マルチメディア通信と分散処理 79－6
（1996. 11. 14）

Pseudo-Active Replication in Heterogeneous Clusters

Kenji Shima, Hiroaki Higaki, and Makoto Takizawa
Tokyo Denki University
e-mail {sima, hig, taki}@takilab.k.dendai.ac.jp

Some distributed systems are required to support fault-tolerant services while including widely available components like workstations and networks. An approach towards making the system fault-tolerant is to replicate the processes. In these systems, the replicas are realized in different kinds of processors. In the active replication, the replicas are computed and communicated in the same synchronous way. The computation speed of the process depends on the slowest replica. In this paper, we would like to discuss a new replication scheme named a *pseudo-active* one where events may occur not simultaneously, not in the same order, and may not occur in the replicas. The process can issue succeeding requests to the replicas without waiting for the completion of the slower replicas.

不均質クラスタにおける疑似能動的多重化

島 健司　桧垣 博章　滝沢 誠
東京電機大学理工学部経営工学科

分散型システムでは、システムの信頼性を向上させるためにフォールトトレラント技法が用いられ、多重化が広く利用されている。Isis等で用いられている能動的多重化では、プロセスの全レプリカは、同一の動作を行なう。本論文では、レプリカが異なったプロセッサに配置された環境で、システムの性能の低下をもたらさない多重化方式として疑似能動的多重化を提案する。この方式では、性能差のあるプロセッサ上にレプリカが配置された場合に、一番遅いプロセッサの性能によりシステム全体の性能が決ってしまう。

1 Introduction

In various kinds of distributed applications, multiple autonomous application processes are cooperated to achieve some objectives by communicating with each other through networks. The processes in the distributed system may suffer from faults, i.e. *stop* and *Byzantine faults* [6]. An approach towards making the system fault-tolerant is to replicate the processes. In this paper, each process is replicated into a collection of multiple *replicas*, which is named a *cluster*. There are two replication schemes, (1) *active replication* [10] and (2) *passive replication* [3]. In the active one, every replica does the same computation and communication. Isis [2] adopts the active one. In the passive one, only one primary replica does the computation and communication. The other back-up replicas change the states only on receipt of the checkpoint from the primary one. The replicas can continuously support the service in the presence of the faults in the active replication, while it takes time to recover from the fault of the primary replica in the passive one.

The distributed system includes kinds of processors, i.e. processors have different processing speeds and levels of reliability. If the replicas in a cluster are allocated in different types of processors, the cluster is *heterogeneous*. The computation of the process completes only if the computations of all the replicas complete. Furthermore, if the slowest processor does not satisfy the response time requirement, the process cannot be actively replicated. In this paper, we would like to propose a new *pseudo-active* one where the computation of the process can be completed without waiting for the completion of the slower replicas. In this paper, we would like to discuss how to detect the slower replicas and how the slower replicas can catch up with the faster replicas. The pseudo-active replication could support faster than the active replication.

In section 2, we present the replication schemes. In section 3, we present the system model. In section 4, we present the pseudo-active replication.

2 Replication Schemes

We would like to discuss how to replicate a process p_i into replicas $p_{i1}, \ldots, p_{il_i}$ ($l_i \geq 1$). A collection $\{p_{i1}, \ldots, p_{il_i}\}$ of the replicas of p_i is a *cluster* c_i. There are two kinds of approaches towards replicating p_i [3, 10]: (1) *active* replication and (2) *passive* replication. In the active one [10], every replica p_{ij} is modeled as a deterministic finite state machine. That is, every p_{ij} ($j = 1, \ldots, l_i$) in c_i does the same computation by receiving and sending the same messages in the same order. Even if some p_{ij} is faulty, the computation of

p_i is continued without stopping as long as some number of the replicas are operational.

In the passive one [3], there is one *primary* replica p_{i1}. The others $p_{i2}, \ldots, p_{il_i}$ are named *backup* ones. p_{i1} receives and sends messages and does the computation of the messages received while no backup replica does the computation. p_{i1} takes a checkpoint and sends the local state information saved at the checkpoint to all the backup ones. Then, every backup replica changes the local state. If p_{i1} is faulty, one of the backup replicas start to compute from the checkpoint as a new primary replica. Hence, it takes time to recover from the fault of the primary replica.

