Vol. 46 No. 2 IPSJ Journal Feb. 2005

Regular Paper

Distributed Scalable Multi-player Online Game Servers on Peer-to-Peer Networks

Takuji Iimura,† Hiroaki Hazeyama† and Youki Kadobayashi†

Today's Multi-player Online Games (MOGs) are challenged by infrastructure requirements because of their server-centric nature. Peer-to-peer overlay networks are an interesting alternative if they can implement the set of functions that are traditionally performed by centric game servers. In this paper, we propose a *Zoned Federation Model* (ZFM) to adapt MOGs to peer-to-peer overlay networks. We also introduce the concept of *zone* and *zone owner* to MOGs. A zone is some part of the whole game world, and a zone owner is a game sever of a specific zone. According to the demands of the game program, each node actively changes its role to a zone owner. By dividing the whole game world into several zones, workloads of the centric game server can be distributed to a federation of zones. In order to reduce response latency overhead on data exchanges between a zone owner and its clients, we limit the use of a Distributed Hash Table (DHT) to the rendezvous point of each zone; actual data exchanges are carried out through direct TCP connection between a zone owner and its members. We also use the DHT as backup storage media to cope with the resignation of a zone owner. We have implemented this zoned federation model as a middle layer between the game program and the DHT, and we evaluate our implementation with a prototypical multi-player game. Evaluation results indicate that our approach enables game creators to design scalable MOGs on the peer-to-peer environment with a short response latency which is acceptable for MOGs.

1. Introduction

Today's Multi-player Online Games (MOGs) are constructed in a server-centric model. To achieve scalability, MOGs usually employ clusters of game servers. In spite of scalability, clustering technologies cost game creators co-location fees. Therefore, starting a new MOG for small or medium enterprises is difficult. Also, in today's server-centric solutions, users cannot play the game when the centric game server stops its services.

In this paper, we try to achieve an alternative MOG infrastructure by using peer-to-peer overlay technologies. The peer-to-peer overlay network is a highly distributed network or computing environment. Nodes on a peer-to-peer overlay network compose a network on the application layer and share resources of each node.

To construct a peer-to-peer overlay infrastructure for MOG, we model a "Zoned Federation Model (ZFM)". ZFM is a latency optimizing approach based on peer-to-peer overlay technologies. The key idea of the ZFM is as follows: numerous participating user nodes on an peer-to-peer overlay network compose a large game server cluster. The ZFM creates several client-server groups on a peer-to-peer overlay network, as shown in **Fig. 1**. Constructing a small-scale client-server model in each zone, the ZFM can achieve as short a response latency as that of the server-centric MOG solutions.

In the ZFM, the tasks of a centric-server are distributed into a peer-to-peer overlay network. We partition the game world into several *zones* by the locality of the game world or the locality of the game data's features. In addition, we let participating nodes play a cluster of the game server on each zone. The node playing a cluster of the game server establishes a direct connection to each client node in the same way that the direct connection between the server and clients in the client-server model is connected. Hence, each server node can serve the latest game status with almost the same performance as that of a centric-server. Each zone is independent from each other; therefore, a user node can become the server on several zones and play the client of some zones.

The cluster of the game server in the ZFM is maintained by participating user nodes autonomously. According to the peer-to-peer overlay environment or the game sequence, each participating user node changes its role to either a cluster of the game server on some zone, the client of the game on other zones, or to resources for constructing a large network or

† Graduate School of Information Science, Nara Institute of Science and Technology

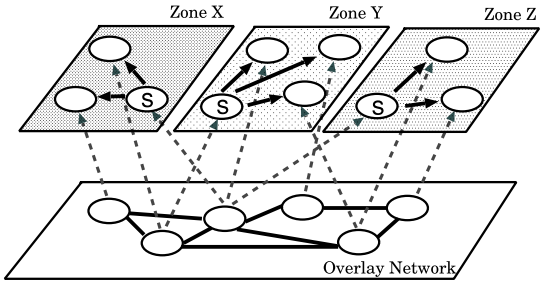

Fig. 1 Zones on the overlay network.

storage. Of course, each participating node can easily resign from the game server task, stop the client role, or leave from the peer-to-peer overlay network. Employing a peer-to-peer overlay network as a backup storage media, the ZFM provides a mechanism for all participating nodes to record the latest game status serialized by the game server role node, to inherit the tasks of the old server role node or the game sequence completely, and to continue serving the game data.

We have implemented the ZFM as a library of a zoning layer, a middle layer between a game program layer, and a TCP/IP stack, or between the game program layer and the Distributed Hash Table (DHT), which is a technique for constructing a peer-to-peer overlay network. We also evaluated the performance of the ZFM implementation with a prototypical multi-player game.

The rest of this paper is organized as follows: we describe the features of today's MOGs and server-centric solutions in Section 2. We mention details of the ZFM, and describe its implementation in Sections 3 and 4, respectively. Section 5 shows the evaluation result of our ZFM implementation with a focus on the response latency.

We refer to related work in Section 7, and finally, we discuss future work and conclude this paper in Sections 8 and 9.

2. Multi-player Online Games

MOGs are such games such that several game users share a game world and play on the game world by exchanging shared game data. We call such shared game data, Global Status Data (GSD). There are several types of MOGs, such as racing, First Person Shooter (FPS), Real Time Strategy (RTS), or Role Playing Game (RPG), etc. Some MOGs, called Massive Multi-player Online Games, are played by thousands or even tens thousands of users.

Typically, a MOG requires a short response latency and a consistency of GSD among users. Short response latency is needed to provide a comfortable game response without stress for users [1)~6)], and the consistency of GSD is required to produce the same game world for all users, as well as to prevent cheating or unfairness in play [7)]. Today's MOG is usually constructed in the client-server model to manage the consistency of GSD, and several clustering or distributing techniques are employed to achieve a short response latency and scalability.

2.1 Global Status Data

In MOG, users have to share GSD to play in the same game world. GSD can be changed according to the game players' demands or the game sequence. Changes of GSD should be serialized and should be synchronized among game players to keep to the consistency of the game world. Therefore, MOG requires some authoritative nodes to provide serializability of state changes and to ensure the consistency of changes. In the client-server model, a centric game server works as this type of authoritative node.

The GSD of the MOG has several localities of interest, and several large-scale MOGs employ interest management techniques [8)]. The examples of localities of interest on a game world are as follows: locality of the network infrastructure, locality of part of game world, locality of the group sharing specific GSD, locality of personal information, etc.

Distributing or clustering techniques use these localities to partition the tasks of a single centric server or the whole GSD. In other words, partitioning GSD or server tasks into several groups by locality enable a reduction in overhead on a single server machine, to achieve scalability or responsibility. Contents Distribution Network technologies focus on the locality of network infrastructure, and SimMud [9)] partitions the game world into several regions with focus on the locality of the area of the game world. Server clusters are constructed by the locality of the group sharing specific GSD, the locality of personal information, or the locality of the numbers of access.

In modeling the ZFM, we focus on the locality of GSD, that is, the locality of the group sharing specific GSD, the locality of personal information, and the locality of the number of access.

