A Distributed Event Delivery Method with Load Balancing for MMORPG

Shinya Yamamoto, Yoshihiro Murata, Keiichi Yasumoto and Minoru Ito
Graduate School of Information Science, Nara Institute of Science and Technology
Ikoma, Nara 630-0192, Japan
(shiny-ya,yosihi-m,yasumoto,ito)@is.naist.jp

ABSTRACT

In this paper, we propose a new distributed event delivery method for MMORPG (Massively Multiplayer Online Role Playing Games). In our method, the whole game space is divided into multiple sub spaces with the same size and some player nodes are selected as responsible nodes to deliver game events occurring in their responsible sub spaces. Our method includes (1) a load balancing mechanism which allows each responsible node for the crowded sub space to dynamically construct a tree of multiple nodes and deliver events along the tree to reduce event forwarding overhead per node, (2) a technique to reduce end-to-end event delivery delay by dynamically replacing nodes in the tree, and (3) a technique to efficiently and seamlessly switch sub spaces to be observed while each player's view moves around in the game space. Through experiments, we show that our method achieves practical performance for MMORPG.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

algorithms

Keywords

Multiplayer Game, Load Balancing, Event Delivery Architecture

1. INTRODUCTION

In recent years, MMOG (Massively Multiplayer Online Games) has become popular. Most of commercial MMOGs have been implemented as client server systems where the global game state consisting of positions and states of all players and so on is managed in a centralized way. The centralized control has advantages in keeping security high and implementation easy. However, it has several drawbacks in scalability on the number of players, management cost, and so on. In order to save the management cost, it is desirable to compose the game system only of player nodes in a P2P fashion without specific servers or high-speed networks. So, we need a method to manage the global game state of the MMOG in a distributed way by multiple player nodes with P2P overlay techniques.

When we let multiple player nodes to cooperatively manage the global game state, the following criteria should be satisfied: (1) the computation and communication overhead of each node is regulated below the specified threshold independently of the number of players and (2) the node failure does not influence the progress of the game. For RPG and FPS (First Person Shooter) type games, the following criterion should also be satisfied: (3) the update interval of the game state is small enough.

There are several research efforts which divide the global game state into multiple sub states, and let multiple nodes to maintain those sub states [1, 5, 6, 8]D [1] has proposed a method where a game space is divided into multiple sub spaces with the same size and the game state in each sub space is managed by a player node. In [6], a game space is dynamically divided depending on the positions of player characters using a technique of Voronoi graph partition algorithm so that player nodes whose characters in the same partitioned region directly exchange game events and keep the same game state. In [5, 8], the global game state is divided into multiple sub states (i.e., the partial game state of player nodes in the same sub space or with the same membership). For each sub state, a selected player node is assigned to manage it, and other player nodes which require the sub state can easily retrieve it through distributed hash tables such as Pastry [2]. These techniques allow player nodes to obtain any sub state when the membership changes, e.g., by moving to different location in the game space. These techniques are also robust against node failure by letting more than one node to backup each sub state.

Techniques in [1, 6, 8] satisfy the above criterion (3) since they allow player nodes with the same sub state to exchange game events among those nodes directly or through a management node. However, as the number of player nodes with a sub state increases, the amount of communication messages sent from each player node or its management node also increases. So the above criterion (1) is not satisfied completely. Also, [6] does not satisfy the criterion (1) since positions of player characters are managed in a centralized way. On the other hand, [5] uses a overlay multicast technique called SCRIBE [3] to deliver the game events to all player nodes with the same sub state. So, the amount of communication messages which each node must forward can be regulated within a threshold, and thus the criterion (1) is satisfied. However, in this technique, when the number of player nodes with a sub state becomes large or the membership of those nodes frequently changes, event delivery delay and cost for updating distributed hash tables increase. So, in such a case, it would be difficult to satisfy the above criterion (3). For the criteria (2), [5, 8] have mechanisms to cope with failure and leaving of nodes in hash tables.

Moreover, if we target MMORPG (Massively Multiplayer Online Role Playing Games), the global game state should be divided into many pieces, since so many players may join the same game at the same time. If the game space is divided into multiple small sub spaces, then each player's view may overlap with multiple sub spaces. So the game system should satisfy the following criterion: (4) sub spaces overlapped with each player's view must be switched quickly and seamlessly as the view moves around in the game space. The existing studies [1, 5, 6, 8] do not propose techniques for achieving the criterion (4) in detail.

