Vol. 39 No. 2 Transactions of Information Processing Society of Japan Feb. 1998

Regular Paper

Pseudo-active Replication in Heterogeneous Clusters

KENJI SHIMA,† HIROAKI HIGAKI,† and MAKOTO TAKIZAWA†

One approach to making distributed systems fault-tolerant is to replicate the processes. In systems composed of widely available commercial products, the replicas have to be realized in various types of processors. In active replication, the replicas are computed and communicated in the same synchronous way, and the computation speed of the process depends on that of the slowest replica. In this paper, we discuss a novel "pseudo-active" replication scheme in which events do not necessarily occur simultaneously or in the same order, and do not necessarily occur in the replicas. New requests can be issued to the replicas if some, but not necessarily all, replies are received from the replicas, without waiting for the completion of the slower replicas.

1. Introduction

In distributed applications, multiple autonomous application processes cooperate to achieve some objectives by exchanging messages. Mission-critical disributed applications require the system to be fault-tolerant. Processes may, for example, suffer from *stop* and *Byzantine*[9),13)] faults. One approach to making a system fault-tolerant is to replicate the processes in the system. In this paper, a collection of *replicas* is named a *cluster*. In the *active* replication[14)] adopted by Isis[3)], every replica performs the same computation and communication. In *passive* replication[4)], only one primary replica performs the computation and communication. In active replication, the replicas can provide continuous service in the presence of faults, while in passive replication it takes time to recover from a fault in the primary replica.

If the replicas in a cluster are allocated to different types of computer, the cluster is *heterogeneous*. The computers have various processing speeds and levels of reliability. A process is completed only if the computations of all the replicas are completed. In this paper, we discuss a novel pseudo-active replication scheme that reduces the response time and the total processing time and provides the same level of reliability as the active replication scheme. Here, a process can be completed if the faster replicas complete their computation without waiting for the slower replicas. The slower replicas have to catch up with the faster ones. We discuss a *distributed* way for each replica to detect the slower replicas by using the vector clock[10)] carried by messages. In addition, we discuss how the slower replicas can catch up with the faster ones by omitting events and changing the order of occurrence of events. In pseudo-active replication, the response time and total computation time of the replicas are shorter than in active replication, even if the slower computers are included. In addition, pseudo-active replication offers the same level of reliability as active replication; that is, the process continues as long as at least one replica is operational.

In Section 2, we overview the replication schemes. In Sections 3 and 4, we present the system model and explain pseudo-active replication. In Section 5, we evaluate pseudo-active replication by comparing the total computation time with that in active replication.

2. Replication Schemes

A process p_i is replicated in order to make p_i fault-tolerant. A collection $\{p_{i1}, \ldots, p_{il_i}\}$ ($l_i \geq 1$) of replicas of p_i is a *cluster* c_i. There are two approaches: *active*[14)] and *passive*[4)] replication. In active replication[3),14)], every replica p_{ij} ($j = 1, \ldots, l_i$) in c_i performs the same computation by receiving and sending the same messages in the same order. p_i is operational as long as at least one replica is operational, provided only stop-faults occur.

In passive replication[4)], c_i contains one *primary* replica p_{i1} and *backup* replicas $p_{i2}, \ldots, p_{il_i}$. Replica p_{i1} exchanges messages and computes the messages received, while no backup replica performs any computation. p_{i1} takes a checkpoint and sends the local state in-

† Department of Computers and Systems Engineering, Tokyo Denki University

formation saved at the checkpoint to all the backup replicas. Then, every backup replica changes its local state. If p_{i1} is faulty, one of the backup replicas, say p_{ik}, is selected to be primary and starts to compute from the checkpoint taken most recently. Hence, it takes time to recover from a fault in the primary replica; in other words, less time is available.

The active replication approach involves more redundant processing and communication than the passive one, but the computation can be continued as long as at least one replica is operational. Hence, we adopt active replication to realize the highly available applications.

To reduce the communication overhead in active replication, we propose *hybrid* replication[18].

3. System Model

3.1 Heterogeneous Clusters

A distributed system is composed of multiple computers interconnected by a communication network. A distributed application is realized through cooperation of multiple processes. Each process is computed in a computer. A *group* G is a collection of cooperating autonomous processes $p_1, \ldots, p_n$, (**Fig. 1**). Each p_i is replicated in a collection $\{p_{i1}, \ldots, p_{il_i}\}$ ($l_i \geq 1$) of replicas, i.e. *cluster* c_i. This is a *one-replica* cluster iff it includes only one replica. It is *homogeneous* iff all the replicas it contains are in the same types of computer. Each computer is characterized in terms of processing speed and reliability level. Cluster c_i is *heterogeneous* iff some replicas are in different types of computer; for example, it is heterogeneous if a replica p_{ij} is computed in a faster computer such as an UltraSparc station and another replica p_{ik} is computed in a slower computer such as a Sparc5.

In this paper, the processes are assumed to stop as a result of faults.

3.2 Causal Precedence

Let $s_{ij}(m)$ and $r_{ij}(m)$ denote the sending and receiving events of a message m in a replica p_{ij}, respectively.

