Multimed Tools Appl (2009) 41:305–331
DOI 10.1007/s11042-008-0230-3

Hack-proof synchronization protocol for multi-player online games

Yeung Siu Fung · John C. S. Lui

Published online: 7 October 2008

Abstract Synchronization protocols based on "dead-reckoning" are vulnerable to a popular type of cheat called speed-hack. A speed-hack helps a cheater to gain unfair advantages by essentially speeding up the actions of the avatar controlled by the cheater, so that the cheater can move, explore and gather items faster than honest players. This paper presents a novel version of a dead-reckoning protocol that is invulnerable to speed-hacks. Existing games based on dead-reckoning can easily be modified to use this hack-proof dead-reckoning protocol and how the protocol works on both client-server architecture and peer-to-peer (P2P) architecture will be demonstrated in this paper.

Keywords Cheat prevention · Multiplayer online game · Speed-hack

1 Introduction

Modern multi-player online games are popular and attractive because they provide a sense of virtual world experience to users: players can interact with each other on the Internet but perceive a local area network responsiveness. To make this possible, most modern multi-player online games use similar networking architecture that aims to hide the effects of network latency, packet loss, and high variance of delay from players. Because real-time interactivity is a crucial feature from a player's point of view, any delay perceived by a player can affect his/her performance [16]. Therefore, the game client must be able to run and accept new user commands continuously regardless of the condition of the underlying communication channel,

Y. S. Fung · J. C. S. Lui (✉)
Department of Computer Science & Engineering, The Chinese University of Hong Kong, Ma Liu Shui, China
e-mail: cslui@cse.cuhk.edu.hk

Y. S. Fung
e-mail: sfyeung@cse.cuhk.edu.hk

and that it will not stop responding because of waiting for update packets from other players. To make this possible, multi-player online games typically use protocols based on “dead-reckoning” [5, 6, 9] which allows loose synchronization between players.

However, dead-reckoning protocol is susceptible to some security attack or exploitation. In particular, the type of cheat that exploits this vulnerability is called speed-hack [3] and it has become so widely available and easily accessible because the implementation of a speed-hack is very simple. Speed-hack cheats exist virtually in all popular commercial multi-player online games [15]. Existing countermeasures target on the cheats themselves, i.e. they scan for and block any known cheating software, or observe any abnormal network traffic and ban that player from the game. These methods cannot safeguard against all potential speed-hacks, and honest players may be accidentally recognized as cheaters due to the false positive nature of detection software.

Figure 1a and b are screenshots from a popular commercial massively multiplayer online role-playing game (MMORPG) called *World of Warcraft*. In an MMORPG, each player controls the action of an avatar inside a virtual world. For example, the player can move the avatar from one place to another, gather different items by moving the avatar towards them, use different weapons and magic spells to attack other avatars and move the avatar to avoid being attacked. Therefore, a player with a fast moving avatar has definite advantages over players with slower moving avatars. Normally, an avatar can move faster only after it has obtained some particular items. However, when using speed-hack an avatar can move arbitrarily faster. Figure 4 illustrates the effect of using speed-hack in an MMORPG. In Fig. 4c and d, player $\mathcal{P}$ is using a speed-hack. We can see that $\mathcal{P}$’s avatar moves faster than that in Fig. 4a and b.

This paper presents a novel dead-reckoning protocol that is immune from the speed-hack cheats. We assume the cheater can modify any binary code or game data, e.g. the OS’s clock speed, the memory data, the incoming and outgoing packets, etc. However, we will prove that the invulnerability of our protocol does not depend on what the cheater can do and even the cheater can modify the outgoing packets,

(a) (b)

Fig. 1 **a** Some avatars moving inside a virtual world, each of them is controlled by an individual player. **b** Several avatars attacking each other using different weapons only very limited advantages can be gained. Since our protocol is based on the conventional dead-reckoning protocol, existing games can easily be modified to become resistant to speed-hack. Our protocol can be adapted to both client-server architecture and P2P architecture in a very similar way.

The remaining parts of this paper are organized as follows. In Section 2, background informations on dead-reckoning and speed-hack are presented. In Section 3, we present our hack-proof dead-reckoning protocol and we proof the invulnerability of our protocol. In Section 4, we present the simulation results of our protocol. Related works in related fields are presented in Section 5. Section 6 concludes.

2 Backgrounds

2.1 Dead-reckoning

In most multi-player online games, before the game starts, a player is able to select among a number of avatars, each having different abilities and characteristics, such as appearance, health point, magic point, speed, etc. When the game begins, or when a player joins an existing game session, the avatar will be given an initial speeding capability. This speeding capability may be different according to which avatar the player has chosen. This speeding capability limits how fast the avatar can move in the virtual world. The avatar can be moving or stationary at any moment during the game, but while it is moving, its speed is fixed. Throughout this paper, we call this speed the *legal speed* of the avatar. The legal speed of the avatar can be changed when the game is in progress. It can be achieved by either gaining enough experience points to upgrade the avatar's abilities or by obtaining special items which will affect the avatar's abilities. In a client-server architecture, the change of an avatar's legal speed needs to be granted by the game server and the game server will broadcast the new legal speed of that avatar to all clients. In a peer-to-peer architecture, the change of an avatar's legal speed needs to be verified by all peers. For example, all peers must agree that the avatar has obtained the specific item successfully and so they will update its legal speed accordingly. Therefore, the change of the legal speed of an avatar works under a tight synchronization requirement.

Synchronization protocols based on dead-reckoning are commonly used in multiplayer online games because they do not require synchronization at every state change. In a game using dead-reckoning, each client sends update packet to the server (in client-server architecture) or to the peers (in peer-to-peer architecture) at a constant interval called timeframe, instead of at each state change. An update packet consists of a timestamp of the game states and a dead-reckoning vector while a dead-reckoning vector consists of the current coordinates and moving direction of the avatar. Using the latest received update packet, each client can predict the movement of another player before the next packet arrives. When a new packet arrives, correction will be made if there is any deviation induced by the prediction. Therefore, players do not maintain strictly synchronized views at every state change. Instead, their views will only be re-synchronized each time when the synchronization takes place.

An important advantage of this loose synchronization is that the rate of graphics rendering at each client side can be made independent to the rate of synchronization.

In order to produce smooth display, the graphics should be rendered at a rate no less than 30 frames per seconds (fps). However, synchronization in MMORPGs typically takes place in a much slower rate. This is because synchronization can consume a significant amount of processing power and network bandwidth the server since the number of connected clients are typically in the order of thousands. The situation is even more severe in peer-to-peer games, since IP multicast is still not yet widely available, a peer-to-peer game client may resort to sending separate update packets to every peer. Because of this, synchronizations in MMORPGs typically take place at a rate less than 10 updates per second, i.e. a timeframe of 100 ms. If a client only renders moving objects to their new coordinates each time when an update packet arrives, i.e. it renders the graphics at a rate of 10 fps, the animation will look choppy and jittery, which will definitely destroy the game's playability. However, under dead-reckoning, since prediction is carried out before any newer packet is available, each client can render the movement of objects at the fastest rate which only depends on the processing power of the client machine.

In order to predict an object's movement from its previous game states, simple linear extrapolation can be used. Using the dead-reckoning vector in the last received packet, the client can extrapolate a linear movement from the object's last known coordinates which head towards the last known direction. When a new update packet arrives, the accurate coordinates may be different from the current coordinates predicted by the extrapolation. Algorithms such as [1] and [11] can be used to hide the effect of any extrapolation error emerged in rendering the movements. Under the dead-reckoning protocol with the use of extrapolation, all clients can render the movement of all avatars at the fastest possible rate, which only depends on the computational power of the client side. If an update packet is late on arrival or is even lost, the graphics rendering will still not be affected and therefore smooth gameplay can be ensured.

2.1.1 Linear extrapolation

We give an example to illustrate a simple linear extrapolation algorithm. Referring to Fig. 2, when a client sends an update packet at time t_1, it is reported that avatar $\mathcal{P}$ is at (x_1, y_1) heading at an angle r. Before the next synchronization scheduled at time t_2 occurs, other clients render $\mathcal{P}$'s movement by linearly extrapolating the position of $\mathcal{P}$ based on $\mathcal{P}$'s dead-reckoning vector sent at time t_1, as follows:

$$\left.\begin{aligned} x(t) &= x_1 + (t - t_1) * \mathit{legal\ speed\ of}\ \mathcal{P} * \sin(r) \\ y(t) &= y_1 + (t - t_1) * \mathit{legal\ speed\ of}\ \mathcal{P} * \cos(r) \end{aligned}\right\} \text{for } t \geq t_1$$

Dead-reckoning protocol provides a means of loose synchronization among players. It is especially necessary when a massive number of concurrent players are interacting with each other. The larger the number of concurrent players the higher the change of having someone's update packet congested or lost in the network. Without dead-reckoning, at the end of each timeframe, all game clients must be halted and wait for update packets from all other players. This will cause significant amount of jitter to the graphics rendering and slow down the response to the player's control, and therefore implies unpleasant gaming experiences. However, by using dead-reckoning protocol, late arrived packets or lost packets can simply be ignored.

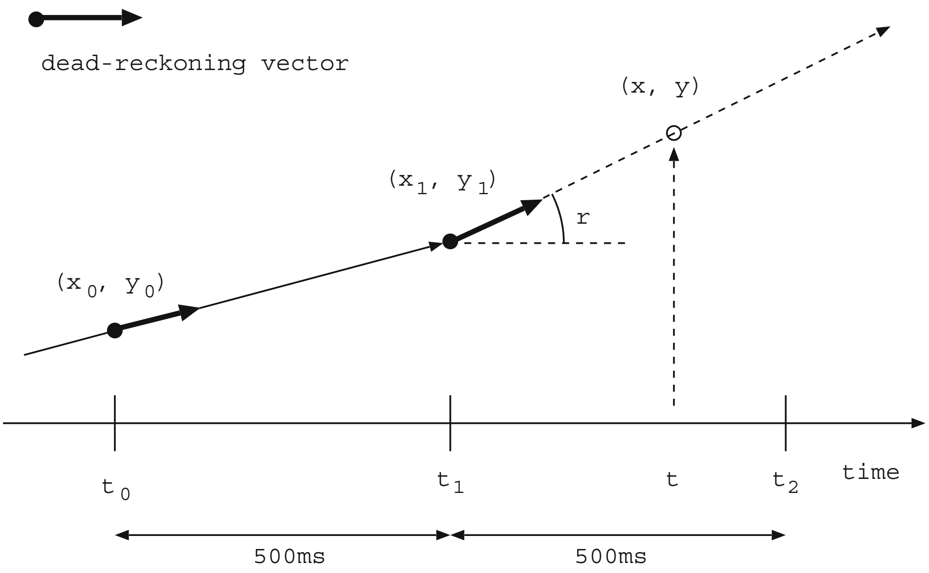


Fig. 2 Extrapolation of (x, y) from the latest dead-reckoning vector

To fill in the missing packets, extrapolation is used to predict the missing game states, therefore the game clients will never be required to halt at any circumstance.