In this paper, we propose a new distributed event delivery method for MMORPG. Similarly to [1], in the proposed method, the whole game space is divided into multiple sub spaces with the same size. For each sub space, a player node called the *responsible node* is selected among all player nodes to deriver events which occur in the sub space. In MMORPG, many player characters are likely to converge to a specific location in the game space due to special events or so on. In the proposed method, even when the number of player characters in a sub space becomes large, communication and computation overhead of the responsible node is regulated below the specified threshold, keeping the end-to-end event delivery as small as possible. To satisfy the criterion (1), we let each responsible node observe the number of player characters, and dynamically construct/extend a tree of multiple nodes (called *load balancing tree*) so that the communication and computation overhead of each node in the tree does not exceed the predefined threshold by delivering event messages through the tree. For the criterion (2), a *backup node* is assigned to each responsible node so that it can seamlessly take over the event delivery in the sub space when the responsible node fails or leaves. To satisfy the criterion (3), the backup node is also used to shorten the end-to-end event delivery latency while events are delivered through the load balancing tree. For the purpose, the backup node is replaced with one of intermediate nodes in the tree if the end-to-end latency improves. We also provide an efficient mechanism for satisfying the criterion (4), where player nodes can quickly and seamlessly switch responsible nodes for subscription while the player characters move over sub spaces.

Through experiments with our prototype system running on LAN and simulations on ns-2, we have confirmed that the proposed method can achieve practical performance in terms of overhead and end-to-end latency required for MMORPG.

2. OUTLINE OF PROPOSED METHOD

In this paper, we suppose MMORPG where multiple players share the same game space and time and each player has a bird's eye view of a part of the game space.

2.1 Definition

We assume that there is a *lobby server* which manages login/ logout of player nodes and keeps the list of all player nodes currently joining the game.

We define that the whole game space consists of the background image and multiple objects. Each object is either a player character, a moving object (e.g., character controlled by the system), or a static object (e.g., weapon, jewel, etc). Each object has several properties such as the position and state. We call an incident which may change the properties of an object, an *event*. For example, movement of an object, attack to an enemy character, or change of object's color/shape, is an event. We call the information of current properties of all objects in the whole game space, the *global game state*. We denote the global game state at time t by $GS(t)$. $GS(t)$ changes as the game progresses. In principle, in order to keep consistency of the game progress, $GS(t)$ must be shared among all players. We define that the *view* of each player is a sub space centered around the position of the player character in the game space. The view of each player corresponds to the part of the whole game space which the player can see through his/her computer display, and it is represented as a rectangle. The view of each player may include several parts of the whole game space which are not adjacent. For time t and any sub space v, let $GS(t, v)$ denote the information of the current properties of all objects which are in v. That is, $GS(t, v)$ denotes the game state of sub space v at time t. For each sub space v and each time t, we define that the game progress is *consistent* if all players whose views overlap with v at time t have $GS(t, v)$.

2.2 Basic Ideas of Proposed Method

In order to achieve the consistent game progress without specific servers in a P2P environment as well as the criteria (1) to (4) in Sect. 1, we propose the following mechanisms.

- **Distributed Event Delivery Mechanism:** As shown in Fig. 1, we divide the whole game space into multiple sub spaces with the same size. We assign a player node (called the *responsible node*) to each sub space so that it receives events from all players in the sub space and delivers those events to the player nodes.

- **Event Delivery Control Depending on Player's View:** Since each player's view covers multiple sub spaces, as player character moves in the game space, the sub spaces in its view also change. So, we propose a mechanism to quickly and seamlessly switch the event delivery paths between the player nodes and the responsible nodes of the sub spaces in the view.

- **Dynamic Load balancing:** When the number of player characters in a sub space becomes large, the responsible node of the sub space would be overloaded due to communication and computation overhead. So, we allow each responsible node to dynamically construct a tree of multiple player nodes called the *load balancing tree* and deliver events through the tree when the sub space gets crowded. When game events are delivered through the load balancing tree, the end-to-end event delivery delay becomes larger than without the tree. So, we propose a technique to shorten the event delivery delay through the tree.

The above three mechanisms are explained in detail in Sect. 3, Sect. 4, and Sect. 5, respectively.

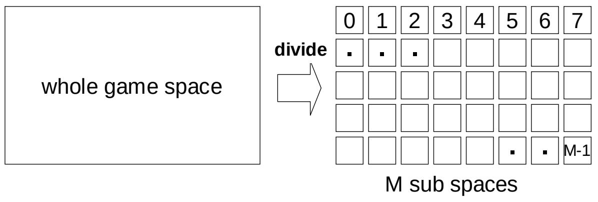

Figure 1: Division of Whole Game Space

3. EVENT DELIVERY MECHANISM

In our proposed method, the whole game space is divided into M sub spaces with the same size rectangles (Fig. 1). Let $v_0, v_1, ..., v_{M-1}$ denote sub spaces. The responsible node keeps the list of players in the sub space, and receives/forwards events from/to those players. The responsible nodes are selected from all player nodes by the lobby server, according to their computation powers and available bandwidths.