[**Causal precedence**][8] An event 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) $e_1 = s_{ij}(m)$ and $e_2 = r_{kl}(m)$.

(3) $e_1 \rightarrow e_3 \rightarrow e_2$ for some event e_3. □

[**Definition**] A message m_1 *causally precedes*

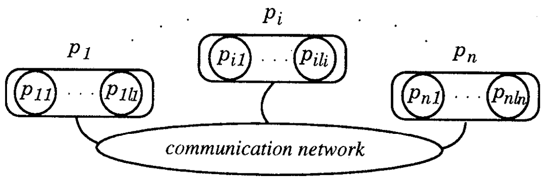


Fig. 1 Group G.

m_2 ($m_1 \rightarrow m_2$) iff $s_{ij}(m_1) \rightarrow s_{kl}(m_2)$. □

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[3),5),7),11),12),16),17)] have been proposed to support a group of multiple processes with causally ordered delivery of messages.

To deliver messages causally, each p_{ij} manipulates the vector clock[3),10)] $V = \langle V_{kh} \mid k = 1, \cdots, n, h = 1, \cdots, l_i \rangle$. Each element V_{kh} shows the local clock of p_{kh}, which p_{ij} knows. Initially, $V_{kh} = 0$. Each time p_{ij} sends a message m, V_{ij} is incremented by 1. Message m carries the local clock $m.V$; that is, $m.V_{kh} = V_{kh}$ for every k and h. On receipt of a message m, p_{ij} manipulates V as $V_{kh} := \max(V_{kh}, m.V_{kh})$ for every k and h. The vector clock satisfies the following property[10)]:

[**Property**] 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 messages received are sequenced in "$\rightarrow$". However, a *gap* between messages, that is, a message loss, cannot be detected. Hence, the network is assumed to support the reliable data transmission of messages to multiple destinations in Isis. Nakamura and Takizawa[12)] present a method by which not only are messages causally ordered, but also gaps can be detected by using the vector of message sequence numbers.

In this paper, the network is assumed to support all the replicas in each cluster with totally and causally ordered delivery of messages. That is, every replica can receive all the messages in the same causal order.

4. Pseudo-active Replication

4.1 Active Replication

Replicas $p_{i1}, \ldots, p_{il_i}$ of the process p_i are often obliged to be distributed to various types of existing computers, because it is expensive to replace existing computers with a set of computers all of the same type; in this case, the cluster c_i is *heterogeneous*. In active replica-

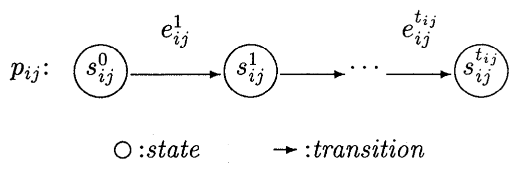


Fig. 2 State transition.

tion, every replica is required to perform the same computation and communication at the same time as the other replicas [14]. Here, suppose that a process p_s sends a request to p_{ij} and p_{ik}. If p_{ij} is faster than p_{ik}, p_s has to wait for the response from p_{ik} even after receiving the response from p_{ij}. Thus, every replica has to wait for the slowest replica in c_i. We therefore propose a *pseudo-active* replication by relaxing the constraints of *active* replication in order to decrease the response time in heterogeneous clusters.

Each replica p_{ij} can be also modeled as a deterministic finite state machine (**Fig. 2**). Let s^0_{ij} denote the initial state of p_{ij}. Here, $s^0_{i1} = \cdots = s^0_{il_i}$. If an event e^1_{ij} occurs in s^0_{ij}, s^0_{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. Here, p_{ij} is represented in a sequence of the events $e^1_{ij} \circ \cdots \circ e^{t_{ij}}_{ij}$. Let $e^h_{ij}(s^{h-1}_{ij})$ denote a state s^h_{ij} and let $e^{h-1}_{ij} \circ e^h_{ij}(s^{h-2}_{ij}) = e^h_{ij}(s^{h-1}_{ij}) = s^h_{ij}$. There occur local events and communication events, namely, *sending* and *receiving* events. e^h_{ij} denotes an instance of an event e^h in p_{ij}.

p_i is *actively replicated* in a cluster $c_i = \{p_{i1}, \ldots, p_{il_i}\}$ if the following conditions hold:

[**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}, $e^h_{ij} \to e^{h+1}_{ik}$ and $e^h_{ik} \to e^{h+1}_{ij}$ for every h.
- AR3: No operational replica p_{ij} loses any event. □

AR1 means that every replica performs the same computation and communication in c_i, that is, the same events occur in the same order. AR2 means that every event occurs simultaneously in every replica. Every event e^h_{ij} occurs in p_{ij} after e^{h-1}_{ik} occurs in every p_{ik}. p_{ij} and p_{ik} are *synchronized* iff they satisfy AR2. AR3 means that every p_{ij} performs the same computation as a cluster including only p_{ij}. If every replica misses some event, AR3 does not hold, although AR1 and AR2 hold. p_{ij} *includes* p_{ik} iff every event occurring in p_{ik} occurs in p_{ij}.