The active replication implies more redundant processing and communication than the passive one because all the replicas do the same computation by sending and receiving the same messages. However, it requires less time-overhead for recovering from faults. On the other hand, the passive replication implies less overhead but it requires more time-overhead for recovering from the faults. We adopt the active replication to realize the highly available application since it is no time for recovering from the fault.

3 System Model

A distributed system is composed of multiple processors interconnected by the communication network. A distributed application is realized by the cooperation of multiple processes computed on the processors and communicating with each other. A collection of processes is a *group* $G = \{p_1, \ldots, p_n\}$ $(n \geq 2)$, where the processes are cooperated to achieve some objectives. In order to tolerate the process faults, each process is replicated to a collection of replicas, i.e. *cluster*. Hence, G is viewed to be composed of multiple *clusters*. Let c_i be a cluster which is a collection $\{p_{i1}, \ldots, p_{il_i}\}$ $(l_i \geq 1)$ of the replicas of p_i.

A cluster c_i in G is *homogeneous* iff all the replicas in c_i are allocated in the same types of the processors. Each processor is characterized in terms of processing speed and reliability level. c_i is *heterogeneous* iff the replicas are allocated in different processors. For example, a p_{ij} is computed in an UltraSparc and p_{ik} in a Sparc10.

The processes are assumed to suffer from the stop-faults. We assume that each replica is located in different processors. That is, the replicas are *independently* faulty.

We assume that the the communication network is *reliable* and *synchronous*, i.e. messages are neither lost, contaminated, nor duplicated, and the propagation delay time is bounded. We also assume that messages are transmitted in the first-in first-out order. If a replica p_{ik} sends p_{jh} m_1 before m_2, p_{jh} receives m_1 before m_2.

Let $s_i(m)$ and $r_i(m)$ denote the sending and receipt events of a message m in a process p_i, respectively.

[Causal precedence[5]] For every pair of events e_1 and e_2, e_1 *causally precedes* e_2 $(e_1 \rightarrow e_2)$ iff one of the following conditions holds:

(1) e_1 occurs before e_2 in some process.
(2) $e_1 = s_i(m)$ and $e_2 = r_j(m)$.
(3) For some event e_3, $e_1 \rightarrow e_3 \rightarrow e_2$. □

Two events e_1 and e_2 are *concurrent* $(e_1 \parallel e_2)$ iff neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$.

[Definition] For every pair of messages m_1 and m_2, m_1 *causally precedes* m_2 $(m_1 \rightarrow m_2)$ iff $s_i(m_1) \rightarrow s_j(m_2)$. □

Here, m_1 and m_2 are concurrent $(m_1 \parallel m_2)$ iff neither $m_1 \rightarrow m_2$ nor $m_2 \rightarrow m_1$. The replicas have to deliver m_1 before m_2 if $m_1 \rightarrow m_2$. Many group communication protocols [2, 8, 9, 12, 13] have been proposed to support the causally ordered delivery of messages.

In order to causally deliver messages, the vector clock [7] is used. Each message m carries the vector clock V where each element V_{ij} shows the local clock of p_{ij}. Each time p_{ij} sends a message m, V_{ij} is incremented by one. On receipt of m, $V_{kl} := \max(V_{kl}, m.V_{kl})$ for every k and l. If p_{ij} sends a message m, $m.V_{kl} := V_{kl}$ for every k and l. For every pair of messages m_1 and m_2, $m_1 \rightarrow m_2$ iff $m_1.V < m_2.V$. By using the vector clock, the replicas order the messages received in the causal order. However, message loss cannot be detected. The paper [9] presents a method by which the message loss can be detected by using the vector of message sequence numbers.

