Semi-Lock: An Efficient Cheat-Proof Synchronization Mechanism for Peer-to-Peer Game Systems

Huaping Shen[1], Sajal K. Das[2] and Mohan Kumar[2], and Zhijun Wang[3]

[1] Ask Jeeves Inc., Piscataway, NJ, 08854, USA,
hshen@ask.com
[2] Dept of Computer Science and Engineering,
University of Texas at Arlington, Arlington, TX 76019, USA,
das,kumar@cse.uta.edu
[3] The Dept of Computing, Hong Kong Polytechnic University, Hong Kong,
cszjwang@comp.polyu.edu.hk

Abstract. With more and more online game players having access to broadband Internet connections and high performance computers, peer-to-peer architecture offers an attractive solution for online multiplayer game design. However, message synchronization and cheat proof are two major challenges in implementing a fully distributed peer-to-peer game system. In this paper, a novel scheme, called *Semi-Lock*, is proposed to support message synchronization and prevent protocol level cheats for such systems. In Semi-Lock, peers encrypt each message by using cryptographically secure one-way hash function. A two-step validation is applied at destination peers to verify the integrity and correctness of the received messages. Trace driven simulations are conducted to verify the performance of our proposed scheme. Simulation results show that it significantly outperforms existing algorithms in terms of message latency and message synchronization ratio, with only little extra computational overhead on each peer.

1 Introduction

Real-time multiplayer online games are becoming increasingly popular due to the advancement of game design and the availability of broadband Internet access to the end users. As predicted in [8], the revenue of worldwide online gaming will grow to 5.2 billion dollars by 2006.

Existing online multiplayer games typically use a client-server architecture. Players send messages to a central server and the server broadcasts the update messages to all players. This architecture has advantages that a single authoritative server orders messages, acts as a central repository for data, and is easy to secure. However, the centralized client-server architecture also has several disadvantages. First, since every command must go from client to server and then be re-sent by the server to other clients. This adds additional latency over the minimum cost of sending commands directly to other clients. Second, the server becomes a single point of failure in the game, and traffic at the server increases with the number of players which may result in localized congestion. Last, this architecture is limited by the computational power of the server. Although, scalability can be achieved by employing server clusters or computing grid, this solution incurs significant cost.

Recently, a fully distributed peer-to-peer architecture is introduced to address the above problems [4][5]. Unlike the client-server architecture, in which a single authoritative copy of the game state is kept at the server, in this peer-to-peer game architecture, each client keeps a copy of the entire game state. Each peer is allowed to send messages directly to other peers, which reduces the latency of message and eliminates localized congestion and single point failure. In order to support massively multiplayer online games, all peers are divided into different peer groups according to the locality interest in the virtual world [5], so that the event in one group will not affect other groups of the game, and players in one group form a peer-to-peer network in order to exchange event messages. A peer can be elected as the leader of one group to handle the message exchanges with other multiple groups [4].

In a peer-to-peer game architecture, each peer may experience different network delays in receiving messages from different peers. Messages from a peer further away or from one with congested link may take longer time to arrive at destination peer than that from a nearby peer or one with better network condition. Thus, to support the fairness of a peer-to-peer game, a synchronization mechanism is required at each peer to ensure the fairness of the messaging among the peers and to guarantee the consistency of the game state at each peer. Another challenge lying in the peer-to-peer game architecture is to prevent game cheats. As the definition given in [4], a game cheat is any action by a player that gives her an unfair advantage over another player. Instead of using an authoritative game server to maintain the game state, each peer keeps a copy of the game state. Thus, the peer-to-peer game architecture increases the opportunities of game cheating than the client-server architecture.

