Zoned Federation of Game Servers: a Peer-to-peer Approach to Scalable Multi-player Online Games

Takuji Iimura
Nara Institute of Science and Technology
Nara, 630-0192, Japan
takuji-i@is.naist.jp

Hiroaki Hazeyama
Nara Institute of Science and Technology
Nara, 630-0192, Japan
hiroa-ha@is.naist.jp

Youki Kadobayashi
Nara Institute of Science and Technology
Nara, 630-0192, Japan
youki-k@is.naist.jp

ABSTRACT

Today's Multi-player Online Games (MOGs) are challenged by infrastructure requirements, because of their server-centric nature. Peer-to-peer networks are an interesting alternative, if they can implement the set of functions that are traditionally performed by centralized authoritative servers. In this paper, we propose a *zoned federation model* to adapt MOG to peer-to-peer networks. In this model, *zoning layer* is inserted between the game program and peer-to-peer networks. We introduce the concept of *zone* and *zone owner* to MOG. Zone is some part of the whole game world, and zone owner is an authoritative server of a specific zone. According to the demands of the game program, each node actively changes its role to zone owner and works in the same way as a centralized authoritative server. By dividing the whole game world into several zones, workloads of the centralized authoritative game server can be distributed to a federation of nodes. We have implemented the zoned federation model, and evaluate it with a prototypical multi-player game. Evaluation results indicate that our proposed approach is applicable to small and medium-sized MOGs, where the number of nodes is less than 500.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols—*Applications*

General Terms

Design, Performance

1. INTRODUCTION

Today's Multi-player Online Games (MOGs) are usually constructed using the client-server model. In the client-server model, a centralized authoritative server is required as storage for the global state, as a judge to avoid conflicts among clients, and as a game master to update global states and to announce updated data according to game program sequences or clients demands. In order to handle frequent requests from multiple players without interrupting game sequences, the authoritative server is usually constructed as a cluster, in which each cluster node is provisioned with high performance hardware with large storage and wide bandwidth. Therefore, running and managing an MOG requires initial investments and continuous maintenance fees. This prevents small or medium enterprises from starting up a new MOG.

In this paper, we attempt to adapt MOG to peer-to-peer networks. If peer-to-peer network nodes can collaboratively form a game server cluster, game creators may be able to run MOG programs without the co-location fees.

We use Distributed Hash Table (DHT) to construct the peer-to-peer network. DHT is a distributed data placement and lookup algorithm for peer-to-peer networks. There are some variations of DHT, e.g., Chord [1], CAN [2], Pastry [3], and Tapestry [4].

To construct game server clusters on DHT, we propose the *zoned federation model*. The zoned federation model is a model of MOG based on DHT and *zoning layer*. In the zoned federation model, all global states of an MOG are separated into several *zones* and distributed onto the DHT. A node in the zoned federation model changes its role according to demands of the game program. A node may work as the game server on some zone, or may become a game client node.

To achieve the correct game sequence and low latency on data updates, we inserted zoning layers between a game program and the DHT or the network layer (Fig.1). We limited the use of DHT to the backup data storage and to the initial rendezvous point to an authoritative node. That is, once the authoritative node for a particular global state is found by DHT, a client node keeps the connection to the authoritative node in order to receive latest updates directly and to request modification of the data. In other words, zoning layer confines the update request and notification within single zone, and zoning layer depends on DHT layer for data storage and rendezvous. By doing so, zoned federation model can provide latency comparable to client-server model MOG in spite of the initial rendezvous overhead.

We have implemented this zoned federation model; zoning layer has been implemented as a middle layer between MOG layer and DHT layer. Its performance is evaluated with a prototypical game.

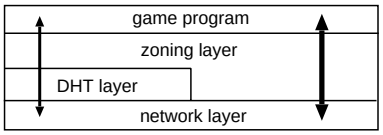

Figure 1: Zoning Layer

SIGCOMM'04 Workshops, Aug. 30+Sept. 3, 2004, Portland, Oregon, USA.

