A Distributed Architecture for Massively Multiplayer Online Services with Peer-to-Peer Support

Keiichi Endo[1], Minoru Kawahara[2], and Yutaka Takahashi[3]

[1] Kyoto University, Kyoto 606-8501, Japan (endo@sys.i.kyoto-u.ac.jp)
[2] Ehime University, Matsuyama 790-8577, Japan (kawahara@cite.ehime-u.ac.jp)
[3] Kyoto University, Kyoto 606-8501, Japan (takahashi@i.kyoto-u.ac.jp)

Abstract. This paper deals with Massively Multiplayer Online Services, users of which communicate interactively with one another in real time. At present, this kind of services is generally provided in a Client-Server Model, in which users connect directly with a central server managed by a service provider. However, in this model, since a computational load and a communication load concentrate on the central server, high-performance server machines and high-speed data links are required. In this paper, we propose an architecture for distributing the loads by delegating part of the server's function to users' machines, which is based on a Peer-to-Peer Model.
When we adopt a Peer-to-Peer Model, we face a security problem. In our research, we let multiple machines of users manage the same data and apply a decision by majority rule. This mechanism adds robustness against halts of users' machines and data tampering by malicious users to the service.

Key words: peer-to-peer, cheat-proof, interactive, real-time applications

1 Introduction

In recent years, more and more people make use of those interactive services such as online games, online chats, online auctions, and online trades, users of which send and receive information in real time. In the services, many users connected to the Internet share the database. In this paper, this kind of services is called Massively Multiplayer Online (MMO) services.

At present, MMO services are generally provided in a Client-Server Model, in which users connect directly with a central server managed by a service provider. However, this model has a disadvantage that a computational load and a communication load concentrate on the central server. High-performance server machines and high-speed data links are required to provide an MMO service in the model. Therefore, many researchers have devised methods to provide the service in a Peer-to-Peer (P2P) Model. In a P2P Model, users contribute CPU cycles, memories, and bandwidths to the service, and the load

on the server machine is reduced. However, there are some problems to be solved for a practical use. First, the service must have robustness against halts of users' machines. That is, the service must not be stopped or lose data by user's sudden disconnection. Second, the service must overcome network congestion. Users' machines are apt to get communication interruption over several seconds. Finally, the service must not be influenced by cheating. Delegating part of the server's function to users' machines increases the opportunities for illegal acts such as data tampering.

In this paper, we try to solve the above-mentioned problems by letting multiple machines of users manage the same data and applying a decision by majority rule. We compare the proposed model with Client-Server Model and an existing P2P Model by performance analysis. We focus mainly on MMO role playing game services here, however, the model with appropriate modifications can be applied to other services such as online chats, online auctions, and online trades.

The rest of this paper is organized as follows. Section 2 presents related work. In Section 3, we discuss our model. Section 4 describes the detailed algorithms used in our system. Performance analysis is stated in Section 5. We conclude and discuss future work in Section 6.

2 Related Work

Bucket Synchronization [1] is a distributed synchronization mechanism which enables a P2P multiplayer game to maintain state consistency regardless of network delay. This mechanism divides time into fixed-length periods and a *bucket* is associated with each period. A user sends an action message to other users when he takes an action. Receiving the message, a user stores it in the bucket corresponding to the period during which it was sent. Actions in a bucket are executed to compute each user's local view some time after the end of the period for the bucket. The game *Age of Empires* is an example of real applications based on Bucket Synchronization. This mechanism has several problems about cheating. For instance, a user can unjustly gain an advantage by deciding his action after getting the information of what other users do for the current period, which is called *lookahead cheat.*

Lockstep Protocol [2] solves this problem by forcing each user to commit all the actions for the current period before the actions are revealed. In this protocol, each user at first sends a cryptographic hash of his action, instead of the content of the action. However under the protocol, *action delay*, i.e. time required for an action to be displayed on users' screen after the action is taken by a user, is longer than three times the latency of the slowest link between any two users. *Asynchronous synchronization* is also suggested to shorten the average of action delay, but it worsens *jitter* (how much action delay varies). This property is unfavorable especially for game applications.

Adaptive Pipeline Protocol [3] improves the jitter by adjusting its parameter to adapt to network conditions. However, the action delay under the protocol is still rather long.

Reference [4] proposes a way to make action delay shorter by permitting rollbacks when state data become inconsistent among users. However, displaying wrong information temporarily is not acceptable for many kinds of applications. Though it is possible to decrease the frequency that rollbacks are carried out at the cost of action delay [5], the delay becomes too long if we attempt to make the probability low enough because of network congestion.