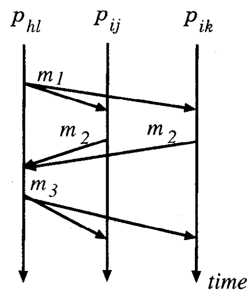


Fig. 3 Synchronized replicas.

Here, let m^{ij} show an instance of a message m sent by p_{ij}.

[**Proposition 1**] p_{ij} and p_{ik} are *synchronized* if $m_1 \to m^{ij}_2$ iff $m_1 \to m^{ik}_2$ for every pair of messages m_1 and m_2 respectively received and sent by p_{ij} and p_{ik}. □

AR2 holds if all the replicas in c_i are synchronized. In **Fig. 3**, 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^{ij}_2 and m^{ik}_2. Since $m^{ij}_2 \to m_3$ and $m^{ik}_2 \to m_3$, p_{ij} and p_{ik} are synchronized.

4.2 Following Relation

The replicas in the faster computers support a shorter response time than the slower ones. The computation speed of the cluster c_i depends on the slowest replica in c_i, because AR2 has to be satisfied. The response time can be reduced if the computation of c_i is completed before that of every replica. For example, suppose that the computation of a request m from p_h is completed in the fastest replica p_{ij} while the other replicas are still computing m in c_i. Replica p_{ij} sends the response of m to p_h. Here, p_h considers that m is completed in c_i, although p_h has not received all the responses from c_i.

First, we consider a case in which AR1 holds but AR2 does not. That is, every event occurs in the same order but does not necessarily occur simultaneously in every replica.

[**Definition**] p_{ik} *follows* p_{ij} ($p_{ij} \Rightarrow p_{ik}$) iff $e_{ij} = e_{ik}$, $e'_{ij} = e'_{ik}$, $e_{ij} \to e'_{ij}$, $e_{ik} \to e'_{ik}$, and $e_{ij} \to e'_{ik}$, but $e_{ik} \not\to e'_{ij}$ for some events e_{ij} and e'_{ij} occurring in p_{ij} and e_{ik} and e'_{ik} in p_{ik}. □

p_{ij} and p_{ik} are synchronized iff neither $p_{ij} \Rightarrow p_{ik}$ nor $p_{ik} \Rightarrow p_{ij}$. If $p_{ij} \Rightarrow p_{ik}$ and $p_{ik} \Rightarrow p_{ij}$, p_{ij} and p_{ik} are *thrashed*. p_{ij} and p_{ik} may be

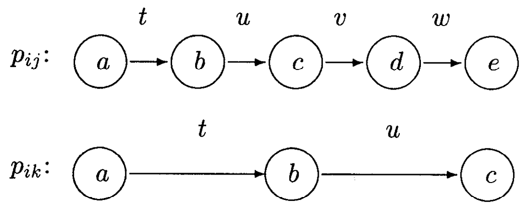


Fig. 4 Heterogeneous replicas.

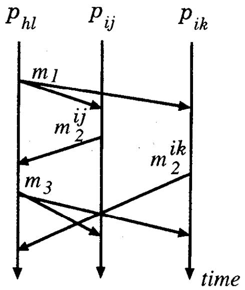


Fig. 5 p_{ik} follows p_{ij}.

thrashed if either p_{ij} or p_{ik} sometimes becomes slower because of overload. **Figure 4** shows that p_{ik} follows p_{ij}. p_{ik} is still in state c while p_{ij} is already in e, because $u_{ij} \to u_{ik}$.

[**Definition**] p_{ik} *fully follows* p_{ij} $(p_{ij} \rightsquigarrow p_{ik})$ iff $e'_{ij} \to e'_{ik}$ if $e_{ij} \to e_{ik}$ for every events e_{ij} and e'_{ij} in p_{ij}, and e_{ik} and e'_{ik} in p_{ik} such that $e_{ij} = e_{ik}$, $e'_{ij} = e'_{ik}$, $e_{ij} \to e'_{ij}$, and $e_{ik} \to e'_{ik}$. □

If the computer of p_{ij} is always faster than that of p_{ik}, then p_{ik} fully follows p_{ij}. It is trivial to show that $p_{ij} \Rightarrow p_{ik}$ if $p_{ij} \rightsquigarrow p_{ik}$.

A cluster c_i is *regular* iff every pair of replicas p_{ij} and p_{ik} are synchronized or one of p_{ij} and p_{ik} fully follows the other. Here, both $p_{ij} \Rightarrow p_{ik}$ and $p_{ik} \Rightarrow p_{ij}$ may hold in some p_{ik}. That is, the processing speed of some replica is dynamically changed.

