Supporting Continuous Consistency in Multiplayer Online Games

Frederick W.B. Li[†] Lewis W.F. Li[‡] Rynson W.H. Lau[‡]

[†] Department of Computing, The Hong Kong Polytechnic University, Hong Kong
[‡] Department of CEIT, City University of Hong Kong, Hong Kong

ABSTRACT

Multiplayer online games have become very popular in recent years. However, they generally suffer from network latency problem. If a player changes its states, it will take some time before the changes are reflected to other concurrent players. This significantly affects the interactivity of the game. Sometimes, it may even cause disputes among the players. In this paper, we present a continuous consistency control mechanism to support collaborative game applications. Specifically, we propose a relaxed consistency control model for continuous events. Based on this model, we have developed a method to provide a global-wise continuous synchronization on the states of dynamic game objects presented among concurrent game players. We show the performance of the proposed method through some experiments.

Categories and Subject Descriptors: C.2.4 [**Computer- Communication Networks**]: Distributed Systems - *Distributed applications*
General Terms: Performance, Human Factors.

Keywords: Consistency control, distributed synchronization, collaborative gaming, distributed virtual environments.

1. INTRODUCTION

Due to the existence of relatively high network latency of the Internet, a player of a multiplayer online game may perceive significant delay in receiving updated state information of other players as these players move around in the game environment. For example, in Final Fantasy XI [5], which is one of the popular online games currently available in the market, a player usually receives position updates of other game players with almost a second delay. In order to reduce the effect of such delay, some restrictions are imposed on the game itself. First, players can only attack enemy objects, but not each other. Second, the enemy objects are designed to move very little while they are under attack by a player. Such game rules significantly limits the game features and the type of games that can be developed.

Consistency control in distributed applications has been explored in different disciplines, including distributed systems [6], database systems [1] and collaborative editing systems [8]. These systems generally regard state updates as *discrete* events. Hence, their applications could work well by only ensuring that state updates are presented to the relevant users in a correct order, without considering the exact moment that the state updates are presented to the users. However, this method could hardly satisfy the requirements of multiplayer online games or distributed walkthrough systems [3], where events are usually *continuous* [7] and must be presented to users within an accepted period of time in order to maintain the interactivity of the game.

To address the network latency problem of multiplayer games, Mauve et al. [7] adopt a local-lag mechanism. When a player (sender) issues an event, the event will be sent and presented to other players immediately, but not to the sender itself until after the local-lag period is expired in order to reduce the discrepancy between the sender and the receivers. However, with this method, each sender is limited to have only a single local-lag value. Hence, this method can only be used to synchronize between two players, the sender and the receiver; it cannot be used to maintain consistency among a number of players. On the other hand, in Half-Life [2], game players make use of the reference state information from the server to correct the states of their local copies of the dynamic objects. However, there is no control mechanism to guarantee that the actions performed at a player would be synchronized with those at other players.

In this paper, we propose a continuous consistency control mechanism to support collaborative gaming. The rest of this paper is organized as follows. Section 2 presents a consistency model for collaborative game applications. Section 3 describes our continuous synchronization scheme. Section 4 discusses the experimental results of our method, and briefly concludes our work.

2. CONSISTENCY CONTROL

In traditional distributed systems, consistency control is typically performed by maintaining the causality [9] of events presented to each replicated site, where the events are not independent of each other. This is modeled as follows:

DEFINITION 2.1. *Given two events, E_a and E_b, if $E_a \rightarrow E_b$, then E_a must be executed before E_b at each replicated site, where $\rightarrow$ denotes the causal order relation.*

However, this model is not sufficient to handle multiplayer online games, as it does not impose any requirement on the exact moment when the states must be presented to the users. Hence, the users may perceive different states of a shared object at certain moments in time. This may cause visual inconsistency among the users and lead them into making incorrect decisions. To facilitate collaboration among users, ideally, given an updated state of a shared data

MM'04, October 10-16, 2004, New York, New York, USA.

object, the same update should be revealed to all users at the same time. This could be modeled by a time-dependent consistency control model as follows:

DEFINITION 2.2. *Given two distinct sites, i and j, which have replicated a common shared object o, the states of o, $s_i(t)$ and $s_j(t)$ at site i and site j, respectively, should be the same at any time moment t.*

In practice, the above requirement is hard to enforced, due to the existence of network latency. Events generated at a site are usually received by other sites after some delay, which causes state discrepancy among these sites. Although Mauve et al. [7] apply a local-lag mechanism to minimize the discrepancy in state updates between a pair of sites, as discussed in Section 1, the method is unable to provide a global-wise consistency control. To provide continuous consistency control, instead of enforcing *strict consistency* on individual state, we propose a *relaxed consistency control model* as follows:

DEFINITION 2.3. *Given that two distinct sites i and j have replicated a shared object and the states of the replicated object at time t are $s_i(t)$ and $s_j(t)$, respectively, the state discrepancy D of the object at the two sites during any time period between T_a and T_b should be smaller than an application specific tolerance, ξ. Therefore,*

$$D = \int_{T_a}^{T_b} |s_i(t) - s_j(t)|dt \ < \ \xi \qquad (1)$$

This model considers the *continuous* property of state changes in interactive applications such as online games. We have observed that in this kind of applications, the users are more concerned about the *state trajectory* of each moving object. This state trajectory is the trajectory of a continuous sequence of state updates regarding to a moving object. The user would decide how to respond the changes of an object based on observing its state trajectory, instead of its individual states. Hence, we propose the *relaxed consistency control model* that relaxes the strict time-dependent consistency control on individual states of a replicated object as in definition 2.2 to allow the state trajectories of the replicated object to deviate from that of the correct one by an acceptable amount. In fact, the relaxed model will become the strict model if we shorten the time period so that $T_a = T_b$ and set $\xi = 0$.

3. CONTINUOUS SYNCHRONIZATION

To show how to incorporate the relaxed consistency control model in an interactive application, we consider a client-server based game system. The server will run a simulator of each dynamic object in the game. We refer to this simulator as the *reference simulator* of the object. A dynamic object can be either an autonomous object, which is a software controlled object, or an avatar, which is a player controlled object representing the player itself. Each client is required to synchronize any dynamic objects stored with the corresponding reference simulators at the server.

In a traditional client-server system, if a client wants to communicate with another client, the message has to go through the server. In a multiplayer online game, if a client (or player) is interested in a dynamic object managed by another client, it will need to perform the synchronization via the server. This will cause the client to suffer from a double round-trip time delay to obtain the state information of the object. By running the reference simulator at the server, however, the latency is reduced to a single round-trip delay.

In an online game, a player typically controls its own avatar by issuing motion commands through manipulating a keyboard or a game pad. In our method, a player may initiate or modify the current motion of its avatar by issuing a motion command, based on which a motion vector is computed as the navigation vector of the avatar. Subsequent movement of the avatar will follow a motion predictor. In our implementation, we use a first-order predictor as follows:

$$p_{new} = p + t \times V \qquad (2)$$

where p is the current position of the avatar, t is a motion timer difference between p and p_{new}, and V is the motion vector of the avatar. Note that other motion predictors can also be used here [4].

3.1 Client-Server Synchronization

To illustrate the interactions between the client machine of a game player and the server, we consider a situation where the player is moving in the game environment and the server is running a reference simulator of the player, as shown in figure 1. Here, two motion timers T_s and T_c are maintained at the server and the client, respectively. They are the virtual clocks indicating how long the avatar has been performing certain movement as perceived by the server and by the client. Assuming that both the avatar and its reference simulator move with the same motion vector, then the motions of these two objects are expected to be synchronized if T_s has the same value as T_c.

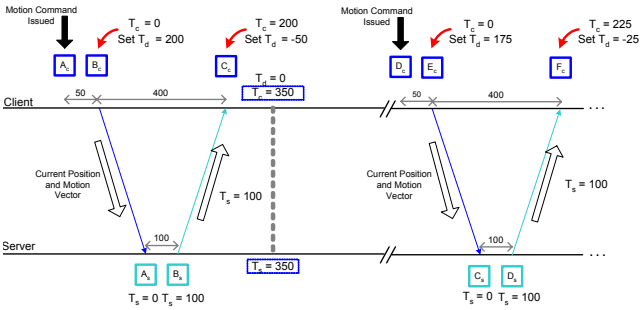

Figure 1: Client-server interactions.