Each player is informed of the initial position in the game space as well as the addresses of the responsible nodes whose responsible sub spaces overlap with his/her view by the lobby server, when he/she joins the game. Each responsible node knows each player in its responsible sub space when the player subscribes to the node.

The responsible node of a sub space is allowed to be the responsible node of another sub space, the *intermediate node* of the load balancing tree (explained in Sect. 5.1) and / or the *backup node* (explained in Sect. 3.2) at the same time, as long as it has extra computation power and available bandwidth.

The responsible node of each sub space $v_i (0 \leq i \leq M-1)$ receives events occurred in v_i and forwards them to players (nodes) who can observe v_i (i.e., his/her view overlaps with v_i). In our method, we use a publish/subscribe system for event delivery. That is, when a player executes an event in v_i, the event message is sent to the responsible node of v_i (i.e., the event is published), and the responsible node forwards the event message only to player nodes which have subscribed to v_i. Each player node renews game state $GS(t, v_i)$ with published event messages of sub space v_i. To avoid inconsistency of the game state among players, event messages are collected and delivered every fixed time unit called *timeslot*. Also, to prevent the game from being disturbed due to node failure (e.g., physical breakdown, leaving the game, network congestion, etc), we assign a *backup node* to each responsible node. Below, we will explain the details.

3.1 Event Delivery and Game State Update

Let $V(p)$ denote the view of player p. Also, let $R(v_i)$ denote the responsible node of sub space v_i.

The clocks of all player nodes are synchronized loosely with NTP (Network Time Protocol). We represent the time in the game by the sequence numbers of timeslots. Time slot t_i represents the time $[T_0 + i \times \Delta, T_0 + (i+1) \times \Delta)$ where the length of each timeslot is Δ and the beginning time of the game is T_0.

In our method, event delivery is carried out in the following steps.

1. At the beginning time of the game, each player (node) p subscribes to each responsible node $R(v_j)$ such that v_j overlaps with $V(p)$.

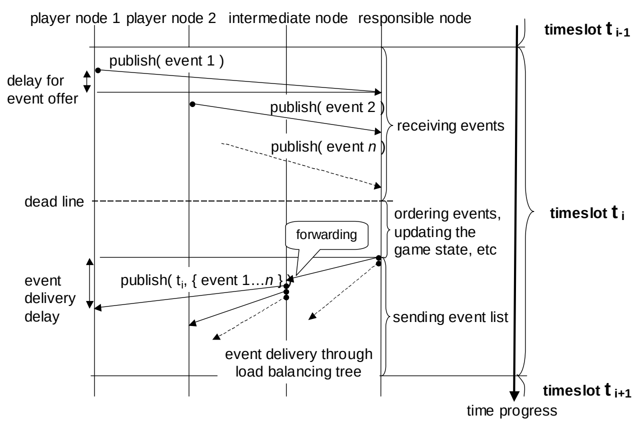

Figure 2: timeslot

2. When $R(v_j)$ receives the subscription from p, it sends the latest game state $GS(t, v_j)$ to p.

3. When p wants to publish an event, it sends an event message to the responsible node of sub space v_m where the location of the event is included in v_m [1].

4. Each responsible node $R(v_j)$ receives event messages from player nodes until deadline of the current timeslot t_i (see Fig. 2) and determines the orders of the received event messages [2]. Then $R(v_j)$ updates the game state $GS(t_i, v_j)$ according to the order of events and sends a message with the ordered event list to $R(v_j)$'s subscribers (i.e., player nodes who have subscribed to $R(v_j)$).

5. Each player node updates the game state to $GS(t_i, v_j)$ based on the received event list.

In our method, each player node updates and manages the latest game state $GS(t, v)$ by itself. This approach is effective when the amount of events occurring in a time slot is smaller than that of the game state. In MMORPG, the amount of the game state of each sub space is likely to become large. Also, with this approach, the recovery is easier when responsible nodes fail.

When a player node cannot receive an event list in a timeslot due to message loss or so on, it can request the lost message or the latest game state by sending a NACK message with the timeslot number to the responsible node. This makes it easy for each player node to keep the consistency of the game state.

3.2 Coping with Node Failure

We let the lobby server select player nodes which have higher computation powers and more available bandwidths, and keep these nodes' addresses in a buffer called the *backup node queue*.

[1]When a player wants to publish an event in two or more locations over multiple sub spaces simultaneously, it sends messages to all responsible nodes of those sub spaces. In this case, these events are not always processed at the same time slot, because they may be received in those responsible nodes at different time.