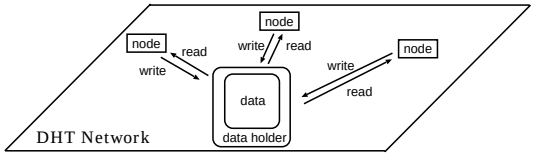

Figure 2: Flat model

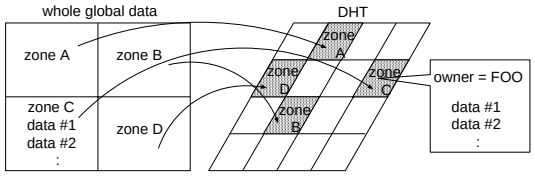

Figure 3: Mapping Zones on DHT

In Section 2, we model MOG on a peer-to-peer network, and describe the details of node behaviors and procedures on our proposed model. We mention an implementation of our proposed model based on Pastry[3] in Section 3 and evaluate it in Section 4. In Section 5, we refer to related work, and finally, we conclude this paper in Section 6 and discuss future work in Section 7.

2. MODELING MOG ON PEER-TO-PEER NETWORKS

In order to adapt MOG to peer-to-peer networks, we have to consider how to distribute the tasks of the authoritative server. The tasks of the authoritative server are storing global states, modification of global states, and broadcasting state changes to clients to keep synchronization among each client.

2.1 Flat model

First of all, we consider a simple model which distributes the storage of all global states. We designed a flat model based on DHT (Fig.2). Although each node can read and write global states, a specific global state is always stored uniquely on the peer-to-peer network because of the inherent consistency of data on DHT.

However, incorrect game sequence easily occurs due to the lack of serializability. As an example of incorrect game sequence, a player can be suddenly dead by very small damage after full recovery from near death. Furthermore, there is no mechanism on DHT to announce changes of global states to nodes. Each node must read global states frequently, in order to keep synchronization among each node's game progress.

2.2 Zone model

In order to establish serializability on game, we consider a second model, named *zone model* (Fig.4).

On MOG, most nodes exhibit locality; they only access fraction of information in the whole game world. Therefore, we divide the whole global state into several *zones*. A zone is a set of several global states gathered by a set of keys or some specific feature (Fig.3). In other words, zone is the minimum unit on distributing global states into DHT. For example, if the whole map data is partitioned into some areas with a set of keys, a zone data contains all global states of a specific area in the whole game world. Also, it is possible to bind all status data of a single player to a zone. There is no restriction on partitioning all global states into zones; it is possible to assign a zone to one global state. It is up to each MOG to determine how to decompose global states into zones. Each zone is associated with a participant node according to DHT's data distributing algorithm. In this paper, we call a node, which stores some zone data in its own storage, as a *data holder* node.

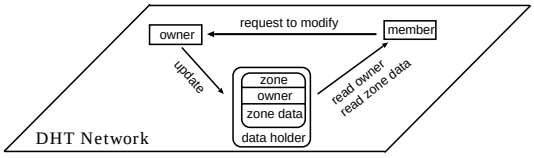

Figure 4: Zone model

In order to serialize global state changes within each zone, some node changes its role to the authoritative node, called *zone owner*, according to situations. When a node works as the zone owner on some specific zone, the node can modify global states on the zone. Meanwhile, the zone owner is responsible to accept other nodes' requests, to judge conflicts, and to serialize all changes of each global state on its governing zone.

Initially, each node starts at the *independent* node state. When the game program on some node tries to modify some global state, the node checks which node is working as the zone owner. Along with global states, the address of the zone owner is stored on a zone. Whenever the address of the zone owner is not stored, the detecting node writes its own address in the zone and becomes new zone owner. If the game program on a zone owner no longer requires the global states of its zone any more, the zone owner can resign its authoritative role.

An independent node changes its status to *zone member* if the node wants to change some global states whose zone owner already exists. A zone member node looks up the address of the zone owner, and asks the zone owner to modify some global state on the zone. Each zone member reads the latest update from the data holder node through DHT.