4 Pseudo-Active Replication

4.1 Active replication

The replicas of the process are often obliged to be distributed to different kinds of processors because it is expensive to newly obtain the same kinds of processors. Hence, a cluster is rather *heterogeneous*. In the active replication, every replica is required to do the same computation and communication at the same time [10]. If the process p_i is actively replicated to be multiple replicas $p_{i1}, \ldots, p_{il_i}$ in the heterogeneous cluster c_i, the processing speed and reliability levels of each p_{ij} are different from each other. For example, some replica p_{ij} is computed in a faster processor like Ultra Sparc while another p_{ik} is in a slower processor like Sparc2. Here, the computation of p_{ij} is faster than p_{ik}. Here, suppose that some process sends a request to p_{ij} and p_{ik}. The computation of the request in p_{ij} is faster than p_{ik}. The process has to wait for the response from p_{ik} after receiving the response from p_{ij}. Thus, every replica has to wait for the completion of the computation of the slowest replica in c_i even if the responses from the replicas in the faster processors are received. Hence, we propose a new replication scheme named a *pseudo-active* replication by relaxing the constraints of the *active* replica-

tion in order to decrease the response time in the heterogeneous clusters.

The computation of each replica p_{ij} is modeled as the deterministic finite state machine [Figure 1]. Let s^0_{ij} denote the initial state of p_{ij}. Here, suppose that $s^0_{i1} = \cdots = s^0_{il_i}$. If an event e^1_{ij} occurs in s^0_{ij}, the state s^0_{ij} of p_{ij} is transited to the 1st state s^1_{ij}. Thus, the hth state s^h_{ij} is transited to s^{h+1}_{ij} if an event e^{h+1}_{ij} occurs in s^h_{ij}. Here, p_{ij} is represented in a sequence of the events $e^1_{ij} \circ e^2_{ij} \circ \cdots \circ e^{t_{ij}}_{ij}$. Here, let $e^h_{ij}(s^{h-1}_{ij})$ denote a state s^h_{ij}. $e^1_{ij} \circ \cdots \circ e^{t_{ij}}_{ij}(s^0_{ij}) = s^{t_{ij}}_{ij}$.

The computation of p_i is viewed to be a state transition, which is realized by the cooperation of the replicas $p_{i1}, \ldots, p_{il_i}$ in c_i. If a p_{ij} has the same state transition as p_i, i.e. $s^h_i = s^h_{ij}$ and $e^h_i = e^h_{ij}$ for every h, p_{ij} is *complete* for p_i. Here, e_{ij} denotes an instance of an event e occurs in p_{ij}.

The cluster $c_i = \{p_{i1}, \ldots, p_{il_i}\}$ of p_i is *actively replicated* if the following conditions hold. Here, $p_{i1}, \ldots, p_{il_i}$ are operational replicas of p_i.

[Active replication (AR) conditions]

AR1 : For every pair of operational replicas p_{ij} and p_{ik}, $s^h_{ij} = s^h_{ik}$ and $e^h_{ij} = e^h_{ik}$ for every h.

AR2 : For every pair of operational replicas p_{ij} and p_{ik}, $e^h_{ij} \rightarrow e^{h+1}_{ik}$ and $e^h_{ik} \rightarrow e^{h+1}_{ij}$.

AR3 : Every operational p_{ij} is complete. □

AR1 means that every replica does the same computation and communication in the cluster c_i. In every p_{ij}, the same state transitions occur and the same events occur in the same order. AR2 means that every event occurs simultaneously in every replica of c_i. Every event e^h_{ij} occurs in p_{ij} after the events $e^{h-1}_{i1}, \ldots, e^{h-1}_{il_i}$ occur in $p_{i1}, \ldots, p_{il_i}$, respectively. Two replicas p_{ij} and p_{ik} are *synchronized* iff p_{ij} and p_{ik} satisfy AR2. AR3 means that every replica p_{ij} does the same computation as p_i if p_{ij} is computed alone. If every replica misses one event, AR3 does not hold although AR1 and AR2 hold.

There are two kinds of events, i.e. local events and communication events. These are two kinds of communication events, i.e. *sending* and *receipt* events. Let m^{ij} denote an instance of a message m sent by a replica p_{ij}.