In this paper, we focus on the message synchronization and cheat-proof in the peer-to-peer game architecture. A novel mechanism, called Semi-Lock, is proposed to support message synchronization and prevent protocol level cheats while providing low message latency in such systems. In order to prevent protocol level cheats, when a peer initializes an action message, the message is encrypted by using cryptographically secure one-way hash function and is sent to other peers. After waiting a period of D_{max}, where D_{max} is a tuning parameter based on the network condition, the peer resend the plain-text message to other peers. After receiving a plain-text message, a two-step validation is applied to verify the integrity and correctness of the received message. In Semi-Lock, the game time is divided into equal frame intervals (FI). At each peer, valid messages received in one FI are executed according the rule of game application, and the game state is updated and saved at the end of each FI. Rollback operation is conducted to correct the game state if late messages are received. By adjusting the FI and D_{max} parameters, we show that Semi-Lock can meet different requirements of game designs. Trace driven simulations are conducted to verify the performance of our proposed scheme. Simulation results show that it significantly outperforms existing algorithms in terms of message latency and message synchronization ratio, with only little extra computational overhead on each peer.

The rest of this paper is organized as follows. Section 2 proposes the Semi-Lock algorithm. Implementation issues of the proposed algorithm are discussed in Section 3. Section 4 presents performance evaluation of Semi-Lock and compares it with existing approaches. Section 5 concludes the paper and discusses future research.

2 Proposed Semi-Lock Scheme

As defined in [4], game cheats can be classified by layer in which they occur, such as application level cheats, protocol level cheats, and network level cheats. Application level cheats happen when cheaters modify the code of game or the operating system to gain unfair benefit. Network level cheats happens when cheaters use inherent properties of network layer. A denial of service (DoS) attack is an example of a network level cheat. On the other hand, protocol level cheats happen when cheaters read, modify or blocks the packets of game protocol to gain unfair benefit. One typical protocol level cheats is *timestamp cheat*. In peer-to-peer games, since each peer maintains its own game state, each game message must be timestamped when generated so that they can be executed at the same relative time by each peer. In timestamp cheat, after receiving a message from peer 1, peer 2 issues a message whose timestamp is before peer 1's and sends out to all peers. Thus, if peer 1 wants to attack peer 2 in his message, peer 2 can dodge the attack by cheating.

Semi-Lock focuses on the prevention of protocol level cheats, while providing low latency messaging synchronization for the peer-to-peer game architecture. In order to simplify the discussion, let P_i denote peer i in a game session and F_k denote the kth game frame interval (FI). m_{i,t_s} is a message that is generated by P_i at time t_s. We use $S_{i,F_k}^{t_k}$ to denote the game state that is generated by P_i at frame interval F_k and t_k is the commit time of last message executed in F_k. $P = \{P_1, P_2, ..., P_n\}$ is the set of peers participating in the game session. Let $D_{i,j}$ denote the network delay between P_i and P_j. We define a system parameter D_{max} as the maximal link delay among all peers in terms of number of FIs, which is given as follows:

$$D_{max} = \lceil \frac{\max\{D_{i,j} | \forall P_i, P_j \in P\}}{FI} \rceil \times FI \qquad (1)$$

2.1 Semi-lock Algorithm

In Semi-Lock, all peers in one game session are assumed to have a synchronized clock by using Network Time Protocol (NTP). Network delay measurement algorithms [1] are used to acquire delay information $D_{i,j}$ between peers. When a peer, say P_i generates a game message m_{i,t_s}, the timestamp t_s is included into the message. At the same time, P_i sends out hash of message, $H(m_{i,t_s})$, to other peers in the game session. After waiting for a time period of D_{max}, the peer P_i sends out the plain-text message m_{i,t_s} to other peers. Since Semi-Lock commits the hashed message instantly after message generation, we call t_s as the *commit time* of message m_{i,t_s}, and call the time when P_i sends out the plain-text message m_{i,t_s} as the *reveal time* of this message. When a peer, say P_j, receives a hashed message $H(m_{i,t_s})$, P_j keeps $H(m_{i,t_s})$ in a hash message list L_j^H. The plain-text messages are kept in a list L_i^m, which is sorted by commit time of each message.

