Asynchronous Recovery in Distributed Systems

Hiroaki Higaki, Kenji Shima, Takayuki Tachikawa, and Makoto Takizawa
Department of Computers and Systems Engineering
Tokyo Denki University
{hig,sima,tachi,taki}@takilab.k.dendai.ac.jp

This paper proposes a novel algorithm for taking checkpoints and rolling back the processes for recovery in asynchronous distributed systems. The algorithm has the following properties: (1) Multiple processes can simultaneously initiate the checkpointing. (2) No additional message is transmitted for taking checkpoints. (3) A set of local checkpoints taken by multiple processes denotes a consistent global state. (4) Multiple processes can initiate simultaneously the rollback recovery. (5) The minimum number of processes are rolled back. (6) Each process is rolled back asynchronously. The number of messages for rolling back the processes is $O(l)$ where l is the number of channels. Therefore, the system is kept highly available by the algorithm presented in this paper.

1 Introduction

Information systems become distributed and are getting larger by including various kinds of component systems and interconnecting with various systems, e.g., by the Internet, in the world. The distributed systems are designed and developed by using widely available products including free softwares rather than specially designed hardwares and softwares. Distributed applications are realized by cooperation of multiple processes executed in multiple computers. These components are not always guaranteed to support enough reliability and availability for the applications. It is critical to discuss how to make and keep the systems so reliable and available that even fault-tolerant applications could be computed in the systems.

Checkpointing and rollback recovery are well-known time-redundant techniques to allow processes to make progress even if some processes fail. The processes take checkpoints by saving their state information in the local logs while executing the applications. If the processes fail in the system, the processes are rolled back to the checkpoints by restoring the saved state information and then the applications are restarted to be executed from the checkpoints. In this paper, we assume that every failure is *transient*, e.g., hardware errors, process crashes, transaction aborts, and communication deadlocks. The failures are unlikely to recur after the processes are restarted.

We have to consider how to keep the system consistent when taking checkpoints and rolling back the processes. The consistency of the global state is formalized by Chandy and Lamport [3]. Many papers [2, 3, 5–7, 9–14] have discussed so far how to take the consistent checkpoints among multiple processes. In addition, we have to discuss how to roll back the processes for recovery if some process fails. If each process is rolled back independently of the other processes, the system state may be inconsistent. One idea [7] is to synchronize all the processes to be rolled back by using the protocols similar to the two-phase commitment protocol [1]. However, it takes time to exchange messages among the processes. In this paper, we would like to discuss a new method where the processes are allowed to be asynchronously rolled back and restarted.

In section 2, the conventional checkpointing and rollback recovery methods are reviewed. In section 3, we show a basic algorithm for taking checkpoints and rolling back processes. In section 4, we make clear how livelocks occur in the rollback recovery. A livelock-free algorithm is proposed in section 5. The evaluation of the algorithm is presented in section 6.

2 Checkpoint and Rollback

2.1 Consistent state

A distributed system is composed of multiple processes interconnected by channels, i.e., $\langle V, L \rangle$ where $V = \{p_1, \ldots, p_n\}$ is a set of processes and $L \subseteq V^2$ is a set of channels. $\langle p_i, p_j \rangle \in L$ indicates a channel from p_i to p_j. In the distributed system, three kinds of events occur: message-sending, message-receiving and local events. A state of the process is changed when an event occurs. An event e_s *happens before* e_t ($e_s \rightarrow e_t$) iff one of the following conditions is satisfied [8]:

- e_s occurs before e_t in the same process.
- e_s is a message-sending event for a message m and e_t is a message-receiving event for m.
- There is an event e_u such that e_s happens before e_u and e_u happens before e_t.

A local state of p_i is determined by the initial state and the sequence of events occurring in p_i. Messages are transmitted from p_i to p_j via a channel $\langle p_i, p_j \rangle$. Here, $\langle p_i, p_j \rangle$ is named a *channel of* p_i. If there is a channel $\langle p_i, p_j \rangle$, p_j is referred to as a *neighbor* process of p_i. A state of $\langle p_i, p_j \rangle$ is defined as a sequence of messages sent by p_i but

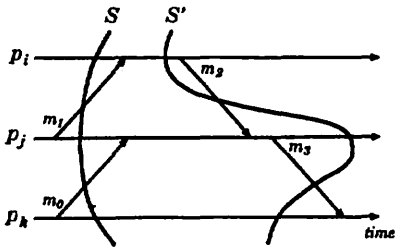

Figure 1: Consistent global state

not yet received by p_j, i.e., messages in transmission from p_i to p_j. A global state of $\langle V, L \rangle$ is a collection of states of the processes in V and of the channels in L.