[Proposition] p_{ij} and p_{ik} are *synchronized* if $m_1 \rightarrow m^{ij}_2$ iff $m_1 \rightarrow m^{ik}_2$ for every pair of messages m_1 sent by p_{ij} and m_2 received by p_{ik}. □

It is sure that AR2 is satisfied if all the replicas of the process are synchronized. In Figure 2, p_{ij} and p_{ik} receive a message m_1. After receiving m_1, p_{ij} and p_{ik} send m^{ij}_2 and m^{ik}_2 respectively. p_{hl} sends m_3 after receiving m_2 from p_{ij} and p_{ik}. Since $m^{ij}_2 \rightarrow m_3$ and $m^{ik}_2 \rightarrow$ Here, suppose that m_1 is a request and m_2 is a response of m_1. p_{ij} and p_{ik} are synchronized if p_{hl} sends a request m_3 after receiving the responses of m_1 from p_{ij} and p_{ik}. m_3, p_{ij} and p_{ik} are synchronized.

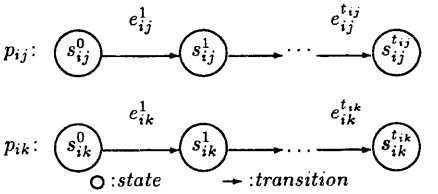

Figure 1: State transition.

4.2 Heterogeneous clusters

A heterogeneous cluster c_i includes replicas distributed in different types of processors. The processors are *different* if they have different processing speeds and support different levels of reliability. The replicas in different types of processors are computed at different speeds. If the heterogeneous cluster c_i is actively replicated, the computation of c_i has to wait for the completion of the slowest replica in c_i, because AR2 is required to be satisfied. In order to improve the response time and throughput, the computation of c_i is tried to be completed before the computation of every replica completes in this paper. For example, suppose that the fastest replica p_{ij} completes the computation while the others are still computing in a cluster c_i.

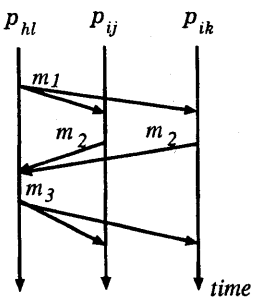

Figure 2: Synchronized replicas.

First, let us like to consider that AR2 does not hold while AR1 is satisfied. That is, events may not occur simultaneously in every replica while every event occurs in every replica in the same order.

[Definition] Let p_{ij} and p_{ik} be two different replicas of a process p_i. p_{ik} *follows* p_{ij} ($p_{ij} \Rightarrow p_{ik}$) iff for some events e_{ij} and e'_{ij} occurring in p_{ij} and e_{ik} and e'_{ik} in p_{ik}, $e_{ij} = e_{ik}$, $e'_{ij} = e'_{ik}$, $e_{ij} \rightarrow e'_{ij}$, $e_{ik} \rightarrow e'_{ik}$, and $e_{ij} \rightarrow e'_{ik}$, but $e'_{ik} \not\rightarrow e'_{ij}$. □

p_{ij} and p_{ik} are not synchronized ($p_{ij} \parallel p_{ik}$) iff $p_{ij} \Rightarrow p_{ik}$ or $p_{ik} \Rightarrow p_{ij}$. If $p_{ij} \Rightarrow p_{ik}$ and $p_{ik} \Rightarrow p_{ij}$, p_{ij} and p_{ik} are *heated* ($p_{ij} \equiv p_{ik}$).

[Definition] p_{ik} *completely follows* p_{ij} iff for every events e_{ij} and e'_{ij} in p_{ij}, and e_{ik} and e'_{ih} in p_{ik} such the $e_{ij} \rightarrow e'_{ij}$, $e_{ij} = e'_{ij}$, $e_{ik} = e'_{ik}$, and $e_{ik} \rightarrow e'_{ik}$, $e_{ij} \rightarrow e_{ik}$ if $e'_{ij} \rightarrow e'_{ik}$. □

If the processor computing p_{ij} is faster than p_{ik}, p_{ik} completely follows p_{ij}. Figure ?? shows two replicas p_{ij} and p_{ik} while p_{ij} is followed by p_{ik}. p_{ik} is still in a state c where p_{ij} is already in a state e.

