to prevent collisions with any obstructions that the vehicle has passed through in the meantime.

These trials, though limited in scope, demonstrate both the abilities of the client side delay compensation mechanism as well as some of the fundamental limitations of such a strategy. However, it is effective as a proof of concept for the strategy to be implemented in the final game.

Final Game Testing:

These tests were performed utilizing two machine connected via a 100Mbps Ethernet as described above. The server machine is a 2.2Ghz Pentium 4 with 680 MB of memory, running Windows XP Professional. The machine hosting the client and the middleman is a 1.2Ghz Athlon with 256MB of memory, running Windows ME. Each of the data points below are based on the time performance of two identical AI's on a fixed course consisting of a single buoy 1000 feet distance to which the agents race. The data points each represent the average of 15 separate trials under each testing condition. In each case one agent runs at the

client and is subject to the effects of the latency and any delay compensation. The other agent runs at the server and experiences no latency effects.

There were eight latencies tested - 0, 50, 100, 200, 300, 700, 1100, and 1500ms (a latency of 0 indicates that the middleman immediately forwards all packets it receives). Tests were run with several different environmental scenarios that remain fixed for each trial. The details of each will be listed with the data.

The values in the charts that follow reflect percent difference between the server agent and the client agent in the time required to complete the race. For instance, a value of 0.5 means that the client AI required 50% longer to complete the race as compared to the identical AI running on the server.

Two different agents are used in the tests below. The Simple AI uses a feedback based decision mechanism for adjusting the sail which adjusts the sail and examines the boat's speed to determine if the change was harmful or beneficial thus playing much like an inexperienced human player might. The Optimal AI calculates the optimal sail position given the boat's velocity and the wind velocity and implements it. Optimal sail position is calculated as follows:

(10) $\theta = \rho/2$

θ is the optimal sail angle (for boat velocity) and ρ is the angle between the apparent wind and the axis of the boat, U and CL respectively (see figure 3 in the preceeding section).

Both agents share a simple, feedback-based rudder control mechanism that simply steers the boat towards the buoy.

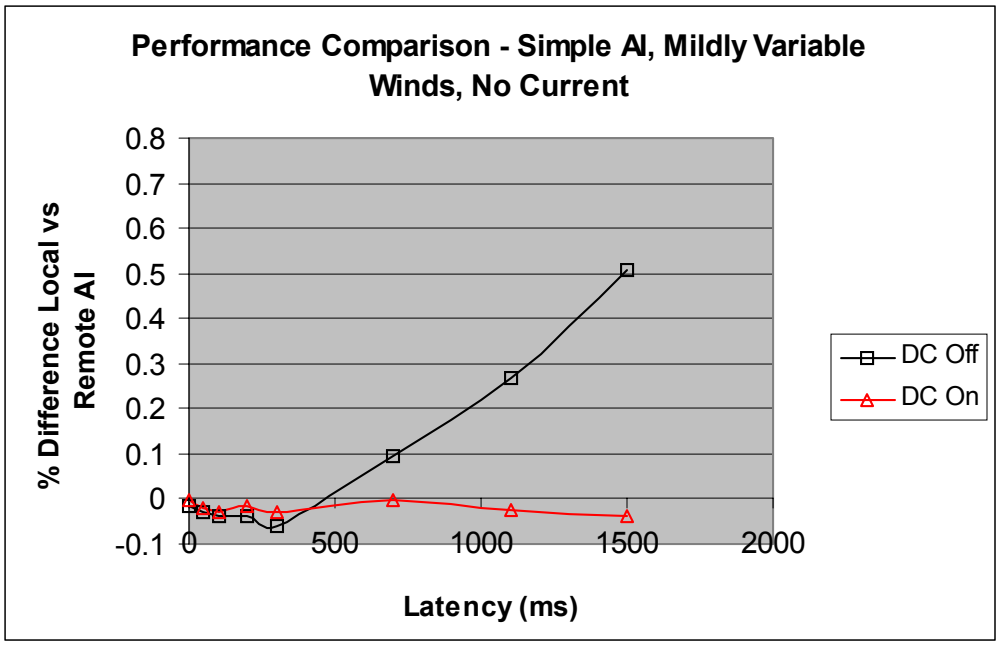


Figure 9. Comparison of Time Performance of Synthetic Player with Delay Compensation and Without.

The mildly variable wind is at 20 knots and shifts 90 degrees every 20 seconds, from W to S and back again.