When a player issues a motion command to drive its avatar to move in certain direction (state A_c), the motion command is first buffered for a very short period, about 50ms in our implementation. All the motion commands received during this period will be combined and only the resultant motion vector will be sent to the server at the end of the buffering period. This buffering period acts as a filtering process to limit the input frequency from the keyboard or game pad.

Our implementation approach is to gradually synchronize the client with the server. Let us assume that the avatar's position is updated every Δt in order to be able to render the most updated frame. Hence, Δt may be considered as the duration between two consecutive frames. During each frame, the client updates the position of the avatar by updating the following variables:

$$t = \Delta t - sgn(T_d) \times \epsilon \qquad (3)$$

$$T_d = T_d - sgn(T_d) \times \epsilon \qquad (4)$$

$$T_c = T_c + t \qquad (5)$$

where t is the one used in Equation 2 for computing the new position of the avatar. T_d is the estimated time difference between T_c and T_s. ϵ is computed as $\min(|T_d|, \frac{\Delta t}{2})$. Equation 3 adjusts the duration of t in order to control the amount of motion of an avatar. This approach helps gradually align the motion of the avatar in the client to synchronize with the reference simulator in the server. Equation 4 acts as a counter so that the adjustment process will continue until $T_d = 0$, when synchronization is achieved (eg., when $T_c = T_s = 350$ in figure 1).

At state C_c of figure 1, the client receives the value of T_s from the server. It can then evaluate the updated round-trip time RT, where RT is approximately equal to half of the absolute difference between T_s and the time duration from state B_c to state C_c. In addition, as the updated RT is obtained, T_d will need to be adjusted (in addition to Equation 4) with the new round-trip delay as follows:

$$T_d = T_d + (RT - RT_{accum}) \quad (6)$$

$$RT_{accum} = \alpha \cdot RT_{accum} + (1 - \alpha) \cdot RT \quad (7)$$

where RT_{accum} is an accumulative weighted average of RT. α is an application dependent weighting scalar for estimating future RT. In our game system, setting it to 0.5 seems to give reasonably good predictions most of the time. While the command button is still pressed (states A_c to D_c), the motion of the player will be extrapolated using the motion predictor shown in Equation 2, until the command button is released or a new motion command is issued (state D_c). If a new motion command is detected, another synchronization cycle would be initiated (state D_c).

3.2 Client-Client Synchronization

Our reference simulator approach natively handles the client-client synchronization problem. Given a player **A**, there is an avatar representing the player in the game environment and a reference simulator running in the server to model its motion. The motion of **A** (and therefore its avatar) is adjusted gradually to synchronize with that of the reference simulator at the server. If another player **B** needs to interact with **A**, client **B** will send a request to the server to gather the motion information of **A**, and at the same time, it will create a simulator locally to model the motion of **A**. Figure 2 depicts this process.

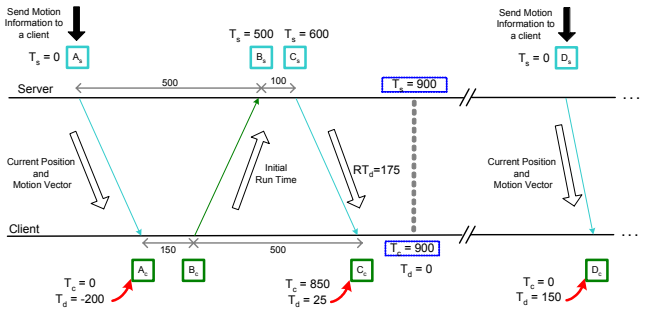

Figure 2: Client-server interactions in the client-client synchronization process.