In this paper, we assume that every p_{ij} can decide how messages received by p_{ij} are causally preceded, e.g. by using the vector clock [6]. Here, suppose that p_{ij} and p_{ik} send messages m_2^{ij} and m_2^{ik}, respectively, after receiving m_1 before m_3 as shown in Figure 3. In the active replication, p_{hl} is required to send m_3 after receiving m_2 from every replica. However, p_{hl} sends m_3 after receiving m_2^{ij} without waiting for m_2^{ik} in order to decrease the response time. On receipt of m_3, p_{ij} and p_{ik} know that $m_2^{ij} \rightarrow m_3$ but $m_2^{ik} \not\rightarrow m_3$. That is, p_{ij} and p_{ik} know that m_3 is sent before receiving m_2^{ik} from p_{ik}. Hence, p_{ij} and p_{ik} can decide how one of them follows the other according to the following proposition.

[Proposition] A replica p_{ik} *follows* p_{ij} ($p_{ij} \Rightarrow p_{ik}$) if $m_2^{ij} \rightarrow m_3$ but $m_2^{ik} \not\rightarrow m_3$ for some messages m_3 received and m_2 sent by p_{ij} and p_{ik}. □

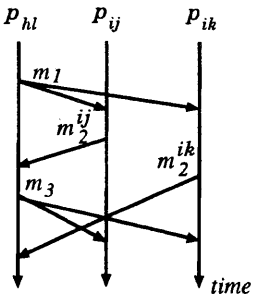

Figure 3: p_{ik} follows p_{ij}.

In Figure 3, p_{ik} follows p_{ij} since $m_2^{ij} \rightarrow m_3$ but $m_3 \parallel m_2^{ik}$. That is, p_{ij} and p_{ik} are not synchronized. For example, suppose that m_1 is a request and m_2 is the response of m_1, and m_3 is a request. On completion of the computation of m_1, p_{ij} sends back the reply m_2 to p_{kl}. Here, suppose that it takes longer to compute m_1 in p_{ik} than p_{ij}. That is, p_{ik} is slower than p_{ij}. If p_{hl} waits for all the replies from p_{ij} and p_{ik}, it takes longer to compute m_1. p_{hl} can send the succeeding request m_3 to p_{ij} and p_{ik} on receipt of m_2^{ij} if m_2^{ik} would be surely received by p_{hl}. Hence, p_{hl} receives m_2^{ik} after sending m_3. By this method, the performance of the system can be improved even if the system includes the slower processors. Thus, $p_{ij} \Rightarrow p_{ik}$ means that p_{ik} is slower than p_{ij}.

Next, we consider how each p_{ij} can know if p_{ij} follows or succeeds other replicas. Each message m carries the vector clock $m.V$, where each element $m.V_{ij}$ denotes the local clock for p_{ij}. Suppose that p_{ij} receives m. p_{ij} knows that p_{ik} follows p_{ih} if $m.V_{ik} > m.V_{ih}$.

p_{ij} can decide how the replicas are followed by the following F rule.

[Follow (F) rule] For every pair of replicas p_{ih} and p_{ik},

(1) p_{ik} follows p_{ih} if $m.V_{ik} < m.V_{ih}$,

(2) p_{ik} and p_{ih} are synchronized if $m.V_{ik} = m.V_{ih}$. □

If p_{ij} decides that $p_{ih} \Rightarrow p_{ik}$, p_{ik} follows p_{ih} in p_{ij} ($p_{ih} \Rightarrow_{ij} p_{ik}$). Here, p_{ih} is *fastest* in c_i if $m.V_{ih}$ is $max(m.V_{i1}, \ldots, m.V_{il_i})$. p_{ih} is the *slowest* if $m.V_{ih}$ is $min(m.V_{i1}, \ldots, m.V_{il_i})$.

[Definition] A message m is referred to as *delayed* on p_{ij} if $m.V_{ij} < m.V_{ik}$ for some p_{ik}. □

If p_{ij} receives the delayed message, p_{ij} knows that p_{ij} follows some replica. If p_{ij} knows that p_{ij} follows the fastest p_{ik}, p_{ij} has to catch up with p_{ik}.