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 **Fig. 5**. In 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}. On receipt of m_3, p_{ij} and p_{ik} know that $m_2^{ij} \to m_3$ but $m_2^{ik} \not\to m_3$, that is, they know that p_{hl} sends m_3 before receiving m_2^{ik} from p_{ik}. Hence, p_{ij} and p_{ik} can decide which one should follow the other by using the following theorem:

[**Theorem 2**] p_{ik} *follows* p_{ij} $(p_{ij} \Rightarrow p_{ik})$ if $m_2^{ij} \to m_3$ but $m_2^{ik} \not\to m_3$ for some messages m_3 and m_2 respectively received and sent by p_{ij} and p_{ik}. □

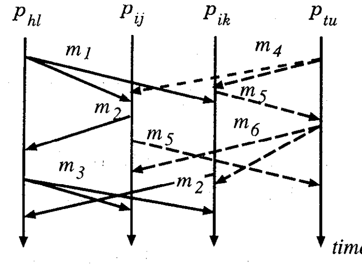


Fig. 6 Thrashing.

In Fig. 5, p_{ik} follows p_{ij} on receipt of m_3, since $m_2^{ij} \to m_3$ but $m_3 \parallel m_2^{ik}$. p_{ij} and p_{ik} are synchronized before receipt of m_3. For example, suppose m_1 and m_3 are requests and m_2 is the response to m_1. Here, suppose that it takes longer to compute request m_1 in p_{ik} than in p_{ij}. That is, p_{ik} is slower than p_{ij}. Without waiting for all the responses of m_1, p_{hl} can send another request m_3 to p_{ij} and p_{ik} on receipt of m_2^{ij}. By this method, the response time can be reduced even if the system includes slower computers.

4.3 Decision Rule

Next, we consider how each replica p_{ij} can decide in a distributed way whether p_{ij} follows or succeeds other replicas in the cluster c_i. Suppose that p_{ij} receives a message m carrying the vector clock $m.V$ [9]. From Theorem 2, p_{ik} knows that p_{ik} follows p_{ih} if $m.V_{ik} < m.V_{ih}$. Hence, p_{ij} decides how the replicas are followed on receipt of m according to the F rule.

[**Following (F) rule**] For every pair of replicas p_{ih} and p_{ik} in c_i,

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

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

p_{ik} *follows* p_{ih} *in* p_{ij} $(p_{ih} \Rightarrow_{ij} p_{ik})$ if p_{ij} decides that $p_{ih} \Rightarrow p_{ik}$. This means that p_{ij} knows that p_{ik} follows p_{ih}. Here, p_{ij} considers that p_{ih} is the *fastest* and *slowest* in c_i if $m.V_{ih}$ is maximum and minimum in $m.V$, respectively.

Suppose that p_{hl} in c_h sends m_1 and m_3 to p_{ij} and p_{ik} in c_i, and that p_{tu} in c_t sends m_4 and m_6, as shown in **Fig. 6**. p_{hl} sends m_3 on receipt of m_2^{ij} from p_{ij} before receiving m_2^{ik} from p_{ik}. Hence, on receipt of m_3, $p_{ij} \Rightarrow_{ij} p_{ik}$ and $p_{ij} \Rightarrow_{ik} p_{ik}$. p_{tu} sends m_6 on receipt of m_5^{ik} from p_{ik} before receiving m_5^{ij}. Hence, on receipt of m_6, p_{ij} and p_{ik} know that $p_{ik} \Rightarrow p_{ij}$. On receipt of m_3 and m_5, p_{ij} and p_{ik} are thrashed; that is, $p_{ij} \Rightarrow p_{ik}$ and $p_{ik} \Rightarrow p_{ij}$.

It is straightforward to show that the following theorem holds, since every replica is assumed to receive all messages in the same order.

[Theorem 3] On receipt of a message m, $p_{ij} \Rightarrow_{ih} p_{ik}$ iff $p_{ij} \Rightarrow_{il} p_{ik}$ for every pair of operational replicas p_{ih} and p_{il}. □

If p_{ih} knows that p_{ij} follows p_{ik} by the F rule on receipt of m, another p_{ih} receiving m is sure that $p_{ij} \Rightarrow_{ih} p_{ik}$.

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

If p_{ij} receives a message delayed on p_{ij}, p_{ij} knows that p_{ij} follows some replica.

In active replication, p_{hl} sends a request message m_1 to p_{ij} and p_{ik}. On receipt of the responses m_2^{ij} and m_2^{ik}, p_{hl} considers that the computation of m_1 is completed, and sends a request m_3 to p_{ij} and p_{ik}. In pseudo-active replication, p_{hl} does not wait for the responses from all the replicas. Since only stop-faults are assumed to occur, p_{hl} can send m_3 after p_{hl} receives one response from one replica.

Next, suppose that the replicas in c_i receive requests m_1 and m_2. If $m_1 \parallel m_2$, the replicas in c_i may receive m_1 and m_2 in different orders. If p_{ij} and p_{ik} receive write requests m_1 and m_2 in different orders, p_{ij} and p_{ik} become inconsistent. Hence, we assume that every p_{ij} in c_i takes messages in the total order.

4.4 Equivalent Sequences of Events

The slower replica p_{ik} has to catch up with the faster full replica p_{ij} in c_i. If every event stored in the receipt queue is required to occur in p_{ik}, p_{ik} cannot catch up with p_{ij}, since p_{ij} is faster than p_{ik}. Hence, we try to make p_{ik} catch up with p_{ij} by omitting events unnecessary and by changing the order of occurrence of the events.

