Sync-MS: Synchronized Messaging Service for Real-Time Multi-Player Distributed Games

Yow-Jian Lin
Department of Computer Science
State University of New York, Stony Brook
yjlin@cs.sunysb.edu

Katherine Guo Sanjoy Paul
Center for Networking Research
Lucent Bell Labs
{kguo, sanjoy}@bell-labs.com

Abstract

Real-time, online multi-player games are becoming increasingly popular due to advances in game design and the proliferation of broadband Internet access. However, fairness remains a major challenge when players over large geographic areas participate in a client-server based game together. This paper proposes a game-independent, network-based service, called Sync-MS, that balances the trade-off between response time and fairness. Sync-MS uses two mechanisms, Sync-out and Sync-in, to address state update fairness and player action fairness, respectively. Two metrics, ahead and behind measured against the fair order, are defined to evaluate Sync-MS's fairness performance. Simulation results show that Sync-MS dramatically improves player action fairness for all players while slightly increases average response time for players with shorter network delay to the game server.

Key Words: *online multi-player games, real-time, fairness, response time, network delay.*

1 Introduction

This paper describes a value-added service that supports real-time, online multi-player distributed games using advanced networking technologies. Real-time, multi-user distributed applications, such as online multi-player games or distributed interactive simulations (DIS), are becoming increasingly popular due to advances in game design and the proliferation of broadband Internet access such as DSL and cable modem. When players over a wide geographic area play a game together, however, variants in their network performance (such as delay) pose a major fairness challenge. Currently each game application deals with these variants at the application level, a practice that yields drastically different and often unsatisfactory gaming experiences. To gaming network service providers, better network support for more consistent gaming performance could not only attract more subscribers, but also keep them on the network longer, both lead to larger revenue.

Most online multi-player games today are implemented based on a client-server model instead of a peer-to-peer model [3, 4, 6]. In the client-server model, an authoritative game *server* is set up and all players or *clients* logon to this game server to play the game against one another. Player actions are forwarded from each player station to the game server in *action messages*. The game server then processes the actions in sequence, and notifies player stations of the effects of their actions in *state update messages* or simply *update messages*. The only communication in the system is between the game server and player stations. Because of the real-time nature of these games, the majority of action and update messages are sent over UDP. To update states timely, *dead reckoning* is a technique commonly used to compensate for late arrival or lost packets [14, 7].

We identify two fairness issues in client-server based online multi-player games. One concerns the fairness among players in accessing state updates. Since players may experience different network delays in receiving the same state update from the game server, some players can take early actions against the latest update before others have chances to react. The other concerns the order that the game server processes action messages from players. Action messages from a player further away from the game server or connected to the server through congested links may take longer time to reach the server. Without compensating for execution environment differences, the server will process these messages out of the order of their real-time occurrence with respect to state changes.

An improved network support for better fairness in online multi-player games must synchronize the delivery of one-to-many update messages from the server to all players. At the same time, for action messages coming back from players to the server, it must deliver the messages to the server based on the order these messages react to global state updates. To be game-independent, the support should

not require the use of any data embedded in application messages.

1.1 Sync-MS Solution

We propose a network support service called Sync-MS (Synchronized Messaging Service). Sync-MS addresses two fairness issues of online multi-player games, namely: *state update fairness*, and *player action fairness*. Note that Sync-MS does not attempt to shorten message delay for each player station; delay reduction could come from advances in CPU, link speed, or some game specific features.

State update fairness means that all players should receive each game state update from the game server at the same time. When an update message arrives at some player stations before others, Sync-MS uses a *Sync-out* mechanism to properly queue up the message at the player stations and deliver it to the game application only after the same update message has arrived at all player stations. As a result, all players can react to the same global state fairly.

Player action fairness means that the game server will process action messages from all player stations in a fair order based on their real-time occurrence[1]. Because action messages from different players exhibit different delays to reach the server, Sync-MS uses a *Sync-in* mechanism to enforce a sufficient waiting period on each action message dynamically in order to guarantee fair processing of all action messages. In reality, the waiting period at the server is bounded because of the real-time nature of interactive games. We propose several Sync-in algorithms that can trade higher delay at some players against improved fairness among all players.

Sync-MS is designed to be transparent to real-time, multi-user distributed applications, and is well suited for the type of online multi-player games such as client-server based, first person shooter games and role playing games in which a fair order of player actions is critical to the outcome. Sync-MS provides better overall fairness without degrading the perceived interactive nature of online multi-player games.

The rest of the paper starts with a description of related work in Section 2, and the system model for online multi-player games in Section 3. Section 4 describes the Sync-out mechanism for state update fairness, and Section 5 the Sync-in mechanism for player action fairness. The simulation results of various Sync-MS mechanisms are evaluated in Section 6, followed by concluding remarks in Section 7.

[1]This requires synchronized clocks, which is the assumption we make in Section 3.

2 Related Work

Borella [5] and Färber [6] are a few that have studied network game traffic models. In first person shooter games, the study found that the inter-arrival time of action messages in client traffic has the characteristics of either extreme values or deterministic distribution, or a combination of both. The parameters of distribution for each client vary depending on the type of hardware used. The server traffic, mainly update messages, also follows an extreme value distribution.

Regarding the impact of delay on game players, Armitage [2] suggests that a round-trip delay of more than 150 ms could lead to un-smooth gaming experience for players. IEEE DIS (distributed interactive simulation) standards specifies that a round-trip delay of 100–300 ms is appropriate for military simulations [9]. From empirical results of user behaviors, Henderson [8] concludes that application level delay, in other words, response time is in the range of 50–300 ms for most players in first person shooter games.