2.2 Speed-hack

Dead-reckoning protocol is popular because of its advantage listed above, that is, all players can have a perception of smooth gameplay even though the underlying communication channel is in fact error-prone, congested and has high delay variance. However, it hints the potential vulnerability to a form of very popular and highly available cheat called *speed-hack*. When using a speed-hack, a cheater can speed up all movements of his/her avatar and thus gain an unfair advantage over other honest players.

A speed-hack essentially speeds up the timing of the cheating game client, and this can be done quite easily, especially under the dead-reckoning protocol. This is due to the fact that most of the game clients depend on a time source, such as software programmable timer or system library calls, to count the time elapsed and then applies it to the Newton's first law of motion to project the movements of moving objects in the virtual world. Here, we illustrate how most online games handle player movement. According to Fig. 3, the avatar is at position p_0 at time t_0. The player moves the avatar by clicking the mouse at the point d_0 in the virtual world. The game client then stores this coordinates into memory and initiates the avatar to move towards this destination. However, before the avatar reaches the destination d_0, if the player issues another mouse click at the point d_1 when the avatar is at coordinates p_1 at time t_1, the game client will initiate a new movement towards d_1 from p_1. Similarly, at time t_2, when the player issues another mouse click at d_2 before the avatar reaches d_1, therefore, the avatar will change its direction at p_2 and moves towards d_2. At any time t' after the player issues a destination point d_i at time t_i, but before the avatar

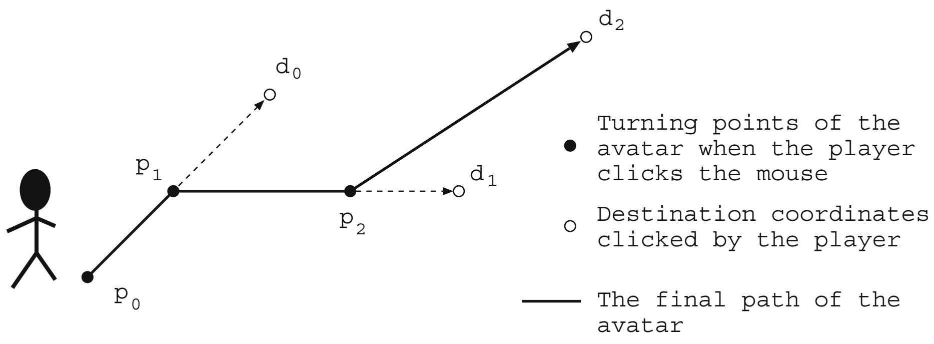

d0
d1
d2
p0
p1
p2
Turning points of the avatar when the player clicks the mouse
Destination coordinates clicked by the player
The final path of the avatar

Fig. 3 Movement of an avatar in typical massively multiplayer online games

reaches there from p_i, the game client will update the avatar's position as follows. Let T_j be the journey time of an avatar,

$$\begin{aligned} T_j &= \mathit{journey\ \ time} \\ &= \frac{\sqrt{(x_{d_i} - x_{p_i})^2 + (y_{d_i} - y_{p_i})^2}}{\text{legal speed of the avatar}} \end{aligned}$$

and the computation of the new coordinates will be:

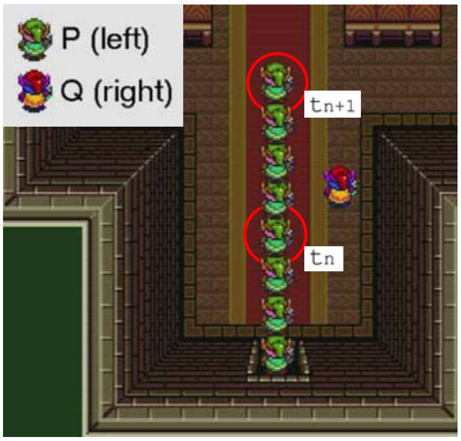
$$x(t) = x_{p_i} + (x_{d_i} - x_{p_i})\frac{t - t_i}{T_j}$$

$$y(t) = y_{p_i} + (y_{d_i} - y_{p_i})\frac{t - t_i}{T_j}$$

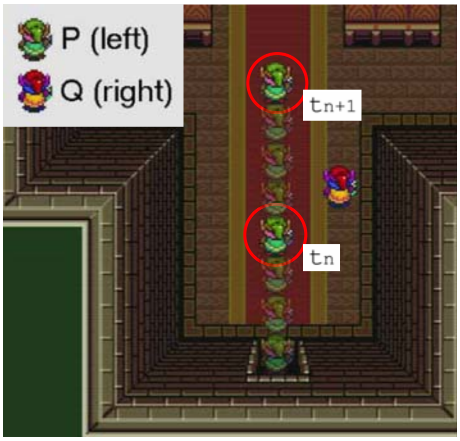
In order to speed up a game client, a speed-hack alters its own time source to count time faster, or intercepts the genuine time source and injects a malicious one that counts time faster, i.e. it makes the value of t advances at a faster rate. All local objects in the hacked game client will therefore move faster in the cheater's local view. Under dead-reckoning protocol, the game client simply reports in its update packet about the coordinates of the cheater's avatar computed in the cheater's local view. Upon receiving the cheater's update packet, a client will move the cheater's avatar to that new position as reported in the update packet. Therefore, all players will perceive that the cheater's avatar moves at a faster speed.

Figure 4a and b illustrate the views of two interacting honest players $\mathcal{P}$ and $\mathcal{Q}$ respectively. In the figures, $\mathcal{P}$'s avatar is moving upward while $\mathcal{Q}$'s avatar stays motionless. $\mathcal{P}$ sends two updates at time t_n and t_{n+1} respectively, giving $\mathcal{Q}$ the information to render the two opaque avatars corresponding to $\mathcal{P}$'s position at time t_n and t_{n+1} respectively. However, when rendering $\mathcal{P}$'s position between time t_n and t_{n+1}, where no exact information about $\mathcal{P}$'s position is available, $\mathcal{Q}$ extrapolates it from the position at time t_n to fill in the positions between time t_n and t_{n+1}.

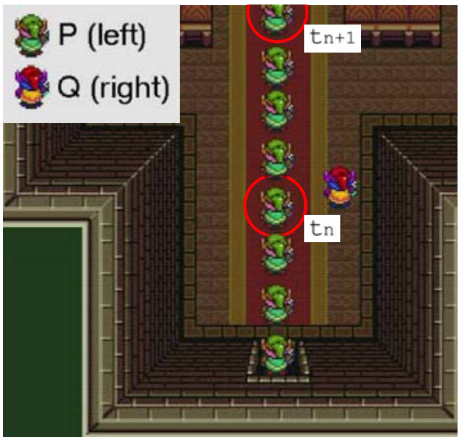
Figure 4c and d illustrate the views of two interacting players $\mathcal{P}$ and $\mathcal{Q}$ respectively, where $\mathcal{P}$ is using a speed-hack. In the figures, $\mathcal{P}$'s avatar is moving upward while $\mathcal{Q}$'s avatar stays motionless. The speed-hack speeds up $\mathcal{P}$'s game client so that $\mathcal{P}$ is able to move at a faster speed and therefore travels farther at time t_{n+1} compared to that in



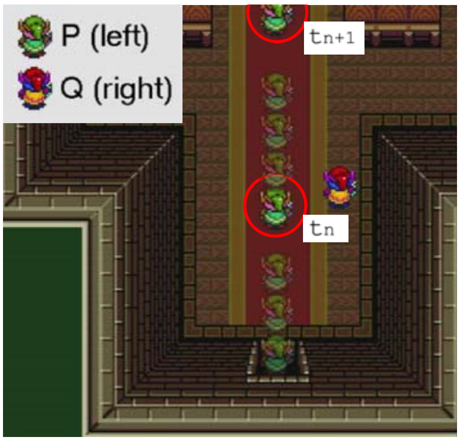
(a) P's own view, P is not cheating



(b) Q's view, P is not cheating



(c) P's own view, P is cheating



(d) Q's view, P is cheating

Fig. 4 Overlapped successive frames observed by two interacting players $\mathcal{P}$ (*left*) and $\mathcal{Q}$ (*right*) (**a**–**d**). Opaque avatars represent accurate positions given by the dead-reckoning vectors. Transparent avatars represent positions predicted by extrapolations

Fig. 4a. When synchronization takes place at t_{n+1}, $\mathcal{P}$'s dead-reckoning vector reports the same position as what $\mathcal{P}$ perceives locally. Therefore $\mathcal{Q}$ updates $\mathcal{P}$'s avatar to that farther position and therefore perceives $\mathcal{P}$'s avatar moving at a faster speed compared to the scenario shown at Fig. 4b.

3 Hack-proof synchronization protocol

In this section, we present a dead-reckoning protocol that is invulnerable to speed-hacks. The invulnerable protocol completely preserves the latency-hiding characteristic of conventional dead-reckoning protocol. Extrapolations are still allowed to smooth out the graphics rending under the enhanced protocol.

We first describe a baseline countermeasure to act against the speed-hack. Inspired by this baseline countermeasure, we can then propose a modified dead-reckoning protocol which is a slightly modified version of the conventional dead-reckoning protocol. The modified protocol is invulnerable to speed-hack; however, it cannot handle some synchronization scenarios which are common in real games. Therefore, we will propose another enhanced version of the invulnerable protocol which is based on the modified protocol but is more sophisticated and is able to handle all possible synchronization scenarios.

3.1 Countermeasure

The first countermeasure to act against speed-hack under dead-reckoning protocol is to verify the new coordinates of the avatar during each synchronization before accepting them so as to ensure that the avatar has only moved within a legitimate displacement since the last synchronization.

To verify the new coordinates stated in a dead-reckoning update packet, the server (or the peers) can use the elapsed time and the avatar's current legal speed to compute the maximum possible displacement of the avatar as

$$d_{\max} = (t_i - t_{i-1}) * \text{legal speed}$$

Under this simple approach, a game client can detect if a player is using speed-hack and hence restrict the movement of an avatar within its maximum possible displacement in each timeframe. To illustrate, we should have a look at Fig. 5. The cheater uses a speed-hack so that the avatar's displacement between each synchronization is larger than its maximum possible displacement $d_{\max}$ in the cheater's local

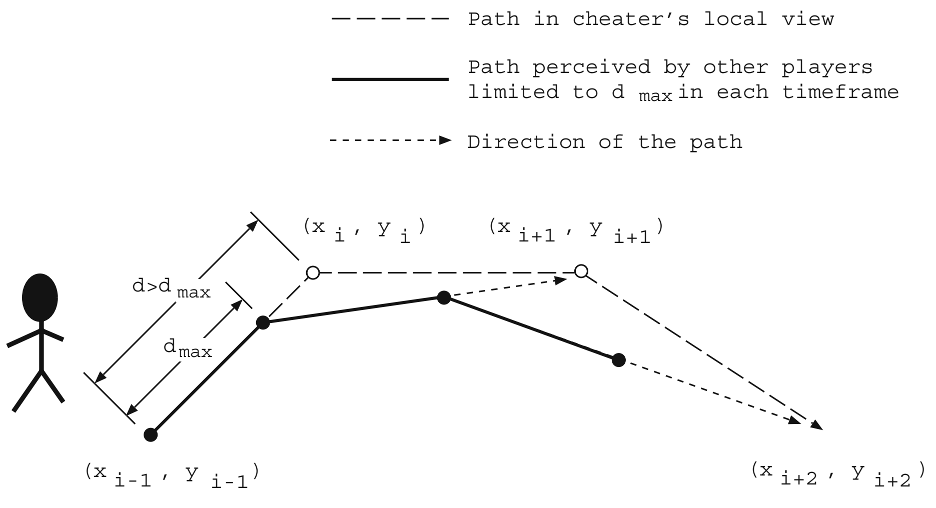

Path in cheater's local view
Path perceived by other players limited to d max in each timeframe
Direction of the path
(x i , y i)
(x i+1 , y i+1)
d>d max
d max
(x i-1 , y i-1)
(x i+2 , y i+2)

Fig. 5 Limiting the displacement of any avatar within its maximum possible value in each timeframe

view. However, when other clients receive the cheater's update packet, they will compute the displacement of the avatar of the last synchronization as

$$d = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

and conclude that

$$d > d_{\max}$$

If d is much greater than $d_{\max}$ in several consecutive timeframes, then obviously the player is using a speed-hack and the server can consider to kick that cheater out of the game. However, sometimes a cheater may only speed up a little bit just to gain an advantage over honest players. A conservative scheme is to accumulate the excess displacements over an extended period of time. For example, if an avatar moves on average 10% faster than its legal speed in a period of 10 s, then the player should be kicked out of the game. To avoid a cheater from gaining enough advantage within the grace period, such as successfully obtaining an important item because he/she moves faster than other honest players, we should limit the actual displacement of an avatar within each timeframe to its maximum value $d_{\max}$, and this is illustrated in Fig. 5.