Semi-Lock uses a hybrid approach to synchronize game messages at each peer. The game time is broken into equal frame intervals (FI). Different from existing game protocols, in Semi-Lock, a peer is allowed to send multiple messages in an FI. At the end of each FI, all plain-text messages received in this interval are sequentially executed according to their timestamps. Before each execution, each plain-text message needs to be validated to prevent cheats as two steps as follows: *(i), the hash value of plain-text message is compared to the hash message in the hash message list to check the integrity of plain-text message; (ii), the difference between message commit time and current time should be less than* $2D_{max}$ *and more than* D_{max}. After the execution of all messages in a frame interval, the current game state is updated and saved in a list, called *state list* L_i^s. Each peer maintains a state list which is sorted by the generation time of each state. In one execution, if the commit time of a message is earlier than the commit time of a message in the previous saved game state, a rollback operation is conducted to correct all out-of-order messages, and the states after out-of-order message are replaced by correct game states. In order to reduce memory requirement, the game states whose commit time of last message is D_{max} earlier than current time, are removed from the game state list. The detailed description of Semi-Lock algorithm is given in Figure 1.

Figure 2 illustrates the Semi-Lock scheme. At time t_1, peer P_1 issues a game message m_{1,t_1} and commits the hash of message $H(m_{1,t_1})$ to the game session. (For clarity purpose, only game states of P_2 is illustrated and we omit the message exchange between P_1 and P_3.) Next, at time t_2, peer P_3 issues a message m_{3,t_2} and commits the hash of message $H(m_{3,t_2})$. After time period of D_{max} ($D_{max} = 4FI$) from their commit time, both P_1 and P_3 send out their plain-text messages. The delay between P_3 and P_2 is less than that between P_1 and P_2. Although, P_1 sends m_{1,t_1} before P_3 sending m_{3,t_2}, the message m_{3,t_2} arrives P_2 before m_{1,t_1}'s arrival. P_2 executes m_{3,t_2} in the frame interval after state S_2 and generates new state S_1. When m_{1,t_1} arrives at P_2 in the frame interval after state S_1, the game state rollbacks to S_2, and S_1 is recalculated by re-executing m_{3,t_2}. Finally, new state S_0 is generated and saved in the state list.

Theorem 1. *The Semi-Lock algorithm is safe: For any message m, no message that is generated after m's reveal time will be executed before m at any game instance of peers.*

```
Begin Sender Procedure{}
    At time $t_s$, peer $i$ (i.e., $P_i$) issues a game message $m_{i,t_s}$
    $P_i$ sends out hash message $H(m_{i,t_s})$ to other peers
    Wait for $D_{max}$ period
    $P_i$ sends out plain-text message $m_{i,t_s}$ to other peers
End
Begin Receiver Procedure{}
    If $P_i$ receives a hashed message $H(m_{j,t_s})$
        Keep $H(m_{j,t_s})$ in list $L_i^H$
    If $P_i$ receives a plain-text message $m_{j,t_s}$
        Keep $m_{j,t_s}$ in list $L_i^m$ sorted by $t_s$
End
Begin Frame Generation Procedure{}
    At frame interval $F_k$, when $P_i$ generates a game frame
    Two-step validation for each message in the list $L_i^m$
    $M_r = \{m_{j,t_s} | \forall m_{j,t_s} \in L_i^m,\ t_s < t_{k-1} | S_{i,F_{k-1}}^{t_{k-1}} \in L_i^s\}$
    $M_c = L_i^m - M_r$
    If $M_r \neq null$
        The latest state $S_{i,F_q}^{t_q}$ whose $t_q < \min\{t_s | \forall m_{j,t_s} \in M_r\}$
            Rollback and correct all states $S_{i,F_e}^{t_e} |\ q < e < k$
    Execute all messages in $M_c$
    Save state $S_{i,F_k}^{t_k}$ where $t_k = \max\{t_s | \forall m_{j,t_s} \in M_c\}$
End
```

Fig. 1. Semi-Lock Algorithm

For the limited space, the proof of the safety of Semi-Lock algorithm is skipped in the paper.

2.2 Optimized Semi-Lock Algorithm

The basic Semi-Lock algorithm reduces the response time of each message by allowing peers to optimistically advance their game without waiting to receive all message from other peers. This leads to a better performance of responsiveness for Semi-Lock than the Lockstep protocol. However, since each peer still needs to wait for a period of D_{max} to reveal its plain-text message, Semi-Lock still suffers from poor responsiveness if there exists a peer with bad connection in the game session. In the following, we discuss two methods for optimizing the performance of Semi-Lock while still maintaining the same level security as the basic Semi-Lock algorithm.