After client **B** has received the motion information of **A** from the server (state A_c), it sets $T_d = -RT_{accum}$ to start the simulation locally. T_d is set to a negative value since we anticipate that the simulator running in client **B** would start with some delay relative to the reference simulator of **A** at the server. Similar to the client-server synchronization process, whenever a client needs to update the position of an avatar, it will need to update the relevant variables as shown in Equations 3, 4 and 5. In addition, the client will also send an initial run time, i.e., the time duration between state A_c and B_c, to the server to request for the updated round-trip time information to prepare for its subsequent synchronization process. As a result, the gradual synchronization process is performed in a way similar to the one shown in Section 3.1. Finally, since the motion of any local simulator modeling the motion of a remote player would be synchronized against the reference simulator at the server, motion of any particular avatar could ultimately be synchronized among all relevant client machines.

4. RESULTS AND CONCLUSION

To study the performance of our method, we incorporate it into a network game engine that we have developed. This game engine is based on the DVE system that we developed earlier [3]. We have experimented it with three avatar movement patterns: *linear*, *zig-zag* and *circular* paths, under different network latencies, including 0.64ms (for LAN), 10 ms (for Internet - within a city) and 160ms (for Internet - international). In the experiments, player **A** navigates in the game environment and reports its states to the server, where the reference simulator of **A** is being run. Player **B** is the one interested in the motion of **A**, and requests for motion information of **A** from the server. To determine how well our synchronization method works, we monitor the motion timers of various parties for a period of time. We try to determine the discrepancy among the relevant parties. Referring to the motion predictor shown in Equation 2, the motion timer is the only parameter to determine the difference in spatial location between two copies of a shared dynamic object, since they are supposed to move with the same motion vector.

Figure 3 shows the performance of our synchronization scheme. To improve the readability of the results, we use two different scales to depict the state discrepancies under different network environments. Since the state discrepancy values collected from the LAN and from the local Internet connection are generally smaller, they are plotted based on the scale on the primary y-axis (the vertical axis on the left). However, the state discrepancy values collected from the overseas connection are in general much higher and are plotted based on the scale on the secondary y-axis (the vertical axis on the right). As shown in the diagrams, all the relevant parties would in general experience a large state discrepancy when a motion command is just being sent out.

Consider the first set of results as shown in Figures 3(a-c). Since the results from different network environments generally exhibit similar behavior, we focus our discussion on those from the overseas connection, as the effect here is more obvious. Since player **A** navigates in a linear path, only a single motion command is issued. Refer to figure 3(a). At around 0.2s, the server just receives a motion command from the player. Due to network latency, there is a large motion timer difference between the server and **A** at this moment. However, since **A** would start the gradual synchronization process as soon as it has sent a motion command to the server, the difference will be quickly reduced to about half of the round-trip time. In addition, the synchronization process will gradually reduce the discrepancy within a very short period (about 0.45s). Figure 3(b) depicts the motion