Referring to the figure, it is seen that when the server (or the peers) verifies that the displacement of a certain avatar is larger than its maximum possible value $d_{\max}$, the avatar's position stated in the update packet will be ignored. Instead, the recipient will compute a restricted position along the same path but with a shortened displacement, by linearly extrapolating from the last synchronized position as follows,

$$\begin{aligned} x_i' &= x_{i-1} + (t_i - t_{i-1}) * \text{legal speed} * \sin(r) \\ &= x_{i-1} + (t_i - t_{i-1}) * \text{legal speed} * \frac{y_i - y_{i-1}}{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}} \end{aligned}$$

and

$$\begin{aligned} y_i' &= y_{i-1} + (t_i - t_{i-1}) * \text{legal speed} * \cos(r) \\ &= y_{i-1} + (t_i - t_{i-1}) * \text{legal speed} * \frac{x_i - x_{i-1}}{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}} \end{aligned}$$

3.1.1 Invulnerability

The only possible way for a cheater to spoof other players is by tagging a larger timestamp t_i in the latest update packet, resulting in a larger $d_{\max}$ for the cheater. However, the exaggeration in t_i is limited by the traveling time of the update packet from the sender to the server (or the peers) which is the network latency. For example, if a game client sends out an update packet at time t_i, and the network latency between it and the server is 20 ms. The server will receive the update packet at time $t_i + 20$ ms and the cheating client cannot replace the timestamp with a value larger than $t + 20$ ms or otherwise it will be detected and the packet may be treated as corrupted and simply being ignored. Moreover, when the next synchronization starts, the corresponding elapsed time will be counted from $t_i + 20$ ms and thus the exaggeration cannot be accumulated over time.

3.1.2 Handling missing packets

An important feature of the dead-reckoning protocol is that it allows packet loss. Therefore, the above countermeasure should also preserve this important feature. Assume that there are some missing packets before an update packet arrives with timestamp t_j, we can generalize the maximum displacement $d_{\max}$ used for the verification as

$$d_{\max} = (t_j - t_i) * \text{legal speed},$$

where t_i is the timestamp of the last valid dead-reckoning position received. If the position is verified to be invalid, the recipient will compute a restricted position by the generalized equations as follows,

$$\begin{aligned} x'_j &= x_i + (t_j - t_i) * \text{legal speed} * \sin(r) \\ &= x_i + (t_j - t_i) * \text{legal speed} * \frac{y_j - y_i}{\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}} \end{aligned}$$

and

$$\begin{aligned} y'_j &= y_i + (t_j - t_i) * \text{legal speed} * \cos(r) \\ &= y_i + (t_j - t_i) * \text{legal speed} * \frac{x_j - x_i}{\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}} \end{aligned}$$

3.2 Modified dead-reckoning protocol

Under the above countermeasure, all dead-reckoning vectors in update packets have to be verified at first. If a dead-reckoning position is found to be illegal, additional computation will then be required to adjust the position. In fact, we can modify the protocol so that we can eliminate any client from sending illegal dead-reckoning vectors out in the first place.

Instead of computing the maximum possible displacement and verifying the integrity of the coordinates in each synchronization, we modify the dead-reckoning protocol so that the position vector will not be transmitted directly. Synchronization parameters are computed from the position vectors and a recipient can reveal the position vector from these parameters. The computation of the synchronization parameters and the reverse computation do not depend on the timing of any single machine, but only depend on a global clock. We assume the cheater can modify any binary code or game data, e.g. the OS's clock speed, the memory data, the incoming and outgoing packets, etc. We will show that the cheater can only gain very few advantages under the modified protocol, and the advantages cannot be accumulated over time.

We assume that the game server and all game clients are synchronized to a global clock, the Network Time Protocol (NTP) [14] or similar protocols [18] can be used to achieve this purpose. A client must firstly be synchronized to an appointed NTP time source before joining a game session. Otherwise, the game server will reject the connection if the client's clock differs too much from the server's. During the game, the clients and the server are only required to synchronize with the NTP server at a moderate interval, but their clocks will be incremented locally. The synchronization is only used to ensure that each clock is always kept within an acceptable amount of deviation. Since the clock of an innocent client normally will not diverge from the NTP server significantly within several minutes, a synchronization interval of 1 min is typically sufficient.

Moreover, the invulnerability of our protocol does not depend on the strictly synchronized clocks. Instead, when a client's clock diverges too much from the NTP server, it will generate malicious timestamps in its update packets and the server and other clients will consider it as cheating. However, if the cheater tries to modify the packets to pretend generating correct timestamping, only very limited advantages can be gained under our protocol and the advantages cannot be accumulated over later updates. Therefore, under our protocol, a synchronized clock is not a requirement for the invulnerability but is only necessary for a client to manifest its honesty.

Here we present the details of our modified dead-reckoning protocol. Instead of providing the current coordinates directly in the update packet, several parameters are provided such that the recipients can compute the corresponding avatar's new coordinates accordingly. Attempts to modify these parameters will not give the cheater any advantage. The parameters are the *tangent of angle* r, where r is the angular coordinates of the current dead-reckoning coordinates with respect to the previous dead-reckoning coordinates. The parameter r is transmitted in the update packet in the form of (T_y, T_x) where $\tan(r) = \frac{T_y}{T_x}$. Therefore, we can simply take (T_y, T_x) as the offset from the previous dead-reckoning coordinates to the current dead-reckoning coordinates. Figure 6 shows an example that illustrates the idea. The avatar is at (x_n, y_n) at time t_n. When synchronization takes place at time t_{n+1}, and the avatar has moved to (x_{n+1}, y_{n+1}). The angular coordinates of (x_{n+1}, y_{n+1}) with respect to (x_n, y_n) is 45°. The parameter (T_y, T_x) is given by $(y_{n+1} - y_n, x_{n+1} - x_n)$ or (1, 1), since $tan(45°) = \frac{1}{1}$. Upon receiving this update packet, the server (or the peers) can

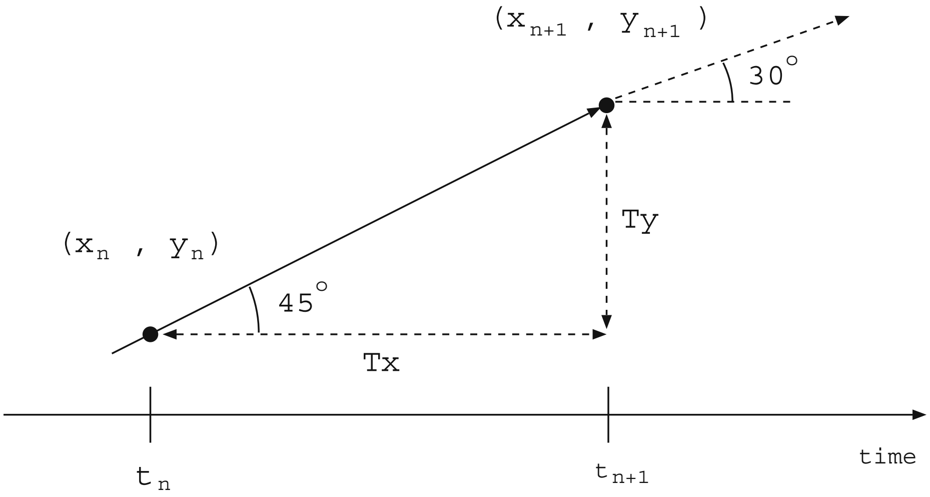


Fig. 6 The parameter (T_y, T_x) and the current direction when synchronization takes place at time t_{n+1}

compute that avatar's new coordinates from (x_n, y_n), the timestamp tagged on the packet and the avatar's current legal speed, as follows:

$$\cos(r) = \frac{x_{n+1} - x_n}{\text{legal speed} * \text{elapsed time}}$$

$$x_{n+1} = x_n + (\text{legal speed} * \text{elapsed time}) * \cos\left(\arctan\left(\frac{T_y}{T_x}\right)\right)$$

and

$$\sin(r) = \frac{y_{n+1} - y_n}{\text{legal speed} * \text{elapsed time}}$$

$$y_{n+1} = y_n + (\text{legal speed} * \text{elapsed time}) * \sin\left(\arctan\left(\frac{T_y}{T_x}\right)\right)$$

3.2.1 Invulnerability

Suppose a cheater wants to gain an advantage by maliciously tagging a larger timestamp on the packet, and he/she intents to produce a farther displacement from the above computations. The exaggeration in the timestamp is limited by the traveling time of the update packet from the sender to the server (or the peers) which is the network latency. Since an over-large timestamp can be detected easily as all machines are synchronized to the same global clock. Moreover, since at each update the elapsed time is computed against the timestamp of the previous update, the exaggerated elapsed time cannot be accumulated over time but will be bounded within only one single-trip latency in general. We will prove it in Section 3.5.

3.2.2 Extension

If the avatar does not have any movement since the last synchronization, then $(T_y, T_x) = (0, 0)$ can be used to indicate such a special case. However, this simple protocol can only express either completely motionless or completely nonstop movement in the whole timeframe. Every movement must start or stop at the beginning of a timeframe, and then keep moving or motionless until the next timeframe begins. This may be impractical for real games because it impedes the game's responsiveness to player's control. Hence, in the next section we propose an enhanced version of this invulnerable protocol which is more sophisticated in handling various situations.