Bernier [3] discussed latency compensating methods commonly used in client-server based first person shooter games. The methods used are at the application level, and are proprietary to each game. These methods are complimentary to our Sync-MS in the sense that they are aimed at making large delays and message loss tolerable for players.

The fairness issue has been addressed in the form of consistency in peer-to-peer model. MiMaze [7, 13], an online multi-player game, uses a bucket synchronization mechanism. To overcome message delay variants (for delays up to 100 ms), all player stations process each message in the sampling period 100 ms after its sending time. The bucket approach for player stations to process and display latest states "at the same time" is similar to the playout buffer mechanism used in delivering packet audio. Mauve [11] describes a *local lag* concept in networked games and distributed virtual reality applications. Participants delay their local actions for at least the maximum of average network delays between any two participants in order to minimize missing any remote actions in transient. In contrast, Sync-MS deals with client-server based games. The Sync-out mechanism for broadcast messages from the server to all clients is similar to the other two, except that the server is the one dictating the delivery time dynamically. For ordering messages from many clients to one server, since in the Sync-in mechanism the server is the only decision maker, it can better balance the trade-off between delay and fairness without concerning consistency. Synchronizing the trade-off will be difficult in peer-to-peer model.

3 System Model

This section describes the framework for client-server based multi-player games with Sync-MS support. As de-

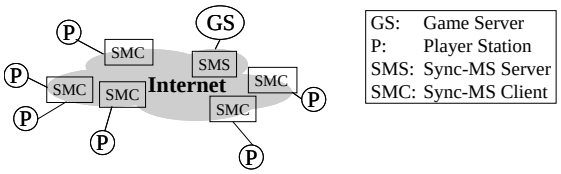


Figure 1. System model for Sync-MS for online multi-player games

picted in Figure 1, *Sync-MS Client (SMC)* and *Sync-MS Server (SMS)* refer to network support modules that interact with player station and game server applications, respectively. They are responsible for delivering messages to game applications in a synchronized and fair fashion. The modules can be collocated with game applications on the same host, or they can run on the edge routers that the hosts are connected to (provided the network delay from a host to its respective edge router is compensated in the algorithm). Communication between the game server and player stations must pass through their associated SMS and SMCs. We assume that hosts and edge routers are capable of identifying and redirecting game service subscribers' data traffic to Sync-MS support modules based on some packet classification schemes. Each player station is connected to an SMC, and each SMC can serve multiple player stations.

We assume that hosts and routers have synchronized clocks using Network Time Protocol (NTP) [12] or Global Positioning Systems (GPS). This paper concentrates on the interaction between the SMS and SMCs, and assumes that the game server and player stations have negligible message delays to their directly connected SMS and SMCs.

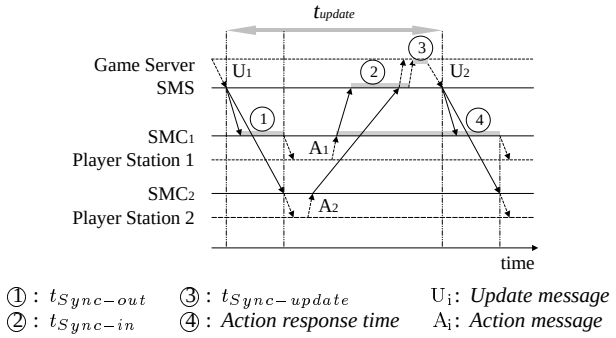


Figure 2. Message passing sequence in client-server model with Sync-MS support

Figure 2 presents a message passing sequence for Sync-out and Sync-in mechanisms in Sync-MS. Detailed descriptions of both mechanisms will follow in Sections 4 and 5.

Sync-out is for synchronized delivery of state update messages to all player stations through SMCs. Typically the game server sends periodical update messages to all player stations. The update period t_{update} could be 40 ms for a smooth showing of game states at 25 frames per second [13], or a configurable value such as 50–200 ms in the Quake game [4, 15]. For each update message, SMS determines how to accomplish Sync-out based on its knowledge of one-way delays from itself to SMCs. Mark ① in Figure 2 indicates the added delay $t_{Sync-out}$ due to Sync-out.

Sync-in concerns fair delivery order of action messages to the game server. Players respond to state updates with action messages, which arrives at the SMS through SMCs. Upon receiving an action message, the SMS must decide how long it needs to hold on to the message before delivering it to the game server for Sync-in. For example, in Figure 2, the sending time of A_2 from Player Station 2, comparing to that of A_1 from Player Station 1, is closer to the arrival time of update message U_1. It implies that Player 2 reacts to the same update faster. Hence, the SMS should not deliver A_1 until it has received and delivered A_2, which we term a *fair order* (cf. Section 6.1.1 for a precise definition of fair order proposed in our study). Mark ② in Figure 2 represents the added delay $t_{Sync-in}$ due to Sync-in.

The effect of an action message is unknown until the game server announces it in the next update message. Mark ③ in Figure 2 refers to the time lag $t_{Sync-update}$ due to periodical state updates.