First, we try to relax the AR1 condition.

[Definition] An event e is an *identity* event in p_{ij} iff $e(s_{ij}) = s_{ij}$ for every state s_{ij}. An event sequence S is *idempotent* in p_{ij} iff $S \circ S(s_{ij}) = S(s_{ij})$ for every state s_{ij} of p_{ij}. □

For example, an execution of an SQL *select* statement is an identity event.

[Definition] An event sequence S_1 is *absorbed* by S_2 iff $S_1 \circ S_2(s_{ij}) = S_2(s_{ij})$ for every state s_{ij} of p_{ij}. □

Suppose a write event w_1 of a value v_1 occurs before w_2 of v_2 in p_{ij}. Because v_1 is overwritten by v_2, w_2 absorbs w_1.

[Definition] Two event sequences S_1 and S_2 are *commutative* in p_{ij} iff $S_1 \circ S_2(s_{ij}) = S_2 \circ S_1(s_{ij})$ for every state s_{ij} of p_{ij}. □

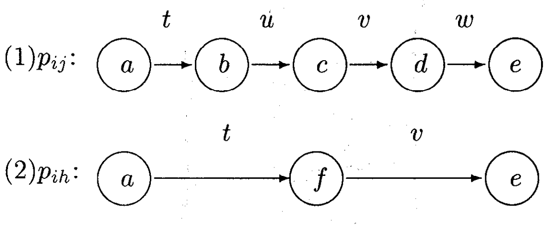


Fig. 7 Replicas.

Unless S_1 and S_2 are commutative, S_1 and S_2 *conflict*. For example, two *select* statements are *commutative* but *select* conflicts with *update*.

[Definition] Let S_1 be a sequence of events $e_{11} \circ \cdots \circ e_{1k_1}$, and let S_2 be $e_{21} \circ \cdots \circ e_{2k_2}$. S_1 is *equivalent* to S_2 in p_{ij} ($S_1 \equiv S_2$) iff $e_{11} \circ \cdots \circ e_{1k_1}(s_{ij}) = e_{21} \circ \cdots \circ e_{2k}(s_{ij})$ for every state s_{ij} of p_{ij}. □

In **Fig. 7**, p_{ij} includes p_{ih}. If u and w are identities, $c = b$ and $e = d$, $t \circ v \equiv t \circ u \circ v \circ w$. Furthermore, if t and v are commutative, $v \circ t \equiv t \circ v$. Here, p_{ih} can catch up with p_{ij} by omitting two identity events u and w.

[Omission rules] Let S_1 and S_2 be event sequences and let e be events.

(1) $S_1 \circ e \circ S_2 \equiv S_1 \circ S_2$ if e is an identity.

(2) $e \circ S_1 \circ e \equiv S_1 \circ e$ if e is idempotent and S_1 includes no event conflicting with e. □

That is, the slower replicas can omit the identity and idempotent events occurring in the faster replicas.

Next, we consider how to exchange events.

[Exchanging rule] Let S be an event sequence and let e_1 and e_2 be events. $e_1 \circ S \circ e_2 \equiv e_2 \circ S \circ e_1$ if e_1 and e_2 are commutative and S includes no event conflicting with e_1 and e_2. □

Suppose that w_1 and w_2 are events writing objects x and y, respectively, and that r is an event reading z. Here, $w_1 \circ r \circ w_2 \equiv w_2 \circ r \circ w_1$ if x, y, and z are pair-wise different.

4.5 Catching up

We discuss how the slower replicas catch up with the faster ones by using the omission and exchanging rules. In **Fig. 8**, suppose p_{ik} follows p_{ij}. In the computation of the request m_1, p_{ik} receives the requests m_3 and m_5 from p_{hl}. Here, $m_3 \to m_5$ and m_3 and m_5 are delayed. Since p_{ik} is sure that p_{ij} completes m_3 and p_{hl} receives the response m_4 of m_3 from p_{ij}, p_{ik} does not need to compute m_3. However, p_{ik} is not sure that p_{ij} completes m_5.

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

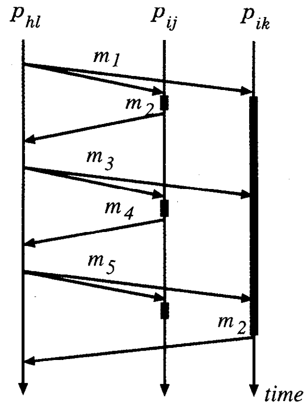


Fig. 8 Obsolete message.

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

(2) m is delayed on p_{ij}, and

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

Thus, p_{ij} can omit an *obsolete* request m, since m is already finished in the faster replica p_{ik}.