A node may work as a zone owner on some zone, while at the time joining another zone as a zone member.

Introducing zone owner as an authoritative node, zone model achieves correct game sequence and global state consistency. However, this model has limited applicability since many MOGs require low latency[5]. In this model, all messages and modifications have to go through DHT for the purpose of keeping data consistency. DHT requires several routing hops to reach a data holder node because of DHT lookup algorithm[1, 2, 3, 4]. Searching DHT, whenever the game program wishes to modify global states, causes high latency which makes game progress intolerably slow.

2.3 Zoned federation model

In order to amend zone model for the latency requirement, we came up with the third model, where we limit the use of DHT to the backup storage of global states and to the initial rendezvous point. We also employ caching techniques to reduce latency. The third model is named zoned federation model (Fig.5).

2.3.1 Data caching

Zone owner has the write permission to the master data of global states on its governing zone. However, updating master data through DHT causes more latency than modifying data on local storage of zone owner. So, using local cache on a zone owner as master data of global states on its zone, it is able to cut the latency caused by searching the data holder to modify global states through DHT.

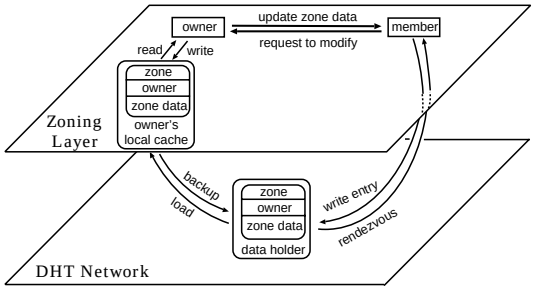

Figure 5: Zoned federation model

Table 1: Zoning Layer API

initialize()	to connect to the game world
step_up()	to *step up* to zone owner
join()	to join a zone as zone member and listen *update* messages
update()	to *update* modified global states
commit()	to send a commit message
release()	to release direct connection to the zone owner
step_down()	to *step down* from zone owner and close all connections to zone members

Therefore, we employ DHT as a backup data storage, in other words, we use local cache of zone data on its zone owner as a master data. When a node becomes the zone owner of some zone, the new zone owner uploads its local cache of zone data from the data holder through DHT at first. After then, the zone owner reads and writes the local cache on its own storage as a master data of its zone. Once a zone owner has uploaded zone master data into its local cache, the zone owner doesn't need to search the data holder location to modify global states. To avoid losses of the latest master data of some zone when the zone owner moves out from the zone, each zone owner updates zone data on its data holder to leave a backup of master data on its zone.

2.3.2 Connection caching

Along with data caching on each zone owner, we combine connection caching in the zone model. Some global states, such as the location of a character, are likely to be changed frequently. It is wasteful to close the connection between a zone owner and a zone member whenever a notification of state changes finishes.

Therefore, we take DHT as a rendezvous point between a zone owner and its zone member. As a rendezvous point, global states on a data holder are stored with not only its zone owner's location, but also with the addresses of each zone member. When an independent node becomes a zone member, the new zone member writes its address in the DHT while checking its zone owner location by using DHT. Once a zone member finds a zone owner through the DHT, the zone member keeps the connection to the zone owner until the zone member no longer needs any global states on the zone. Establishing the direct connection between the zone member and the zone owner, the zone member makes requests to the zone owner through the direct connection.

On the other hand, if no zone owner exists on some zone but its zone members are still alive, and if some node becomes new zone owner on the zone, the new zone owner can grasp all its members' addresses uploading all zone data, because zone members addresses are also described in zone data stored as a backup data on the zone's data holder. Therefore, new zone owner can updates all its members global states directly as soon as the zone owner finishes uploading the latest backup data stored on its data holder.

Using DHT as rendezvous points and backup storage, the zoned federation model lets each zone owner play the authoritative game server on its governing zone. In a sense, this model represents some kind of clustering game servers arranged on peer-to-peer network sparsely. In this model, each zone owner can update zone members' global states directly while keeping the consistency and the serialization of global states on each zone. We can consider that the zoned federation model can provide the necessary and sufficient low latency characteristics required by MOG.