3.2.3 Handling missing packets

The new dead-reckoning protocol still allows late packet arrival. Extrapolation is used to predict the avatar's movement until an update packet arrives, just as it is used in conventional dead-reckoning protocol. However, since the synchronization parameters in each synchronization is based on the previous synchronized position, any lost packets must be re-transmitted or otherwise the path of the avatar's movement cannot be reconstructed. A simple approach can be used to overcome this problem. When there is only a single packet being dropped, i.e. a sending client does not receive any acknowledgment until the next synchronization takes place, the client may simply re-transmit the last parameters along with the new parameters so that the recipient can compute the two latest dead-reckoning positions at once.

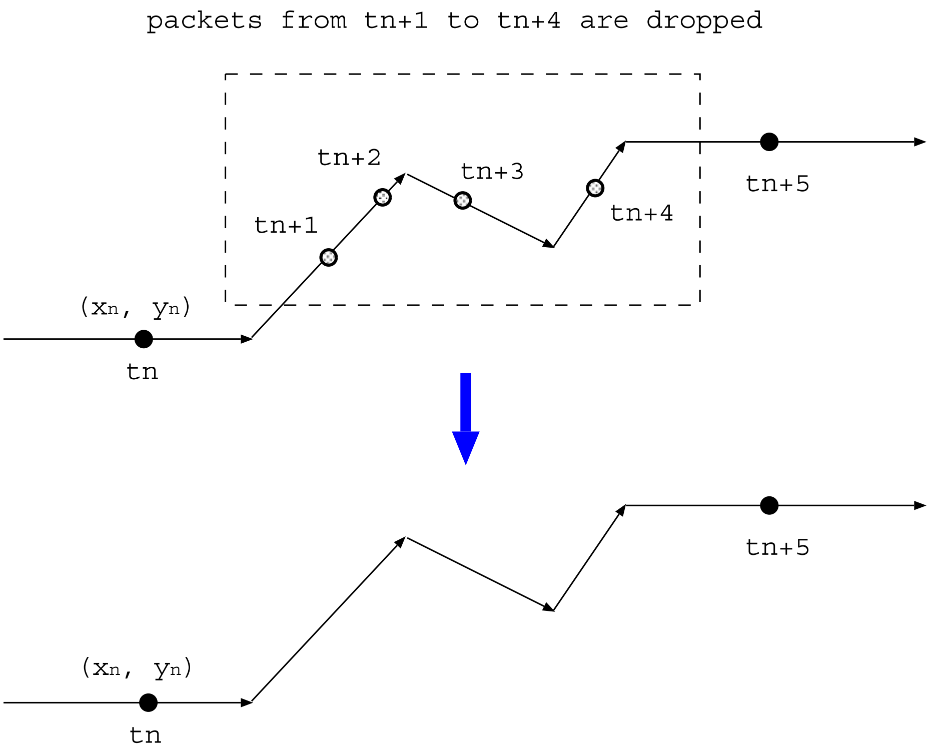


Fig. 7 When more than one packet is dropped, the sender includes an extra parameter $t_{\text{ack}} = t_n$ in its update packet and computes the synchronization parameters at t_{n+4} based on the latest synchronized position (x_n, y_n) at t_n

If there is more than one packet being dropped, i.e. the sending client does not receive any acknowledgment for several consecutive timeframes, re-transmission of all of the parameters may induce additional loads to the network. In this situation, the protocol have to allow packets to be dropped permanently. To realize it, the sender includes an extra parameter t_{ack} in its update packet, which is the timestamp of the latest acknowledged update packet. For example, in Fig. 7, the sending client does not receive acknowledgments of the synchronizations at t_{n+1} to t_{n+3}. At t_{n+4} the sender computes the synchronization parameters based on the avatar's position at t_n, so that the recipients can determine the corresponding dead-reckoning position based on the position synchronized at t_n. Therefore, all of the parameters in the dropped packets can be ignored.

3.3 Enhanced invulnerable protocol

In this section, we enhance the above invulnerable protocol so that it becomes more sophisticated and can tolerate malicious timestamping.

In each update packet, a game client sends the current timestamp and three parameters F, R_1 and R_2 to the server (or the peers) as illustrated in Fig. 8. The solid arrowed line represents the actual path taken by avatar $\mathcal{P}$; the two points $M(x_1, y_1)$

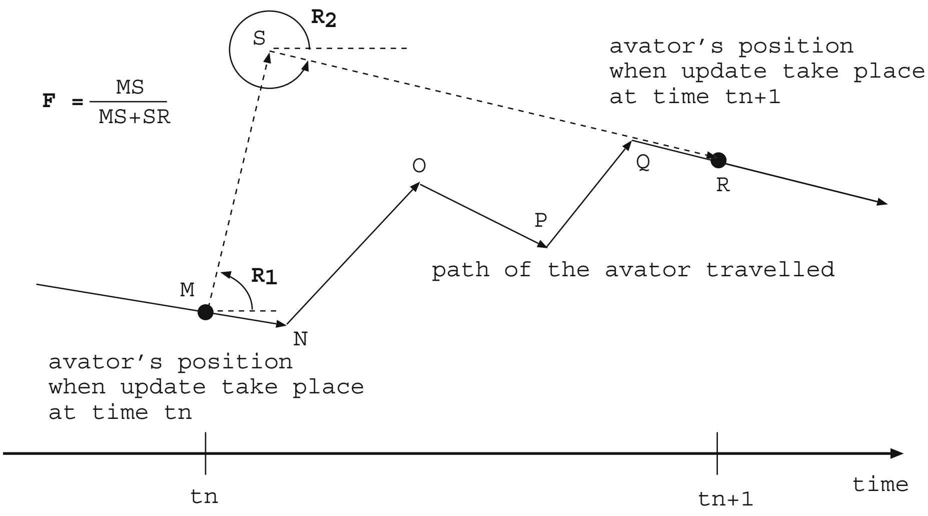


Fig. 8 The three synchronization parameters R_1, R_2 and F when synchronization take places at time t_{n+1}

and $R(x_2, y_2)$ on the path indicate $\mathcal{P}$'s coordinates when update packets are sent at time t_n and t_{n+1} respectively.

To illustrate how to compute the three parameters for the update packet at time t_{n+1}, we construct a triangle MST as shown in Fig. 9. The line SRT is extended from the avatar's velocity vector. The line MT is constructed such that the angle included with the line SRT equals to $360° - R_2$, where R_2 is the current direction of $\mathcal{P}$'s movement. Let d be the length of the path $MNOPQR$ taken by $\mathcal{P}$ and the length of the line segment SR be l, we need to find out l such that the total length of the two line segments MS and SR equals to d.

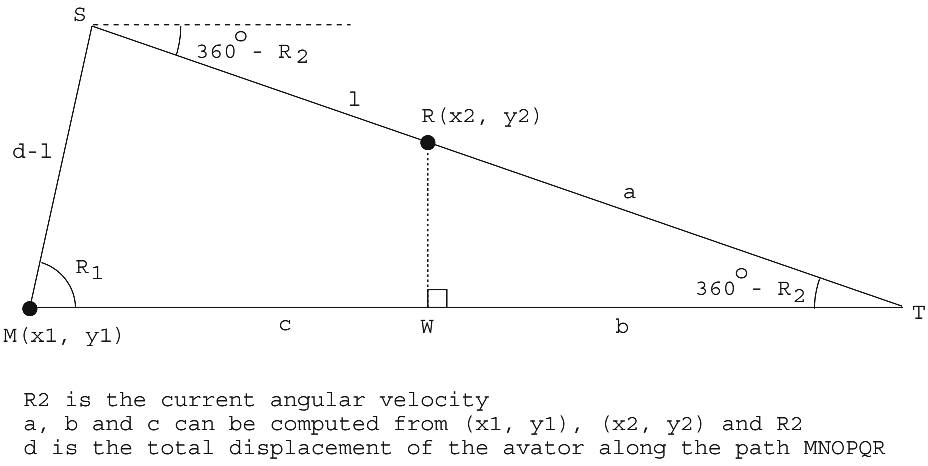

S
360° - R2
l
R(x2, y2)
d-l
a
R1
360° - R2
T
M(x1, y1)
c
W
b
R2 is the current angular velocity
a, b and c can be computed from (x1, y1), (x2, y2) and R2
d is the total displacement of the avator along the path MNOPQR

Fig. 9 Computation of the three parameters in the hack-proof dead-reckoning protocol

The value of l is given by

$$\begin{aligned}
\cos(\angle STM) &= \frac{ST^2 + MT^2 - MS^2}{2(ST)(MT)} \\
\cos(360^\circ - R_2) &= \frac{(l+a)^2 + (b+c)^2 - (d-l)^2}{2(l+a)(b+c)} \\
\cos(R_2) &= \frac{(l+a)^2 + (b+c)^2 - (d-l)^2}{2(l+a)(b+c)} \\
2(l+a)(b+c)\cos(R_2) &= l^2 + 2al + a^2 + b^2 + 2bc + c^2 - d^2 + 2dl - l^2 \\
2l(b+c)\cos(R_2) &= 2l(a+d) + (a^2 + b^2 + 2bc + c^2 - d^2) \\
&\quad -2a(b+c)\cos(R_2) \\
2l(b+c)\cos(R_2) - 2l(a+d) &= (a^2 + b^2 + 2bc + c^2 - d^2) - 2a(b+c)\cos(R_2) \\
2l\{(b+c)\cos(R_2) - (a+d)\} &= (a^2 + b^2 + 2bc + c^2 - d^2) - 2a(b+c)\cos(R_2)
\end{aligned}$$

$$l = \frac{(a^2 + b^2 + 2bc + c^2 - d^2) - 2a(b+c)\cos(R_2)}{2\{(b+c)\cos(R_2) - (a+d)\}}$$

Having l, we can then compute the parameter F by

$$\begin{aligned}
F &= \frac{MS}{MS + SR} \\
&= \frac{d-l}{d}
\end{aligned}$$

Finally, we can compute R_1 by

$$\begin{aligned}
\frac{MS}{\sin(\angle MTS)} &= \frac{ST}{\sin(\angle SMT)} \\
\frac{d-l}{\sin(360^\circ - R_2)} &= \frac{l+a}{\sin(R_1)} \\
\sin(R_1) &= \frac{(l+a)\sin(R_2)}{l-d}
\end{aligned}$$

Working Example We now use an example to illustrate the computation of the synchronization parameters and illustrate how a server (or the peers) can compute the new coordinates and direction from the received synchronization parameters. Referring to Fig. 10, let $\mathcal{P}$ was at $M(15, 18)$ at time $t_n = 14900$ ms. At time $t_{n+1} = 15000$ ms $\mathcal{P}$ was moved $d = 8$ units to $R(20, 20)$ and was heading at an angle of 315°.