Also, GSD should keep its consistency by using an authority to produce the same game world for all users. In the client-server model, a centric server or centric server clusters judge conflicts among users, modify the GSD according to game sequence, serialize changes, synchronize updated GSD on all users to prevent a mismatch, and audit GSD to find cheating or unfairness. Processing these tasks on a distributed environment such as on peer-to-peer overlay networks is difficult. Some distributed agreement protocol [10] can resolve inconsistency on peer-to-peer overlay networks, but such distributed agreement protocol is likely to be complex.

3. Zoned Federation Model

We design the Zoned Federation Model (ZFM) as a model of a MOG on an overlay network by employing five techniques to construct the ZFM: *zone*, the *Distributed Hash Table* (DHT), *zoning*, *mapping*, and *zoned federation*. A zone is a judgment and data transfer block of the GSD to distribute whole GSD into a peer-to-peer overlay network. DHT is a peer-to-peer overlay network technique we employ to construct the ZFM. Zoning is a framework of partitioning whole GSD into several zones to use the DHT as a backup storage media and as a rendezvous point to zones. Zoned federation is a mechanism to let participating user nodes manage each zone and the whole game world autonomously.

3.1 Assumptions

Before describing the ZFM in detail, we assume the following:

- GSD on the DHT overlay network never disappears
- No malicious user nodes join to the MOGs
- Network infrastructure provides short and stable network delay over the inter-domain networks.

3.2 Zone

First, we introduce the concept of a *zone*. A zone is a piece of the whole GSD partitioned by locality of interest. When dividing the whole GSD into zones by locality, the GSD on some zones is required by some of all the participating nodes. We define a zone as a judgment block and a data transfer block. According to this definition of zone, we define a server role as existing on each zone, and design a mechanism to manage the server role and the GSD on each zone.

As in the ZFM, the SimMud [9] partitions whole GSD by locality of interest. However, SimMud divides the game world into several regions only by locality of area in the game world. The ZFM partitions whole GSD into several zones not only by locality of area in the game world, but also by locality of group sharing specific GSD or the locality of personal information. The ZFM can produce a flexible data structure for any type of MOG.

Along with the concept of *zone*, we introduce node status. Basically, each user node has one of three statuses for each zone: *independent*, *zone member*, and *zone owner*.

The zone owner represents the server role on a zone, and the zone member represents the client role on a zone. When a user node wants to receive updates of the GSD of a particular zone, the node changes its status as a zone member. A zone member node can request the zone owner node to modify some GSD on the zone. If a user node wants to modify the GSD on the zone when there is no zone owner, then the node tries to change its role to zone owner. Although the zone owner node has a right to change all GSD on the zone, the zone owner node has responsibility for judging conflicts among requests from zone members, modifying GSD along with the requests, serializing GSD according to the game sequence, and announcing any updates of GSD to all zone members.

If a user node is not interested in the GSD of some zone, the node then has an independent status to the zone. Next, we add a new definition for "zone" When we describe "a node joining to a zone" or "the zone owner leaving from the zone", we use "zone" to represent the membership who manages the zone owner role and GSD.

Along with the basic three node statuses, we add one extra node status: *data holder*. In the ZFM, whole GSD is distributed into a peer-to-peer overlay network representing several zones. Futuremore, some nodes contribute their own local storage to part of the shared storage on the peer-to-peer overlay network. We call the node contributing its storage for storing GSD of some zone, a data holder. Therefore, a data holder node has all GSD of some zone. Data holder nodes are selected by the data sharing algorithm of the employed overlay technique.

3.3 Distributed Hash Table

In the ZFM, we employ the DHT as at tech-

nique to construct an peer-to-peer overlay network. The DHT is a distributed data placement and a data lookup algorithm for an overlay network. Basically, the DHT stores the mapping between a key and a value, and the DHT appears as an ordinary hash table for each user node. When a node tries to search some data, the node looks up its own DHT as a hash table. A key points to the identifier of some node, then, the node pointed to by the key contains true data on the hash table. Therefore, a user node refers the true data to some other node whose identifier is obtained through the DHT. Variations of DHT, e.g., Chord [11], CAN [12], Pastry [13], Tapestry [14], etc. exist.

Using the DHT, all nodes on the same DHT overlay network use one hash table, that is, all nodes share the same data. DHT ensures the pairing of a key and the corresponding data or the identifier of some data holder node; therefore, the mismatching of data among nodes never occurs.

However, DHT has several drawbacks. First, the DHT has accessibility to data, that is, every node can modify every piece of data on the DHT. Therefore, some nodes can modify some data when the data is not a match for the node. The DHT does not have a judge to avoid conflicts of changing data among user nodes. Also, the DHT ensures every node will share the same data; hence, the accessibility of the DHT can easily provide inconsistency of data on a time sequence.

Second, the DHT has a tradeoff between the size of the routing table and routing hops. If the order of the routing table size is $O(logN)$ at most, then the order of the routing hops becomes $O(logN)$ [11),13)]. Hence, read or write operations to the DHT require $O(logN)$ routing hops. Several Application Layer Multicast (ALM) methods using the DHT are proposed [15]; these ALMs require several routing hops on the application layer. Therefore, using the DHT as a data transfer method may provide long network latency when the overlay network topology is large. Such long network latency may be a critical overhead for some MOGs, racing games or first person shooter games.

To reduce the number of nodes access the GSD through the DHT, we use the DHT as a backup storage media, and as a rendezvous point of zones; that is, as the routing map to zones.

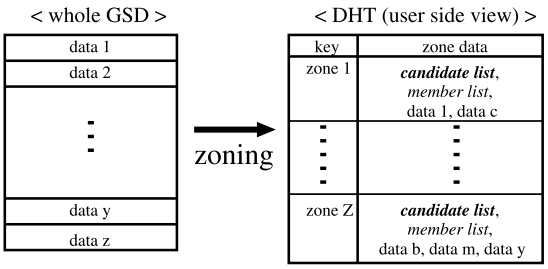

Fig. 2 Zoning.

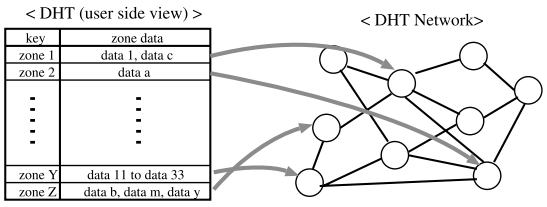

Fig. 3 Mapping.

3.4 Zoning and Mapping

We introduce zoning and mapping methods for using the DHT as the rendezvous point to zones. To reduce message forwarding through the DHT, an independent node should know the latest membership of each zone by using only a single access to the DHT. In other words, the ZFM requires a mechanism by which an independent node can learn which node is the zone owner and which nodes are zone members when the independent node accesses to the zone data on the DHT overlay network.

Zoning makes each zone a judgment and a data transfer block of GSD by adding the information about the membership on each zone. Each zone contains some piece of the whole GSD (see Section 3.2). We add *candidate list* and *members list* to each zone by zoning. The candidate list shows which node is the current zone owner, or tells which nodes are the candidates for a new zone owner. On the other hand, the members' list expresses which nodes have the zone member state for the zone. **Figure 2** shows how the information about each zone looks on the user side of the DHT.