NEO Protocol [6] achieves a shorter action delay without the need of rollbacks by letting each user accept an action message from another user if the majority of users receive it during a fixed-length period. Though this protocol realizes a high-performance and cheat-proof system on a fixed topology, cheat-proof processing methods of storing state data and user's moving into another *group* (what is called *site* in the next section) are not stated, and this problem is probably difficult to solve. Moreover, users with high-latency data links are not able to use the service. For those users, making use of the service with some disadvantages due to the slow link would be better than being refused to use the service. A service provider also would like to get as many users as possible, provided that slow users do not cause other users inconvenience (e.g. longer action delay).

In the architectures discussed above, each user manages state data which he needs to know, and directly receives action messages from other users. We call these *user-centric architectures* in this paper. The architecture has some drawbacks. One of those is that a user must receive an action message with a message header, which consists of an IP header etc., separately from each user. That is quite inefficient especially in a game application where users frequently send small packets to one another.

The following studies discuss *server-centric architectures*. In the architecture, only one user has an authority to update a certain portion of state data, which provides a consistent environment for all users.

Mercury [7] is a distributed publish-subscribe system for game applications. The routing algorithm in this architecture takes $O(n)$ hops on average, where n is the number of users. References [8] and [9] reduce required routing hops to $O(\log n)$ by the use of Distributed Hash Table. Reference [8] also achieves high durability of state data by replicating them. In all these architectures stated in this paragraph, each part of state data is exclusively managed by one user. Therefore, these architectures have cheating problems such as data tampering.

3 Proposed P2P Model

The P2P Model we propose is similar to server-centric architectures stated above, but differs in that *multiple* users manage each part of state data.

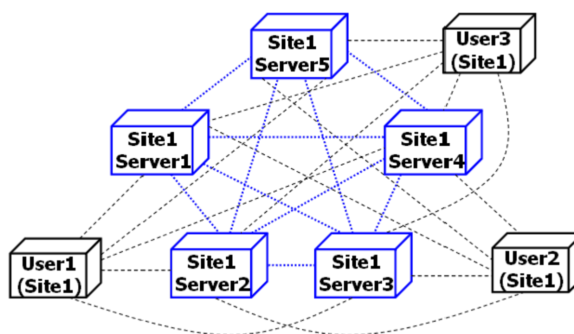

Fig. 1. Model of managing a site.

Although the MMO service has many simultaneous users, a user can always see a limited area, which is called a site in this paper. That is, there are some sites in the service, and each user belongs to one of them. A user can move from one site to another. User's action is made known in real time to only those users staying on the same site. This property makes it possible to transfer the service function of a site to users' machines. User's machine managing a site is called a Site Server.

Fig. 1 illustrates how Site Servers manage one site in our model. Each user is connected with all the Site Servers which manage the site where the user is. All the Site Servers managing the same site form a complete graph. When a user takes an action, the user sends a message to all the Site Servers in order to tell the action. Then, Site Servers update state data, which are classified into two types: those of a user and those of a site. For instance, in a role playing game service, the position of a user and information of items put on a map (site) correspond to state data of a user and those of a site, respectively. When updating state data, Site Servers synchronize processing order of users' actions to have the same state data as one another's. Finally, the Site Servers notify the users on the site of the update. For robustness against cheating, each user accepts the update which the majority of the Site Servers send.

4 Algorithms

4.1 User's Logging in

When a user starts using the MMO service, the first thing to do is to connect[1] with the Administration Server. Then the Administration Server authenticates the user.

[1] All connections in our system are TCP connections.

4.2 Transferring the Service Function of a Site

While there are only a few users, the MMO service is provided in a normal Client-Server Model, i.e., the service functions of all the sites are in the Administration Server. As the number of users increases, the load on the Administration Server becomes higher and higher. Therefore, the service function of a site is transferred to users' machines on a certain condition. For example, the transfer is carried out when the number of users on a site reaches a given parameter. By this mechanism, the users' machines, which are called "peers" in the following discussion, gradually form a (Hybrid) P2P Network.

When the Administration Server decides to transfer the service function of a site, it chooses N_{server} peers which the service function is delegated to, where N_{server} is a fixed positive odd number. Although the selection can be made on various strategies, currently we adopt a simple way, which is to make a random choice from all the peers not managing a site yet. In order to decrease the influence of cheating, the Administration Server also avoids selecting a user on the site whose service function is about to be transferred.