[2]The decision method can be either random, the order of receiving time, their hybrid or so on.

In order to cope with failure or leaving of responsible nodes, one backup node is assigned to each responsible node picked up from the backup node queue.

We let each responsible node $R(v)$ communicate with its backup node (denoted by r_b) periodically so that r_b checks whether $R(v)$ is working. Also, r_b receives the list of subscribers from $R(v)$. When r_b cannot communicate with $R(v)$, we replace r_b with $R(v)$ and let r_b behave as the new responsible node. After this replacement, r_b broadcasts the *replacement notification message* to the subscribers and asks a subscriber to send the latest game state.

4. EVENT DELIVERY CONTROL DEPENDING ON PLAYER'S VIEW

When we manage the global game state as the set of the sub game states which are managed in the corresponding sub spaces separately, we need a mechanism to allow each player to *dynamically* and *seamlessly* switch responsible nodes for subscription as his/her view moves in the game space. [1] simplifies this mechanism by assuming that each player's view always corresponds to a sub space with the same size in the whole space (see Fig. 1). [5, 8] allow each player to have a view including two or more sub spaces. However, they do not provide a detailed method of how each player dynamically switches the set of sub spaces depending on movement of player's view.

4.1 Dynamic Switching of Responsible Nodes for Subscription

Below, we show the proposed method to achieve the dynamic switching of sub spaces depending on movement of player's view.

Let $\{v_{p_1}, ..., v_{p_m}\}$ denote a set of sub spaces overlapping with player p's view. p subscribes to the responsible node of each sub space $v_{p_i} (1 \leq i \leq m)$. We assume that each player can obtain addresses of responsible nodes whose sub spaces overlap with his/her initial view informed by the lobby server, when he/she joins the game.

When a player p's character moves and a sub space v_{p_i} gets out of p's view, player node p sends the *unsubscribe message* to the responsible node $R(v_{p_i})$. When $R(v_{p_i})$ receives this message, it removes this player from the list of subscribers. Similarly, when p's character moves and a new sub space v_{p_j} comes into p's view, p sends the subscribe message to the responsible node $R(v_{p_j})$.

To allow each player node p to get addresses of the responsible nodes of neighbor sub spaces (hereafter called neighbor responsible nodes), we let each responsible node $R(v_{p_j})$ retain addresses of v_{p_j}'s eight neighbor responsible nodes (8 directions: north, south, east, west and oblique angles). When each player node sends a subscribe message to a responsible node of sub space v, it receives as a reply the latest game state $GS(t, v)$ with addresses of v's neighbor responsible nodes.

When a responsible node is replaced with a backup node due to node failure or so on (Sect. 3.2) , the new responsible node sends its address to its neighbor responsible nodes.

4.2 Seamless Switching of Responsible Nodes for Subscription

In the method where the global game state is maintained separately in sub spaces, each player cannot receive events

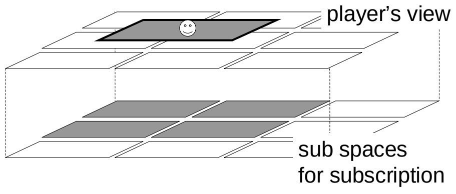

Figure 3: Sub Spaces and Player's View

occurred in unsubscribed sub spaces. Therefore, if each player subscribes to a responsible node after a new sub space comes into his/her view, the player may miss receiving events which occurred in the new sub space just before the space comes into his/her view. To cope with this problem, we define the *subscription range*. The subscription range of player p is defined by a rectangle which has the same center as p's view and has longer sides than p's view by constant length $2 * d_1$. We let each player p subscribe to sub spaces overlapping with this range instead of p's view.

When the border of the subscription range and the border of a sub space are close to each other, and when a player character moves around in a narrow space, subscription and unsubscription to the same sub space may be repeated redundantly. To avoid this problem, we define the *unsubscription range*. The unsubscription range of player p is defined by a rectangle which has the same center of p's view and has longer sides than p's subscription range by constant length $2 * d_2$. Each player node p cancels the subscription to sub space v only when v gets out of this range.

With these technique, when a new sub space v approaches near the border of a player p's view within distance d_1, p can subscribe to $R(v)$ and obtain the latest game state $GS(t, v)$ before v is displayed on his/her display. Also, whenever a player character moves around between two points within distance d_2, the redundant repetitions of subscription and unsubscription are avoided. In general, use of larger values of d_1 and d_2 can increase the effects, but may produce more control massages. So, we should carefully decide appropriate values considering tradeoff.