A player station will not realize the effect of an action message until it receives a state update message containing the effect. We define the *response time* of an action message to be the time interval between the moment an SMC sends the action message and the moment the SMC delivers the update message that contains the effect of the action. Response time consists of five parts: network delay from SMC to SMS; Sync-in delay $t_{Sync-in}$; the time lag at SMS due to state update $t_{Sync-update}$; network delay from SMS to SMC; and Sync-out delay $t_{Sync-out}$. Mark ④ in Figure 2 indicates the response time of action message A_1. Note that, because the effect of A_1 can only appear in the next periodical update U_2, the added Sync-in delay does not increase A_1's response time. However, with Sync-in, message A_2 will receive the added benefit of fair ordering. In general, the response time of an action message could increase in multiples of t_{update}, with the value of $\lceil \frac{t_{Sync-in}}{t_{update}} \rceil \times t_{update}$ in the worst case.

We assume that each update message sent by the SMS includes a tag of the scheduled delivery time. An SMC must deliver each update message no earlier than the time specified in the tag for Sync-out. We also assume that all SMCs will timestamp each action message they send to the SMS with the message sending time, so that the SMS can use it as the basis for Sync-in. The tagging does not depend on any game specific data embedded in the messages, which

makes the proposed approach game-independent.

4 Sync-out

The goal of the Sync-out mechanism is for all SMCs to deliver each update message to all player stations simultaneously. For each update message from the game server to the player stations, the SMS selects a delivery time, and SMCs enforce the delivery schedule.

The SMS chooses a delivery time for each update message based on an up-to-date information it maintains about one-way delay from itself to every SMC. Upon receiving an update message u from the game server at time t, the SMS chooses a delivery time $T = t + D$ for u, where D is greater than the longest one-way delay. It then tags the message u with the time T and forwards u to all SMCs. The SMS can use network delay measurement algorithms [1] to acquire delay information, or rely on the delay guarantees offered through quality assured services.

With the clocks at the SMS and SMCs synchronized, each SMC simply uses its local clock to schedule the delivery of update messages received from the SMS. Assume the update message u with a delivery time tag T arrives at an SMC z at time t_z. If $t_z \leq T$, SMC z delivers the packet at time T. Occasionally, an update message may arrive at an SMC later than the time specified in the delivery time tag. In such cases, the SMC may deliver the update message without further delay, discard the late message, or take other actions depending on specific applications. SMCs can discard late packets because real-time gaming applications have built-in mechanisms to handle late packets or lost packets [3].

5 Sync-in

The SMS runs a Sync-in mechanism to deliver the action messages it receives from SMCs fairly to the game server. We propose three different Sync-in algorithms; each offers a different trade-off between the action response time of individual SMCs and the fairness among all SMCs.

For simplicity, we assume there are n SMCs in the system, and each SMC_i, $1 \leq i \leq n$, handles only one player station. The SMS keeps track of estimated one-way action message delay, $cs(i)$, from SMC_i to the SMS. For each action message A it receives, the SMS retrieves the sending time $s(A)$ from A's tag, and notes the receiving time $r(A)$.

Figure 3 illustrates the concepts of *waiting period* in a Sync-in execution. Assume that an update message U_1 sent by the SMS has reached three SMCs, SMC_1, SMC_2 and SMC_3 at the same time t_1 by virtue of Sync-out. SMC_1 does not react to state updates in U_1, whereas SMC_2 and SMC_3 react by sending action messages A_2 and A_3 at time

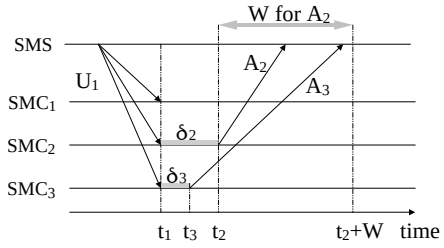


Figure 3. Waiting period in Sync-in

t_2 and t_3 respectively. Because SMC_3 reacts to the state update faster ($\delta_3 < \delta_2$, and thus $t_3 < t_2$), the SMS should deliver A_3 to the game server ahead of A_2, even though it receives A_2 first. To do so, the SMS let A_2 wait in a delivery queue until it is reasonably sure it has received all earlier action messages (such as A_3) from other SMCs. We define the *waiting period* for an action message A to be the time from the sending time $s(A)$ of A till the time the SMS is ready to deliver A to the game server.

The Sync-in objective is to deliver action messages to the game server based on when SMCs generate the actions with respect to state updates. With a Sync-out mechanism in place for state update fairness, the SMS can simply deliver action messages based on their sending time tag, which reflects their fair oder.

The SMS implements a Sync-in algorithm as follows. It maintains a queue of received action messages that are pending for delivery to the game server.The messages in the queue are sorted based on their sending time in ascending order. Let H be the head message in the queue and $s(H)$ be its sending time. The SMS decides a waiting period W for H. If H is still the head message at time $s(H) + W$, then the SMS takes it out of the queue and delivers it to the game server. Otherwise, the SMS repeats the same to the new head message. The three proposed Sync-in algorithms, *MaxWait*, *SelectWait*, and *TossWait* differ in the choice of the waiting period W for the message at the head of the processing queue.

5.1 MaxWait

MaxWait is a straightforward, conservative Sync-in algorithm. By choosing $W = W_M = \max_{1 \leq i \leq n}\{cs(i)\}$, the longest one-way delay from all SMCs to the SMS, the SMS can deliver the head message H at time $s(H) + W_M$, and be reasonably sure that it has received and delivered all action messages sent to it from any SMC earlier than $s(H)$.

There is no guarantee that waiting for W_M will always be long enough to cover all network delay cases. Occasionally a newly arrived message may be sent earlier than the last delivered message. The algorithm considers the new message too late to be processed in a fair order.