Data of each zone and data holder nodes on the DHT overlay network correspond to mapping methods. A DHT algorithm selects data holder nodes randomly, and assigns the key of each zone to the identifier of a corresponding data holder node (**Fig. 3**). Because each key on the user side of the DHT points to the identifier or the address of the corresponding data holder node, the user node can use the DHT as

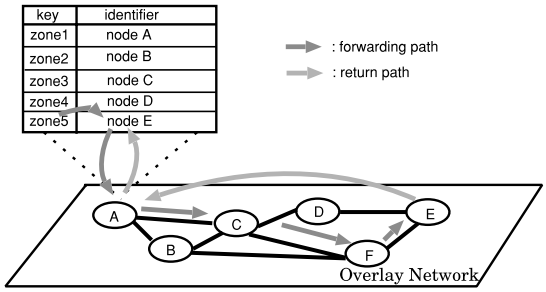

Fig. 4 Message forwarding on DHT.

the routing map to each zone.

Figure 4 shows message forwarding on the DHT. On the user node side, when a user node wants to read zone data, the user node gets the key generated by an employed hash function and a key word, and then user node can look up the identifier of the data holder node pointed out by the key. The user node then sends a message to the data holder node via several routing hops, and receives the zone data from the data holder node. If some user node wants to change some zone data, the user node sends a message to the data holder via several forwarding hops, and the data holder node changes the zone data according to the received message.

3.5 Zoned Federation

A zoned federation is a mechanism to manage the information of membership and consistency of the GSD on each zone. Zoning and mapping methods enable each participating node on the DHT overlay network to access each zone and to grasp the zone data of the accessed zone. For the nature of the DHT overlay network, the ZFM should let each participating user node join and leave the DHT overlay network easily. Therefore, the procedures to inherit the information of the zone data on the zone from the zone owner node must be tolerant of the frequent changes of the zone owner role node. The zoned federation provides a mechanism to inherit the current situation of each zone and to manage each zone autonomously.

3.5.1 Data Backup

To succeed in taking the current GSD from the old zone owner, the ZFM uses the DHT overlay network as a backup storage media. A zone owner is the centric game server on particular zone; therefore, the zone owner accepts zone members' requests, judges conflicts among requests, modifies GSD, serializes the changes of GSD, and announces any updates of GSD to all zone members. Also, the zone owner updates the current GSD on the corresponding data holder node whenever any updates about GSD occur.

By recording the current zone's GSD on the DHT overlay network, all participating user nodes can get the latest information about each zone from the DHT even when no zone owner exists in a specific zone. If a participating user node changes its status to the zone owner on some zone, the node can know all zone member nodes and the latest GSD from the zone data on the corresponding data holder node; therefore, the new zone owner node can accomplish all the tasks left by the old zone owner.

Requests about changes to GSD and announcements of the latest GSD are transmitted between the zone owner and zone members directly; in other words, no intermediate hop on the application layer level is employed in transmitting GSD-related messages in the ZFM. On the other hand, updating GSD on the data holder node is processed through the message forwarding mechanism on the DHT overlay network. Using the message forwarding of the DHT requires several routing hops; however, the updated zone data on the data holder node is processed in parallel by announcing the latest GSD to each zone member. Hence, backing up the zone data to The DHT overlay network doesn't influence the response latency of message exchanges between the zone owner and the zone members.

3.5.2 Zone Membership Management

In the ZFM, each user node accesses zones where the GSD required by the user node is stored. Also, the node playing the zone owner role is changed dynamically. The candidate list and the member list on each zone provides management mechanism of the zone owner role and grasping current zone members.

Each user node changes its status regarding each zone as illustrated in like **Fig. 5**. An user node becomes the zone owner from the independent state or the zone member state by the *step up* procedure. An independent state node can join the zone as a zone member by following the *join* procedure. The zone member state node leaves from the zone by the *leave* procedure, and the zone owner role node can resign its game server task by the *step down* procedure and change to the independent state. Basically, these status changes and procedures are announced to other nodes by the candidate list and the members' list of each zone.

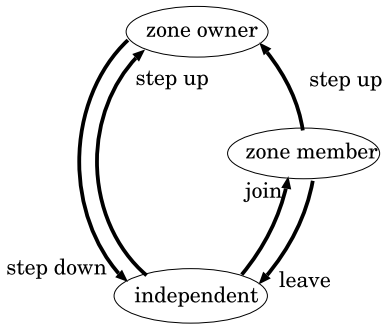

Fig. 5 Node status changes.

The candidate list shows which node is the current zone owner, or which node has the right to play the zone owner role. When there is no zone owner in a zone and a user node tries to become the new zone owner of the zone, the user node writes its identifier into the candidate list of the zone and reads the latest zone data through DHT message forwarding. If the identifier of the user node is listed on the top of the loaded candidate list, the user node can become a new zone owner, and then the user node can start working as the zone owner. When the zone owner wants to leave the zone, the zone owner has to remove its identifier from the candidate list on the data holder node. After removing its identifier from the candidate list, the zone owner can leave the zone.

On the other hand, when a user node wants to join a zone as a zone member, the user node writes its identifier into the members' list on the data holder node, reads the latest situation about the zone from the DHT overlay network, and sends a *join* message to the zone owner who is listed on the top of the candidate list. When the zone owner receives a join message from a new zone member, the zone owner adds the identifier of the new zone member into its own members' list. Next, the zone owner and the new zone member establish a connection and exchange messages directly. When a zone member node tries to leave the zone, the zone member node removes its identifier from the members' list on the DHT, that is, the members' list on the data holder node. The zone owner can realize the disappearance of the zone member when the connection between the zone owner and the zone member is closed.

The details of the procedures for managing the membership of zones and GSD on each zone are described in Section 4.3.

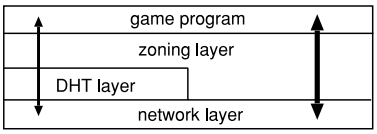

Fig. 6 Zoning layer.

Table 1 APIs of Pastry for ZFM.

function	definition
join()	join to DHT network
query(key)	get the current zone data pointed out from the hash key to DHT
set(key, data)	add new data to zone data on DHT
delete(key, data)	delete the specific data from zone data on DHT

4. Implementation

In this section, we describe our implementation of the Zoned Federation Model (ZFM). We have inserted a *zoning layer* as a middle layer between game programs and TCP/IP stacks, and between game programs and the DHT layer.

The zoning layer covers both the DHT layer and the TCP/IP stacks (**Fig. 6**). By using the zoning layer, game programs don't have to consider whether a message should be exchanged through the DHT or not.

Our ZFM optimizes the response latency on a MOG. In the implementation, we introduced several techniques to reduce or to optimize response latency.

We implemented this zoning layer as a C library; we call this zoning layer library a *libcookai*, and a C library of Pastry customized for the ZFM. We also implemented a sample MOG using libcookai.

4.1 Pastry for ZFM

Table 1 shows APIs of our pastry implementation in C language. These APIs are the interfaces of the Pastry DHT overlay network for the zoning layer (Table 1).