5. DYNAMIC LOAD BALANCING

When the number of players increases in a sub space, load (i.e., computation and communication overhead) due to receiving and forwarding event messages at the responsible node also increases. In the proposed method, the *load balancing tree* is used to distribute load of a responsible node to multiple player nodes.

5.1 Construction of Load Balancing Tree

Let $Player(r)$ denote a set of players who have subscribed to the responsible node r of a sub space. For a given integer constant C, when $|Player(r)| > C$, the responsible node r dynamically constructs the load balancing tree as a k-ary tree as shown in Fig. 4. We assume that C and k are given as constant integers in advance and that $C \geq k$.

To construct the load balancing tree, a player node is picked up from the backup node queue and assigned as an *intermediate node*, which relays event messages between r and each node in $Player(r)$. r sends event messages along the load balancing tree so that the load is regulated below the specified threshold C at each node. The load balancing

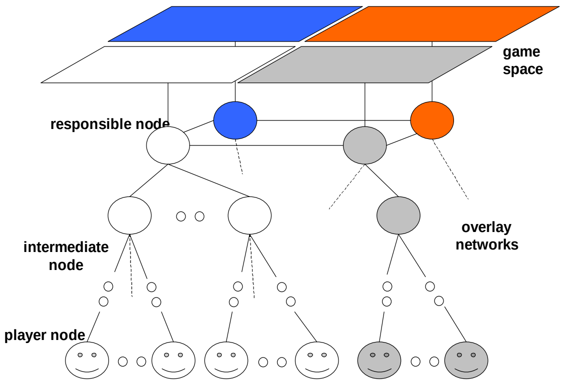

Figure 4: Load Balancing Tree for Sub Space

tree construction algorithm is described as follows.

- (Step 1) The responsible node r observes the number of intermediate nodes (denoted by m) on the load balancing tree and the number of subscribers $|Player(r)|$. r distributes $Player(r)$ among intermediate nodes so that each intermediate node retains no more than C subscribers. Let $Player^*(r)$ denote the set of subscribers retained by the tree whose root is r. When subscribers increase and $|Player^*(r)| > m \times C$ holds, a new intermediate node is picked up from the backup node queue and assigned as $m+1$-th child node of r and the value of m is increased. Conversely, if subscribers decrease and $|Player^*(r)| \leq (m-1) \times C$ holds, an intermediate node is removed and the value of m is decreased. When a new player subscribes to r, r forwards the subscription message to one of its child nodes with less than C subscribers.
- (Step 2) Let $r_1, r_2, ..., r_k$ denote r's child nodes, and r_1 denote r's leftmost child node. If $m > k$ holds by adding an intermediate node, a new node (from the backup node queue) is assigned as r_1's child node (denoted by $r_{1,1}$). Here, $|Player(r_1)|$ is reduced to $C-k$, and k subscribers are taken over by $r_{1,1}$. Let m_1 denote the number of child nodes of r_1. Let $Player^*(r_1)$ denotes the set of subscribers retained by the sub tree whose root is r_1. While $|Player^*(r_1)| \leq k \times C$ holds with new subscriptions and unsubscriptions, r_1 adds and/or removes intermediate nodes (r_1's child nodes) in a similar way to (Step 1). When $|Player^*(r_1)| > k \times C$ holds, (step 2) is applied to $r_2, ..., r_k$ in this order. When $|Player^*(r_k)| > k \times C$ holds, (step 2) is applied to $r_{1,1}, r_{1,2}, ...,$ recursively.

In the above algorithm, for seamless event delivery, we should allocate each intermediate node and distribute subscribers to the node before the number of subscribers exceeds the threshold C. In our method, assignment of an intermediate node and taking over of subscribers can be carried out in advance, for example, when condition $|Player^*(r_i)| > (m_i - 1) \times C + \frac{1}{2}C$ holds. The actual event delivery through the intermediate node (r_i's m_i+1-th child) can be done when condition $|Player^*(r_i)| > m_i \times C$ holds.

5.2 Reduction of End-to-end Event Delivery Delay

As the number of subscribers increases in a sub space, the depth of the load balancing tree becomes larger, and thus the end-to-end event delivery delay becomes larger as well. However, as we stated as the criterion (3) in Sect. 1, realtime event delivery for the constant game state update is essential for MMORPG. According to the report in [7], the game state must be updated at least every 400 msec to prevent players from feeling something wrong. So, we propose a method to reduce end-to-end delay by dynamically replacing intermediate nodes in the load balancing tree.