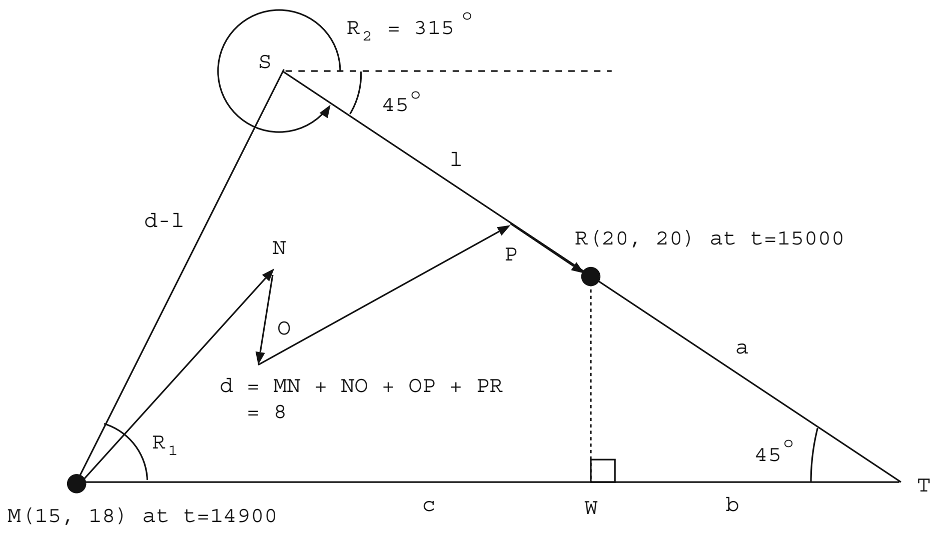


Fig. 10 Working example of the enhanced invulnerable dead-reckoning protocol where the host of $\mathcal{P}$ computes its synchronization parameters

The host of $\mathcal{P}$ computes that $R_2 = 315°, c = x_2 - x_1 = 20 - 15 = 5, b = \frac{RW}{\tan 45°} = y_2 - y_1 = 2, a = \frac{b}{\cos 45°} = \sqrt{8}$. Therefore, we get

$$\begin{aligned} l &= \frac{(a^2 + b^2 + 2bc + c^2 - d^2) - 2a(b+c)\cos(R_2)}{2\{(b+c)\cos(R_2) - (a+d)\}} \\ &= \frac{(\sqrt{8}^2 + 2^2 + 2*2*5 + 5^2 - 8^2) - 2\sqrt{8}(2+5)\cos(45°)}{2\{(2+5)\cos(45°) - (\sqrt{8}+8)\}} \\ &= \frac{(8+4+20+25-64) - 2\sqrt{2}(7)\frac{1}{\sqrt{8}}}{2\left\{(7)\frac{1}{\sqrt{8}} - (\sqrt{8}+8)\right\}} \\ &= 2.9768589 \end{aligned}$$

hence

$$\begin{aligned} F = \frac{d-l}{d} &= \frac{8 - 2.9768589}{8} \\ &= 0.6278926375 \end{aligned}$$

and

$$\begin{aligned} \sin(R_1) = \frac{(l+a)\sin(R_2)}{l-d} &= \frac{(2.9768589 + \sqrt{8})\sin(45°)}{2.9768589 - 8} \\ R_1 &= 54.8063918° \end{aligned}$$

On receiving the update packet (*timestamp*, F, R_1, R_2), the server (or the peers) computes the elapsed time between $\mathcal{P}$'s two latest synchronizations as $t_{n+1} - t_n = 15000 - 14900 = 100$ ms, and use the legal speed of $\mathcal{P}$, i.e. 0.08 units/ms, to compute that $d = 0.08 * 100 = 8$ units. To illustrate the computation of $\mathcal{P}$'s new coordinates, we construct two triangles as shown in Fig. 11. First, the server computes the coordinates of S by

$$\begin{aligned}
S &= (15 + MX, 18 + XS) \\
&= (15 + \cos R_1 * MS, 18 + \sin R_1 * MS) \\
&= (15 + \cos R_1 * (d * F), 18 + \sin R_1 * (d * F)) \\
&= (15 + \cos 54.8063918 * 8 * 0.6278926375, 18 + \sin 54.8063918 * 8 * 0.6278926375) \\
&= (17.8950429, 22.1049571)
\end{aligned}$$

and then computes the coordinates of R by

$$\begin{aligned}
R = {} & (17.8950429 + \sin(R_2 - 270^\circ) * SR, \\
& \qquad 22.1049571 - \cos(R_2 - 270^\circ) * SR) \\
= {} & (17.8950429 + \sin(45^\circ) * d(1 - F), \\
& \qquad 22.1049571 - \cos(45^\circ) * d(1 - F)) \\
= {} & (17.8950429 + \sin(45^\circ) * 8(1 - 0.6278926375), \\
& \qquad 22.1049571 - \cos(45^\circ) * 8(1 - 0.6278926375)) \\
= {} & (20, 20)
\end{aligned}$$

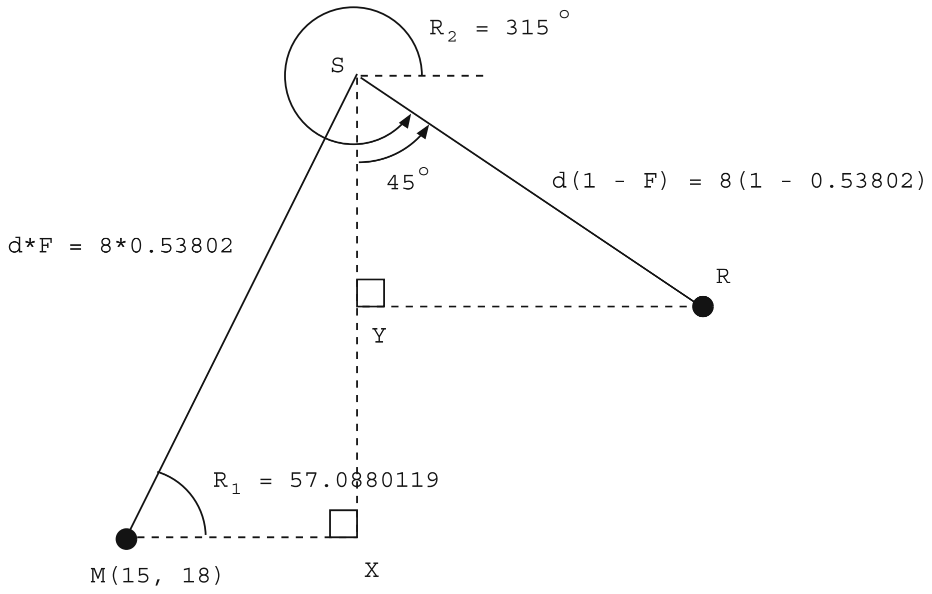


Fig. 11 Working example of the hack-proof dead-reckoning protocol where the server determines $\mathcal{P}$'s coordinates and the direction from $\mathcal{P}$'s synchronization parameters

which is the correct new coordinates of $\mathcal{P}$ at $t = 14900$ ms, and the new direction of $\mathcal{P}$'s movement can be simply given by $R_2 = 315°$.

3.3.1 Handling missing packets

The enhanced protocol can handle missing packets in the same way as the modified dead-reckoning protocol. Readers can refer it to Section 3.2.3.

3.4 Extensions

In this section, we use different scenarios to illustrate some additional issues and how we can extend the protocol to handle these cases.

Scenario 1 Referring to Fig. 12, suppose player $\mathcal{P}$ has moved and stopped occasionally, or has accelerated and decelerated occasionally, between time t_n and t_{n+1} so that the total length of the path $\mathcal{P}$ taken is shorter than the maximum possible displacement if $\mathcal{P}$ moves continuously with its legal speed. We re-define the value of d for a greater generality:

$$d = legal\ speed\ of\ \mathcal{P} * \text{elapsed time}$$

Therefore, when the host of $\mathcal{P}$ computes its synchronization parameters and when the server (or the peers) computes $\mathcal{P}$'s new coordinates from the synchronization parameters, they will always have the same value of d and therefore the correct coordinates of $\mathcal{P}$ can be determined even if $\mathcal{P}$ has stopped or decelerated occasionally since the last synchronization.

Scenario 2 Suppose player $\mathcal{P}$ has not moved since the last synchronization at time t_n and remains stationary until time t_{n+1}, i.e. the final displacement is zero, but the direction may or may not have changed. In this scenario, the client may simply use $F = 0.5$ and $R_1 = NULL$ to report the server (or the peers) to render no movement

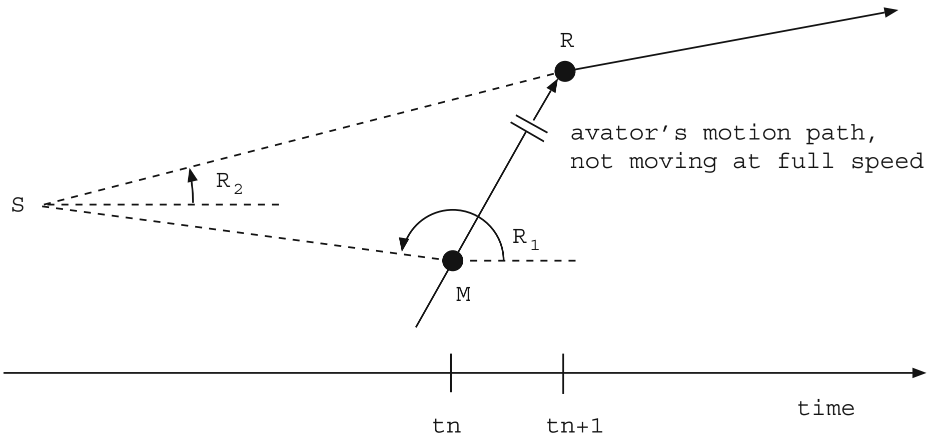


Fig. 12 Player $\mathcal{P}$ has stopped or decelerated occasionally between time t_n and t_{n+1}, therefore the path MR taken by $\mathcal{P}$ is shorter than $d = MS + SR$

in this synchronization. Notice that the synchronization parameter R_2 is still useful in this scenario, because $\mathcal{P}$ may have changed the direction, i.e. it has local motion since time t_n but without having any global motion. The value of R_2 tells other players to render $\mathcal{P}$ turning to the direction given by R_2.

3.5 Proof of invulnerability

Theorem *Let a cheater be capable of modifying the content of the update packets, the extra displacement that an avatar $\mathcal{P}$ can gain over the whole game session is bounded by*

$$\textit{single-trip latency} * \textit{legal speed of } \mathcal{P}.$$

Proof In Fig. 11, the largest displacement that avatar $\mathcal{P}$ can travel between two successive synchronizations is given by

$$\begin{aligned} \textit{displacement} &= MS + SR = d * F + d(1-F) = d \\ &= \textit{legal speed of } \mathcal{P} * \textit{elapsed time}. \end{aligned}$$

Therefore, the overall displacement that an avatar can travel over the whole game session is bounded by

$$\begin{aligned} &\textit{overall displacement} \\ &\quad \le \textit{displacement}_1 + \textit{displacement}_2 + ... + \textit{displacement}_n \\ &\quad = \textit{legal speed of } \mathcal{P} * (\textit{elapsed time}_1 + \textit{elapsed time}_2 + ... + \textit{elapsed time}_n) \\ &\quad = \textit{legal speed of } \mathcal{P} * \{(t_1 - t_0) + (t_2 - t_1) + ... + (t_n - t_{n-1})\}. \end{aligned}$$

Note that the legal speed of an avatar $\mathcal{P}$ is authorized on either the server side (client-server architecture), or on the peers (P2P architecture), its value only changes when it is granted by the server or agreed by all peers in particular game events. Therefore, its value cannot be spoofed by the cheater.