5.2 SelectWait

SelectWait is more aggressive in delivering action messages. The idea is to wait for messages from only the subset of SMCs that can generate earlier actions.

We observe that each SMC takes some time to send consecutive action messages. This time interval could be either due to player station hardware or enforced by network support modules. Let $b(i)$ be the time between actions of SMC_i, and $e(i)$ be the sending time of the last known action message sent by SMC_i. The SMS concludes that SMC_i will not send another action message until some time after $e(i) + b(i)$. Hence, for the head message H in the queue, SelectWait chooses $W = W_S = \max\{cs(i) | e(i) + b(i) \leq s(H), 1 \leq i \leq n\}$. Thus, it considers only the $cs(i)$ values of those SMCs that could have generated actions earlier than the sending time of H.

Since the waiting period selection is now based on the message delay of a subset of SMCs, the average added delay $t_{Sync-in}$ of SelectWait is shorter compared to that of MaxWait. Nevertheless, since SelectWait explores the time gap between action messages and is not a random guess, the algorithm does not compromise much to the overall fairness. When the data about the time between actions $b(i)$ are accurate, SelectWait could achieve the best fairness possible with the least added Sync-in delay.

5.3 TossWait

TossWait is a probabilistic approach to the selection of waiting period. As stated earlier, even the use of W_M in MaxWait cannot guarantee the delivery of action messages in 100% fair order. TossWait seeks to add unfairness randomly in exchange for better action responsiveness.

Assume that the current head message is H, with its respective sending and receiving time $s(H)$ and $r(H)$. Note that MaxWait chooses a waiting period $W = W_M = \max_{1 \leq i \leq n}\{cs(i)\}$ and delivers the message H at the time $s(H) + W_M$. Therefore, any choice of W, $r(H) - s(H) \leq W \leq W_M$, can yield some measure of fairness. The closer the value of W is to $r(H) - s(H)$, the lower is the added Sync-in delay and hence the better is the average response time. On the other hand, choosing a W closer to W_M improves fairness for the players with longer message delays.

One simple way to choose the waiting period W in TossWait is to select a value between $r(H) - s(H)$ and $\max_{1 \leq i \leq n}\{cs(i)\}$ with a uniform probability distribution. The simple strategy, nonetheless, may be extremely unfair to the module SMC_k with the longest message delay $cs(k)$. Because of the uniform distribution, the probability of the SMS choosing $W = cs(k)$ as a waiting period is close to 0. As a result, most of the time the SMS will have delivered the messages it received from other SMCs before it receives the one sent at about the same time by SMC_k.

We propose to select waiting periods using a probability function on percentile one-way message delay of SMCs. Let $W_p(X)$ be the X^{th}-percentile delay values, that is, X^{th}-percentile of $cs(i)$, $1 \leq i \leq n$, are less than or equal to $W_p(X)$. The SMS then follows a set of percentile-probability pairs, called *toss vector* of the form $\{(X_j, Y_j) | \sum_j Y_j = 100\%\}$, and uses $W = W_p(X_j)$ Y_j-percent of the time. Note that MaxWait is an extreme instance $\{(100^{th}, 100\%)\}$ of this TossWait strategy.

Given a set of (X_j, Y_j) pairs, we expect any SMC whose message delay is less than $W_p(X)$ will be treated fairly $\sum_{j|X_j \geq X} Y_j\%$ of the time. For example, with a toss vector {(100th,50%) (90th,30%) (80th,20%)}, the SMS running TossWait should treat a SMC with 85th percentile message delay fairly 30%+50%= 80% of the time.

6 Performance Evaluation

This section presents performance results of the Sync-MS mechanisms in client-server based, first person shooter games. The results are acquired by simulating message exchanges between 10 player stations and a dedicated server using Matlab [10]. The server sends out update messages every 40 ms. The inter-arrival time of action messages generated by each player station follows an extreme value distribution [5]. We assume the one-way message delay between the SMS and each SMC is symmetric and exponentially distributed. We also assume the delay between each player station and its corresponding SMC is negligible, same is the delay between the sever and the SMS.

Table 1 summarizes the parameters used in the experiments. We use the terms "player" and "player station" interchangeably, given that commonly there is an one-to-one relationship between them. To evaluate the effectiveness of Sync-MS on a variety of network delays, we assign mean delay values to players almost uniformly from 4 to 30 ms. Players with smaller IDs have shorter delay mean values. The low-end mean values simulate players residing in the same LAN environment as the server, whereas the high-end ones represent players being some distance away from the server or using low speed connections. Their delay variance is roughly proportional to the delay mean, and is small to reflect the assumption that game service providers offer quality network service. Players having the same action message inter-arrival time signifies their use of similar player station hardware. We have run experiments with various mixes of delay and inter-arrival time parameters and found minor differences from the results presented here.