The load balancing tree for the sub space v consists of the responsible node r (root node), intermediate nodes and subscribers' nodes (leaf nodes) as shown in Fig. 4. Among these nodes, subscribers' nodes are likely to leave the tree when sub space v gets out of their views. On the other hand, the responsible node r does not change until it leaves from the game. Therefore, as a basic policy, we replace each intermediate node with a backup node of r to reduce the end-to-end delay. The algorithm is described as follows (see also Fig. 5).

(1) r attaches time stamp to the list of event messages collected in each time slot, and sends it to each child node $r_i (1 \leq i \leq k)$. It also sends the current subscriber list $Player(r)$ to the backup node r_b after attaching time stamp to it.

(2) When each child node r_i receives the event list, it computes delay D_i by the difference between the message arrival time and time stamp in the message, and sends D_i to r_b. To mitigate the jitter of delay, D_i can be computed as the average of delays measured over two or more time slots. Here, we assume that the clocks of all player nodes are synchronized loosely (a few msec for the maximum), for example, with NTP.

(3) When r_b receives the message with $Player(r)$ and time stamp, it computes delay D_b by the difference between the message arrival time and time stamp in the message. It also receives the message with delay D_i from $r_i (1 \leq i \leq k)$, and computes the maximum delay $D_m (1 \leq m \leq k)$. If $D_m > D_b$, it sends the *replacement message* to r so that r_b is replaced with r_m.

(4) If r receives the replacement message, it sends the message with $Player(r_m)$ (i.e., the list of subscribers assigned to r_m) to r_b, and it forwards the list of event messages to r_b, hereafter.

(5) When r_b receives the forwarded event list, it sends the *replacement completed message* to r_m and behaves as the m-th child of r, hereafter.

(6) If old r_m receives the replacement completed message, it is added to the tail of the backup node queue in the lobby server. Then, a new node is picked up from the backup node queue and assigned as the new backup node of r. The above steps (1) to (6) are repeated until no new replacement happens.

In the case that the height of the load balancing tree is three or more, the steps (1) to (6) are applied to each child node of $r_i (1 \leq i \leq k)$ recursively, after finishing replacement of all child nodes of r.

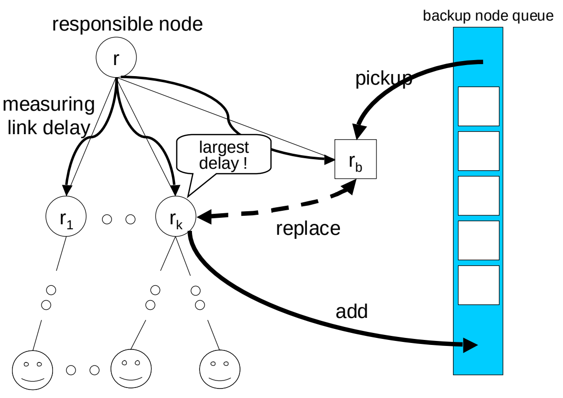

Figure 5: Replacement of Intermediate Nodes in Load Balancing Tree

6. EXPERIMENTS AND EVALUATION

To evaluate the proposed method, we measured computation and communication overhead at a responsible node and an intermediate node of the load balancing tree for a sub space, changing the number of players in the sub space (**Experiment 1**).

We also measured end-to-end event delivery delays through the load balancing tree in WAN by simulations with ns-2 (**Experiment 2**).

Experiment 1

We have implemented a prototype system based on the proposed method and executed the system on eight PCs connected via a 100BASE-T LAN, where we used two PCs a and b for player nodes, one PC c for the responsible node, and five PCs d, e, f, g and h for intermediate nodes in the load balancing tree. The specifications of PCs are as follows: the responsible node c: Pentium4 3GHZ and 1GB Memory; and other nodes: Pentium 3/4 0.6 – 3 GHZ and 256 – 1024 MB memory. Debian GNU Linux is installed on all of these PCs.

To simplify the experiment, we have implemented two modules for imitating the behavior of all player nodes in the sub space: the receiver module and the sender module which were executed on node a and node b, respectively. The sender module running at node a sends the same number of event messages as the number of supposed player nodes in the sub space every timeslot. The receiver module running at node b receives packets with the list of events from the responsible node or via the load balancing tree.

The responsible node c receives event messages from node a until the deadline of each time slot, composes a list of received events, and forwards it to b or to several internal nodes when the load balancing tree exists.

In the experiment, since we used 64 byte packet for each event message, the size of each event list would be 64 byte $\times$ the number of players [3]. We used $\Delta = 400$ms for each timeslot, and $C = 20$, $k = 5$ for the load balancing tree. The deadline of each timeslot was set to 200ms point from the beginning of the timeslot.