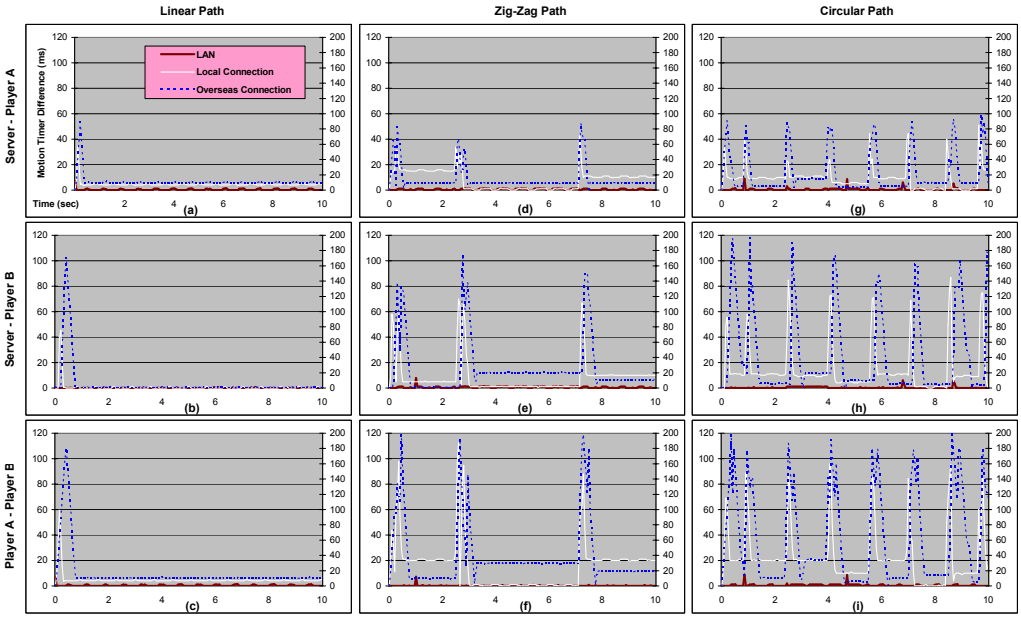

Figure 3: Experimental results of our synchronization scheme.

timer difference between the server and player **B**, which is interested in motion information of **A**. At 0.4s, when **B** has just received the motion information of **A** from the server, there is a large motion timer difference between **B** and the server. Since there is no trivial way to correct the discrepancy at this moment, the amount of discrepancy is equal to the round-trip time. Finally, due to the double round-trip delay between **A** and **B**, the discrepancy between them is expected to be large at the beginning. However, since **A** has started the gradual synchronization process with the server earlier, the discrepancy between **A** and the server would have been much reduced by the time when **B** has received the motion command from **A**. Hence, the discrepancy between **A** and **B** is reduced to about a single round-trip time.

Figures 3(d-f) show the second set of results. Here, player **A** moves in a zig-zag path by pressing different command buttons alternatively to change the movement directions. From the diagrams, we can see that whenever **A** issues a new motion command, a discrepancy pulse appears. We can see that the period from the start of a pulse to the start of the next pulse, e.g., the period from 0.3s to 2.5s in Figure 3(d), exhibits similar behavior as in the first set of the results. This is because during each of these periods, the avatar of **A** essentially moves in a straight line.

Figures 3(g-i) show the final set of results. Here, player **A** moves in a circular path by pressing different command buttons continuously. Overall, this set of results exhibits similar behavior as the second set. In fact, both cases are conceptually similar as they both require the player to continuously issue different motion commands in order for the avatar to move according to the desired paths. The difference is that in this set of results, there are more discrepancy pulses generated within the same period of time as it requires the player to change the command buttons more frequently in order for the avatar to move circularly.

Finally, we can see from the results that some small discrepancy pulses occurs in most of the diagrams even after our synchronization operations, e.g., the period from 0.5s to 2.5s in figure 3(d). This is because our method attempts to correct the discrepancies based on estimating the network latency from recent network communications. This estimation would sometime cause small errors when the network traffics fluctuate significantly.

In summary, we have presented in this paper a method to provide a global-wise continuous synchronization for multiplayer online games. Experimental results demonstrate that our method could provide good consistency control.

Acknowledgments

The work described in this paper was partially supported by a CERG grant from the Research Grants Council of Hong Kong (RGC Reference No.: CityU 1308/03E) and by a DAG grant from City University of Hong Kong (Project No.: 7100264).

5. REFERENCES

[1] P. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, **13**(2):185–221, 1981.

[2] Y. Bernier. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. In *Proc. the Game Developers Conference*, 2001.

[3] J. Chim, R. Lau, H. Leong, and A. Si. CyberWalk: A Web-based Distributed Virtual Walkthrough Environment. *IEEE Trans. on Multimedia*, **5**(4):503–515, December 2003.

[4] DIS Steering Committee. IEEE Standard for Distributed Interactive Simulation - Application Protocols, 1998. IEEE Standard 1278.

[5] Final Fantasy XI. http://www.playonline.com/ff11/home/.

[6] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, **21**(7):558–565, 1978.

[7] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Trans. on Multimedia*, **6**(1):47–57, 2004.

[8] C. Sun and D. Chen. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. *ACM Trans. on Computer-Human Interaction*, **9**(1):1–41, 2002.

[9] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Trans. on Computer-Human Interaction*, **5**(1):63–108, 1998.