In Semi-Lock scheme, unicast, multicast, or structured peer-to-peer overlay could be used to send messages among peers. Message transmission between any two different peers P_i and P_j requires time $D_{i,j} > 0$. According to Theorem 1, if we guarantee that after receiving plain-text message m, no peer can issue a message that will be executed before m, the safety of algorithm is not compromised. Thus, for any peer P_i, the waiting period D_{max} between commit time

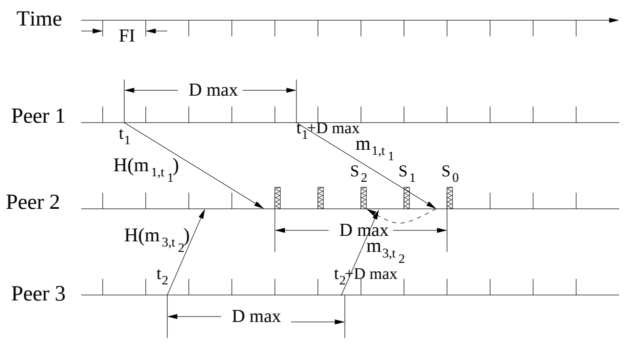

Fig. 2. Illustration of Semi-Lock

and reveal time can be reduced by D^i_{min}, which is P_i's minimal delay to other peers, i.e., $D^i_{min} = \min\{D_{i,j} | \forall j \neq i, P_j \in P\}$.

Furthermore, if peer-to-peer games use unicast to exchange messages among peers, optimized Semi-Lock can further reduce the waiting period of each message while still maintaining the same level security as the basic Semi-Lock algorithm. With unicast communication, after commit time of a message m, P_i will wait for time period of $D_{max} - D_{i,j}$ before sending plain-text message m to P_j, where $D_{i,j}$ is the link delay between P_i and P_j. If the link delay estimation is accurate, the plain-text message m will arrive P_j late than D_{max} after message's commit time, so that after receiving plain-text m no peer can issue a message that can be executed before m.

Since each message will arrive its destination peer D_{max} time later than its commit time, all messages will be sequentially executed according to their commit time. Therefore, no rollback operation is required for optimized Semi-Lock algorithm under unicast communication when delay estimation is accurate. In later performance evaluation section, we show that optimized Semi-Lock can also achieve zero rollback performance under multicast communication when the game session uses only one multicast group to disseminate messages.

3 Implementation Issues

In order to use Semi-Lock in real peer-to-peer game systems, there are several issues that need to be addressed. In this section, we address two critical implementation issues of Semi-Lock in real network environments.

3.1 Handling Message Loss

In above section, we assume there exist reliable communication channels among peers and messages never get lost. However, in real networks, loss of packets always happens. We need to consider three different cases for Semi-Lock when messages get lost. *Case 1*: The hash message is lost but the plain-text message is received. In order to differentiate this situation from the cheats, the peer will send out a verification request to other peers which includes the plain-text message. The other peers compare the plain-text message with their valid message and send back the verification confirmation. If the majority of other peers confirm the validation, the plain-text message can be correctly executed. *Case 2*: If the hash message is received, but the plain-text message is lost, the peer can retrieve the valid plain-text message from other peers excluding the source peer. *Case 3*: Both hash message and plain-text message are lost. In this case, the peer still can reconcile the game state with other peers by retrieving the valid messages from other peers. Therefore, Semi-Lock can provide consistent game states to all peers even in network environments with packet losses.

3.2 Concealing Rollback

The other concern is how to conceal the visual artifacts that may result from the rollback operations of Semi-Lock. Occasionally, rollbacks will cause some drastic changes, such as the player in a first person shooting (FPS) game coming back to life when he is thought to be killed. One approach to minimize the occurrence of these artifacts is to classify the messages into different levels of significance. We delay the execution of significant messages such as a player's death in case there is a rollback.

4 Performance Evaluation

In this section, we evaluate Semi-Lock under different parameter settings. Two existing cheat-proof protocols, namely Lockstep [2] and NEO [4] are compared with Semi-Lock in the performance evaluations.