A local checkpoint c^i of p_i is taken by recording the state information of p_i in the local log. A global checkpoint is a set of local checkpoints taken by all the processes in V, i.e., $\{c^1, \ldots, c^n\}$. If the processes take the local checkpoints and are rolled back to the local checkpoints independently of the other processes, there may exist two kinds of inconsistent messages: *lost messages* and *orphan messages*. Here, suppose that p_i sends a message m to p_j. m is lost iff a message-sending event of m occurs before p_i takes a checkpoint and a message-receiving event of m occurs after p_j takes a checkpoint. m is an orphan iff a message-sending event of m occurs after p_i takes a checkpoint and a message-receiving event of m occurs before p_j takes a checkpoint. By recording the received messages in the log after p_j takes a local checkpoint, lost messages can be received by taking them out of the log after p_j is rolled back. However, if there exist orphan messages, the system becomes inconsistent after the rollback recovery. Even if the processes record messages in the log each time message-sending events occur, the nondeterministic processes may send messages different from the messages in the log after the rollback recovery. Therefore, the global state of the system can be defined to be *consistent* iff there is no orphan message. Figure 1 shows three processes p_i, p_j and p_k. A global state S is consistent because there is no orphan message while m_0 is lost. However, another global state S' is inconsistent because m_2 is an orphan message.

2.2 Checkpointing

There are two approaches to taking checkpoints among multiple processes: *asynchronous* and *synchronous* checkpointing. In the asynchronous checkpointing [2, 5, 13, 14], the processes take the checkpoints without cooperating with the other processes. This approach implies less overhead because it requires no communication among the processes. However, *domino effects* may occur [11]. On the other hand, in the synchronous checkpointing [3, 6, 7, 11, 12], multiple processes are coordinated to take the checkpoints by using the protocols similar to the two-phase commitment protocol [1]. Here, the overhead for taking checkpoints is higher than the asynchronous checkpointing while no domino effect occurs. This paper discusses the synchronous checkpointing.

In the conventional checkpointing [3, 5, 7, 11–14], if some process takes a local checkpoint, all the processes in the system are required to take local checkpoints. Moreover, in the conventional synchronous checkpointing, additional messages to take the checkpoints are transmitted and the processes are suspended during the checkpointing procedure. However, all the processes are not always needed to take local checkpoints to keep the system consistent after the rollback recovery. We would like to discuss which processes have to take the local checkpoints. We define a *semi-consistent* global state.

Definition (semi-consistent) For a distributed system $\langle V, L \rangle$, let P be a subset of V. A global state S is *semi-consistent* for P iff there is no orphan message for every channel of each process in P. □

The system is kept consistent after the rollback recovery iff a global state S is semi-consistent for a subset P of processes and only and all the processes in P are rolled back. Here, suppose that a process p_i takes a checkpoint c^i. If p_i sends a message m to p_j after taking c^i, m is referred to as a *checkpoint message* of c^i to p_j. In our checkpointing algorithm, m contains the information on whether p_i takes c^i before sending m or not. In order that a global checkpoint denotes a semi-consistent global state, p_j takes a checkpoint c^j if the following condition is satisfied:

Checkpoint If a message-receiving event e for a checkpoint message m occurs in a process p_j and p_j does not take a local checkpoint c^j, p_j takes c^j just before e. □

This means that p_j can newly take a local checkpoint after p_j is rolled back but cannot if p_j had taken a local checkpoint. By using this checkpointing algorithm, the minimum number of processes take checkpoints and no additional message is transmitted to take the checkpoints.