With the above experimental setting, we measured the CPU load (with the system call `clock()`), the communication overhead, and the forwarding delay (i.e., time interval between the deadline and the time when starting to send the event list) at the responsible node c for both cases with and without the load balancing tree. The results are shown in Fig. 6, Fig. 7, and Fig. 8, respectively.

[3]In each timeslot, some players may not execute any event. So, actual size of each event list would be smaller.

From Fig. 6 and Fig. 7, we see that the CPU load and the communication overhead increase monotonously as the number of player nodes increases, in the case without the load balancing tree ($C = 100$). When the number of players increases to about 100, the CPU load was still low enough (0.3%), but the communication overhead increased fatally (more than 10Mbps). With the load balancing tree (C=20), the CPU load and the communication overhead were regulated below the reasonable values (below 0.2% and 0.75 Mbps, respectively). This is because the load balancing tree was constructed and the event list was delivered through the tree after the number of players exceeded $C = 20$. If we set the smaller value for C, we can make the CPU load and the communication overhead lower.

Fig. 6 suggests that the load balancing tree can reduce CPU load of the responsible node to roughly half of that without the tree. Therefore, with our technique, each responsible node can treat twice more players with the same computation power.

We suppose MMORPG with 50 to 100 players per sub space at peak time (10 to 100 thousand players in the whole game space). As shown in Fig. 7, when the number of players is 100, the required network bandwidth at the responsible node is around 12 Mbps if the load balancing tree is not used. If the tree is used, the bandwidth is reduced to 0.75 Mbps. This shows that the proposed method can keep the communication overheads at the responsible nodes small enough for practical use.

In Fig. 8, we see that the forwarding delay at the responsible node increases linearly if the load balancing tree is not used (10ms when the number of players is 100). If we use the load balancing tree with $C = 20$, the delay is regulated around 3 msec independently of the number of players.

In summary, we confirmed that the proposed method with the load balancing tree reduces the CPU load, the communication overhead and the forwarding delay. However, the proposed method increases the end-to-end event delivery delay since the event messages are sent through intermediate nodes. So, we must carefully decide the values of k and C, considering tradeoff.

Experiment 2

To evaluate an effect of the proposed technique in Sect. 5.2, we measured the end-to-end event delivery delay in WAN by simulations with ns-2.

In simulations, we generated hierarchical network topologies with a topology generator called Tiers[10]. Each topology contains 1011 nodes with a WAN, 10 MANs per WAN and 10 LANs per MAN, where each WAN, each MAN and each LAN include one node, one node and ten nodes, respectively. We set that link delays within the same LAN are 2ms, link delays between different LANs through a MAN are 10ms, and link delays between different MANs through a WAN are 100ms.

Supposing a sub space in the game space, one responsible node and 125 player nodes were selected at random from 1011 nodes.