4.1 Experiment Setup

We design a trace driven simulator to evaluate the performance of Semi-Lock. Because there is no peer-to-peer game available in the Internet, we use Quake III [9] which is a popular first person shooting game based on client-server architecture to collect the trace data. We use five Quake III clients to connect to one Quake III server in the Internet and record the game traffic locally with tcpdump. We then use Ethereal [10], which has a built-in packet decoder for Quake III, to filter out everything except the game messages sent out by our client. In these trace files, there are 30-50 action messages per second that are generated by clients and sent to the server. We use the trace data in 270 seconds from game start time of each trace file to simulate a game session that lasts for 270 seconds.

In our simulator, the Transit-Stub method of GT-ITM model [7] is used to generate a hierarchical interconnecting network. Similar to SimMud [5], we use Pastry peer-to-peer overlay [6] to support peer-to-peer routing and also use Scribe that is built on top of Pastry to provide application level message multicast in our simulations.

4.2 Experiment Results

In each of the following experiments, if there are n peers in one game session, they are randomly selected from 1600 nodes in the simulated network. 10 experiments with different random seeds are conducted, and the average result of these experiments is recorded as one final result. The primary metric we use to measure the performance is the message response time which is the time from when a peer first sends out action message to when the message is executed and displayed on the screen of the other peers.

Basic Semi-Lock vs Optimized Semi-Lock In this experiment, we explore the performance of the basic Semi-Lock, say *SemiLock(B)*, under different frame intervals and number of peers, and it is compared with optimized Semi-Lock, say *SemiLock(O)*. We change the game frame interval, FI, of each peer and evaluate the message response time and the number of rollbacks of Semi-Lock under different number of peers in the game session.

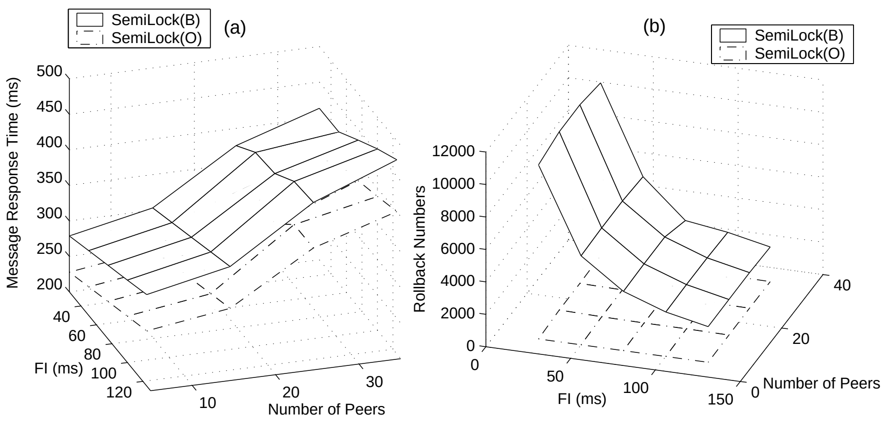

Fig. 3. Performances under different frame intervals and number of peers

As shown in Figure 3(a), the message response time is reduced as the frame interval decreases, but it increases as the number of peers increases in one game session. This is because when the FI decreases, each message needs to wait for less time for execution after being received by the destination peer. The increasing of number of peers incurs the increase of maximal link latency, D_{max}, among peers in one game session which leads to the increase of message response time. Figure 3(a) also shows that the optimized Semi-Lock algorithm reduces the message response time by about 50 ms.

Let the rollback number be the number of rollback operations that are executed in one peer in one games session of 270 seconds. In Figure 3(b), the rollback number drops as the frame interval increases, and slightly increases as more peers participate in one game session. As FI increases, more messages will be synchronized in one frame interval, so less misording messages need to be corrected by rollback operations. This results in the drop of rollbacks. As shown in Figure 3(b), the optimized Semi-Lock significantly reduces rollback operations as compared to the basic Semi-Lock. In simulations, we use the Scribe [3] multicast protocol to multicast the game messages, and all peers in the game session join one multicast group. In Scribe, a multicast tree is built among all peers to disseminate the messages in one group, so every multicast message is routed to the root node of the multicast tree, then sent to other peers. So the minimal delay D^i_{min} of each peer P_i is the delay between P_i and the peer that is closest to root of the multicast tree. Since the optimized Semi-Lock algorithm reduces the waiting time of each message by minimal delay D^i_{min} for each peer P_i, it makes all message arrive the root peer in the same sequence as the messages were generated, thus few rollbacks are needed to correct the misording messages.