2.3 Rollback

In the conventional rollback algorithms [3, 5, 7, 11–14], the processes have to be synchronised to be restarted by the following procedure:

1) Request messages are transmitted from the *coordinator process* to all the other processes called *cohort processes*.
2) Each cohort is rolled back to the checkpoint and a reply message is transmitted from each cohort to the coordinator.
3) The coordinator transmits restart messages to all the cohorts. Each cohort is restarted from the checkpoint.

One of the disadvantages of the algorithms is that all the processes are suspended and additional

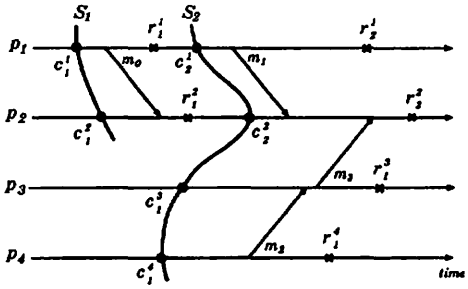

Figure 2: Semi-consistent global state.

messages are transmitted to synchronize all the processes. The larger the system becomes, the longer the processes are suspended. Thus, the system becomes less available. In order to keep the system highly available with the rollback recovery, we would like to discuss a method where the processes are asynchronously restarted from the checkpoints.

3 Basic Algorithm

In this section, we would like to show a basic algorithm for taking checkpoints and rolling back processes by using an example shown in Figure 2. The system consists of four processes $\{p_1, p_2, p_3, p_4\}$ fully connected by bidirectional channels. Each process p_i takes a checkpoint c_s^i. If some process fails, p_i is rolled back to c_s^i and restarted from c_s^i. c_s^i represents the sth checkpoint taken by p_i. r_s^i represents a possible rollback event where p_i is rolled back to c_s^i. c_s^i is *active* if p_i takes c_s^i and r_s^i does not occur. For example, c_2^1 is active when p_1 sends m_1. After r_2^1 occurs and p_1 is restarted from c_2^1, c_2^1 is not active.

If p_i fails, it is not sufficient to restart p_i from an active checkpoint c_s^i because there may be orphan messages. For example, if r_1^1 occurs in p_1 and p_1 is restarted from c_1^1, m_0 is an orphan message. In order to obtain a semi-consistent global state after the rollback recovery, p_2 has to be restarted from c_1^2. That is, if p_i is restarted from c_s^i, an event e_t^j in p_j where $c_s^i \rightarrow e_t^j$ has to be canceled by a rollback recovery in p_j. Even if p_1 and p_2 are rolled back and restarted from c_1^1 and c_1^2, respectively, p_3 and p_4 are not required to be rolled back. That is, if no event e_t^j where $c_s^i \rightarrow e_t^j$ occurs in p_j, p_j is not required to be rolled back. Thus, if p_i is rolled back, we have to identify which processes are required to be rolled back in the system.

First, we would like to define the precedence relation among the active checkpoints.

Definition (checkpoint precedence) Let c_s^i and c_t^j be active checkpoints taken by processes p_i and p_j, respectively. Let e^i and e^j be events such that $c_s^i \rightarrow e^i$ and $c_t^j \rightarrow e^j$. c_s^i *precedes* c_t^j ($c_s^i \Rightarrow c_t^j$) if $e^i \rightarrow e^j$. □

In Figure 2, when p_2 receives m_3, $c_2^1 \Rightarrow c_2^2$, $c_1^4 \Rightarrow c_1^3$, $c_1^3 \Rightarrow c_2^2$ and $c_1^4 \Rightarrow c_2^2$. Next, we define a *rollback domain* $D(p_i)$ for determining a set of processes to be rolled back if p_i is rolled back.

Definition (rollback domain) A *rollback domain* $D(p_i)$ of a process p_i is defined to be a following subset of processes in the system:

1) $p_i \in D(p_i)$ if there is an active checkpoint c_s^i in p_i. Otherwise, $D(p_i) = \emptyset$.
2) $p_j \in D(p_i)$ if c_t^j is active in p_j and $c_t^j \Rightarrow c_u^k$ or $c_u^k \Rightarrow c_t^j$ where c_u^k is active in $p_k \in D(p_i)$.
3) Only the processes satisfying 1) and 2) are included in $D(p_i)$. □

In Figure 2, m_0 is transmitted from p_1 to p_2, p_2 takes c_1^2 before accepting m_0. Here, $D(p_1) = D(p_2) = \{p_1, p_2\}$.