The only way to spoof a larger elapsed time is to provide a larger timestamp in an update packet. However, a large value of timestamp can be detected easily since all machines are synchronized to the same global clock. Therefore, the exaggeration on the elapsed time is bounded by a single-trip latency from the sender to the recipient where

$$t_{exaggerated} \le t_{loyal} + \textit{single-trip latency},$$

or else $t_{exaggerated}$ will be larger than the system time of the recipient when receiving this malicious packet. Moreover, this exaggeration cannot be accumulated since

$$\begin{aligned} \textit{elapsed time}_n &= t_{exaggerated_n} - t_{exaggerated_{n-1}} \\ &\le (t_{loyal_n} + \textit{single-trip latency}) - t_{loyal_{n-1}} \\ &= t_{loyal_n} - t_{loyal_{n-1}}. \end{aligned}$$

That is, when the timestamp in the previous update packet is already exaggerated, the exaggerated timestamp in the current update packet will not be able to produce an enlarged elapsed time again. By induction, the exaggerated timestamp in any later update packets cannot produce any enlarged elapsed time, too. Therefore, the illegal overall displacement is bounded by

$$\begin{aligned}
&\textit{illegal overall displacement}\\
&\quad \leq \textit{legal speed of } \mathcal{P} * \{(t_1 - t_0) + (t_2 - t_1) + \ldots + (t_n + \textit{single-trip latency} - t_{n-1})\}\\
&\quad = \textit{legal speed of } \mathcal{P} * (t_n + \textit{single-trip latency} - t_0)\\
&\quad = \textit{legal speed of } \mathcal{P} * (t_n - t_0) + \textit{legal speed of } \mathcal{P} * \textit{single-trip latency}\\
&\quad = \textit{legal displacement} + (\textit{legal speed of } \mathcal{P} * \textit{single-trip latency})
\end{aligned}$$

The extra displacement is *legal speed of* $\mathcal{P} *$ *single-trip latency* and the theorem is hence proved. □

4 Implementation

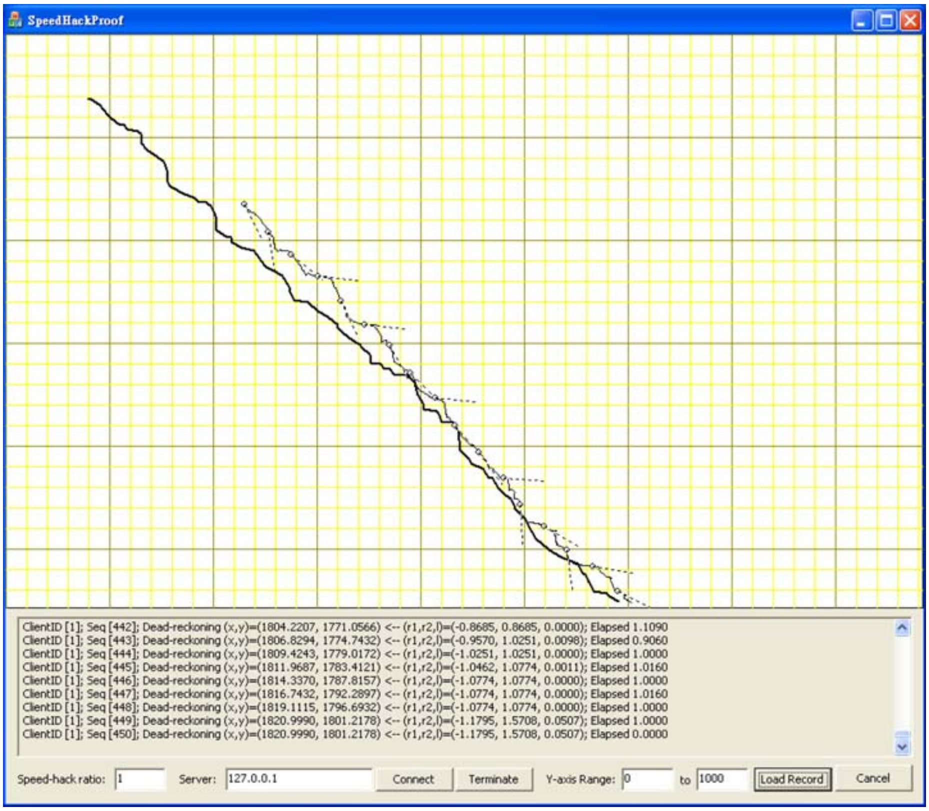
We have implemented a prototype server and a prototype client to demonstrate the feasibility of our proposed protocol. The prototype server only acts as a broadcaster which forwards dead-reckoning packets to all clients. The prototype client automatically generates random moves continuously and sends out dead-reckoning parameters at an interval of 1 s. Both client and server are coded with Visual C++ using Windows Socket API. We tested our prototype on Windows XP platforms.

All clients are synchronized to the same NTP time server. In our implementation, we used the public NTP server available at stdtime.gov.hk. Each client queries the NTP server at a 30-s interval to ensure that the clocks of all clients are loosely synchronized throughout the whole game session. However, between successive NTP updates, a client increments its time with its local system clock.

Rounding error may occur when a sender converts a position vector to dead-reckoning parameters and when a recipient converts back the parameters to a position vector. Since at each synchronization, the position vector is computed based on the previous one, the errors will accumulate over time. To overcome this, a sender simply runs the same routine used by the recipients with the parameters it sends out, and adjust the position of its local avatar accordingly. The position of the local avatar is hence adjusted with an amount equals to the rounding errors emerged from the computations which is a very small value that will not be noticeable on the rendered graphics. Using this simple scheme, the accumulation of the rounding errors is eliminated.

Figures 13 and 14 shows the rendered paths of two connected clients in a duration of 10 min. The paths are generated by the clients randomly. There is no boundary on the movements (clients can move freely to any direction at every moment), and collision test was omitted (clients may overlap together at the same coordinates). These two requirements are necessary for real games, standard boundary and collision test for conventional dead-reckoning protocol are appropriate for our new protocol; however, missing these details does not disprove the correctness of our protocol.

The thicker lines are the actual paths of the 2 avatars on their local machines respectively. The thinner lines represent the extrapolated paths of non-local avatars. The extrapolated paths are projected from the dead-reckoning vectors computed



```
ClientID [1]; Seq [442]; Dead-reckoning (x,y)=(1804.2207, 1771.0566) <-- (r1,r2,l)=(-0.8685, 0.8685, 0.0000); Elapsed 1.1090
ClientID [1]; Seq [443]; Dead-reckoning (x,y)=(1806.8294, 1774.7432) <-- (r1,r2,l)=(-0.9570, 1.0251, 0.0098); Elapsed 0.9060
ClientID [1]; Seq [444]; Dead-reckoning (x,y)=(1809.4243, 1779.0172) <-- (r1,r2,l)=(-1.0251, 1.0251, 0.0000); Elapsed 1.0000
ClientID [1]; Seq [445]; Dead-reckoning (x,y)=(1811.9687, 1783.4121) <-- (r1,r2,l)=(-1.0462, 1.0774, 0.0011); Elapsed 1.0160
ClientID [1]; Seq [446]; Dead-reckoning (x,y)=(1814.3370, 1787.8157) <-- (r1,r2,l)=(-1.0774, 1.0774, 0.0000); Elapsed 1.0000
ClientID [1]; Seq [447]; Dead-reckoning (x,y)=(1816.7432, 1792.2897) <-- (r1,r2,l)=(-1.0774, 1.0774, 0.0000); Elapsed 1.0160
ClientID [1]; Seq [448]; Dead-reckoning (x,y)=(1819.1115, 1796.6932) <-- (r1,r2,l)=(-1.0774, 1.0774, 0.0000); Elapsed 1.0000
ClientID [1]; Seq [449]; Dead-reckoning (x,y)=(1820.9990, 1801.2178) <-- (r1,r2,l)=(-1.1795, 1.5708, 0.0507); Elapsed 1.0000
ClientID [1]; Seq [450]; Dead-reckoning (x,y)=(1820.9990, 1801.2178) <-- (r1,r2,l)=(-1.1795, 1.5708, 0.0507); Elapsed 0.0000
```

Speed-hack ratio: 1 Server: 127.0.0.1 Y-axis Range: 0 to 1000

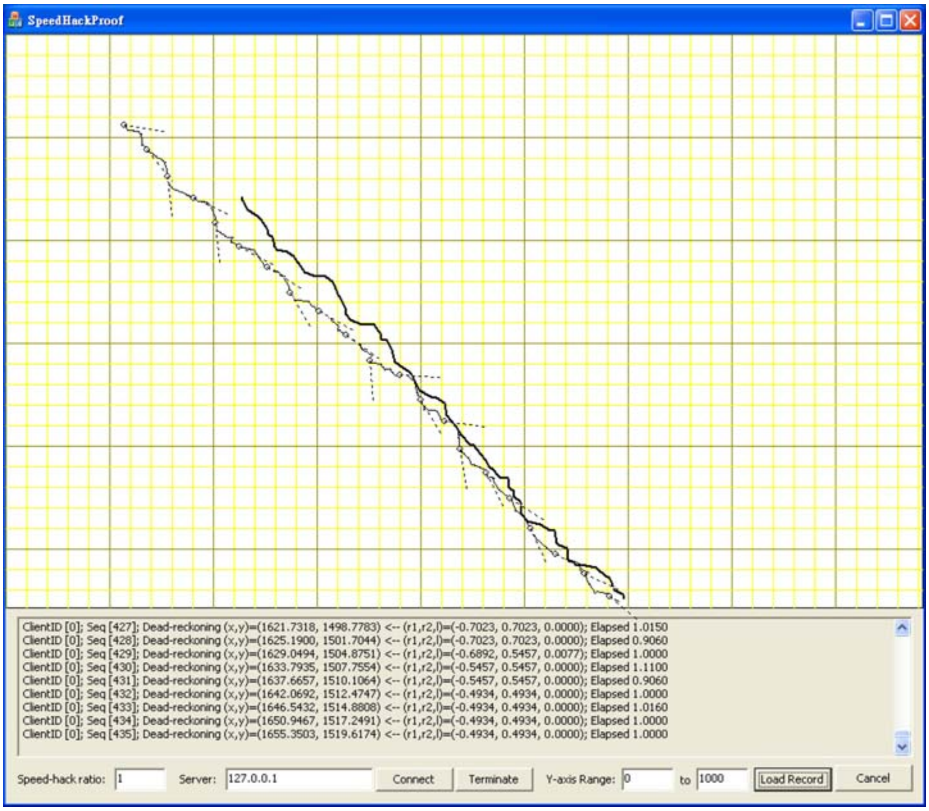
Fig. 13 The path of the local avatar $\mathcal{P}$ (*thicker line*) and the path of the non-local avatar $\mathcal{Q}$ (*thinner line*) rendered on $\mathcal{P}$'s local machine. The *circled tails* represent the dead-reckoning positions computed from the received synchronization parameters and the *dashed lines* represent the directions of the dead-reckoning vectors

from the received synchronization parameters. The dead-reckoning vectors are illustrated as dashed lines with circled tails. Only some of the dead-reckoning vectors are displayed on the figures for a clear view of the paths. Figures 15 and 16 zoom into the last 60 s of Figs. 13 and 14 respectively. We can see that the clients are still synchronized correctly after 10 min of simulation.