4.3 Handling Action Messages

When a user takes an action except logging in or out, the user makes an action message, which represents the user's behavior. If the user is on a site which is managed by the Administration Server, the user sends the message to the server. After receiving the message, the server immediately processes it and updates state data. The message can change not only state data of the user but also state data of the site and/or those of other users on the site. Then, by sending an update message to all the users on the site (or to users who have the necessity of knowing the state update), the server lets the change of conditions be known.

On the other hand, if the site is managed by Site Servers, the procedure stated in the rest of this subsection is carried out. The user sends the action message to all the Site Servers which manage the site. The message includes an action number for identification. After receiving the message, each of the Site Servers memorizes the user number, the content of the message, and the time when the message is received. The clocks of the Site Servers are synchronized in advance by a mechanism like NTP (Network Time Protocol) [10]. Unlike the case of an Administration Server, the Site Server does not process the message at this time. A mechanism which makes the processing order of action messages the same between the Site Servers is required. In our system, the processing order of action messages is determined using the median of the arrival time at each Site Server. The reason why a median time is used is to diminish the influence of cheat. The next paragraph shows the details.

At intervals of a fixed span T_{timeslot}, a Site Server sends a Timeslot Message to all the other Site Servers of the same site. The message consists of the site number, the transmission time (time stamp), and the sets of {user number,

action number, arrival time} for all the action messages which are received after the last transmission of a Timeslot Message. When a Site Server receives a Timeslot Message, the server memorizes the message and updates state data in the algorithm shown below.

1. From actions such that the information of them is received from at least N_{majority} Site Servers including the server itself, the server picks out the actions whose N_{majority}-th earliest arrival times are older than any time stamp of the last Timeslot Message from each Site Server. N_{majority} is a minimum integer which is greater than the half of the number of Site Servers managing the same site, i.e.,

$$N_{\mathrm{majority}} = \frac{N_{\mathrm{server}} + 1}{2}. \tag{1}$$

   This step means that the server singles out the actions whose median arrival times are determined. If there are not such actions, the update of state data is not done.
2. The Site Server updates state data by processing the action messages corresponding to the picked actions in the order of N_{majority}-th earliest arrival time. If there is an action message which the server has not received yet, the server waits for the arrival of the message from the user who has taken the action.
3. The server sends an update message to the users.

A user receives the same update messages in the same order from all the Site Servers of the site unless there are malicious users or network congestion. The user does not accept an update message until the user receives N_{majority} update messages with the same contents from different Site Servers for fear that the MMO service should be attacked by some cheaters. After the update message is accepted, the image on the screen is updated.

To reduce communication load, $(N_{\mathrm{majority}} - 1)$ Site Servers actually sends only a hashed value computed from an update message.

4.4 Overcoming Network Congestion

The algorithm stated in the foregoing subsection needs to be improved because the action delay is strongly affected by network congestion. For instance, when Timeslot Messages from a Site Server to not less than N_{majority} Site Servers are delayed, the service function of the site is stopped. We solve this problem in the way stated below.

We let T_{timeout} be a fixed span which is longer than T_{timeslot}. If a Site Server has not received a Timeslot Message from another Site Server at T_{timeout} after the time stamp of the last Timeslot Message, the server sends a Timeslot Request Message (TRM) to all the other Site Servers of the same site except the server which the Timeslot Message has not come from. If a Site Server receives a

TRM, the server checks whether the server has received the requested Timeslot Message before. If it has, the server sends back the requested message.

If the server receives TRMs from all the Site Servers which the server has sent the TRM to, or if a given length $T_{\mathrm{req_timeout}}$ has passed since the transmission of the TRM, the server ignores the delayed Site Server for a period of T_{ignore}. Ignoring a Site Server means disregarding Timeslot Messages from the server and performing ordering of users' actions without the server when updating state data. In this algorithm, a Site Server which is ignored by one Site Server is ignored by all the other Site Servers of the same site, too. If the delayed message is not received within the period of T_{ignore}, the Administration Server is requested to choose an alternative Site Server. This mechanism adds robustness against halts of users' machines to the service.

4.5 Switching Sites

When a user moves to another site, the user gets the IP addresses of the Site Servers managing the site from the Administration Server. The Administration Server receives the state data (or the hashed value of the state data) of the user from all the Site Servers managing the current site. Then, after verifying the rightness of the state data by comparison, the Administration Server sends the data to all the Site Servers managing the destination site.