Each user node also participates in the Pastry DHT overlay network by the *join* function. By using the *query* API, the zoning layer reads current zone data from the data holder node on the Pastry DHT overlay network. If any node tries to change the same zone data asynchronously, some inconsistency of the zone data may occur. To avoid inconsistencies of data, we divide the 'write operation' of the zone data into *set* and *delete*. The zoning layer calls the *set* function

Table 2 Zoning layer API.

function	definition
initialize()	to connect to the game world
step_up(zone)	to *step up* to zone owner
join(zone)	to become zone member and listen to *update* messages
update(zone, data)	to *updated* modified GSD
commit(zone, data)	to send a commit message
release(zone)	to release direct connection to the zone owner
step_down(zone)	to *step down* from a zone owner and close all connections to zone members

to add a new data value of zone data on the DHT, and calls the *delete* function to remove the old value of the zone data.

4.2 Zoning Layer

The zoning layer controls network access. When the game program on a user node tries to send a message, the zoning layer chooses Pastry message forwarding or the end-to-end TCP connection according to the node status and the data type of the message. On the game program side, the zoning layer represents the interface necessary to control its node status for each zone, as well as the interface of the network layer including the Pastry overlay network. Game programs access the zoning layer by using the APIs listed in **Table 2**.

4.2.1 Data Structure of Zone

The data structure of the zone is described in **Fig. 7**. *DHT_DATA* is the basic data structure, and it constructs a one-way list. If the data type is *OWNER*, the DHT_DATA contains the identifier of the zone owner or that of a candidate for a new zone owner. The DHT_DATA is used as a part of the members' list when the data type is *MEMBER*. Each GSD is contained in the *DATA* type DHT_DATA. Example of data on zone data list is described in **Fig. 8**.

New DHT_DATA is added in the tail of the zone data list by the *set* function of Pastry for the ZFM. Focusing on *OWNER* type data, the zone data list is attached to the candidate list of the zone. The zone data structure is a one-way list; therefore, when a user node searches the zone owner on some zone, the zone owner is the node whose identifier is contained in the first *OWNER*-type data listed in the zone data.

4.2.2 Latency Optimizing Techniques

Most types of MOGs require a short response latency of less than 200 ms [3]. We designed the ZFM to optimize response latency. In addition, we add two latency optimizing techniques onto

```
enum{
   OWNER,
   MEMBER,
   DATA,
};
struct DHT_DATA{
   unsigned int type;
   unsinged int data_length;
   char data[];
   struct DHT_DATA *next;
};
```

Fig. 7 Data structure of zone.

```
zone data{
   OWNER{
     data_length
     "node1.example.com:8472"
   }
   MEMBER{
    data_length
    "node2.example.com:8472"
  }
  DATA{
    data_length
    binary_data
    }
  MEMBER{
    data_length
    "node3.example.com:8472"
  }

}
```

Fig. 8 Example of data on zone data list.

the implementation of the zoning layer; namely, data caching, and connection caching.

The zone owner has the permission to write to the master data of GSD on its governing zone. However, updating data on a data holder node through the DHT forwarding paths causes more latency than the modifying data on the local storage of a zone owner. Therefore, by using a local cache of the zone data list on a zone owner as master data of the GSD on its zone, the local cache is able to cut the response latency caused by searching the data holder node to modify the GSD through DHT message forwarding.

Along with data caching on each zone owner, we combine connection caching to reduce the response latency on the announcing updated GSD. Each zone member of a zone is listed on the zone data list of the zone; therefore, a new zone owner can understand all current zone members. When a user node becomes a new zone owner, the new zone owner establishes an end-to-end TCP connection to each zone member, and the zone owner keeps these TCP connections until the zone owner leaves the zone. We call each TCP connection between a zone owner and a zone member a *transfer path*. When a zone owner changes the GSD on its local cache of the zone data, the zone owner announces updated GSD for each zone member directly through transfer paths. In parallel

with updating zone members' GSD, the zone owner also updates GSD on the data holder node through DHT message forwarding to avoid loss of the current GSD of the zone. Hence, every node can understand new zone owner by following in sequence the latest GSD from DHT overlay network.

4.3 Procedures

Our zoning layer implementation provides several APIs for game programs (Table 2). Using these APIs, game programs should only be concerned about their own node status for each zone. On the assumption that game programs are written in an event driven model, we have designed and implemented APIs. **Figure 9** shows pseudo codes of a zoning layer API.

Initialize function is used when a user node joins to the DHT overlay network where a MOG runs. Initially, for the user node just joining the DHT network, the user's node state is an independent state for all zones. The node tries to participant in several zones which have the GSD that the user node must read or change according to the game sequence.

When a user node wants to change the GSD of a zone, the user node tries to become the new zone owner of the zone. Next, the user node takes a *step up* procedure. First, the user node adds its identifier to the zone data list on the data holder node through the DHT forwarding path, and reads the current zone data list from the data holder node. If the user node finds the *OWNER*-type data it wrote by itself first when the user node searched the current candidate nodes, then the user node changes its node status to zone owner, reads the current zone data list again, comprehends all current zone member nodes, and establishes a transfer path to each zone member. Establishing a transfer path tells each zone member about the arrival of a new zone owner. If the user node finds an other node's identifier first when searching the candidate nodes, the user node realizes that the other node is the zone owner, then the user node removes the *OWNER*-type data it committed by itself from the zone data list.

If the zone owner already exists when an independent node wants to modify the GSD, or when an independent state node wants to just read the current GSD or to receive the updated GSD from the zone owner, the independent node changes its status to zone member by a *join* function. *Joining* steps are as follows: an independent node changes its status to zone member, writes its identifier in the tail of the zone data list as *MEMBER*-type data, and reads the latest zone data list from the DHT overlay network. Next, the new zone member checks the zone owner identifier and tries to establish a transfer path by sending a *join message* to the zone owner. When the zone owner receives the join message, the zone owner adds the new zone member's identifier to the zone data list on its own local cache, and establishes the new transfer path.

When a zone member node wants to change some GSD value, the zone member sends a *commit* message with new GSD to the zone owner. When the zone owner receives a request for modification from some zone member or the zone owner wants to modify some GSD on its own zone, the zone owner calls an *update* function. After judging the conflict or the consis-

```
function initialize(){
  DHT_join();
}
function step_up(zone){
   DHT_append(zone, mydata.hostdata + type::OWNER);
   data_list = DHT_get(zone);
   foreach element (data_list){
     switch(element.type){
       case MEMBER:
         regist_member(zone, element.hostdata);
         break;
       case OWNER:
         if(element.hostdata == mydata.hostdata){
           return success;
         }
         if(alive_check(element.hostdata) == alive){
           DHT_delete(zone, mydata.hostdata + type::OWNER);
           return fail;
         }else{
           DHT_delete(zone, element.hostdata + type::OWNER);
         }
         break;
     }
   }
   return fail;
}
function join(zone){
   DHT_append(zone, mydata.hostdata + type::MEMBER);
   data_list = DHT_get(zone);
   foreach element (data_list){
     switch(element.type){
       case OWNER:
         regist(zone, element.hostdata);
         break;
       case DATA:
         mydata.zone.data = element.data;
         break;
     }
   }
}
function commit(new_data.zone.data){
   send(owner.hostdata, new_data.zone.data);
}
function update (mydata.zone.data){
   foreach member (zone.member_list){
   send(member, new_data.zone.data);
   }
   DHT_delete(zone, old_data.zone.data);
   DHT_write(zone, new_data.zone.data);
} function step_down(zone){
   DHT_delete(zone, mydata.hostdata + type::OWNER);
   foreach member (zone.member_list) {
   disconnect(member);
   }
}
function release(zone){
   DHT_delete(zone, mydata.hostdata + type::MEMBER);
   disconnect(zone::owner);
}
```

Fig. 9 Zoning API.

tency of the GSD, the zone owner modifies the GSD on its local cache, announces new GSD to each zone member through transfer paths, and sends the new GSD to the data holder node via the DHT message forwarding for the purpose of backup.