On receipt of a message m, p_{ij} stores m in the receipt queue RQ_{ij}. The messages in RQ_{ij} are sequenced in causal order. p_{ij} takes a top message m from RQ_{ij} and computes m; that is, the messages are causally delivered to p_{ij}. In Fig. 8, m_3 and m_5 are stored in RQ_{ij} and $m_3 \rightarrow m_5$ during the computation of m_1. After sending m_2, p_{ik} takes m_3 from RQ_{ij}.

p_{ij} executes the following procedure to check whether a top message in RQ_{ij} is obsolete.

[**Receipt**] A message m arrives at p_{ij}.

(1) m is stored in RQ_{ij} in the causal order by using the vector clock.

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

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

(3-1) If m' is an identity request, m' is marked as *omissible.*

(3-2) If m' is idempotent and the same kind of request m'' as m' precedes m and succeeds m' in RQ_{ij}, m' is marked as *omissible* if there is no message between m and m' in RQ_{ij} that conflicts with m'. □

p_{ij} takes the top message m from RQ_{ij}. If m is marked *omissible*, p_{ij} removes m and sends back the dummy response of m with no result.

In Fig. 8, suppose m_2 and m_4 are idempotent requests. m_3 arrives at p_{ik} during the computation of the request m_1. Here, m_3 is delayed on p_{ik} but is not obsolete. m_3 is enqueued into RQ_{ij}. Then, the request m_5 arrives at p_{ik}. m_5 is delayed on p_{ik}. The messages in RQ_{ij} are checked by the receipt procedure if they are obsolete. m_3 in RQ_{ij} is obsolete, since $m_3 \rightarrow m_5$ and m_5 is delayed. Therefore, m_3 is marked as *omissible.* After the completion of m_1, that is, after sending m_5, p_{ik} takes m_3 from RQ_{ij}. Since m_3 is marked as omissible, p_{ik} omits m_3 and sends back a dummy response for m_3 to p_{hl}. Then, p_{ik} starts to compute m_5, since m_5 is not obsolete while it is delayed.

4.6 Correctness

Let c_i be a cluster of replicas $p_{i1}, \ldots, p_{il_i}$ of a process p_i. Let d_i be another cluster of replicas $q_{i1}, \ldots, q_{ik_i}$ of p_i. Here, suppose that a process p_h receives multiple instances $m^{i1}, \ldots, m^{il_i}$ of a message m from $p_{i1} \ldots p_{il_i}$. p_h takes only one of them, which is delivered to p_h. The sequence of messages taken from c_i is an *output sequence* of c_i to p_h.

[**Definition**] c_i is *equivalent* to $d_i\,(c_i \equiv d_i)$ iff all output sequences of c_i and d_i are the same for every input sequence of messages. □

[**Theorem 4**] In pseudo-active replication, every cluster c_i is equivalent to some one-replica cluster of p_i.

[**Proof**] Every slower replica p_{ij} omits only obsolete messages. Every omissible message m is computed in one replica and the response is received by the sender of m. Lastly, let us consider a case in which the replicas are faulty. If the fastest replica is operational, it is straightforward. Suppose that the fastest replica, say p_{i1}, becomes faulty before sending the response to a request m. Since m is never omissible in any operational replica, some replica is certain to computes m. □

5. Evaluation

Pseudo-active replication supports the same level of reliability as active replication; that is, the cluster can provide service as long as at least one replica is operational, as shown in Theorem 4.

We evaluate the pseudo-active replication cluster $c_i = \{p_{i1}, \ldots, p_{il_i}\}$ by comparing it's response time and computation time with those of an active replication cluster. A process p_h sends requests to the replicas in c_i and receives responses from the replicas. Let δ_i be the propagation delay time between p_h and the replicas in c_i. There are f_i types of requests that p_{ij} can take. π_i^i, π_i^d, and π_i^o denote the probabilities of an identity request, an idempotent, request, and some other request in c_i, respectively. Here, $\pi_i^i + \pi_i^d + \pi_i^o = 1$. We assume that every pair of operations conflict if neither of them is an identity. We assume that p_{i1} is the fastest and p_{il_i} is the slowest in c_i. Suppose that p_h issues w requests to c_i. After sending a request r, p_h sends the subsequent request to c_i on receipt of the response to r. In this paper, we assume that it takes τ_{ij}^i, τ_{ij}^d, and τ_{ij}^o time units to compute each identity event, each idempotent event, and each other event, respectively, in p_{ij}. Let τ_{ij} be the average computation time for each event in p_{ij}; that is, let $\tau_{ij} = \pi_i^i \cdot \tau_{ij}^i + \pi_i^d \cdot \tau_{ij}^d + \pi_i^o \cdot \tau_{ij}^o$. We assume that $\tau_{ij}^i/\tau_{i1}^i = \tau_{ij}^d/\tau_{i1}^d = \tau_{ij}^o/\tau_{i1}^o = \tau_{ij}/\tau_{i1} (\leq 1)$.

Compute w requests in p_{ij} is expected to take $w \cdot \tau_{ij}$ time units. The expected processing time R_P in the pseudo-active replication, is $w \cdot (\tau_{i1} + 2 \cdot \delta_i)$ time units, while the expected processing time R_A in active replication is $w \cdot (\tau_{il_i} + 2 \cdot \delta_i)$. It is clear that $R_P \leq R_A$, since $\tau_{i1} \leq \tau_{il_i}$.