Table 2 lists the combination of algorithms simulated in each run. TossWait1 uses a toss vector [(100th, 50%) (50th, 30%) (10th, 20%)], which means that maximum delay is used for waiting period 50% of the time, 50th percentile

Player		1	2	3	4	5	6	7	8	9	10
Message Delay (between SMS & SMC)	Mean	4	6	8	10	11	15	20	25	28	30
	Variance	0.8	0.8	1	1	1	1	1	2	2	2
Action Message Inter-arrival Time	Mean	18	18	18	18	18	18	18	18	18	18
	Variance	1	1	1	1	1	1	1	1	1	1

Table 1. Message delay and action message inter-arrival time parameters (in ms)

	No Sync-in	With Sync-in
No Sync-out	NsoNsi	NsoWsi (MaxWait Sync-in)
With Sync-out	WsoNsi	MaxWait, SelectWait, TossWait1, TossWait2, TossWait3

Table 2. Combinations of algorithms

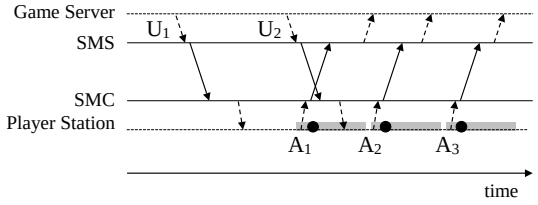


Figure 4. Identifying the respective update message

delay is used 30% of the time, and so on. TossWait2 uses a toss vector [(100^{th}, 30%) (50^{th}, 70%)], and TossWait3 uses [(100^{th}, 20%) (30^{th}, 30%) (10^{th}, 50%)].

In each experimental run, we first generate a sequence of update messages sent by the SMS throughout the simulation time. Each update message has its respective sending time at the SMS and its delay time to each SMC, with the first update message being sent at time 0. We also generate for each player (thus each SMC) a sequence of action messages. Each action message has its sending time relative to time 0 and its delay time to reach the SMS. For each algorithm combination in the run, the message sending time of this action message sequence is then time-shifted based on the arrival time of the first update message to produce the sending time of action messages for the combination. Note that all simulated combinations in a run share the same update message sequence. However, depending on whether a combination includes Sync-out, and whether we use estimated or exact delay when Sync-out is simulated, the arrival time of the first update message at each SMC might be different for different combinations, resulting in different sending time for each action message sequence. Nevertheless, these action message sequences share the same delay and inter-arrival time characteristics, which enables us to compare various proposed algorithms.

We ran five runs of each algorithm combination. The simulation time for each run is 80,000 ms, long enough for each player to send more than 4,000 action messages. The SMS processes over 40,000 action messages for each simulated combination. The results are the average of five runs.

6.1 Terms and Definitions

This section provides detailed description of terms critical to the evaluation of Sync-MS mechanisms. See Section 3 for the definition of terms such as response time.

6.1.1 The Fair Order of Action Messages

The *fair order* of action messages is based on their *virtual sending time*, which we derive from the actual sending time of each action message with respect to the arrival time of update messages. One key question is which update message we should relate to for each action message. To answer it we look closely into how games are implemented. According to Bernier's description [3] of a generic client's frame loop (running at a player station), a client checks for new arrival of update messages immediately after sending out an action message. If an update message were not readily available by then but arrived later during the same frame loop, the client would not render the update in the next frame. As a result, the actions collected and sent during the next frame will be based on the previous update extrapolated over simulation time.

Figure 4 illustrates the point. The three gray bars depict three frame loops run at a player station, whereas the three bullets signify the time the player station attempts to read update messages sent by the server. After the player station sends action message A_1 and is ready to read update messages, only U_1 is available. Hence, the scene rendered during the first bar is based on U_1, which is also the one that actions in A_2 act upon. Similarly, the actions included in A_3 are based on the scene rendered during bar 2, after the player station sends out A_2 and reads U_2 from its input queue. We call the update message U that an action message A is based on the *respective update message* of A, and label it $rum(A)$. For example, in Figure 4, $rum(A_2) = U_1$, and $rum(A_3) = U_2$.

We define the *virtual sending time* of an action message A_i to be $vst(A_i) = s(A_i) - (r(rum(Ai)) - s(rum(A_i)))$,

Delivery Order	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Fair Order	16	18	19	15	20	17	22	23	14	24	25	26	27	28	21
Ahead	-2	-3	-3	-1	-2	-1	-2	-2		-1	-1	-1	-1	-1	
Behind				3		3			8						7

Table 3. An example of the ahead and behind measures

where $s(x)$ and $r(x)$ are the sending and receiving time of message x. That is, the virtual sending time of an action message effectively were the action time should the respective update message take zero delay. The fair order among action messages follows naturally based on their virtual sending time.

6.1.2 The Ahead and Behind Measures

The proposed ahead and behind measures are for evaluating the fairness among delivery orders generated by different combinations of algorithms. An action message is behind by p steps in a delivery order if p action messages behind it in the fair order are delivered ahead of it. Similarly, an action message is ahead by q steps if q action messages ahead of it in the fair order are delivered behind it. Note that an action message could be p messages ahead and q messages behind at the same time, over different sets of messages. Table 3 is a sample of ahead and behind measures taken from an experimental run. Each column corresponds to an action message from some player, with its delivery order at the SMS, its calculated fair order, and its ahead and behind measures. The sender and the ID of each message are irrelevant here and are omitted. For example, the 15th message in terms of the fair order, delivered as the 17th in the delivery order, is one message ahead (of the 14th fair order message, delivered at 22nd place) but three messages behind (the 16th, 18th, and 19th fair order messages).

By summarizing the percentage of behind messages for each player, we can compare how unfairly each player has been treated by the server. Similarly, the percentage ahead measure indicates how often a player's action messages have been taking advantage of others'.