If a zone member *leave* the zone, the zone member shuts down the transfer path and removes the *MEMBER*-type data which contains its own identifier from the data holder node through DHT forwarding path. When the transfer path has been closed by some zone member, the zone owner realizes that the zone member left from the zone, and the zone owner removes the zone member's identifier from its local zone data list.

A zone owner *steps down* to the independent state when the zone owner wants to leave from the managing zone or just wants to resign from the zone owner role. In this case, the zone owner removes its identifier from the *OWNER*-data from the zone data list on its local cache and on the data holder node respectively. Next, the zone owner closes all transfer paths, and changes its status to an independent state. All zone members realize the zone owner has just left the zone by the closed transfer path. If some zone member simply tries to *commit* after the zone owner leaves, the zone member tries to become the zone owner by calling the *step up* function.

4.3.1 Recovery from Failure

In this section we describe the error processing procedure that occurs when a node is suddenly isolated from the peer-to-peer network because of a network failure.

When a zone member is isolated from the network, this event of isolation looks for the "leave" action called by the zone member. The zone owner is the only node which knows that the zone member has left, so the zone owner removes the zone member's entry from the zone data list on its own local cache. If the isolated zone member comes back to the DHT overlay network, then the zone member realizes that the transfer path has been closed. In this case, the zone member must send a join message to the zone owner again.

When a zone owner has been disconnected from the network, its zone members notice that the zone owner has been isolated from the network due to an absence of heartbeat messages from the zone owner. Next, each zone member sends an owner-lost message to the game program by itself. After sending an owner-lost message, one of these zone members deletes the zone owner entry from the DHT. If the game program on some zone demands a change of the GSD, the zone member tries to "step up".

If a zone owner disappears from the network by a network accident when no zone member is listed, the entry of the zone owner remains on the DHT. In this case, a node becomes a zone member because of the remnant of the zone owner entry on the DHT, but the new zone member cannot establish the transfer path to the registered zone owner, so the zone member notices that the zone owner doesn't exist. Then, the zone member deletes the old zone owner entry from the DHT and tries to step up.

4.4 Sample MOG Program

To evaluate the zoning layer, we have implemented a MOG program.

Our sample MOG is "get the game flag" similar to "rally-x", but extended to a multi-player game. Each player drives a small car and players struggle to get flags distributed on a two dimensional world map. Each player can disturb other players by a smokescreen. If a player's car hits a rock or another player's car, the player has to restart.

To divide the whole world into several zones on this MOG, we used maps and positions as features of the zones. A zone of a map image contains a map image which is 128×128 square when one square is defined as the size of a car image. On the other hand, each zone of positions contains the positions of cars, flags, and smokescreens on a specific map image are defined by a zone of the map image. These zones are distributed onto a DHT overlay network.

5. Evaluation

In this section, we evaluate our zoning layer implementation of the zoning layer described in Section 4. For evaluation, we employed a sample MOG program and examined it by focusing on latency overhead as a performance metric.

5.1 Order

We clarify the tradeoff between the order of message forwarding hops on an application layer level and the order of messages to be sent by a node in several models. We describe the ZFM in **Table 3**, in the client-server model in **Table 4**, and in Scribe [15], which is an Application Layer Multicast (ALM) based on DHT message forwarding in **Table 5**, respec-

Table 3 Communication overhead of ZFM.

	num of messages	hop counts
step up	O(N)	O(logN)
join	$O(1)$	O(logN)
step down	$O(1)$	O(logN)
leave	$O(1)$	O(logN)
commit	$O(1)$	$O(1)$
update	O(N)	O(logN)

Table 4 Communication overhead of client server model.

	num of messages	hop counts
recovery	O(N)	$O(1)$
join	$O(1)$	$O(1)$
leave	$O(1)$	$O(1)$
request	$O(1)$	$O(1)$
update	O(N)	$O(1)$

Table 5 Communication overhead of Scribe.

	num of messages	hop counts
create	$O(1)$	O(logN)
subscribe	$O(1)$	O(logN)
unsubscribe	$O(1)$	O(logN)
publish	O(N)	O(logN)

tively. Scribe is employed by SimMud [9] which is another DHT overlay infrastructure model for MOGs.

Because the DHT overlay network can be accessed on all actions but *commit*, ZFM appears to be a hybrid model of a client-server model and scribe. Therefore, the ZFM appears to provide at least the same response latency as Scribe although the recovery steps of the zone owner (*step up*) may become a bottleneck point for the ZFM.

Comparing the recovery steps of the game server, the ZFM, and the client-server model requires $O(N)$ when sending messages to all clients. The ZFM also needs $O(logN)$ hop counts to inherit the latest zone data list from the DHT overlay network.

When joining or leaving from a zone, the ZFM requires $O(logN)$ hop counts to add or to remove its identifier from the zone. In addition, the Scribe needs $Olog(N)$ hop counts to add or to remove its entry from a multicast group.

Updating the GSD, Scribe and the ZFM requires $O(N)$ message forwarding and $O(logN)$ hop counts. However, the Scribe needs $O(logN)$ hop counts because the Scribe employs the DHT as the message forwarding path for all messages. On the other hand, the ZFM requires $O(logN)$ hop counts to back up the current GSD on the DHT overlay network. If only some nodes play the zone owner role of a zone, backing up the current GSD into the DHT overlay network may not be necessary for the zone owner node. Also, updating each zone members' GSD through the direct TCP connections is processed in parallel by backing up the GSD on the DHT overlay network. Therefore, a zone owner works with $O(N)$ message forwarding and $O(1)$ hop counts from the standpoint of zone members (game clients).

5.2 Response Latency Overhead

We have evaluated response latency overhead caused by our zoning layer implementation. MOGs need low latency overhead on message exchanges and on updating the latest GSD in order to provide stress-free interactions among game players. On the evaluation of response latency overhead, we examined two kinds of response latency used by the zoning layer: one on a *step up* action, and the other on *update* action. In Section 5.1, we described how the step up action may be a bottle-neck point because the step up action requires $O(N)$ message sending and $O(logN)$ DHT forwarding hops to inherit and to recover zone owner tasks. Also, we mentioned how update action may achieve a short response latency to zone members without concerns about back up on a DHT overlay network.