If no event is omitted in any p_{ij}, w requests are computed. Hence, the total computation time of c_i in active replication is $\Sigma_{j=1,\ldots,l_i} w \cdot \tau_{ij}$. In pseudo-active replication, some *obsolete* events are omitted.

Let us consider the total computation times of the fastest replica, p_{i1}, and the slowest, p_{il_i}. The total time of p_{ij} is defined as the time from when p_{ij} receives the first request until p_{ij} sends a response to the last request. Let T_A be the total time needed for p_{il_i} to compute the requests in active replication, that is, when no requests are omitted. Let T_P be the total computation time of p_{il_i} in pseudo-active replication. Here, suppose that $\pi_i^i = 0.5$, $\pi_i^d = 0.3$, and $\pi_i^o = 0.2$. That is, 50% of requests are identity ones, and 30% and 20% are idempotent and other requests, respectively. p_h sends $w(= 100)$ requests by randomly selecting operations in $f_i(= 10)$ ones. Let δ be the ratio of the propagation delay δ_i among p_h and the replicas in c_i to the processing time τ_{i1}^i of the identity request in p_{i1}, δ_i/τ_{i1}^i. The total processing times T_P and T_A are obtained by the simulation. The number w_P of operations computed in p_{il_i} in pseudo-active replication is also obtained. **Figure 9** shows T_A/R_A and T_P/R_A with $\delta = 0.1$, 1, 3, 10 for τ_{i1}/τ_{il_i}. The dotted

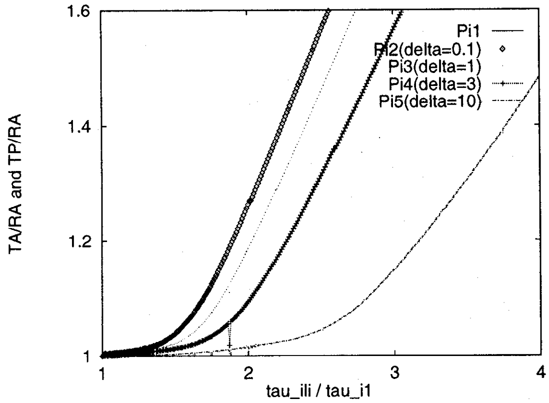


Fig. 9 Ratio of total computation times in active and pseudo-active replication.

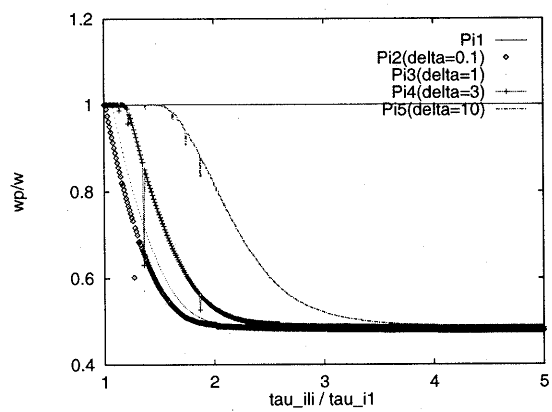


Fig. 10 Ratio of omissible events.

lines show T_P/R_A. **Figure 10** shows w_P/w with δ for τ_{i1}/τ_{il_i}.

Figures 9 and 10, show that pseudo-active replication can reduce the total processing time and the number of operations computed in the slower replicas. Furthermore, the greater the distance between p_h and the replicas, the more efficient the pseudo-active replication. In Fig. 10, let us consider w_P/w for $\delta = 3$. If it takes 10 msec. to compute identity request such as *read* in p_{i1}, the propagation delay is 30 msec. If p_{il_i} is five times slower than p_{i1}, the proportion of requests computed in p_{il_i} is the same, namely, 50%. This means that every identity request is omitted in p_{il_i}, since every request issued to p_{il_i} is queued in RQ_{ij}.

6. Concluding Remarks

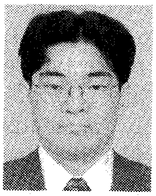
This paper has discussed pseudo-active replication in heterogeneous clusters. In a heterogeneous cluster, the computation of the cluster can be completed if the fastest replica is completed while the slower replicas are still being computed. We have presented a vector-clock-based method by which each replica can decide how to follow others. The slower replicas can catch up with the faster ones by omitting identity and idempotent events and changing commutative events. We have shown that pseudo-active replication implies a shorter response time and total computation time than active replication, while providing the same level of reliability.

Acknowledgments We would like to thank Prof. Lewis A. Davis of Tokyo Denki University for his kind and useful help and encouragement in writing this paper. We also would like to thank Mr. Iwao Shimojo, a graduate student of Tokyo Denki University for his help in revising this paper.