In the active replication, p_{hl} sends a request m_1 to p_{ij} and p_{ik}. On receipt of the responses m_2^{ij} and m_2^{ik} from p_{ij} and p_{ik}, p_{ij} considers that the computation of m_1 completes and sends a request m_3 to p_{hl} and p_{ik}. In the pseudo-active replication, p_{kl} does not wait for the replies from all the replicas since p_{hl} can send m_3 after p_{hl} receives one response from one replica. We assume that the replicas suffer from only the stop-faults.

Each replica p_{ij} receives messages from p_{k1}, $\ldots$, p_{kl_k} in a cluster c_k. Some of the replicas in c_k are slower. We consider how p_{ij} receives the messages from the replicas in c_k. After sending a request m_1 to all the replicas in c_k, p_{ij} sends succeeding messages on receipt of at least one reply m_2 from a replica in c_k. Here, m_1 is *completeed*. After m_1 is completed, p_{ij} receives the reply m_2^{kl} from the slower p_{kl}. On receipt of m_2^{kl}, p_{ij} manipulates only the vector clocks.

[Definition] A message m_1 received is *completed* in p_{ij} iff p_{ij} receives such a message m_2 that some p_{ik} sends m_2 and $m_1 \rightarrow m_2 \rightarrow m_3$. □

Next, let us consider that p_{ij} receives requests from p_{k1}, $\ldots$, p_{kl_k}. On receipt of a request m_1 from p_{kl}, p_{ij} starts to compute the request if m_1 had not been received from no replica. Problem is that p_{ij} receives requests m_1 and m_2 from different clusters c_k and c_h. In this paper, we assume that messages can be totally ordered. Every p_{ij} takes messages in the total order in c_k.

Suppose that a replica p_{hl} sends m_1 and m_3 to p_{ij} and p_{ik}, and p_{tu} sends m_4 and m_5 as shown in Figure 4. p_{hl} sends m_3 on receipt of m_2^{ij} from p_{ij} before receipt of m_2^{ik} from p_{ik}. Hence, on receipt of m_3, p_{ij} and p_{ik} know that p_{ik} follows p_{ij} ($p_{ij} \Rightarrow p_{ik}$). On the other hand, p_{tu} sends m_6 on receipt

of m_5^{ik} from p_{ik} before receipt of m_5^{ij}. Hence, on receipt of m_6, p_{ij} and p_{ik} know that p_{ij} follows p_{ik} ($p_{ik} \Rightarrow p_{ij}$). On receipt of m_3 and m_5, p_{ij} and p_{ik} have the contradiction, i.e. $p_{ij} \Rightarrow p_{ik}$ and $p_{ik} \Rightarrow p_{ij}$. That is, p_{ij} and p_{ik} are heated.

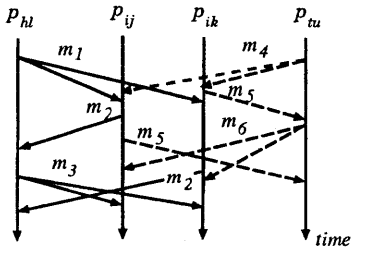

Figure 4: Regular cluster.

A cluster c_i is *regular* iff for every pair of replicas p_{ij} and p_{ik}, p_{ij} completely follows p_{ik} or p_{ij} and p_{ik} are synchronized. If p_{ik} is computed on the slower processor than p_{ij}, it takes longer for p_{ik} to compute every request than p_{ij}. This is the regular cluster. In the regular cluster c_i, either $p_{ij} \Rightarrow p_{ik}$, $p_{ik} \Rightarrow p_{ij}$, or $p_{ij} \parallel p_{ij}$. In the irregular cluster c_i, $p_{ij} \Rightarrow_{ih} p_{ik}$ and $p_{ik} \Rightarrow_{ih} p_{ij}$ in some p_{ih}. p_{ij}, p_{ij} is the *fastest* in c_i.

4.3 Equivalent events

The slower replica p_{ik} has to catch up with the faster complete p_{ij} in c_i. If every event stored in the log is required to occur in p_{ik}, it takes time and p_{ik} may not be able to catch up with p_{ij} since p_{ij} is faster than p_{ik}. Hence, we try to make p_{ik} to catch up with p_{ij} by omitting the events which are not necessary to occur, by preempting an event occurring at present which can be omitted, and by changing the occurrence order of the events.