As already stated, it is undesirable for a site to be managed by a user on the site. Therefore, if a user moves to a site which is managed by the user, the Administration Server selects an alternative Site Server and orders the moving user to send all the data concerning the site to the new Site Server. The Administration Server also notifies the other Site Servers and all the users on the site that the Site Server has been changed.

4.6 User's Logging out

When a user finishes using the MMO service, the Administration Server stores the state data of the user. In case that Site Servers have the data, the Administration Server requests the Site Servers to send back the data and stores the majority of them. If the user has a service function of a site, the user sends all the data about the site to the Administration Server, and the Administration Server gives the data to a newly selected Site Server.

5 Performance Analysis

5.1 Communication Load

In terms of communication load, we compare the proposed P2P Model with Client-Server Model and NEO Protocol, which is considered to be one of the

best user-centric architectures. If only unicast can be used, the total size of messages a user has to send for one action in each model is:

$$\text{Client-Server: } S_{\text{header}} + a, \tag{2}$$
$$\text{NEO: } (n-1)\{S_{\text{header}} + a + \lceil (n-1)/8 \rceil + S_{\text{key}}\}, \tag{3}$$
$$\text{Proposed: } N_{\text{server}}(S_{\text{header}} + a). \tag{4}$$

In the above formula, a represents the size of an action message, and n denotes the number of users in the same site. S_{header} and S_{key} mean the sizes of a message header and a decryption key to prevent lookahead cheat, respectively.

The total size of messages a user must receive to know other users' respective actions in each model is:

$$\text{Client-Server: } S_{\text{header}} + na, \tag{5}$$
$$\text{NEO: } (n-1)\{S_{\text{header}} + a + \lceil (n-1)/8 \rceil + S_{\text{key}}\}, \tag{6}$$
$$\text{Proposed: } N_{\text{majority}}(S_{\text{header}} + na) + (N_{\text{server}} - N_{\text{majority}})(S_{\text{header}} + S_{\text{hash}}), \tag{7}$$

where S_{hash} represents the size of a hashed value computed from an update message. In both P2P Models, we assume that each user sends an action message at the same rate as action messages are processed. For example, if T_{timeslot} equals 100 milliseconds in the proposed model, each user transmits an action message every 100 milliseconds. We also suppose that the size of an update message equals 'the number of users in the same site' times 'the size of an action message.' Under the substitution[2]

$$S_{\text{header}} = 66,\ S_{\text{hash}} = 16,\ S_{\text{key}} = 8,\ N_{\text{server}} = 5,\ n = 30, \tag{8}$$

and $a = 10$ (this is enough for telling a user's move in most cases), the values of (5) – (7) are 366, 2552, and 1262, respectively. Under the condition (8), the proposed model achieves less communication load on users than NEO Protocol if $a \leq 31$. This is because of the advantage that a user can receive the result of multiple users' actions at the same time.

If a user has a service function of a site where n users are staying, he must bear additional load as follows:

$$\text{Send: } n(S_{\text{header}} + na) \quad \text{(if he sends the content of an update message)}, \tag{9}$$
$$n(S_{\text{header}} + S_{\text{hash}}) \quad \text{(if he sends the hash of an update message)}, \tag{10}$$
$$\text{Receive: } n(S_{\text{header}} + a). \tag{11}$$

Moreover, the user and the Administration Server must exchange state data when a user logs in, logs out, and switches sites. However, this is not such a big problem since users who can not afford to manage a site do not have to do that.

[2] $S_{\text{header}} = 20$ *(TCP Header)* $+ 20$ *(IP Header)* $+ 26$ *(Ethernet Header)* $= 66$.

5.2 Action Delay

Average action delays in respective models can be approximated as follows:

$$\text{Client-Server: } 2l_{\text{cu}}, \tag{12}$$

$$\text{NEO: } 2l_{\text{uu}} + (T_{\text{arrival}}/2), \tag{13}$$

$$\text{Proposed: } 2l_{\text{su}} + l_{\text{ss}} + (T_{\text{timeslot}}/2), \tag{14}$$

where l_{cu}, l_{uu}, l_{su}, and l_{ss} denote one-way transfer latencies between a central server and a user, between two users, between a Site Server and a user, and between two Site Servers, respectively. T_{arrival} is what is called *arrival delay* in [6], which is similar to T_{timeslot}. T_{arrival} and T_{timeslot} can be reduced close to zero at the expense of communication load. The proposed model requires one more hop than NEO Protocol, but actually the difference is small since the latencies have the following relation if users with high-speed data links are selected as Site Servers:

$$l_{\text{uu}} > l_{\text{su}} > l_{\text{ss}}. \tag{15}$$

This is because not only users' geographical locations but also the performance of lines for connecting to the Internet has a big impact upon transfer latencies.