6.1.3 Delay and Inter-arrival Time Estimation

In simulating both Sync-out and Sync-in mechanisms, the algorithms must compute expected delay between the SMS and each SMC as well as, in the case of SelectWait, the inter-arrival time of action messages. We use an exponential-weighted window average on the one-way delay of past messages from and to SMC_i to derive the present mean and variance. Given its current delay mean D_{mean}, and variance D_{var}, the most recent message delay d and its variance $d_{var} = \mid d - D_{mean} \mid$, we first update the mean and variance

$$D_{mean} = (1 - \alpha) \times D_{mean} + \alpha \times d$$

$$D_{var} = (1 - \beta) \times D_{var} + \beta \times d_{var}$$

The estimated delay D is then $D = D_{mean} + N \times D_{var}$, where N is a scale function based on the latest variance d_{var}. We use $\alpha = \frac{1}{8}$, $\beta = \frac{1}{4}$, and

$$N = \begin{cases} 4 & \text{if } d_{var} > D_{var} \\ 2 & \text{if } \frac{1}{2} D_{var} \leq d_{var} \leq D_{var} \\ 1 & \text{otherwise} \end{cases}$$

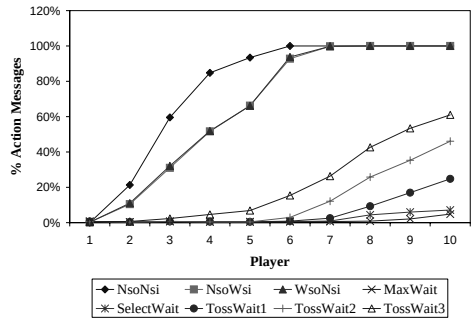
The estimation of action message inter-arrival time follows similar approach. To exclude the effect of inaccurate estimation in our study, we also run experiments based on the exact delay of each message.

6.2 Simulation Results

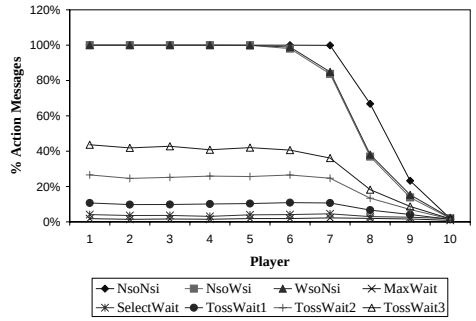
6.2.1 General Results

Figures 5(a) and 5(b) illustrate the fairness improvement due to various Sync-MS mechanisms. The results are the average of five runs using estimated delay and inter-arrival time for each combination of algorithms listed in Table 2. As noted in Table 1, message delay between the SMS and SMCs increases as player ID increases. With both Sync-in and Sync-out of Sync-MS, the fairness improves dramatically over the no Sync-MS case (NsoNsi). As shown in Figure 5(a), players with shorter delays, such as players 1 to 5, experience almost no messages being delivered behind. At the same time, Figure 5(b) shows that only a small portion of their messages (less than 10%, except in TossWait2 and TossWait3) are taking advantage of others and being delivered ahead. Players with longer delay receive much fairer treatment as well. For example, in Figure 5(a), Player 10 only has 7% of messages being delivered behind with SelectWait Sync-in, as opposed to 100% behind without Sync-out or Sync-in (NsoNsi). The lines in Figures 5(a) and 5(b) representing NsoNsi, NsoWsi and WsoNsi cases also show that either Sync-out or Sync-in alone has marginal improvement. However, overall fairness improves significantly, if both update and action messages are delivered fairly.

To accomplish this kind of fairness, Sync-MS increases the response time of players that exhibit shorter message



(a) Behind measures



(b) Ahead measures

Figure 5. Comparison of fairness measures (estimated delay and inter-arrival time used in both Sync-in and Sync-out)

delay. Each line in Figure 6 depicts the average response time of action messages sent by each player over the five runs. MaxWait increases the average response time of all players to the same value which is the maximum response time across all players. Other Sync-in algorithms improve upon the average response time of MaxWait at the expense of a slight decrease in fairness compared to MaxWait, as shown in Figures 5(a) and 5(b). However, as long as the average response time is still within an acceptable range of network gaming (for example, a delay bound of 100–300 ms for distributed interactive simulation [9]), this is a reasonable trade-off. Notice MaxWait has its limitations. If certain players have a response time higher than the tolerable threshold of game players (300 ms), MaxWait will slow down every player to the maximum response time. SelectWait and TossWait are still applicable in this scenario, because they can lower the average response time by sacrificing fairness for the players with longer delays.

As shown in Figures 5(a), 5(b) and 6, among different Sync-in mechanisms, SelectWait constantly exhibits fairness performance slightly lower than but close to MaxWait, with shorter average response time across all players. For each player, TossWait1, TossWait2 and TossWait3 in that

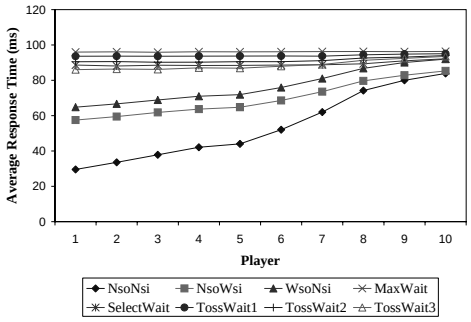


Figure 6. Average response time (estimated delay and inter-arrival time used in both Sync-in and Sync-out)