For each $p_j \in D(p_i)$, it is clear that $p_i \in D(p_j)$ and $D(p_i) = D(p_j)$. $D(p_i) \cap D(p_j) = \emptyset$ if $p_j \notin D(p_i)$. A set $C = D(p_i)$ of processes is referred to as a *rollback class*. According to the definition, two different rollback classes are disjoint. Each time a message-sending or message-receiving event occurs in a process p_i, a rollback class C^i of p_i is changed as follows.

Change of rollback class If p_i in a rollback class C_s^i is changed to be in C_{s+1}^i on sending a message m and p_j in C_t^j is changed to be in C_{t+1}^j on receiving m from p_i, $C_{s+1}^i = C_{t+1}^j = C_s^i \cup C_t^j$. □

In Figure 2, before m_3 is transmitted, $D(p_1) = D(p_2) = C = \{p_1, p_2\}$ and $D(p_3) = D(p_4) = C' = \{p_3, p_4\}$. When m_3 is transmitted from p_3 to p_2, C and C' are changed to $C'' = C \cup C' = \{p_1, p_2, p_3, p_4\}$.

For every state S of a system (V, L) and a subset $V' \subseteq V$, let $S_{V'}$ denote a projection of S into V', i.e., a collection of the local states of the processes in V' and the states of the channels of the processes in V'. Let $A_{V'}$ be a set of the local states denoted by the active checkpoints taken by the processes in V'. If some process $p_i \in V'$ has no active checkpoint, the local state of p_i in S is included in $A_{V'}$.

Theorem 1 For every rollback class C at every system state S, $S_{V-V'} \cup A_{V'}$ is semi-consistent for V' iff 1) $C \subseteq V'$ and 2) $V' \cap C' = \emptyset$ or C' for every rollback class $C' \neq C$. □

That is, if a process p_i in a rollback class C fails, the system state is semi-consistent for C if all the processes in C are rolled back to the active checkpoints. This also means that C is the minimum set of processes to be rolled back for keeping the system semi-consistent after the recovery.

Suppose that p_1, p_2, p_3, and p_4 in Figure 2 are at r_2^1, r_2^2, r_1^3 and r_1^4, respectively. p_1 does not know that p_3 and p_4 are in $D(p_1)$ while knowing that p_2 is in $D(p_1)$. Thus, p_i does not have the complete information on which processes are included in C.

Definition (rollback view) A process p_j is in-

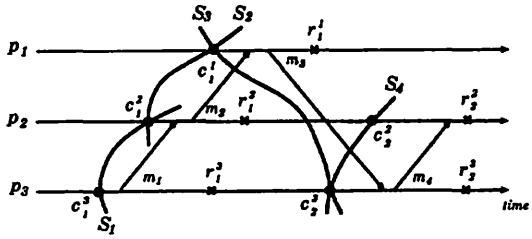

Figure 3: Livelock in rollback recovery.

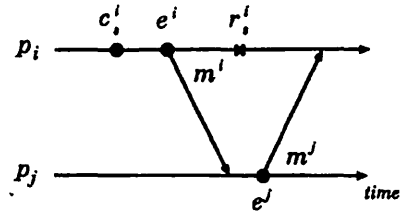

Figure 4: Cause of livelock.

cluded in a p_i's *rollback view* $W(p_i)$ of $D(p_i)$ if p_i knows $p_j \in D(p_i)$. □

Thus, $W(p_i) \subseteq D(p_i)$.

Based on the view $W(p_i)$ of p_i, p_i can be rolled back and restarted from the active checkpoint c_s^i by using the message diffusion protocol [4]:

1) If p_i fails, p_i sends a rollback request m_r to all the processes in $W(p_i)$.
2) On receipt of m_r, p_i also sends m_r to all the processes in $W(p_i)$.
3) If p_i receives m_r from all the processes in $W(p_i)$, p_i is rolled back and restarted from c_s^i.

4 Livelock in Rollback Recovery

Let us consider the following scenario shown in Figure 3.