Speed-hacking on a client is achieved with generic over-clocking software together with a spoofed NTP source. However, doing so produced invalid synchronization parameters which resulted in invalid computation on the recipients. The recipients simply discarded all invalid updates and the cheater was regarded as disconnected.

4.1 Network overhead

In conventional dead-reckoning protocol, the exchange of location information requires four parameters: x-coordinate, y-coordinate, angle, and the timestamp. Typically, a game divides the whole map into smaller areas called zones. Assuming a two-bytes integer is used for a single coordinate, a floating point number for the



```
ClientID [0]; Seq [427]; Dead-reckoning (x,y)=(1621.7318, 1498.7783) <-- (r1,r2,l)=(-0.7023, 0.7023, 0.0000); Elapsed 1.0150
ClientID [0]; Seq [428]; Dead-reckoning (x,y)=(1625.1900, 1501.7044) <-- (r1,r2,l)=(-0.7023, 0.7023, 0.0000); Elapsed 0.9060
ClientID [0]; Seq [429]; Dead-reckoning (x,y)=(1629.0494, 1504.8751) <-- (r1,r2,l)=(-0.6892, 0.5457, 0.0077); Elapsed 1.0000
ClientID [0]; Seq [430]; Dead-reckoning (x,y)=(1633.7935, 1507.7554) <-- (r1,r2,l)=(-0.5457, 0.5457, 0.0000); Elapsed 1.1100
ClientID [0]; Seq [431]; Dead-reckoning (x,y)=(1637.6657, 1510.1064) <-- (r1,r2,l)=(-0.5457, 0.5457, 0.0000); Elapsed 0.9060
ClientID [0]; Seq [432]; Dead-reckoning (x,y)=(1642.0692, 1512.4747) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0000
ClientID [0]; Seq [433]; Dead-reckoning (x,y)=(1646.5432, 1514.8808) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0160
ClientID [0]; Seq [434]; Dead-reckoning (x,y)=(1650.9467, 1517.2491) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0000
ClientID [0]; Seq [435]; Dead-reckoning (x,y)=(1655.3503, 1519.6174) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0000
```

Fig. 14 The path of the local avatar $\mathcal{Q}$ (*thicker line*) and the path of the non-local avatar $\mathcal{P}$ (*thinner line*) rendered on $\mathcal{Q}$'s local machine. The *circled tails* represent the dead-reckoning positions computed from the received synchronization parameters and the *dashed lines* represent the directions of the dead-reckoning vectors

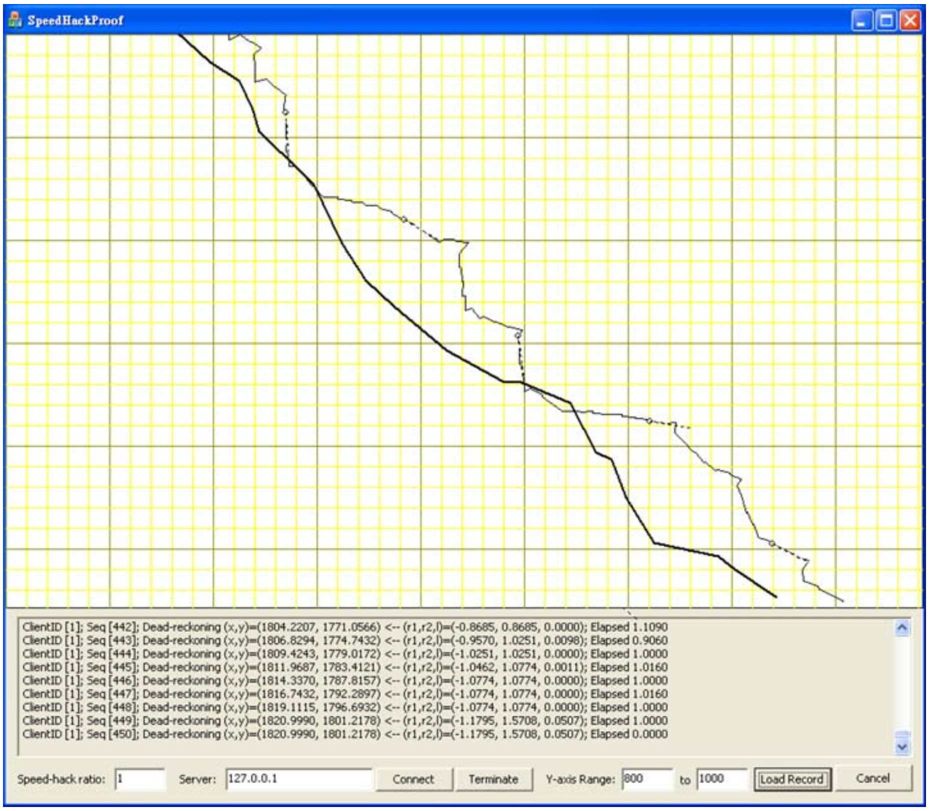
angle in radian, and a double precision timestamp, the total payload for the location information is therefore

$$2 + 2 + 4 + 8 = 16 bytes.$$

In our proposed protocol, the required synchronization parameters F, R_1, R_2 and the timestamp requires four double precision numbers implies a total of 32 bytes. Since 25 frame-per-second or above is enough for a fluent video display, we assume 25 synchronizations per second (in practical 5–10 is usually enough) concurrently which implies a total overhead of

$$(32 - 16)25 = 400 bps$$

which is a small value compares to the 40 Kbps average bandwidth requirement for some commercial multiplayer online games [8]. Therefore, we expect the overhead of our protocol will not induce significant impact on the network traffic.



```
ClientID [1]; Seq [442]; Dead-reckoning (x,y)=(1804.2207, 1771.0566) <-- (r1,r2,l)=(-0.8685, 0.8685, 0.0000); Elapsed 1.1090
ClientID [1]; Seq [443]; Dead-reckoning (x,y)=(1806.8294, 1774.7432) <-- (r1,r2,l)=(-0.9570, 1.0251, 0.0098); Elapsed 0.9060
ClientID [1]; Seq [444]; Dead-reckoning (x,y)=(1809.4243, 1779.0172) <-- (r1,r2,l)=(-1.0251, 1.0251, 0.0000); Elapsed 1.0000
ClientID [1]; Seq [445]; Dead-reckoning (x,y)=(1811.9687, 1783.4121) <-- (r1,r2,l)=(-1.0462, 1.0774, 0.0011); Elapsed 1.0160
ClientID [1]; Seq [446]; Dead-reckoning (x,y)=(1814.3370, 1787.8157) <-- (r1,r2,l)=(-1.0774, 1.0774, 0.0000); Elapsed 1.0000
ClientID [1]; Seq [447]; Dead-reckoning (x,y)=(1816.7432, 1792.2897) <-- (r1,r2,l)=(-1.0774, 1.0774, 0.0000); Elapsed 1.0160
ClientID [1]; Seq [448]; Dead-reckoning (x,y)=(1819.1115, 1796.6932) <-- (r1,r2,l)=(-1.0774, 1.0774, 0.0000); Elapsed 1.0000
ClientID [1]; Seq [449]; Dead-reckoning (x,y)=(1820.9990, 1801.2178) <-- (r1,r2,l)=(-1.1795, 1.5708, 0.0507); Elapsed 1.0000
ClientID [1]; Seq [450]; Dead-reckoning (x,y)=(1820.9990, 1801.2178) <-- (r1,r2,l)=(-1.1795, 1.5708, 0.0507); Elapsed 0.0000
```

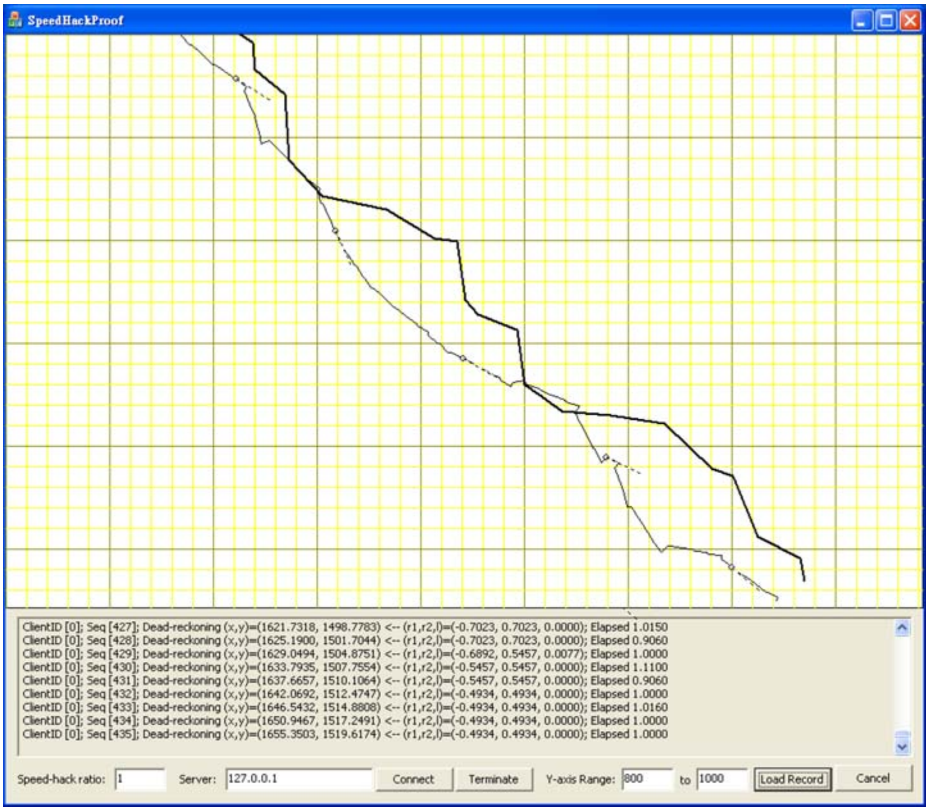
Fig. 15 The path of the local avatar $\mathcal{P}$ (*thicker line*) and the path of the non-local avatar $\mathcal{Q}$ (*thinner line*) rendered on $\mathcal{P}$'s local machine which zoomed into the last 60 s

5 Related works

In [4], the authors proposed the use of runtime verification to verify game codes. This approach mainly targets on cheats that exploit implementation bugs. But this approach is not applicable to cheats that involve modification of client code loading into the memory at runtime. Where most speed-hacks fall into the category of runtime cheats.

PunkBuster [7] is the first client-side cheat prevention system for commercial online games. HLGuard [20], formerly called CSGuard, is a free server-side anti-cheat system for a famous commercial FPS game, Half-Life, and many variations of Half-Life. Besides PunkBuster and HLGuard, there are a few other commercial anti-cheating software [21]. Basically, they are pattern scanners that scan for known cheats in the client machine. The anti-cheating software must be kept up-to-date from time to time since new cheats exist frequently. Cheating still cannot be completely prevented and these anti-cheating software themselves are also vulnerable to hacks.