Response latency on step up action is influenced by two actions: a node becomes a zone owner and establishes direct connections to all zone members. To become a zone owner, a node has to search a data-holder node twice. The first search is needed to write its entry as an owner on the zone-data list, and, after becoming the zone owner, second search is required to fetch all zone data from the data holder-node to use as master GSD, and to comprehend all zone members. Therefore, response latency on step up is affected by the time needed to search a data holder through the DHT forwarding path. Also, response latency on step up is influenced by the number of zone members because of establishing TCP connections between a zone owner and each zone member. In Section 5.2.1, we describe the relation between the number of zone members and the response latency of step up action.

On the other hand, response latency on *update* action is affected by updating a GSD on all zone members and on the data holder. We measured response latency on update action as update time, which is influenced by the number of zone members. Through experiments described

in Section 5.2.2, we try to clarify the relation between update time and the number of zone members on a single zone.

For each evaluation of response latency, we employ a test code which is constructed by libcookai. On the test code, a zone-owner node sends a packet with a 1,024 bytes payload to each zone member node over each TCP session. Each zone member node simply receives the test packet.

The threshold of response latency which users can accept without stress is different among MOG types [3)~6)]. Our sample game program needs the same response latency accepted by tye First Person Shooter (FPS) game. For the evaluation, we set the threshold of the response latency to 200 ms which is acceptable for users on FPS [5)].

5.2.1 Response Latency on Step Up

First, we evaluated the overhead of response latency on the *step up* procedure. In this evaluation, we used an experimental environment consisting of 7 FreeBSD PCs, 3 with 500 MHz processors and the other 4 with 850 MHz, interconnected by a 100base-TX switch. All PCs have 256 M bytes memory. To increase the number of zone members, we simulated multiple zone-member nodes by running zone-member processes on PCs.

We estimated the response latency by dividing several time ranges, for example, DHT Looking-up Time (DLT), Establishing Connections Time (ECT), and Total Stepping-up Time (TST). The relationship among these time ranges is as follows:

- DLT
  The time spent for fetching a zone-data list from the DHT overlay network.
- ECT
  The time spent for establishing each connection between a zone owner and a zone member.
- TST
  The total time spent for stepping up to a zone owner.

In the test-bed environment described above, the zone owner was placed in only one PC. The scenario of this experiment was as follows: First, only zone members run; next, a new independent node steps up to the zone owner. We evaluated these response latencies while increasing the number of zone members gradually. The result of this experiment is shown in **Fig. 10**.

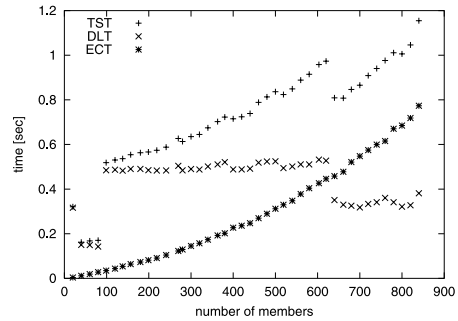

Fig. 10 Response latency on step up.

Obviously, the topology of the Pastry forwarding path is affected on by the TST. The ECT draws the liner curve of $O(N)$, but the DLT draws an incontinuous line. TST When the number of zone members was 700, the TST was less than that on 600 zone members. This was because that the difference of the ECT on 600 zone members versus on 700 members was shorter than the difference of the DLT. Figure 10 shows that a longer Pastry forwarding path is the bottleneck point of response latency on the step up action. The maximum number of zone members which satisfied the threshold was 100 zone members; therefore, a single zone owner can deal with 100 zone members by keeping the TST at less than a 200 ms time threshold.

5.2.2 Response Latency on Updating GSD

Next, we evaluated *update time*. To evaluate the response latency overhead of the update time, we used the experiment on *Starbed* which is a large scale network emulation test-bed environment constructed in Hokuriku IT Open laboratory [16)]. On Starbed, 512 PCs are divided into five partitions and inter-connected through several switches. Each PC has Intel Pentium III 1 GHz, 512 MB main memory, two 100 Base-TX network interfaces, one of which is connected to the control network and the other to the experimental network. We ran FreeBSD 4.7 for the operating system on each PC.

For the evaluation of update time, we used 296 PCs on Starbed with a simple network topology such that each PC connected to the same layer 2 network. In this experimental environment, we did two experiments about update time. We estimated update time on each experiment. The start of the update time was defined as the time when a zone owner sends a test packet, and the end of the update time was

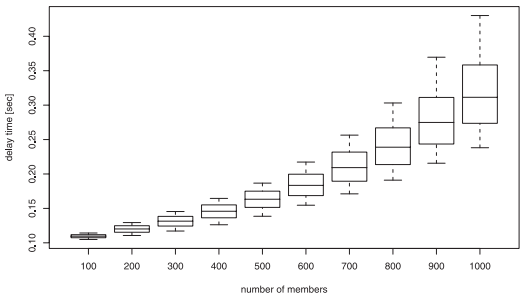

Fig. 11 Update time.

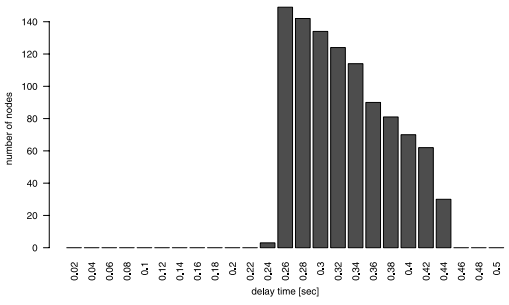

Fig. 12 Histogram of update time.

when a zone member received the test packet. In the experiments, we measured the update time on each zone member node, and drew the results using a box-whisker plot (Figs. 11, 13).

First, we evaluated the effect of the number of zone members on a single zone. In this experiment, we used a PC to run only one zone owner process with a 100 ms delay caused by dummynet, Zone member processes ran on other 295 PCs uniformly.

Figure 11 shows the trend of distribution for the update time, and **Fig. 12** represents the distribution of the update time when the number of zone members was 1,000. According to these figures, although dummynet caused a 100 ms delay, the minimum update time was 240 ms and all update times on each zone member were less than 440 ms, even when the number of zone members was 1,000. The maximum number of zone members which a single zone owner can treat by satisfying the 200 ms threshold was 500 members.

In the second experiment, we measured the effect of the number of zones needed to update time when the total number of zone members was fixed on 297 nodes. We changed the number of zones from 1 to 8, and we distributed zone members to each zone equally. **Figure 13** shows the result of this experiment. In Fig. 13, we divided a zone which has many members

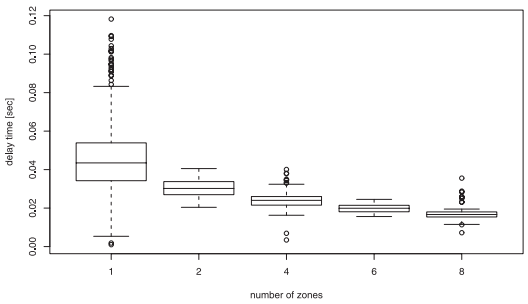

Fig. 13 Effect of the number of zones to the update time.

into several small size zones to enables the reduction of the response delay and to stabilize the distribution of the update time.

From the results of these experiments, we can say that a single zone owner can deal with user nodes as well as a single, not-clustered centric MOG server, and zoning can reduce the overhead on a zone owner and provide MOG the scalability necessary for at number of users of each zone.