1) p_3 takes c_1^3 and sends m_1 to p_2. m_1 is a checkpoint message. Here, c_1^3 is active and $D(p_3) = \{p_2, p_3\}$.
2) p_2 receives m_1 and takes c_1^2 where the local state of p_2 just before receiving m_1 is recorded in the log. Here, c_1^2 is active and $D(p_2) = D(p_3) = \{p_2, p_3\}$. A set of local checkpoints $S_1 = \{c_1^2, c_1^3\}$ is semi-consistent for $C_1 = \{p_2, p_3\}$ because there is no orphan message in the channels of p_2 and p_3.
3) p_2 sends m_2 to p_1. p_1 receives m_2 and takes c_1^1. Here, c_1^1 is active. $D(p_1) = D(p_2) = D(p_3) = \{p_1, p_2, p_3\}$. p_1 does not know that p_3 is in the same rollback class, i.e., $W(p_1) = \{p_1, p_2\} \subset D(p_1)$.
4) p_3 fails and is rolled back from r_1^3 to c_1^3. Now, since c_1^3 becomes inactive, p_3 is not in the rollback class. Here, $D(p_1) = D(p_2) = \{p_1, p_2\}$ and $D(p_3) = \emptyset$. $S_2 = \{c_1^1, c_1^2\}$ is semi-consistent for $C_2 = \{p_1, p_2\}$.
5) p_1 sends m_3 to p_3. p_3 receives m_3 and takes a new checkpoint c_2^3. Here, c_2^3 is active. $D(p_1) = D(p_2) = D(p_3) = \{p_1, p_2, p_3\}$. $W(p_1) = D(p_1) = \{p_1, p_2, p_3\}$, $W(p_2) = \{p_1, p_2\}$ and $W(p_3) = \{p_1, p_3\}$.
6) p_2 is rolled back from r_1^2 to c_1^2 because p_3 is rolled back at step 4). Here, $D(p_1) = D(p_3) = \{p_1, p_3\}$ and $D(p_2) = \emptyset$. $S_3 = \{c_1^1, c_2^3\}$ is semi-consistent for $C_3 = \{p_1, p_3\}$.
7) p_3 sends m_4 to p_2. p_2 receives m_4 and takes c_2^2. $D(p_1) = D(p_2) = D(p_3) = \{p_1, p_2, p_3\}$, $W(p_3) = D(p_3) = \{p_1, p_2, p_3\}$, $W(p_1) = \{p_1, p_2\}$ and $W(p_2) = \{p_2, p_3\}$.
8) p_1 is rolled back from r_1^1 to c_1^1 because p_2 is rolled back at step 6). Here, $D(p_1) = \emptyset$ and $D(p_2) = D(p_3) = \{p_2, p_3\}$. $S_4 = \{c_2^2, c_2^3\}$ is semi-consistent for $C_4 = \{p_2, p_3\}$.

Thus, the rollback class C_s does not become empty and the rollback recovery may be continued forever, i.e. livelock occurs.

One way to resolve the livelock is to synchronize all the processes to be restarted, e.g., by using two-phase commitment protocol. However, it takes time and the control messages have to be transmitted to synchronize the processes. In this paper, we would like to discuss a new method where the processes can be restarted asynchronously. Suppose that a process p_i sends a checkpoint message m^i at an event e^i after taking a local checkpoint c_s^i and p_j sends m^j at e^j after receiving m^i as shown in Figure 4. Here suppose that a rollback r_s^i occurs in p_i. Since $e^i \rightarrow e^j$ and $c_s^i \rightarrow e^i \rightarrow r_s^i$, p_i cannot receive m^j after p_i is rolled back from r_s^i, i.e. e^i is canceled. This is because p_i is sure that p_j would be rolled back owing to the rollback of p_i and the message-receiving event for m^j has to be canceled. If p_i receives m^j, p_i is required to be rolled back again due to the rollback recovery in p_j. Thus, the livelock may occur. Hence, in Figure 3, p_3 is not allowed to receive m_3, i.e., discards m_3 to realize the livelock-free rollback recovery.