This test demonstrates the effectiveness of the delay compensation in allowing for feedback based sail control even at very high latencies. Interestingly, both the compensated and non-compensated client agents slightly outperform the server agent in the lower latency trials. This seems to be primarily the result of the way that the simple AI functions. A small amount of delay does not hinder it but actually allows it to

adjust to wind shifts slightly more effectively. However, at higher latencies the feedback based sail control performs much worse and leads to ineffective sail control. Additionally, the client also tends to over steer on the limited rudder control that it needs to do which is to be expected as it also depends on feedback.

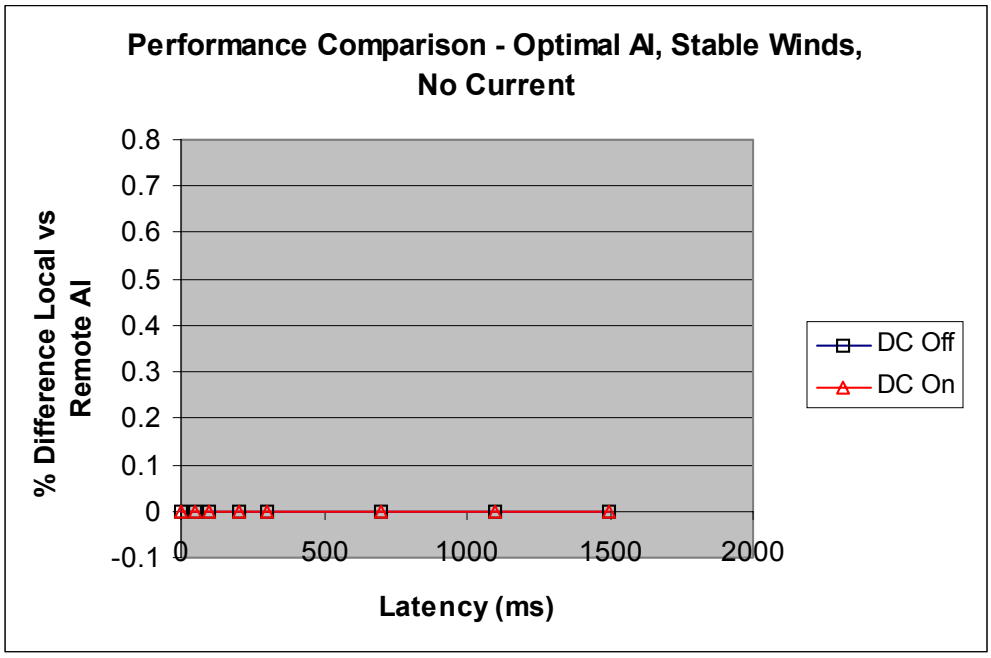


Figure 10. Comparison of Time Performance of Synthetic Player with Delay Compensation and Without.

The stable wind is at 20 knots from the W and does not shift throughout the course of the trial.

As is very apparent, in this scenario the client side agent is at no disadvantage due to the latency that it is experiencing. As a result, the delay compensation has no effect. Because the server side agent is only able to make control inputs at the same rate as the client agent it does not even get an initial jump on the client agent. They remain completely matched throughout the entire trial. Because only the initial control decision is required the latency experienced by subsequent control inputs has no effect.