5.3 Robustness against Network Congestion

To investigate robustness against network congestion, we calculate the probability that a user fails to have his action processed under the condition that a link between two machines becomes temporarily unavailable with the probability p. The probability of the failure in each model is:

$$\text{Client-Server: } p, \tag{16}$$

$$\text{NEO: } \sum_{k=\lceil n/2 \rceil}^{n-1} \binom{n-1}{k} \cdot p^k (1-p)^{n-1-k}, \tag{17}$$

$$\text{Proposed: } \sum_{k=N_{\text{majority}}}^{N_{\text{server}}} \binom{N_{\text{server}}}{k} \cdot p^k (1-p)^{N_{\text{server}}-k}, \tag{18}$$

where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{19}$$

In Client-Server Model an action message from a user is not processed if he can not transmit it to the central server, while in P2P Model an action message is successfully processed in most cases unless half of the destination nodes fail to receive it.

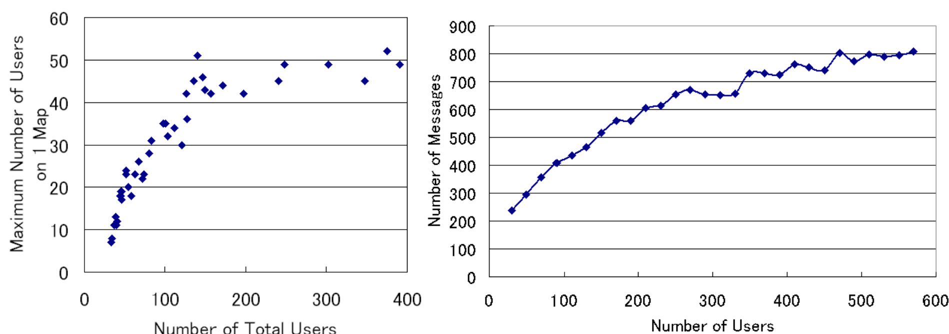

Fig. 2. (a) [left] Number of users on the most popular map. **(b)** [right] Number of messages sent to the most popular chat channel in an hour (averaged at intervals of 20 users).

5.4 Scalability

In this subsection, we consider what happens when the number of users increases.

Fig. 2**(a)** shows the relation between the number of total users and the number of users on the most popular *map* (what we call *site* in this paper). The sample data was collected on August 13, 2004 from a real MMO game service, which is provided in a Client-Server Model. In the game world there are 53 maps, which seem to be ample for 400 users. The number of users in a site seems to have a saturation point.

Fig. 2**(b)** illustrates the relation between the number of users and the number of messages sent to the most popular *chat channel* in an hour. We collected the sample data from the real MMO game service in January through May of 2004. A *chat channel* means a room where users can talk with one another. We can regard it as a site. In the service, users are able to create new chat channels whenever necessary. The maximum number of messages to one chat channel is almost independent of the number of users when lots of users are playing the game.

It is inferred from these data that the number of users on the most popular site is almost stable against the increase of total users when there are a lot of users. This is because gathering of too many users in one site makes the users uncomfortable in most cases. In conclusion, too much load is hardly ever put on Site Servers, provided that there are sufficient sites in the application (or users can make new sites freely). Putting a limit on the number of users in one site may be a good idea, because users in a too crowded site can not normally use the service anyway (e.g. slow drawing on users' screen).

The communication load on the Administration Server in the proposed P2P Model varies directly with the number of total users. However, the server can provide the service for much more users than Client-Server Model, because the

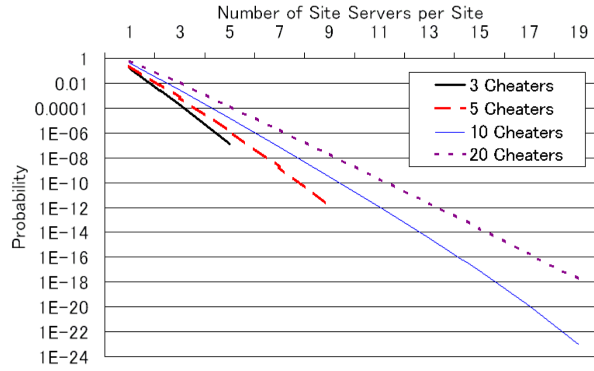

Fig. 3. Probability that colluding cheaters can affect the service.

amount of data sent or received is dramatically decreased by transferring the service function of a site to users' machines. It will also be possible to do away with the Administration Server and make the model completely scalable by the use of other technologies such as distributed storage system.