In order to identify the messages to be discarded, we introduce *generation* concept as follows:

Definition (generation of a process) A generation $g(p_i)$ of a process p_i is given as follows:

1) $g(p_i) = 0$ before r_1^i occurs in p_i.
2) $g(p_i) = s$ between r_s^i and r_{s+1}^i. □

Definition (generation of an event) A generation $g(e)$ of an event e in a process p_i is given when e occurs as follows:

1) $g(e) = s$ if p_i has an active checkpoint c_s^i.
2) $g(e) = \perp$ (unknown) if p_i has no active checkpoint. □

Each time p_i is rolled back, the generation of p_i is incremented. $g(e)$ of an event e occurring in p_i is set to $g(p_i)$ if the checkpoint taken most recently by p_i is active. If an event e^i precedes a message-

sending event of a message m^j in p_j, $g(e^i)$ is piggied back with m^j. On receipt of m^j, p_i knows of $g(e^i)$. If $g(e^i) < g(p_i)$, p_i knows that the message-sending event of m^j would be canceled by the rollback recovery in p_j.

In Figure 4, $g(p_i) = s-1$ when e^i occurs. Thus, $g(e^i) = s-1$ is piggied back with m^i and m^j. When r_s^i occurs in p_i, $g(p_i)$ is changed to s. When p_i receives m_j, p_i obtains $g(e_i) = s-1$ piggied back with m_j while $g(p_i) = s$. Thus, p_i discards m^j because p_i detects $g(e^i) < g(p_i)$.

In the succeeding section, we would like to present the detailed algorithm using vectors of the generations for preventing the livelock.

5 Algorithms

5.1 Assumptions and definitions

A distributed system $S = \langle V, L \rangle$ consists of a finite set $V = \{p_1, \ldots, p_n\}$ of processes and a set $L \subseteq V^2$ of channels. We make the following assumptions on S.

- A1 Every channel in L is bidirectional.
- A2 Every channel in L is reliable.
- A3 In each channel $\langle p_i, p_j \rangle \in L$, messages are transmitted in the first-in-first-out order from p_i to p_j.
- A4 S is asynchronous, i.e. a maximum message transmission delay is unbounded.

The following events occur in p_i:

- Message-receiving: p_i takes out a message m from a channel $\langle p_j, p_i \rangle$ and accepts m from p_j.
- Message-sending: p_i puts a message m to a channel $\langle p_i, p_j \rangle$ to send m to p_j.
- Checkpoint: p_i records the local state information in the log. The sth checkpoint taken by p_i is denoted by c_s^i. Initially, every p_i takes c_0^i.
- Rollback: If p_i fails at r between c_s^i and c_{s+1}^i, p_i is restarted from c_s^i. The rollback to occur at r is denoted by r_s^i. r_s^i exists only if p_i is rolled back to c_s^i.

c_s^i is *active* since c_s^i is taken until r_s^i occurs. c_s^i is discarded if c_{s+1}^i is taken.

A message m contains the data $m.data$ and the following information in the header:

- A flag $m.flag$.
- A vector clock $m.clock = \langle m.cl_1, \ldots, m.cl_n \rangle$.

A process p_i manipulates the following variables. Here, let N^i be a set of neighbor processes of p_i.

- A vector clock $c.CL^i = \langle c.cl_1^i, \ldots, c.cl_n^i \rangle$ named a *checkpoint clock*: Each $c.cl_j^i$ shows the generation of the active checkpoint in p_j that p_i knows. That is, when an event e^i occurs in p_i, $c^j_{c.cl_j^i} \rightarrow e^i$ if $c.cl_j^i \neq \perp$. $c.cl_i^i$ is incremented by one each time p_i takes a local checkpoint. Initially, $c.cl_i^i = 0$ and $c.cl_j^i = \perp$ for $j \neq i$.
- A vector clock $r.CL^i = \langle r.cl_1^i, \ldots, r.cl_n^i \rangle$ named a *rollback clock*: Each $r.cl_j^i$ shows the generation of the rollback most recently occurring in p_j that p_i knows. That is, on receipt of a message m, if p_i has no active checkpoint and $m.cl_j \leq r.cl_j^i$, p_i detects that m is canceled by the rollback recovery. $r.cl_j^i$ is updated to be $c.cl_j^i$ each time a rollback occurs in p_i. Initially, $r.cl_i^i = 0$ and $r.cl_j^i = \perp$ for $j \neq i$.
- A flag $flag^i$: If c_s^i is active, $flag^i = True$. Otherwise, $flag^i = False$.
- A set $W^i \subseteq N^i$ of the neighbor processes of p_i included in the rollback domain $D(p_i)$ of, i.e., $W(p_i)$: Initially, $W^i = \emptyset$.
- A sequence M^i of messages received after taking the active checkpoint in p_i: Initially, $M^i = \emptyset$.