In [2], the authors describe a type of cheat called *suppress-correct cheat* and propose a cheat-proof protocol that resists this type of cheat. Suppose a cheater S



```
ClientID [0]; Seq [427]; Dead-reckoning (x,y)=(1621.7318, 1498.7783) <-- (r1,r2,l)=(-0.7023, 0.7023, 0.0000); Elapsed 1.0150
ClientID [0]; Seq [428]; Dead-reckoning (x,y)=(1625.1900, 1501.7044) <-- (r1,r2,l)=(-0.7023, 0.7023, 0.0000); Elapsed 0.9060
ClientID [0]; Seq [429]; Dead-reckoning (x,y)=(1629.0494, 1504.8751) <-- (r1,r2,l)=(-0.6892, 0.5457, 0.0077); Elapsed 1.0000
ClientID [0]; Seq [430]; Dead-reckoning (x,y)=(1633.7935, 1507.7554) <-- (r1,r2,l)=(-0.5457, 0.5457, 0.0000); Elapsed 1.1100
ClientID [0]; Seq [431]; Dead-reckoning (x,y)=(1637.6657, 1510.1064) <-- (r1,r2,l)=(-0.5457, 0.5457, 0.0000); Elapsed 0.9060
ClientID [0]; Seq [432]; Dead-reckoning (x,y)=(1642.0692, 1512.4747) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0000
ClientID [0]; Seq [433]; Dead-reckoning (x,y)=(1646.5432, 1514.8808) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0160
ClientID [0]; Seq [434]; Dead-reckoning (x,y)=(1650.9467, 1517.2491) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0000
ClientID [0]; Seq [435]; Dead-reckoning (x,y)=(1655.3503, 1519.6174) <-- (r1,r2,l)=(-0.4934, 0.4934, 0.0000); Elapsed 1.0000
```

Speed-hack ratio: 1 Server: 127.0.0.1 Y-axis Range: 800 to 1000

Fig. 16 The path of the local avatar $\mathcal{Q}$ (*thicker line*) and the path of the non-local avatar $\mathcal{P}$ (*thinner line*) rendered on $\mathcal{Q}$'s local machine which zoomed into the last 60 s

uses the suppress-correct cheat and S purposefully drops n packets while receiving n packets from each of other players, other players will be forced to extrapolate the movement of S for n timeframes but cannot confirm where S really is. S then can construct the $n+1$th packet based on the knowledge of the previous n timeframes and it provides him with some advantages. To eliminate suppress-correct cheat, the authors propose a synchronization technique called asynchronous synchronization (AS). Using AS, each host advances in time asynchronously from the other players but enters into the lockstep mode when interaction occurs. When entering the lockstep mode, in every timeframe t each involved player must wait for all packets from other players before advancing to timeframe $t+1$. Because this is a stop-and-wait protocol, extrapolation cannot be used to smooth out any delay caused by the network latency.

In [12], the authors improve the performance of the lockstep protocol by adding pipelines. Extrapolation is still not allowed under the pipelined lockstep protocol. Therefore, if there is an increased network latency and packets are delayed, the game will be stalled.

In [10], the authors propose a sliding pipeline protocol that dynamically adjusts the pipeline depth to reflect current network conditions. The authors also introduce a send buffer to hold the commands generated while the size of the pipeline is adjusted. The sliding pipeline protocol allows extrapolation to smooth out jitters.

Although these protocols are designed to defend against the suppress-correct cheat, it can also prevent speed-hacks when entering into the lock-step mode because players are forced to synchronize within a bounded amount of timeframes. However, speed-hack can still be effective when lock-step mode is not activated. And since these protocols do not allow packets to be dropped, any lost packet must be re-transmitted until they are finally sent and acknowledged. Therefore, the minimum timeframe of the game cannot be shorter than the maximum latency of the player with the slowest connection and all clients must run the game at a speed that even the slowest client can support. Furthermore, any sudden increase in the latency will cause jitters to all players.

Our protocol does not incur any lock-step requirement to game clients while the advantage of loose synchronization in conventional dead-reckoning protocol is completely preserved. Thus, smooth gameplay can be ensured. As we have proved in Section 3.5, a cheater can only cheat by generating malicious timestamps and it can be detected easily and immediately. Therefore, the speed-hack invulnerability of our protocol will be enforced throughout the whole game session so that any action of cheating can be detected immediately.

Moreover, the AS protocol requires a game client to enter the lock-step mode when interaction occurs which requires a major modification of the client code to realize it. However, existing games can be modified easily to adapt our proposed protocol. One can simply add a plugin routine to convert a dead-reckoning vector to the synchronization parameters before sending out the update packets, and add another plugin routine to convert back the synchronization parameters to a dead-reckoning vector on receiving the packets.

The *NEO* protocol [13] is based on [2], the authors describe five forms of cheating and claim that the *NEO* protocol can prevent these cheating.

In [17], the authors show that for the five forms of cheating [13] designed to prevent, it prevents only three. They propose another Secure Event Agreement (*SEA*) protocol that prevents all five forms of cheating which the performance is at worst equal to *NEO* and in some cases better.

In [19], the authors show that both *NEO* and *SEA* suffer from the *undo* cheat. Let P_H denote an honest player and P_C denote a cheater, and M_H, K_H and M_C, K_C represent the message and its key from P_H and P_C respectively. The cheater P_C performs the undo cheat as follows: both players send their encrypted game moves (M_H and M_C) normally in the commit phase. Then, P_H sends key K_H in the reveal phase. However, P_C delays K_C until K_H is received and M_H is revealed. If P_C find that M_C is poor against M_H, P_C will purposely drop K_C and therefore undoing the move M_C. The authors then propose another anti-cheat scheme for P2P games called *RACS* which relies on the existence of a trusted *referee*. The referee is responsible for T1 - receiving player updates, T2 - simulating game play, T3 - validating and resolving conflicts in the simulation, T4 - disseminating updates to clients and T5 - storing the current game state.

The *referee* used in *RACS* works very likely to a traditional game server in conventional client-server architecture. The security of *RACS* completely depends on the referee. For example, speed-hack can be prevented with validating every state updates by the referee. Although *RACS* is more scalable than client-server architecture, it suffers from the same problem that the involvement of a trusted third party is required.

6 Conclusion

In this paper, we presented a synchronization protocol for multi-player online games that support dead-reckoning. Meanwhile, it is invulnerable to a very common type of cheat called speed-hack. The general idea is that the server or peer players can use the legal speed of an avatar to compute its position from a set of update parameters. This eliminates the need to state the avatar's position directly in the update packets. Even if the cheater is able to modify the data in the update packets, the cheater cannot spoof other players to render a faster moving avatar because the displacement an avatar can travel is now bounded by the legal speed of the player that is authorized by the server (in client-server architecture) or among all peers (in P2P architecture). We have used various examples to illustrate our protocol and proved the security feature of our proposal. We have carried out simulations to demonstrate the feasibility of our protocol.

References

1. Banavar H, Aggarwal S, Khandelwal A (2004) Accuracy in dead-reckoning based distributed multi-player games. In: Proceedings of NetGames 2004, Portland, August 2004, pp 161–165
2. Baughman NE, Levine BN (2001) Cheat-proof playout for centralized and distributed online games. In: Proceedings of IEEE INFOCOM. IEEE, Piscataway, pp 104–113
3. Counter Hack (2007) Types of Hacks. http://wiki.counter-hack.net/CategoryGeneralInfo
4. DeLap M et al (2004) Is runtime verification applicable to cheat detection. In: Proceedings of NetGames 2004, Portland, August 2004, pp 134–138
5. Diot C, Gautier L (1999) A distributed architecture for multiplayer interactive applications on the internet. In: IEEE Networks magazine, Jul–Aug 1999
6. Diot C, Gautier L, Kurose J (1999) End-to-end transmission control mechanisms for multiparty interactive applications on the internet. In: Proceedings of IEEE INFOCOM, IEEE, Piscataway
7. Even Balance (2007) Official PunkBuster website. http://www.evenbalance.com
8. Feng WC, Feng WC, Chang F, Walpole J (2005) A traffic characterization of popular online games. IEEE/ACM Trans Netw 13(3):488–500
9. Gautier L, Diot C (1998) Design and evaluation of mimaze, a multiplayer game on the Internet. In: Proceedings of IEEE Multimedia (ICMCS'98). IEEE, Piscataway
10. Jamin S, Cronin E, Filstrup B (2003) Cheat-proofing dead reckoned multiplayer games (extended abstract). In: Proc. of 2nd international conference on application and development of computer games, Hong Kong, 6–7 January 2003
11. Lee FW, Li L, Lau R (2006) A trajectory-preserving synchronization method for collaborative visualization. IEEE Trans Vis Comput Graph 12:989–996 (special issue on IEEE Visualization'06)
12. Lenker S, Lee H, Kozlowski E, Jamin S (2002) Synchronization and cheat-proofing protocol for real-time multiplayer games. In: International Worshop on Entertainment Computing, Makuhari, May 2002
13. Lo V, GauthierDickey C, Zappala D, Marr J (2004) Low latency and cheatproof event ordering for peer-to-peer games. In: ACM NOSSDAV'04, Kinsale, June 2004
14. Mills DL (1992) Network time protocol (version 3) specification, implmentation and analysis. In: RFC-1305, March 1992
15. MPC Forums (2007) Multi-Player Cheats. http://www.mpcforum.com
16. Pantel L, Wolf L (2002) On the impact of delay on real-time multiplayer games. In: ACM NOSSDAV'02, Miami Beach, May 2002

17. Schachte P, Corman AB, Douglas S, Teague V (2006) A secure event agreement (sea) protocol for peer-to-peer games. In: Proceedings of ARES'06, Vienna, 20–22 April 2006, pp 34–41
18. Simpson ZB (2008) A stream based time synchronization technique for networked computer games. http://www.mine-control.com/zack/timesync/timesync.html
19. Soh S, Webb S, Lau W (2007) Racs: a referee anti-cheat scheme for p2p gaming. In: Proceedings of NOSSDAV'07, Urbana-Champaign, 4–5 June 2007, pp 34–42
20. The Z Project (2007) Official HLGuard website. http://www.thezproject.org
21. Wikipedia (2007) Category: Anti-cheat software. http://en.wikipedia.org/wiki/Category:Anti-cheat_software

Yeung Siu Fung is a Ph.D. candidate at the Chinese University of Hong Kong. He received his B.Eng. and M.Phil. degree in the Computer Science and Engineering Department from the Chinese University of Hong Kong. His research are in multimedia technologies, particularly network security and transport protocols. His personal interests include sports and Christian music.

John C. S. Lui received his Ph.D. in Computer Science from UCLA. After his graduation, he joined the IBM Almaden Research Laboratory/San Jose Laboratory and participated in various research and development projects on file systems and parallel I/O architectures. He later joined the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests encompass both systems and theory. His current research interests include theoretic/applied topics in data networks, distributed multimedia systems, network security, OS design issues, mathematical optimization and performance evaluation. John received the CUHK Vice-Chancellor's Exemplary Teaching Award in 2001. He is an Associate Editor of the Performance Evaluation Journal, a member of the ACM, a senior member of the IEEE and an elected member of the IFIP WG 7.3. His personal interests include films and general reading.