Different Message Rates In this experiment, we compare Semi-Lock with NEO and Lockstep under different message rates with five peers in the game session. We set the round time of NEO is 100 ms in the experiment. Since Quake III game has a high message rate, each player generates about 30-50 messages per second. We measure that the average interval between two consecutive messages is around 38 ms based on the measurement of 45 trace files we got. In order to evaluate the performance of each scheme under different game message rates, we vary the message rate by delaying each message multiple times of the original message interval.

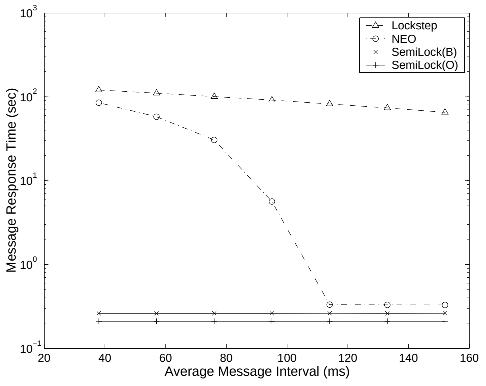

Fig. 4. Different message rates

As shown in Figure 4, Lockstep has the worst response time performance under different message rates. NEO has similar message response time as Semi-Lock at the low message rates. When the message rate increases, (i.e., average message interval decreases), the message response time of NEO significantly increases, especially, when average message interval less than 100 ms. Whereas Semi-Lock constantly has low message response time under different message rates. This is because Semi-Lock allows sending multiple messages in one frame interval. However, NEO allows only one message sending in each round so that messages are congested when the message interval is less than the round duration. Although we can reduce the round duration of NEO to relieve the congestion, this will lead to more misording messages. In Lockstep, each peer needs to receive all committed messages from other peers and then sends out the plain-text message, which results in poor message throughput. Therefore, we conclude that the Lockstep and NEO protocols are not suitable for the game with high message rate.

5 Conclusions

In this paper, we addressed two important issues in the design of peer-to-peer game systems: message synchronization and cheat proof. A novel scheme called *Semi-Lock* is proposed to support message synchronization and prevent protocol level cheats while providing low latency for peer-to-peer game systems. Trace driven simulations are conducted to verify the performance of Semi-Lock. Simulation results show that the Semi-Lock significantly outperforms two existing algorithms in message latency and synchronization with little more computation overhead on each peer. In our future work, we plan to evaluate the performance of Semi-Lock in packet loss network environments. Investigating Semi-Lock in support of massively multiplayer games is also part of our future work.

References

1. M. Allman and V. Paxson, "On Estimation End-to-End Network Path Properties", *In Proc. of ACM SIGCOMM'99*, September 1999.
2. N.E. Baughman and B.N. Levine,"Cheat-proof Playout for Centralized and Distributed Online Games", *IEEE INFOCOM*, 2001.
3. M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron,"Scribe: A Large-scale and Decentralized Application-level Multicast Infrastructure", *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8) pp. 100-111, October 2002.
4. C. GauthierDickey, D. Zappala, V. Lo, and J. Marr,"Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games", *ACM NOSSDAV'04*, June 2004.
5. B. Knutsson, H. Lu, W. Xu and B. Hopkins,"Peer-to-Peer Support for Massively Multiplayer Games", *IEEE INFOCOM*, 2004.
6. A. Rowstron, and P. Druschel,"Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems", *Proc. of IFIP/ACM Intl. Conf. on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany November 2001.
7. Ellen W. Zegura, Kenneth Calvert, and M. Jeff Donahoo, "A Quantitative Comparison of Graph-based Models for Internet Topology" *IEEE/ACM Transactions on Networking*, December 1997.
8. http://www.dfcint.com/game_report/Online_Game_toc.html
9. id Software, 'Quake'. http://www.idsoftware.com/.
10. Ethereal Network Analyzer, http://www.ethereal.com/