5.3 Bandwidth Requirements

A zone owner has to update all zone members' GSD through unicast; therefore, our zoning layer implementation consumes bandwidth when a zone owner updates the GSD. Also, a single user node can become the zone owner on several zones, and if a single user node becomes the zone owner of all zones, the single node requires the same upstream bandwidth as the downstream bandwidth required for a single centric MOG server. Required bandwidth of a user node is described as follows:

about some node j $(j = 1, 2, ..., m)$
in zone i $(i = 1, 2, ...n)$, node j updates GSD as the zone owner

- a_{ij}: whether or not node j is an owner on zone i, that is, a_{ij} is 0 or 1
- N_i: the number of members on the zone i
- F_i: frequency of updating GSD on zone i
- G_i: average size of updating GSD per node on zone i
- M_i: required number of messages sent by the zone owner on zone i
- B_i: required upstream bandwidth consumption on the zone owner on zone i

The total upstream bandwidth consumption on node j (B_{T_j}) is:

$$B_{T_j} = \sum_i a_{ij} B_i$$
$$= \sum_i a_{ij} N_i G_i F_i$$

6. Other MOG Models Based on P2P Overlay Network

SimMud [9] and the PP-CA model [17] are other approaches to MOG infrastructure based on to peer-to-peer overlay network. These approaches provide audit mechanisms for GSD consistency by a third person or by an authority.

SimMud employs Pastry [13] and Scribe [15] as base components of its architecture. In the SimMud approach, the authoritative role is given to a data holder node, which is called "coordinator". By randomly mapping data holder nodes on the DHT, SimMud prevents game players from cheating global states because the coordinator is rarely interested in the GSD stored in its local storage. Also, by preparing several replicas of a coordinator, SimMud provides fault tolerance.

Pellegrino, et al. has proposed the PP-CA model, which is a peer-to-peer overlay MOG infrastructure with a central arbiter server [17]. In the PP-CA model, a central arbiter server only audits inconsistencies of the GSD and resolves the inconsistencies. Other messagings such as updating the GSD are processed by user nodes through a peer-to-peer overlay network.

Pellegrino, et al. analyzed three different models: the client-server model, the peer-to-peer model, and the PP-CA model by using an open source MOG program, BZFlag [18]. The analysis shows that the PP-CA model can reduce the bandwidth requirement of the central arbiter and resolve inconsistencies of the GSD without a complex distributed agreement protocol.

Next, we try to compare ZFM, SimMud, and the PP-CA model (**Table 6**). Durability of GSD on each model is affected by the employed peer-to-peer overlay network.

MOG, ZFM, and PP-CA models are superior to SimMud, because SimMud is a customized model only for a Massive online RPG and MOG, ZFM, and PP-CA models are more adaptable.

SimMud has scalability ensured by a simulation. ZFM also has scalability; however, we have not evaluated ZFM in a simulation with a size as large as that of SimMud. PP-CA has been evaluated only in a small LAN environment; therefore, we cannot discuss the scalability of the PP-CA model.

Table 6 Comparing three models.

	ZFM	SimMud	PP-CA
durability	o	o	o
adaptability	o	x	o
scalability	o	o	-
response latency	o	x	Δ
bandwidth	x	o	Δ
cheat proofing	x	Δ	o
incentive to serve	o	x	o

ZFM is a more latency-optimized approach than SimMud because SimMud uses Pastry [13] as the DHT, and Scribe [15] as the message exchange method based on an application-layer multicast (ALM). While ALM reduces the bandwidth consumption of the coordinator, it incurs network delay by crossing several hops on both the DHT and the ALM. In our zoning layer approach, each node exchanges messages directly; therefore, ZFM can achieve a shorter response latency than the ALM, except for the initial rendezvous by the DHT. But ZFM consumes more bandwidth than Scribe because ZFM updates the GSD through unicast connections.

The PP-CA is a hybrid model of the peer-to-peer and client-server. Of course, the bandwidth consumption of the central arbiter server is low. However, in user nodes, the response latency and bandwidth consumption are affected by the data transfer protocol among user nodes.

The ZFM distributes arbiter servers if every zone owner works with fair play. If we assume that malicious users join in the ZFM as malicious zone owners, then the ZFM has the drawback of cheating or unfairness. Although SimMud equips cheat proofing through a third person, SimMud's cheat proofing can be overwhelmed by overriding numerous malicious nodes. The central arbiter server on the PP-CA is the authority or certificate server of the game; therefore, the PP-CA is tolerant to cheating or unfairness.

The SimMud third person check employs a coordinator who is not interested in the GSD of the managing zone. The game server tasks highly consume the resources of a user node; for this reason, an incentive is needed for the user nodes to process the game server tasks. However, a third person check of SimMud employs a coordinator who is randomly selected and may

be not be interested in the GSD of the managing zone; hence, no incentive or interest for the randomly selected third person exists. If a user is not interested in a particular zone, the user cannot grasp what is needed by most other users on the zone correctly. If several MOGs run on the same DHT overlay network of the SimMud, this forces user nodes to act as the game server for several MOGs. In such a situation, no merit exists on the user's machine. Dealing with several MOGs on the same DHT overlay network is difficult for SimMud.

On the other hand, in both the ZFM and PP-CA, the data transfer and judgment are processed by nodes interested in the same game world. In other words, an incentive to serve GSD in both models exists.

7. Related Work

The scalable data dissemination problem has been addressed in the application-level multicast literatures [15),19)], where large receiver groups are of particular concern. In contrast, our work focuses on zone-local data dissemination with low latency.

A large number of small groups can be supported in small-group multicast protocols [20)]; our work can exploit such infrastructure support, for the purpose of efficient data dissemination from the zone owner.

While we have looked only at the application-layer a topology in this paper, topology-aware overlay [21)] will further reduce the latency of intra-zone communication by optimizing the network-layer topology.

The API described in this paper resembles the CAST interface which is part of the common API effort [22)]. However, the underlying semantics have notable differences: zone-local serializability, and the presence of multiple roles. Typical any-source multicast protocols are not serializable, in the sense that one particular receiver cannot ensure the same order of packet arrival as other receivers. In the MOG context, we believe that the serializability is of particular importance.

8. Future Work

We evaluated the ZFM by focusing on response latency, and showed the scalability of the ZFM. However, our experimental environment was conveyed on a local subnet; therefore, we have to measure the scalability of the ZFM on Internet size topology. Also, we have not had subjective evaluations, so we plan to have several subjective evaluations using our sample game program shown in Section 4.4.

We have constructed the ZFM on the DHT in order to achieve data consistency on the assumption that the employed DHT has strong durability. However, our implementation of Pastry is not durable. When the data holder node leaves the DHT overlay network, no other node can refer to the GSD backed up on the disappeared data holder node until the data holder node returns again to the DHT network. In future work, we need consider the durability of GSD; a method such as OceanStore [23)], which uses the DHT overlay network as its largest storage, would be useful.

In our model, increasing the number of zone members increases the CPU and bandwidth overhead on the zone owner. To solve this problem, increasing the number of zone owners in a zone actively is necessary, in order to distribute a zone owner's tasks. In this case, achieving consistency of the global state is difficult; we should consider a the synchronization of data between multiple zone owners in a specific zone. We may have to consider workaround to reduce the overhead of a zone owner with numerous zone members.