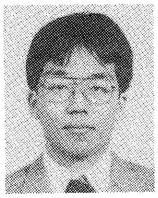
References

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. IEEE ICNP*, pp.365–374 (1995).
2) Amir, Y., Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A. and Ciarfella, P.: The Totem Single-Ring Ordering and Membership Protocol, *Trans. ACM Computer Systems*, Vol.13, No.4, pp.311–342 (1995).
3) Birman, P.K. and Renesse, V.R.: *Reliable Distributed Computing with the Isis Toolkit*, IEEE CS Press (1994).
4) Budhiraja, N., Marzullo, K., Schneider, B.F. and Toueg, S.: The Primary-Backup Approach, *Distributed Computing Systems*, pp.199–221, ACM Press (1994).
5) Ezhilchelvan, P.D., Macedo, R.A. and Shrivastava, S.K.: Newtop: A Fault-Tolerant Group Communication Protocol, *Proc. IEEE ICDCS-15*, pp.296–307 (1995).
6) Fischer, J.M., Nancy, A.L. and Michael, S.P.: Impossibility of Distributed Consensus with One Faulty Process, *ACM TOCS*, Vol.32, No.2, pp.374–382 (1985).
7) Florin, G. and Toinard, C.: A New Way to Design Causally and Totally Ordered Multicast Protocols, *ACM Operating Systems Review*, Vol.26, No.4, pp.77–83 (1992).
8) Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
9) Lamport, L., Shostak, R. and Pease, M.: The Byzantine Generals Problem, *ACM Trans. Programming Languages and Systems*, Vol.4, No.3, pp.382–401 (1982).
10) Mattern, F.: Virtual Time and Global States of Distributed Systems, *Parallel and Distributed Algorithms*, Cosnard, M. and Quinton, P. (Eds.), pp.215–226, North-Holland (1989).
11) Melliar-Smith, P.M., Moser, L.E. and Agrawala, V.: Broadcast Protocols for Distributed Systems, *IEEE Trans. Parallel and Distributed Systems*, Vol.1, No.1, pp.17–25 (1990).
12) Nakamura, A. and Takizawa, M.: Causally Ordering Broadcast Protocol, *Proc. IEEE ICDCS-14*, pp.48–55 (1994).
13) Schneider, B.F.: Byzantine Generals in Action: Implementing Fail-Stop Processors, *ACM Trans. Computing Systems*, Vol.2, No.2, pp.145–154 (1993).
14) Schneider, B.F.: Replication Management Using the State-Machine Approach, *Distributed Computing Systems*, pp.169–197, ACM Press (1993).
15) Shima, K., Higaki, H. and Takizawa, M.: Fault-Tolerant Causal Delivery in Group Communication, *Proc. IEEE ICPADS '96*, pp.302–309 (1996).
16) Tachikawa, T. and Takizawa, M.: Selective Total Ordering Broadcast Protocol, *Proc. IEEE ICNP-94*, pp.212–219 (1994).
17) Tachikawa, T. and Takizawa, M.: Distributed Protocol for Selective Intra-group Communication, *Proc. IEEE ICNP-95*, pp.234–241 (1995).
18) Thomas, L.C. and Mukesh, S.: The Delta-4 Distributed Fault-Tolerant Architecture, *Distributed Computing Systems*, pp.223–247, IEEE CS Press (1990).

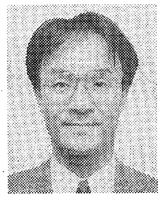
(Received February 27, 1997)
(Accepted September 10, 1997)



Kenji Shima was born in Japan on Jan. 13, 1972. He received his B.E. and M.E. degrees from the Department of Computers and Systems Enginering, Tokyo Denki University in 1995 and 1997, respectively. He is now working for DEC Japan. His research interest includes fault-tolerant distributed systems, computer networks, and communication protocols.



Hiroaki Higaki was born in Tokyo, Japan, on April 6, 1967. He received the B.E. degree from the Department of Mathematical Engineering and Information Physics, the University of Tokyo in 1990. From 1990 to 1996, he was in NTT Software Laboratories. Since 1996, he is in the Faculty of Science and Engineering, Tokyo Denki University. He received his D.E. degree in Computer Science from Tokyo Denki University. His research interest includes distributed algorithms, distributed operating systems and computer network protocols. He received IPSJ Convention Award in 1995. He is a member of IPSJ, ACM and IEICE.



Makoto Takizawa was born in 1950. He received his B.E. and M.E. degrees in Applied Physics from Tohoku University, Japan, in 1973 and 1975, respectively. He received his D.E. in Computer Science from Tohoku University in 1983. From 1975 to 1986, he worked for Japan Information Processing Development Center (JIPDEC) supported by MITI. In 1986, he joins Tokyo Denki University. He is currently a full professor of the Department of Computers and Systems Engineering, Tokyo Denki University. From 1989 to 1990, he was a visiting professor of the GMD-IPSI, Germany. He is also a regular visiting professor of Keele University, England since 1990. He was a vice-chair of IEEE ICDCS, 1994, and is a program co-chair of IEEE ICDCS, 1998. He serves on the program committees of many international conferences. He chairs IPSJ SIGDPS since 1997. His research interest includes communication protocols, group communication, distributed database systems, transaction management, and groupware. He is a member of IPSJ, IEEE, ACM, and IEICE.