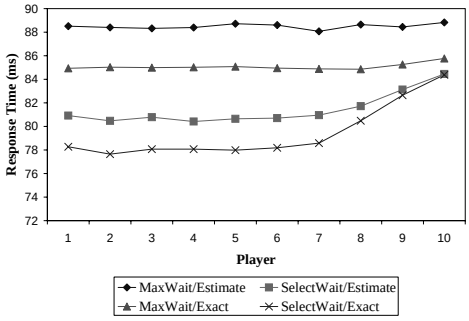
order, exhibits increasing percentage of messages delivered out of fair order, but decreasing average response time. Therefore we can see that with different toss vectors, TossWait can strike a balance between average response time and fairness. Note that the average response time of SelectWait is the best possible given the most fair delivery order possible. As shown in Figure 6, with a toss vector such as the one used by TossWait3, TossWait can even have better average response time than that achieved by SelectWait for players with shorter delay (players 1 to 5) [2]. However, TossWait3 did so by delivering more messages out of fair order for all players than SelectWait did.

Performance of Sync-MS depends on estimated message delay. In these five experimental runs, we chose a more conservative delay estimation for Sync-out ($N = 4$) than the one for Sync-in ($N = 1$, 2, or 4 based on current delay variance). That explains why in Figure 6 the response time of WsoNsi is slightly longer than that of NsoWsi.

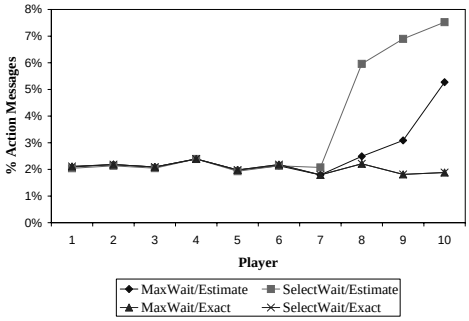
6.2.2 SelectWait vs. MaxWait

SelectWait gears toward the best possible average response time without sacrificing the fairness achievable by MaxWait. Figures 7(a) and 7(b) compare average response time and message behind percentage of both SelectWait and MaxWait. In this comparison, the Sync-out portion is based on exact delay knowledge so that the delivery of each update message is perfectly synchronized. Estimate and Exact differentiate whether Sync-in is based on estimated or exact action message delay information. In addition, for SelectWait it also indicates whether the inter-arrival time of action messages from each player is estimated or known. As shown in Figure 7(a), SelectWait im-

[2] This conclusion depends on the delay profile of different players. We have simulated other environments where most players are in the same LAN as the server and thus have shorter action message delays, and SelectWait exhibits better average response time compared to TossWait3.

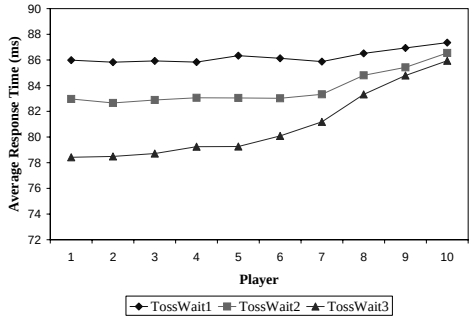


(a) Average response time

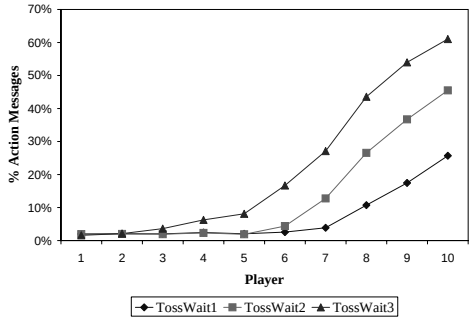


(b) Behind measures

Figure 7. SelectWait vs. MaxWait (with exact delay information used in Sync-out)



(a) Average response time



(b) Behind measures

Figure 8. TossWait performance (with exact delay information used in Sync-out and estimated delay information used in Sync-in)

proves average response time over MaxWait for all players, and the improvement is significant for players with shorter delays. Naturally, in terms of average response time, SelectWait/Estimate performs better than MaxWait/Estimate. In the scenario we have simulated, SelectWait/Estimate even outperforms MaxWait/Exact. In terms of the behind measures shown in Figure 7(b), SelectWait/Estimate exhibits slightly higher percentage for players with longer delays due to estimated inter-arrival times. With exact inter-arrival time and exact delay information for Sync-in, Figure 7(b) also shows that SelectWait/Exact matches MaxWait/Exact completely in delivering fairness.

6.2.3 The Impact of Toss Vectors

While both MaxWait and SelectWait are for achieving the best possible fairness, subject to the accuracy of their delay and inter-arrival time estimation, TossWait allows us to tune the trade-off between average response time and fairness through toss vectors. Figures 8(a) and 8(b) compare the performance of TossWait for different toss vectors. Recall that TossWait1 uses a toss vector [(100^{th}, 50%) (50^{th}, 30%) (10^{th}, 20%)], which means that maximum delay is used for waiting period only 50% of the time, and so on. TossWait2 uses the toss vector [(100^{th}, 30%) (50^{th}, 70%)], and TossWait3 uses [(100^{th}, 20%) (30^{th}, 30%) (10^{th}, 50%)]. To minimize the unfairness introduced by Sync-out, the comparison is based on simulations with exact delay information for Sync-out. TossWait1, TossWait2 and TossWait3 uses less percentage of waiting period based on the 100^{th} percentile delay value in that order (from 50% in TossWait1 to 30% in TossWait2 to 20% in TossWait3). The toss vector of TossWait3 even has 50% chance to wait for 10^{th} percentile delay value. As shown in Figure 8(a), by having a higher percentage of waiting periods to be based on less percentile delay values, TossWait consistently improves average response time across all players.