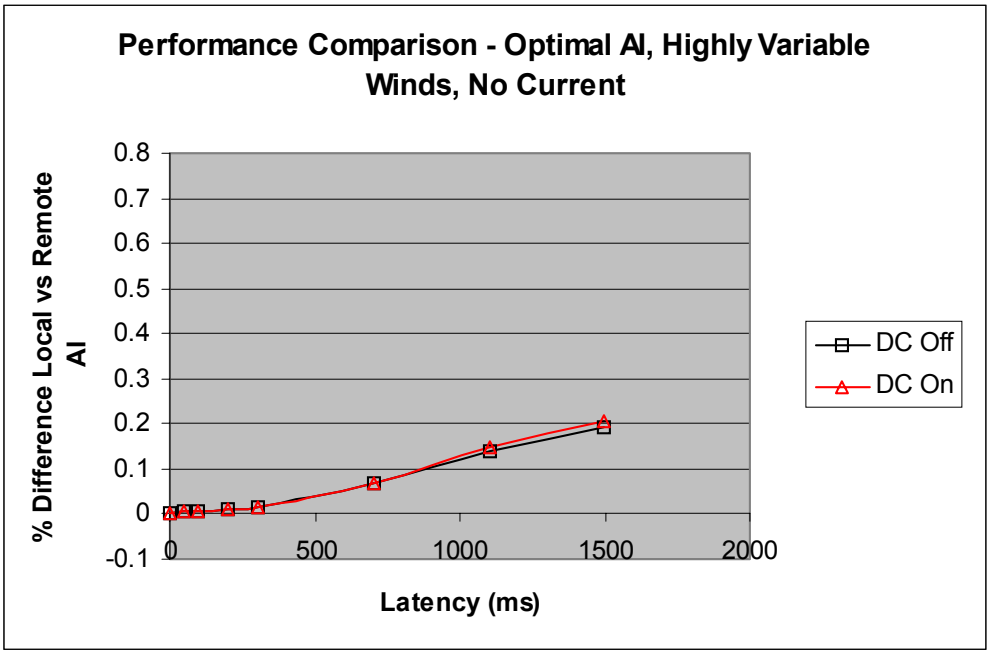


Figure 11. Comparison of Time Performance of Synthetic Player with Delay Compensation and Without

The highly variable wind is at 20 knots and shifts 90 degrees every 5 seconds, from W to S and back again.

Similar to the above trial, the delay compensation does not have a significant effect. However, the client side agent is suffering from the latency on its connection. Because the wind shifts are not at all predictable to the client (similar to the random wind in the prototype game) there is nothing specifically that the delay compensation mechanism can do. Every time the wind shifts the server agent is able to immediate adjust its sail to achieve the new optimal position. However, the remote agent, even when it responds with the correct sail position immediately, must wait one full RTT before it can make the same adjustment. With a rapidly shifting wind these effects begin to become quite significant.

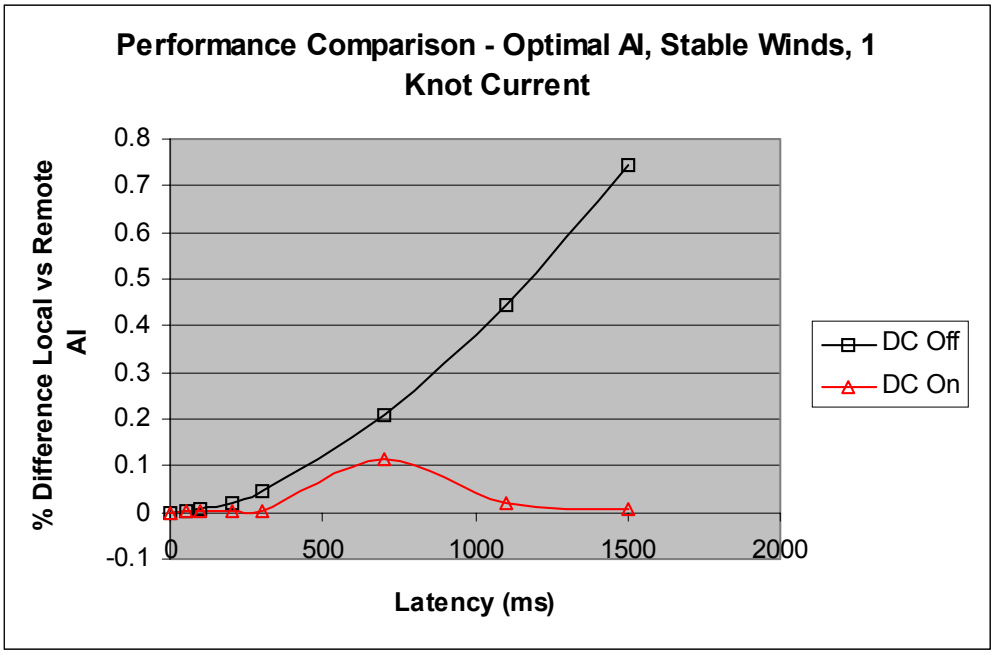


Figure 12. Comparison of Time Performance of Synthetic Player with Delay Compensation and Without

The stable wind is as described above and the current is at one knot from the S.
The effect of the current in this trial is to force both agents to use their rudders more extensively. As can be seen, the feedback based control of the rudder suffers as latency increases. As latency increases so does the tendency to over steer. The lack of feedback causes the AI to over control in one direction and then in the other leading to a steering oscillation that slows the boat. However, human players would quickly adapt to this. This type of control is much less vexing for a human as it is fairly simple to predict the turning rate of the boat given a little experience. An experienced human play can generally achieve the desired steering orientation with 3-4 adjustments without aid of control feedback. While this is worse than the average of one adjustment with control feedback the practical difference in a time trial would be fairly small. Assisted feedback is much more necessary in a more complex or less intuitive system such as the jet car from the prototype.