3. IMPLEMENTATION

In this section, we describe our implementation of the zoned federation model. We have inserted *zoning layer* as a middle layer between game programs and TCP/IP stacks, and between game programs and the DHT layer (Fig.1).

We have implemented the zoning layer as a C++ library that works on NetBSD, FreeBSD, and Solaris. As a DHT implementation, we have also implemented the Pastry[3] which uses SHA1 [6] as the hash function. Our zoning layer implementation provides several APIs for game programs. On the assumption that game programs are written in event driven model, we have designed and implemented APIs (Table 1). Using these APIs, game programs can change its node status along with situations.

3.1 Node Behaviors on Zoning Layer

3.1.1 Change of node status

Whenever a game program on an independent node wants to modify any global states, the independent node has to become the zone owner or zone member of the zone which contains required global states.

First of all, an independent node checks its own DHT whether the zone owner exists or not. If the zone owner already exists, the independent node becomes the zone member of the zone. To become a zone member of some zone, the independent node sends a *join message* to the zone owner. As soon as the zone owner receives the join message from the independent node, the zone owner then adds the independent node as its zone member in zone members list on DHT, and sends an *accept message* to the independent node. When the node receives the accept message, the independent node changes its own status to zone member, and keeps the direct connection to the zone owner until when then game program on the zone member doesn't need global data managed the zone owner any more. This sequence to become zone member is called *join*. After succeeding join, a zone member sends *commit messages* through established direct connections to its zone owner if the game program demands to modify some global state.

Otherwise, if the zone owner of a specific zone doesn't exist, an independent node or a zone member becomes the zone owner. We call this action *step up*. The sequence of "step up" is as follows: an independent node or a zone member registers itself as zone owner on DHT, then, the new zone owner copies or reloads local cache of global states on its zone from DHT to avoid missing any zone members. After finishing step up, a new zone owner then establishes and keeps all connections to its zone members.

The releasing process of the direct connection between a zone owner and a zone member is as follows: if a zone owner wants to

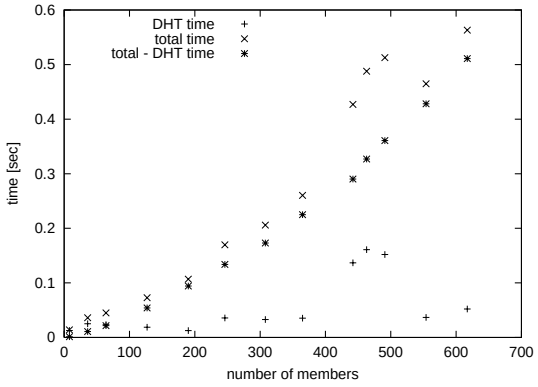

Figure 6: Step Up Time

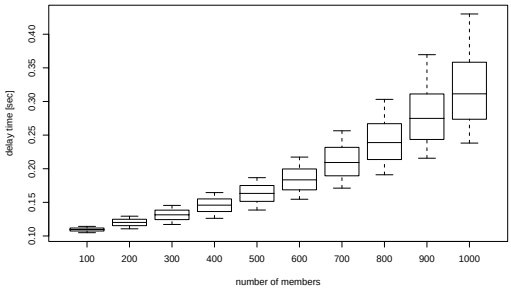

Figure 7: Update latency

step down, to resign the authoritative role, the zone owner deletes its entry as zone owner from the zone data located on DHT, and closes all connections to its zone members. On the other hand, if a zone member returns to a node, it deletes its own registration in the zone member list on the zone data, then closes the connection to the zone owner. We call this procedure "release".

3.1.2 Recovery from Failure

We should mention error processing when a node is suddenly isolated from peer-to-peer network because of some network failure. If a zone member is isolated from the network, we can consider that the zone member carried out a "release" action. The registration of the zone member on DHT still remains, however. The zone owner is the only node which knows the zone member has gone away, so the zone owner removes the zone member's entry from zone members list to keep consistency of the list.