Basically, for our design policy of the ZFM, we assume that the protection for cheating or unfairness on the ZFM is managed by participant users. In Section 6, we discussed how the third-person check employed by SimMud [9)] is not suited for the ZFM. Reputation techniques on a peer-to-peer overlay network [24)] meets the ZFM requirements, because such a reputation system is based on interests.

However, most peer-to-peer overlay network have congenial defects that apply to malicious user nodes, such as catastrophe by a betrayer, hijacking by numerous malicious nodes, or undermining a chain of vouchers from forged multiple identities [24),25)]. A central arbiter server may resolve these threats by providing a consistency check and certificates of the users. Therefore, the hybrid model of the ZFM and PP-CA can be constructed. Such a hybrid model can present short response latency and scalability with consistency and authentication.

9. Conclusion

In this paper, we have proposed the Zoned Federation Model, which adapts MOGs to peer-to-peer overlay networks. In this model, the

whole game world is divided into several zones; each zone is maintained by a federation of nodes: an owner and one or more members. The zone owner plays two critical roles. First, it provides zone-local serializability of state changes by aggregating modifications from all members, and by sending state-change notifications to all members. Second, the ZFM ensures the consistency of changes committed by other member nodes. The DHT harnesses this zoning layer by providing rendezvous capability and by working as a backup storage medium for zone data.

We have applied this model to our prototypical MOG implementation, with which we have evaluated latency and scalability. Our experimental results show the relation between latency of update time and the number of zone members on a single zone, and represents the effectiveness of distributing the functions of a centric authoritative node to several zone owners. Moreover, we have compared other models with ours according to the number of messages and the order of hop counts, and we have described the upstream bandwidth consumption of a zone owner node.

On our zoning layer implementation, the whole game world can be divided into several zones with no restrictions. Therefore, by considering the appropriate number of zones and the permissible number of zone members on a single zone according to our experimental results, we showed how game creators can design scalable MOGs on peer-to-peer environment with the short response latency required by each type of MOG.

References

1) Armitage, G.: Sensitivity of Quake3 Players to Network Latency, *ACM SIGCOMM Internet Measurement Workshop 2001 Work-in-progress Posters Session* (2001).
2) Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S. and Stoica, I.: The Impact of DHT Routing Geometry on Resilience and Proximity, *Proc. ACM SIGCOMM Conference*, pp.381–394, ACM Press (2003).
3) Pantel, L. and Wolf, L.C.: On the impact of delay on real-time multiplayer games, *Proc. 12th International Workshop on Network and operating systems support for digital audio and video*, pp.23–29, ACM Press (2002).
4) Armitage, G.J.: An Experimental Estimation of Latency Sensitivity In Multiplayer Quake 3, *Proc. 11th IEEE International Conference on Networks ICON 2003* (2003).
5) Beigbeder, T., et al.: The Effects of Loss and Latency on User Performance in Unreal Tournament 2003, *Proc. ACM SIGCOMM Workshop Network and System Support for Games NetGames-4* (2004).
6) Quax, P., et al.: Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game, *Proc. ACM SIGCOMM Workshop Network and System Support for Games NetGames-4* (2004).
7) Smed, J., Kaukoranta, T. and Hakonen, H.: Aspects of Networking in Multiplayer Computer Games, *Proc. International Conference on Applications and Development of Computer Games in the 21st Century* (2001).
8) Morse, K.L.: Interest Management in Large-Scale Distributed Simulations, Technical Report ICS-TR-96-27 (1996).
9) Knutsson, B., Lu, H., Xu, W. and Hopkins, B.: Peer-to-Peer Support for Massively Multiplayer Games, *Proc. 23rd Conference of the IEEE Communications Society* (*Infocom 2004*) (2004).
10) Coulouris, G., Dollimore, J. and Kindberg, T.: *Distributed Systems Concepts and Design*, Addison-Wesley (2001).
11) Stoica, I., Morris, R., Karger, D., Kaashoek, F. and Balakrishnan, H.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications, *Proc. 2001 ACM SIGCOMM Conference*, pp.149–160 (2001).
12) Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Schenker, S.: A scalable content-addressable network, *Proc. 2001 Conference on applications, technologies, architectures, and protocols for computer communications*, pp.161–172, ACM Press (2001).
13) Rowstron, A. and Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, *Lecture Notes in Computer Science*, Vol.2218, pp.329–350 (2001).
14) Zhao, B.Y., Kubiatowicz, J.D. and Joseph, A.D.: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing, Technical Report UCB/CSD-01-1141, UC Berkeley (2001).
15) Castro, M., Druschel, P., Kermarrec, A. and Rowstron, A.: Scribe: A large-scale and decentralized application-level multicast infrastructure, *IEEE JSAC*, Vol.20, No.8, p.11 (2002).
16) NiCT: Hokuriku IT Open Laboratory.
17) Pellegrino, J.D., et al.: Bandwidth requirement and state consistency in three mutliplayer game architecture, *Proc. 2nd Work-*

shop on Network and system support for games (*NETGAMES*), pp.52–59, ACM Press (2003).
18) Riker, T.: BZFlag (2002).
19) Zhuang, S., Zhao, B., Joseph, A., Katz, R. and Kubiatowicz, J.: Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination, *Proc. NOSSDAV* (2001).
20) Visoottiviseth, V., Takahashi, Y., Kadobayashi, Y. and Yamaguchi, S.: SIM: Sender Initiated Multicast for small group communications, *INET'2001* (2001).
21) Ratnasamy, S., Handley, M., Karp, R. and Shenker, S.: Topologically-Aware Overlay Construction and Server Selection, *Proc. INFOCOM* (2002).
22) Dabek, F., Zhao, B., Druschel, P. and Stoica, I.: Towards a common API for structured peer-to-peer overlays, *IPTPS '03* (2003).
23) Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C. and Zhao, B.: OceanStore: An Architecture for Global-scale Persistent Storage, *Proc. ACM ASPLOS*, ACM (2000).
24) Kamvar, S.D., Schlosser, M.T. and Garcia-Molina, H.: The EigenTrust Algorithm for Reputation Management in P2P Networks (2003).
25) Doucer, J.R.: The Sybil Attack, *Proceedings of IPTPS '02* (2002).

(Received May 24, 2004)
(Accepted November 1, 2004)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.1, pp.75–90.)

Takuji Iimura received his B.A. degree in Electronic Engineering from The University of Electro-Communications (UEC), Japan, in 2002. He is currently a M.E. course student in Nara Institute of Science and Technology (NAIST). His current interests include overlay network techniques.

Hiroaki Hazeyama received his M.E. degree in Information Science from Nara Institute of Science and Technology (NAIST), Japan, in 2003. He is currently a Ph.D. course student in NAIST. His current interests include overlay network techniques, and network security.

Youki Kadobayashi received his Ph.D. degree in Computer Science from Osaka University in 1997. He is currently an Associate Professor in the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His research interests include content internetworking, overlay networks, quality of services in the application-layer, middleware security, and secure operating systems.