First, we try to relax the AR1 condition. We discuss what operations can be omitted.

[Definition] p_{ij} is *superior* to p_{ik} iff every event occurring in p_{ik} occurs in p_{ij}. □

That is, some event e may not occur in p_{ik} while e occurs in p_{ij}. The complete replica is superior to the others.

[Definition] Let e be an event occurring in p_{ij}. e is *idempotent* in p_{ij} iff for every state s_{ij}, $e(s_{ij}) = s_{ij}$. □

For example, in the replica p_{ij} of the database server, an SQL *select* statement is idempotent. The idempotent event to occur can be omitted in the slower p_{ik}. If the idempotent events occurring in p_{ij} do not occur in p_{ik}, p_{ij} is superior to p_{ik}. The slower replicas can omit the idempotent events occurring in the faster replicas.

Next, the occurrence order of the events in p_{ij} is changed to catch up with the faster replicas.

[Definition] Two events e_1 and e_2 are *compatible* in p_{ij} iff for every state s_{ij} of p_{ij}, $e_1 \circ e_2(s_{ij}) = e_2 \circ e_1(s_{ij})$. □

For example, in the database server, two *select* statements are *compatible*. *deposit* and *withdraw* are compatible in a *bank* object. Suppose that e_1 and e_2 are compatible in p_{ik}. e_1 might be blocked in p_{ik} if the resources required by e_1 are held by other events. If so, p_{ij} can compute e_2 before e_1.

[Definition] Let s_1 be a sequence of events $e_{11} \circ e_{12} \circ \cdots \circ e_{1k_1}$ and s_2 be $e_{12} \circ e_{22} \circ \cdots \circ e_{2k_2}$. s_1 is *equivalent* to s_2 in p_{ij} ($s_1 \equiv s_2$) iff for every state s_{ij} of p_{ij}, $e_{11} \circ \cdots \circ e_{1k_1}(s_{ij}) = e_{21} \circ \cdots e_{2k}(s_{ij})$. □

In Figure 5, p_{ij} is complete and superior to p_{ik} and p_{ih}. If u and w are idempotent, $t \circ v \equiv t \circ u \circ v \circ w$. Furthermore, if t and v are compatible, $v \circ t \equiv t \circ v$. Therefore, p_{ih} and p_{ik} can catch up with p_{ij} by omitting idempotent u and w.

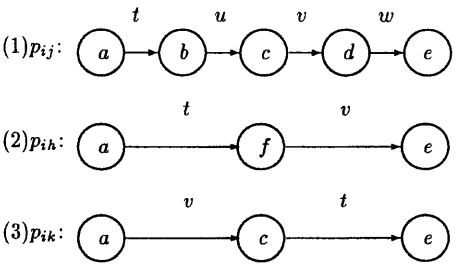

Figure 5: Replicas.

4.4 Catching up

We discuss how the slower replicas catch up with the faster ones by omitting the idempotent events and exchanging the compatible events.

[Definition] A message m is *obsolete* in p_{ij} iff

(1) m is received but not delivered to p_{ij}, and

(2) for some delayed message m' received in p_{ij} such that $m'.V_{ij} < m'.V_{ik}$ for some p_{ik}, $m \rightarrow m'$. □

In Figure 6, p_{ik} follows p_{ij} since it takes longer to compute the request m_1. During the computation of m_1, p_{ik} receives the requests m_3 and m_5 from p_{hl}. m_3 and m_5 are delayed and $m_3 \rightarrow m_5$. Hence, m_3 is obsolete. Since p_{ik} is sure that the computation of m_3 is completed in p_{ij}, p_{ik} does not need to complete m_3. However, p_{ik} might have to compute the request m_3. Thus, if a request message m is *obsolete* in p_{ij}, p_{ij} does not need to receive m and not send the reply of m since the computation for m is finished already. On receipt of m, p_{ij} stores m in the receipt queue RQ_{ij} in the causal order. p_{ij} takes a top message m_f from RQ_{ij}. In Figure 6, m_3 and m_5 are stored in RQ_{ij} and $m_3 \rightarrow m_5$.

p_{ij} can neglect the processing for m if m is obsolete in RQ_{ij}.