In terms of the behind measures, the trade-off between average response time and fairness clearly shows in Figures 8(a) and 8(b). Those toss vectors that yield better average response time (shown in the order of TossWait3, TossWait2 and TossWait1 in Figure 8(a)) in turn give worse fairness results (shown in the same order in Figure 8(b)).

Based on the discussion in Section 5.3, the behind measure of Player 10 in TossWait1 should be (100-50)% = 50%. And for TossWait2 and TossWait3, Player 10's behind measure should be (100-30)% = 70% and (100-20)% = 80%,

respectively. In Figure 8(b), however, the behind measures of Player 10 is much lower than the expected values: 25%, 45% and 65% for TossWait1, TossWait2 and TossWait3 respectively. One key factor is due to the way the SMS delivers messages in the queue. Ideally, an action message, given a waiting period, will be delivered at the end of the period. However, since SMS delivers in order those action messages waiting in the queue, a head message with a long waiting period could delay the delivery of subsequent messages. Consequently, even though a toss vector may suggest to wait at least W for Y% of action messages, the actual waiting period these messages encountered could be longer than W. In other words, less than $(100 - Y)\%$ of action messages have a waiting period shorter than W. Notice only an action message with a waiting period shorter than W could cause another message with delay longer than W to be delivered out of fair order. This explains why the behind measures for TossWait are better than expected.

7 Conclusion and Future Work

We have proposed a network-based service, Sync-MS, for real-time, multi-user distributed applications such as online multi-player games. Sync-MS is game-independent, and balances the trade-off between average response time and fairness of player actions. It is well suited for the type of client-server based, multi-player games in which a fair order of player actions is critical to the outcome. Its Sync-out mechanism dynamically synchronizes the delivery of state updates from the server to all players so that players can react to the same update fairly. Its Sync-in mechanism delivers action messages from all player stations in a fair order to the game server.

We have introduced the concept of fair order among all player action messages based on the real-time occurrence of messages with respect to state updates. Two metrics, ahead and behind measured against the fair order, are defined to evaluate Sync-MS fairness performance. We also proposed three Sync-in algorithms, MaxWait, SelectWait and TossWait, each of which provides different average response time and fairness trade-offs.

Simulation results show that with Sync-MS, action fairness improves dramatically. Players with longer delay can now have less than 10%, instead of 100%, of their action messages being delivered behind. Players with shorter delay will experience increased average response time, but Sync-MS keeps the increase within a reasonable range. Among different Sync-in algorithms, MaxWait offers the best fairness but longest average response time. Compared to MaxWait, SelectWait exhibits similar fairness performance but shorter average response time. TossWait can balance between improved average response time and reduced fairness performance through the use of toss vectors. Nevertheless, many factors also affect the performance outcome, in addition to a particular vector used. We plan to further investigate their effects.

Acknowledgment

The authors thank anonymous reviewers for many useful comments. Our colleagues M.C. Chan, S. Mukherjee, S. Rangarajan, and M. Vernick helped improve an earlier draft of this paper.

References

[1] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *Proc. of ACM SIGCOMM'99*, Sept 1999.

[2] G. Armitage. Sensitivity of Quake3 Players to Network Latency. In *Proc. of IMW'01, Workshop Poster Session*, Nov 2001. URL: `http://www.geocities.com/gj_armitage/q3/quake-results.html`.

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

[4] S. Bonham, D. Grossman, W. Portnoy, and K. Tam. Quake: An Example Multi-User Network Application – Problems and Solutions in Distributed Interactive Simulations. Technical report, CSE 561 Term Project Report, University of Washington, May 2000. URL: `http://www.cs.washington.edu/homes/grossman/projects/561projects/quake/`.

[5] M. S. Borella. Source Models of Network Game Traffic. *Computer Communications*, 23(4):403–410, Feb 2000.

[6] J. Färber. Network Game Traffic Modelling. In *Proc. of NetGames2002*, Apr 2002.

[7] L. Gautier and C. Diot. Design and Evaluation of MiMaze, a Multiplayer Game on the Internet. In *Proc. of IEEE Multimedia (ICMCS'98)*, 1998.

[8] T. Henderson. Latency and User Behavior on a Multiplayer Games Server. In *Proc. of NGC'01*, pages 1–13, Nov 2001.

[9] IEEE. *1278.2–1995, IEEE Standard for Distributed Interactive Simulation – Communication Services and Profiles*, Apr 1996.

[10] MathWorks. URL: `http://mathworks.com`.

[11] M. Mauve. Consistency in Replicated Continuous Interactive Media. In *Proc. of ACM Conference on Computer Supported Cooperative Work (CSCW'00)*, pages 181–190, 2000.

[12] D. Mills. Simple Network Time Protocol (SNTP) version 4 for IPv4, IPv6 and OSI. *RFC-2030*, Oct 1996.

[13] L. Pantel and L. Wolf. On the Impact of Delay on Real-Time Multiplayer Games. In *Proc. of ACM NOSSDAV'02*, May 2002.

[14] S. Singhal and D. Cheriton. Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality. *Presence: Teleoperators and Virtual Environments*, 4(2):169–193, 1995.

[15] R. Swamy. idSoftware Releases Quake 1 Source Code Under the GPL. URL: `http://linuxtoday.com/stories/14111/html`.