When a zone owner has been disconnected from a network, its zone members notice that the zone owner was isolated from the network according to the absence of heartbeat messages from the zone owner. Then, each zone member sends *owner-lost message* to the game program. After sending an owner-lost message, one of these zone members deletes the zone owner entry from DHT. If some zone member game program demands a change of global state, the zone member tries to "step up".

If a zone owner disappears from network when no zone member is listed, the entry of the zone owner remains on DHT. In this case, a node becomes a zone member because of the remnant of the zone owner entry on DHT, but the new zone member receives no heartbeat message from 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 DHT and tries to "step up".

4. EVALUATION

In this section, we evaluate our implementation described in Section 3.

First, we evaluated *step up time*, that is, the time a node spends to become a new zone owner and to establish direct connections to all of its members. We used an experimental environment which consists of seven FreeBSD PCs with 256MB memory, three with 500MHz processors and the other four with 850 MHz, interconnected by a 100base-TX switch. In order to increase the number of zone members, we emulated multiple zone member nodes by running multiple zone member processes on a PC. In this test-bed environment, zone owner was placed in only one PC, and zone members were equally distributed among other six PCs.

The scenario of this experiment was as follows: at first, only zone members are running; next, a new independent node steps up to the zone owner. We evaluated this step up time while increasing the number of zone members gradually. The result of this experiment is shown in Figure 6. The time to establish connections to all zone members ("total - DHT time" in the figure) was proportional to the number of zone members, whereas the rendezvous overhead was almost constant but with inter-node delay fluctuations.

Next, we measured *update latency*, the time spent to update a global state of each zone member from the zone owner, from when a zone owner starts transferring an updated global state. The measurements have been done at Hokuriku IT Open laboratory, where 512 PCs are divided into five partitions and interconnected through switches. Each PC has Intel Pentium III 1GHz, 512 MB main memory, and two 100 Base-TX network interfaces. On each node, one of the NICs is connected to the control network, and another NIC is connected to the experimental network. We ran FreeBSD 4.7 on each PC. For this particular measurement, we used 296 PCs with a flat network topology.

We evaluated the effect of the number of zone members on a single zone. In this experiment, we used a PC for running only one zone owner process with 100 milliseconds artificial delay by dummynet, and zone member processes ran on other 295 PCs.

The update latency exhibits weak exponential trend when we increase the number of zone members from 100 to 1,000 (Figure 7). This trend can be explained by multiplexing of zone members onto single processor. The degree of multiplexing increases as zone member exceeds 295, 590, and 885, respectively.

The update latency of zoned federation model is comparable to client-server model in the case of single zone, since the single zone case is identical to client-server model when we only look at updates.

Next, we measured the effect of the number of zones to update time when total number of zone members was fixed to 297 nodes. We changed the number of zones from one to eight, and we distributed zone members to each zone uniformly. Figure 8 shows the result of this experiment. In contrast to the first experiment, artificial delay was not introduced here. From this figure, we can see that we can reduce the delay and narrow distribution of update time by dividing a large zone into several smaller zones.

5. RELATED WORK

Knutsson et. al. proposed SimMud, an alternative approach to MOG in peer-to-peer environments[7]. SimMud employs Pastry[3] and Scribe[8] as base components of its architecture. In SimMud approach, the authoritative role is given to a data holder node, which

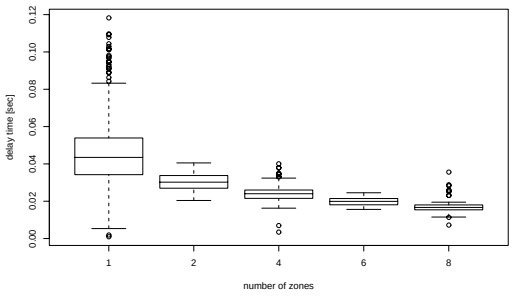

Figure 8: Effect of number of zones to update time

they call "coordinator". By randomly mapping coordinators on DHT, SimMud prevents game players from cheating global states. Also, by preparing several replicas of a coordinator, SimMud provides fault tolerance.