[Neglet procedure] If m is obsolete and is a re-

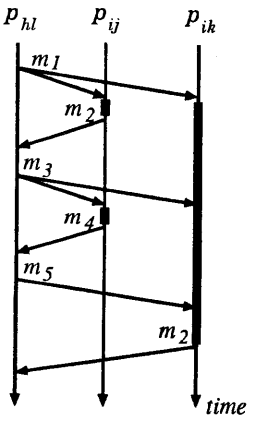

Figure 6: Obsolete message.

quest, p_{ij} neglects m and sends the response of m with null data. □

We have to check if the messages in RQ_{ij} are obsolete. On receipt of a message m, p_{ij} executes the following procedure.

[Receipt procedure] p_{ij} receives a message m .

(1) m is stored in RQ_{ij} in $\rightarrow$.

(2) If m is delayed on p_{ij}, m is marked *delayed.*

(3) If m is delayed, every message m' causally preceding m in RQ_{ij} is searched. m' in RQ_{ij} is marked *obsolete* if $m' \rightarrow m$. □

p_{ij} takes the top message m from RQ_{ij}. If m is marked *obsolete*, p_{ij} checks if m is a request. If m is a request, p_{ij} removes m and sends back the response of m with no result.

5 Concluding Remarks

We have discussed the pseudo-replication in the heterogeneous cluster. In the heterogeneous cluster, the computation of the cluster can complete if the faster replicas complete while the slower replicas are still being completed. The slower replicas can catch up with the complete faster replicas by omitting the idempotent events and changing the compatible events. In the dynamic cluster, the replication scheme is changed so that the requirements for the application and system environments are satisfied. By the dynamic replication, the distributed systems can support the application with the *flexible* service.

References

[1] Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M., and Budhia, R. K., "A reliable Ordered Delivery Protocol for Interconnected Local-Area Networks," *Proc. of IEEE ICNP-95*, 1995, pp.365–374.

[2] Birman, P. Kenneth, and Renesse, V. Robbert, "Reliable Distributed Computing with the Isis Toolkit," *IEEE CS Press*, 1994.

[3] Budhiraja, N., Marzullo, K., Schneider, B. F., and Toueg, S., "The Primary-Backup Approach," *Distributed Computing Systems, ACM Press*, 1994, pp.199-221.

[4] Higaki, H. and Takizawa, M., "Group Communication Protocol for Flexible Distributed Systems," to appear in *Proc. of IEEE ICNP-96*, 1996.

[5] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, Vol.21, No.7, 1978, pp.558–565.

[6] Lamport, L., Shostak, R., and Pease, M., "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems*, Vol.4, No.3, 1982, pp.382–401.

[7] Mattern, F., "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms* (Cosnard, M. and Quinton, P. eds.), *North-Holland*, 1989, pp.215–226.

[8] Melliar-Smith, P. M., Moser, L. E., and Agrawala, V., "Broadcast Protocols for Distributed Systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol.1, No.1, 1990, pp.17–25.

[9] Nakamura, A. and Takizawa, M., "Causally Ordering Broadcast Protocol," *Proc. of 14th IEEE ICDCS*, 1994, pp.48–55.

[10] Schneider, B. F., "Replication Management using the State-Machine Approach," *Distributed Computing Systems, ACM Press*, 1993, pp.169-197.

[11] Shima, K., Higaki, H., and Takizawa, M., "Fault-Tolerant Causal Delivery in Group Communication," *Proc. of IEEE ICPADS'96*, 1996, pp.302-309.

[12] Tachikawa, T. and Takizawa, M., "Selective Total Ordering Broadcast Protocol," *Proc. of IEEE ICNP-94*, 1994, pp.212-219.

[13] Tachikawa, T. and Takizawa, M., "Distributed Protocol for Selective Intra-group Communication," *Proc. of IEEE ICNP-95*, 1995, pp.234-241.