Also apparent is that the immediate feedback provided to the agent by the delay compensation allows for effective rudder control. Curiously, at 700ms the agent's behavior deteriorates by about 10% as compared to the server side agent. Investigating further, it seems that the effect drops to about half at 600ms and 800ms and is completely absent at 500ms and 900ms. Similar to the slight AI difference noted in the first test this seems to be the result of odd interaction between the agent's behavior and the delay compensation

system. The agent gets into a pattern of over control similar to that of the uncompensated agent but more chaotic. The exact mechanism of the interaction is still not clear.

The tests of both the prototype and the final game clearly show the benefits of a client side delay compensation based on state extrapolation and short circuiting of client input. They also show that the method has some clear limitations and that it shares limitations with any technique that would be applied on the client end of the connection.

5. Conclusion

To assess the effectiveness as well as the limitations of different delay compensation techniques two demonstrative network games with different game dynamics were developed. For each, a delay compensation system was developed which utilized two techniques in particular: dead reckoning and input short circuiting. Each game was then tested to ascertain the effectiveness of the compensation as well as its weaknesses through the use of specific performance measures as well as subjective evaluation.

The results of the testing show that the techniques, when combined, have the potential to be quite effective, even in very high latency situation. However, they also show that for unforeseeable events or those with critical timing, client side delay compensation is fundamentally limited. In these scenarios a different form of delay compensation is clearly needed.

This project has been successful in demonstrating the effectiveness and benefits of client side delay compensation as well as some common issues and fundamental limitations thereof. Finally, the framework developed will provide a stable platform for further study of the field including potentially such other techniques as contingency control and server side latency reduction as well as extensions to the techniques used such as improved game state convergence techniques.

References:

[1] The Everquest website can be found at: http://everquest.station.sony.com/

[2] Wu-chang Feng, Francis Chang, Wu-chi Fend, and Johathan Walpole. Provisioning on-line games: A traffic analysis of a busy Counter-Strike server. In Proceedings of the Second Internet Measurement Workshop pages 151-156, 2002.

[3] "Institute for National Measurement Standards - National Research Council", 03/20/03, National Research Council Canada, http://inms-ienm.nrc-cnrc.gc.ca/time_services/network_time_protocol_e.htm

[4] Jolt Online Gaming Ltd., http://www.56k.jolt.co.uk/index.php?page=dod§ion=faqdod.html

[5] Kazuhiro Kosuge and Hideyuki Murayama. "Teleoperation via Computer Network", Electrical Engineering in Japan, Vol. 124, No. 3, 1998.

[6] Jay Lyman. "Report: Broadband Hits the Mainstream", March 6, 2002, NewsFactor Network http://www.newsfactor.com/perl/story/16629.html

[7] Chris Morris. "Everquest – with Wookies: Star Wars enters the persistent world gaming space – but will it make a difference?", June 26, 2003, CNN Money, http://money.cnn.com/2003/06/25/commentary/game_over/column_gaming/

[8] Romi Nijhawan. "Neural delays, visual motion and the flash-lag effect", TRENDS in Cognitive Sciences, Vol. 6, No. 9, September 2002.

[9] Romi Nijhawan and Kuno Kirshfeld. "Analogous Mechanisms Compensate for Neural Delays in the Sensory and Motor Pathways: Evidence from Motor Flash-Lag", Current Biology, Vol. 13, April 2003, pp. 749-753.

[10] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In the 12th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), 2002

[11] Lothar Pantel, Lars Wolf: "On the Suitability of Dead Reckoning Schemes for Games", First Workshop on Network and System Support for Games (NetGames2002), April 16-17, 2002, Braunschweig, Germany.

[12] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. A Review on Networking and Multiplayer Computer Games. Turku Center for Computer Science, 454, April 2002.

[13] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. "Aspects of networking in multiplayer computer games", The Electronic Library, Vol. 20, No. 2, 2002, pp 87-97.

[14] Andreas Thorstensson "Half-Life Netcode Explained", December 23, 2001, http://www.geekboys.org/docs/Half-Life%20Netcode%20Explained.doc

[15] Jose Pable Zagal, Miguel Nussbaum, Ricardo Rosas: "A Model to Support the Design of Multiplayer Games", Presence, Vol. 9, No. 5, October 2000, pp. 448-462.