For a pair of vector clocks $v^i = \langle v_1^i, \ldots, v_n^i \rangle$ and $v^j = \langle v_1^j, \ldots, v_n^j \rangle$, $max(v^i, v^j)$ is defined to be $\langle v_1, \ldots, v_n \rangle$ where each $v_k = v_k^j$ if $v_k^i = \perp$, $v_k = v_k^i$ if $v_k^j = \perp$, $v_k = max(v_k^i, v_k^j)$ otherwise.

5.2 Checkpointing

The checkpointing algorithm has to satisfy the following requirements:

- R1 A process which has an active checkpoint is included in exactly one rollback class.
- R2 The global state is semi-consistent for a set of checkpoints taken by the processes in each rollback class. That is, there is no orphan message.
- R3 Every process p_i has the rollback view W^i on which neighbor processes are included in the same rollback class.
- R4 There is no lost message.

A process p_i takes a local checkpoint c_s^i if one of the following conditions is satisfied:

- C1 If p_i decides to take a local checkpoint by such a trigger as user request or timeout, p_i takes c_s^i.
- C2 If a message-receiving event e occurs in p_i where p_i has no active checkpoint and p_i receives a checkpoint message m transmitted from a neighbor process p_j of p_i, p_i takes c_s^i just before e.

C1 means that the checkpointing can be initiated independently by multiple processes. By taking the checkpoints as presented in C2, there is no orphan message in the channel $\langle p_i, p_j \rangle$. Thus, R2 is satisfied.

Suppose that p_i and p_j have active checkpoints c_s^i and c_t^j, respectively. If p_i sends a message m to p_j, m is a checkpoint message, i.e., $m.flag = True$ because $flag^i = True$. When p_j receives m, p_j does not take another checkpoint even if $m.flag = True$. As presented in the preceding section, if p_i in a rollback class C_u^i sends a checkpoint message to p_j in a rollback class C_v^j where $C_u^i \cap C_v^j = \emptyset$,

C_u^i and C_v^j are changed to C_{u+1}^i and C_{v+1}^j, respectively, where $C_{u+1}^i = C_{v+1}^j = C_u^i \cup C_v^j$. Thus, R1 is satisfied.

The system has to prevent from the livelock caused by the rollback recovery as discussed in the previous section. Here, we would like to present the checkpointing algorithm for the livelock-free rollback recovery. Suppose that p_i and p_j are in the same rollback class C where p_i and p_j have active checkpoints c_s^i and c_t^j, respectively, and p_i receives a checkpoint message m from p_j. Let $e^i(m)$ denote the message-receiving event of m in p_i and $e^j(m)$ denote the message-sending event of m in p_j. If r_s^i occurs before $e^i(m)$, p_i discards m because p_i knows that $e_j(m)$ is canceled eventually. In order to discard m, p_i uses $c.CL^i$, $r.CL^i$ and $m.clock$. Each time p_j sends m, $m.clock = \langle m.cl_1, \ldots, m.cl_n \rangle$ where $m.cl_k = c.cl_k^j (k = 1, \ldots, n)$.

Livelock-free message reception On receipt of a checkpoint message m from p_j, p_i discards m if $m.cl_k \neq \perp$ and $m.cl_k \leq r.cl_k^i$ for some k. □

Suppose that processes p_i, p_j and p_k are in a rollback class C and a checkpoint message is transmitted from p_k to p_j. Suppose that p_k is rolled back and a rollback request m_r is transmitted and that p_j sends a checkpoint message m to p_i before receiving m_r and p_i receives m_r before m. Here, $r.cl_k^i$ is incremented by one if the rollback recovery occurs in p_i. $m.cl_k$ is the same as one given before the rollback recovery. Hence, $m.cl_k \leq r.cl_k^i$ and m is discarded in p_i.