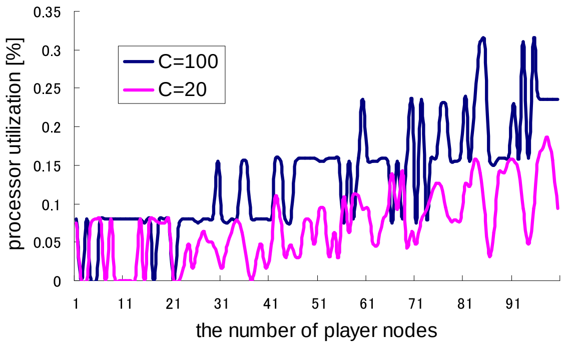

Figure 6: CPU Load

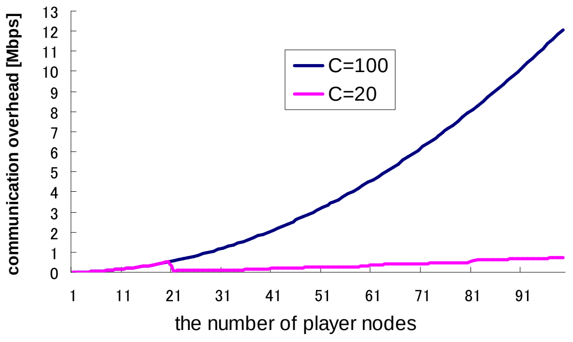

Figure 7: Communication Overhead

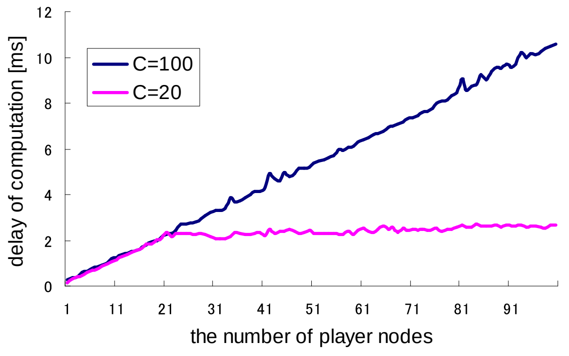

Figure 8: Forwarding Delay

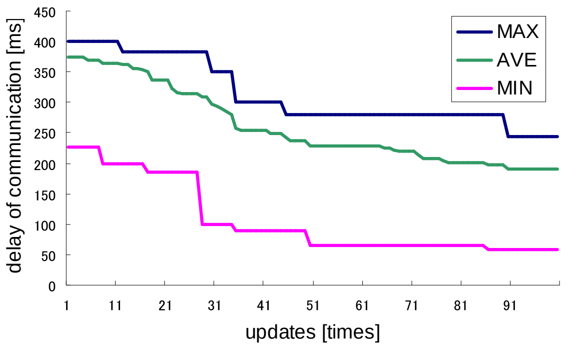

Figure 9: Improvement of End-to-end Delay

Intermediate nodes of the load balancing tree and the backup nodes were selected at random from unused nodes. It is assumed that the frequency of event occurrence is e=2.5 times/second (i.e., timeslot is Δ = 400msec), the threshold value is C=5, and the tree degree is k=5. Here, the height of the load balancing tree was four: the root node (the responsible node), five child nodes, 25 grandchild nodes, and 125 player nodes.

To simplify the experiment, we composed the load balancing tree with all the above intermediate nodes and player nodes in advance, and applied the algorithm of Sect. 5.2 to the tree in the simulation, and measured how the end-to-end event delivery delays change.

The result calculated by the average of the five simulations is shown in Fig. 9.

In the figure, we see that the maximum, average, and minimum (MAX, AVE, and MIN in Fig. 9) of the delivery delay were reduced gradually and converged to the minimum value after about 90 replacements of intermediate nodes. After these replacements, the maximum end-to-end delivery delay was reduced by about 50 percent from the initial delay and even the maximum delay converged to below 250msec.

In an actual environment, since the number of players in a sub space increases gradually, the number of intermediate nodes also increases gradually. Thus, in actual game situations, the end-to-end event delivery delay could be improved more quickly.

7. CONCLUSION

In this paper, we proposed a new distributed event delivery method for MMORPG.

This method includes (1) a load balancing mechanism which allows each responsible node for the crowded sub space to dynamically construct a tree of multiple nodes and deliver events along the tree to reduce event forwarding overhead per node, and (2) a technique to reduce the end-to-end event delivery delay through the load balancing tree by replacing one of intermediate nodes with the backup node incrementally. We also proposed techniques for efficient and seamless switching of sub spaces for subscription while each player's view moves in the game space.

Through experiments with our prototype system running on LAN and simulations on ns-2, we have confirmed that the proposed method can regulate computation and communication overhead of each responsible/intermediate node below the specified threshold, and achieve the end-to-end delay small enough for MMORPG in realistic environments.

As part of future work, we are planning to implement the proposed method as middleware library and evaluate its performance using actual traces of MMORPG.

8. REFERENCES

[1] A. Bharambe, S. Rao and S. Seshan: "Mercury: A Scalable Publish-Subscribe System for Internet Games", *Proc. of 1st Workshop on Network and System Support for Games (NetGames2002)*, 2002.

[2] A. Rowstron and P. Druschel: "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware2001)*, LNCS2218, pp. 329–350, 2001.

[3] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron: "SCRIBE: A large-scale and decentralised application-level multicast infrastructure", *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 2002.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan: "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", *Proc. of ACM SIGCOMM'01*, pp. 149–160, 2001.

[5] B. Knutsson, H. Lu, W. Xu, and B. Hopkins: Peer-to-Peer Support for Massively Multiplayer Games, *Proc. of INFOCOM 2004*, 2004.

[6] S. Y. Hu, G. M. Liao: "Scalable peer-to-peer networked virtual environment", *Proc. of the 3rd Workshop on Network and System Support for Games (NETGAMES 2004)*, pp. 129–133, 2004.

[7] T. Henderson: "Latency and Behaviour on a Multiplayer Game Server", *Proc. of 3rd Int'l. Workshop on Networked Group Communication (NGC2001), LNCS2233*, pp. 1–13, 2001.

[8] T. Iimura, H. Hazeyama and Y. Kadobayashi, "Zoned Federation of Game Servers: a Peer-to-peer Approach to Scalable Multiplayer Online Games", *Proc. of the 3rd Workshop on Network and System Support for Games (NETGAMES 2004)*, 2004.

[9] http://www.isi.edu/nsnam/ns/index.html.

[10] http://www.isi.edu/nsnam/ns/ns-topogen.html.