5.5 Robustness against Cheating

In the proposed P2P Model, multiple cheaters must collude to succeed in cheating (when $N_{\mathrm{server}} > 1$). We let $C_i(N, c)$ represent the probability that there is at least one site where more than half of the Site Servers are controlled by malicious users when there are i sites, c colluding cheaters, and N users in total. Then the probability can be calculated by using the following recurrence equation:

$$C_i(N,c) = 1 - \sum_{k=0}^{N_{\mathrm{majority}}-1} \frac{\binom{c}{k} \cdot \binom{N-c}{N_{\mathrm{server}}-k}}{\binom{N}{N_{\mathrm{server}}}} \cdot \{1 - C_{i-1}(N - N_{\mathrm{server}}, c-k)\} \tag{20}$$

Fig. 3 illustrates how the probabilities $C_{250}(5000, 3)$, $C_{250}(5000, 5)$, $C_{250}(5000, 10)$, and $C_{250}(5000, 20)$ change with N_{server}. The probability that colluding cheaters can affect the service decreases exponentially with the number of Site Servers managing one site.

6 Conclusion

In this paper, we have proposed an architecture for distributing server's loads by delegating part of the server's function to users' machines.

The proposed architecture has the following characteristics compared with other studies.

1. We applied the method to the MMO service that the majority of multiple machines which manage the same data is accepted. It prevents data from being falsified or lost. Moreover, the service has robustness against network congestion.
2. The users whose machines manage a site are selected from those who are not on the site by an Administration Server. It reduces the benefit of cheat and makes conspiracy difficult.
3. The processing order of action messages is determined using the median of the arrival time at each Site Server. It leads to the tolerance for the alteration of time stamps on messages.
4. Not all of the users' machines manage sites. Therefore, if we properly determine the rule of transferring the service function of a site, users with narrow bandwidths, with low-performance machines, or behind firewalls (or NAT routers) can use the service.

As future work, we plan to consider the robustness against cheating in greater detail. We also have a plan to make a practical experiment using a real MMO game service.

References

1. C. Diot and L. Gautier, A Distributed Architecture for Multiplayer Interactive Applications on the Internet, *IEEE Network* **13**(4), 6–15 (1999).
2. N. E. Baughman and B. N. Levine, Cheat-Proof Playout for Centralized and Distributed Online Games, *Proceedings of Infocom 2001* (2001).
3. E. Cronin, B. Filstrup, and S. Jamin, Cheat-Proofing Dead Reckoned Multiplayer Games, *Proceedings of ADCOG 2003* (2003).
4. E. Cronin, A. R. Kurc, B. Filstrup, and S. Jamin, An Efficient Synchronization Mechanism for Mirrored Game Architectures, *Multimedia Tools and Applications* **23**(1), 7–30 (2004).
5. J. Brun, F. Safaei, and P. Boustead, Tailoring Local Lag for Improved Playability in Wide Distributed Network Games, *Proceedings of NSIM 2004* (2004).
6. C. GauthierDickey, D. Zappala, V. Lo, and J. Marr, Low Latency and Cheat-Proof Event Ordering for Peer-to-Peer Games, *Proceedings of NOSSDAV 2004* (2004).
7. A. R. Bharambe, S. Rao, and S. Seshan, Mercury: A Scalable Publish-Subscribe System for Internet Games, *Proceedings of NetGames 2002* (2002).
8. B. Knutsson, H. Lu, W. Xu, and B. Hopkins, Peer-to-Peer Support for Massively Multiplayer Games, *Proceedings of Infocom 2004* (2004).
9. T. Iimura, H. Hazeyama, and Y. Kadobayashi, Zoned Federation of Game Servers: a Peer-to-Peer Approach to Scalable Multi-Player Online Games, *Proceedings of NetGames 2004* (2004).
10. D. L. Mills, Network Time Protocol (Version 3) Specification, Implementation and Analysis, *RFC 1305* (1992).