Since SimMud uses DHT and application-layer multicast (ALM) for message exchange, it incurs network delay by crossing several hops on both DHT and ALM. In contrast, our model is optimized for lower latency: except initial rendezvous by DHT, each node exchanges messages directly. Also, the zoned federation model optimizes the network latency without combining a central arbiter server like as [9].

The scalable data dissemination problem has been addressed in the application-level multicast literatures[8, 10], where large receiver groups are of particular concern. In contrast, our work focuses on zone-local data dissemination with low latency.

The API described in this paper resembles to the CAST interface which is part of the common API effort[11]. However, the underlying semantics have notable differences: zone-local serializability, and the presence of multiple roles. 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 MOG context, we believe that the serializability is of particular importance.

6. CONCLUSION

In this paper, we have proposed the zoned federation model, which adapts MOG to peer-to-peer networks. In this model, the whole game world is divided into several zones, and each zone is maintained by a federation of nodes: an owner and one or more members. 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, it ensures consistency of changes that are committed by other member nodes. DHT harnesses this zoning layer by providing rendezvous capability and by working as a backup medium for zone data.

We have applied this model to our prototypical MOG implementation, with which we have evaluated latency and scalability. Our results, based on synthetic workloads, indicate that the zoned federation model can achieve scalability by decomposing the whole game world into many zones and by maintaining each zone relatively small.

7. FUTURE WORK

We have constructed our model on DHT in order to achieve data consistency. However, if a data holder leaves from network, the zone data disappears until the data holder comes back to peer-to-peer network. In future work, we should consider the durability of global states by replicating states or by adopting loss-resilient codes.

While we have only looked at the application-layer topology, topology-aware overlay[12] will further reduce latency of intra-zone communication by optimizing network-layer topology of the overlay.

Working as the authoritative node is likely to be heavy task for a game player's PC. In our model, we assume that each player node plays fairly and trusts other players. Based on this assumption, we let a node, which wants to change some global state, become the zone owner. Therefore, cheating on the zone owner node is possible in our model. Cheating on the zone owner can be solved by replacing the zone owner role on the data holder node as proposed in SimMud[7]. However, this solution cannot deal with attacks by a large number of malicious nodes. These problems come from the lack of the mechanism for checking the trustworthiness of each node autonomously. If some reputation system like as EigenTrust[13] is available, it can replace an untrusted zone owner node or a malicious data holder node according to the relationship of mutual trust among participant nodes. Achieving the reliability of the zoned federation model is one of our future work.

8. REFERENCES

[1] I. Stoica et. al. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[2] S. Ratnasamy et. al. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[3] A. Rowstron et. al. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2001.

[4] B. Y. Zhao el. al. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.

[5] L. Pantel et .al. On the impact of delay on real-time multiplayer games. In *Proceedings of NOSSDAV*, 2002.

[6] D. Eastlake III et. al. US secure hash algorithm 1 (SHA1). RFC 3174, Internet Engineering Task Force, 2001.

[7] B. Knutsson et. al. Peer-to-peer support for massively multiplayer games. In *Proceedings of INFOCOM*, 2004.

[8] M. Castro et. al. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 2002.

[9] J. D. Pellegrino et .al. Bandwidth requirement and state consistency in three mutliplayer game architecture. In *Proceedings of NETGAMES*, 2003.

[10] S. Zhuang et. al. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of NOSSDAV*, 2001.

[11] F. Dabek et. al. Towards a common api for structured peer-to-peer overlays. In *IPTPS '03*, 2003.

[12] S. Ratnasamy et. al. Topologically-aware overlay construction and server selection. In *Proceedings of INFOCOM*, 2002.

[13] S. D. Kamvar et .al. The eigentrust algorithm for reputatin management in p2p networks. In *Proceedings of The 12th International World Wide Web Comferrence*, 2003.