An identifier of a neighbor process $p_j \in N^i$ is included in the rollback view W^i of p_i if p_i has an active checkpoint c_s^i and one of the following events occurs in p_i:

- A message-receiving event of a checkpoint message m from p_j and $m.cl_j \neq \perp$.
- A message-sending event of a checkpoint message m to p_j.

Hence, p_i knows that p_i and p_j are in the same rollback class even if no checkpoint message is transmitted between p_i and p_j. Therefore, R3 holds.

Moreover, in order to assure that no message is lost after the rollback recovery, if p_i receives a message m from p_j where p_i has an active checkpoint c_s^i and p_j has no active checkpoint, p_i records m in M^i. If p_i is rolled back to c_s^i and is restarted, p_i takes m out of M^i before receiving a message from the channels of p_i. By this algorithm, R4 is satisfied.

The following procedures Send(m) and Receive(m) is executed when a message-sending event of a message m to a process $m.receiver$ and a message-receiving event of a message m from a process $m.sender$ occur in p_i, respectively:

```
Send(m)
  m.flag ← flag^i;
  m.clock ← c.CL^i;
  send m to m.receiver;
Receive(m)
  if m.flag = True
    if m.cl_k ≠⊥ and m.cl_k ≤ r.cl_k^i for some k
      discard m;
    else
      foreach p_j ∈ N^i do
        if m.cl_j ≠⊥
          add p_j to W^i
        fi
      od
      c.CL^i ← max(c.CL^i, m.clock);
      if flag^i = False
        c.cl_i^i ← c.cl_i^i + 1;
        flag^i ← True;
        checkpoint;
      fi
      accept m;
    fi
  else
    if flag^i = True
      add m to M^i;
    fi
    accept m;
  fi
```

5.3 Rollback recovery

If a process p_i fails, a rollback recovery procedure is initiated. The procedure is finished if the rollback class C of p_i becomes empty. The rollback recovery is realized by using the message diffusion protocol [4]. If p_i receives the rollback request m_r from p_j, p_i sends m_r to all the processes in W^i except p_j. Then, p_i restores the state information recorded at the active checkpoint c_s^i taken by p_i and is restarted from c_s^i. Thus, on being rolled back to the checkpoint, p_i can be restarted independently of the other processes while p_i has to be suspended to be synchronised with the other processes in the other algorithms [5,7,11–14]. When p_i is restarted, $r.CL^i$ is updated to be $c.CL^i$.

The following procedure Rollback() is executed when p_i is recovered or p_i receives a rollback request message m_r:

```
Rollback()
  foreach p ∈ W^i do
    send a rollback request m_r to p;
  od
  flag^i ← False;
  r.CL^i ← c.CL^i;
  foreach j ≠ i do
    c.cl_j^i ←⊥;
  od
  restart from the active checkpoint of p_i;
```

6 Evaluation

First, we would like to show the logical properties of the algorithm presented in this paper.

Theorem 2 The rollback algorithm is terminated

Table 1: Overhead.

	Checkpointing		Rollback	
	Message	Time	Message	Time
Koo & Toueg	$O(N)$	$O(D)$	$O(N)$	$O(D)$
Ours	0	0	$O(n)$	$O(d)$

in finite time.

Proof Suppose that a process p_i is included in a rollback class C. Let $|C|$ be the number of processes included in C. Consider a case that p_i receives a checkpoint message from a process $p_j \in C$ and then p_i is rolled back and restarted from the active checkpoint. If p_i receives a checkpoint message m from p_j where $p_j \in C$, p_i discards m because $m.cl_i < r.cl_j^i$. Thus, p_i can be included in C at most $|N^i|$ times where $|N^i|$ is the number of the neighbor processes of p_i. Therefore, $|C|$ is incremented by one at most $I = \sum_i |N^i|$ times. Since the number of processes in the system is finite, I is also finite. On the other hand, $|C|$ is decremented by one if a process $p_k \in C$ receives a rollback request message. In our algorithm, once the rollback recovery is invoked by a process in C, rollback request messages are transmitted among the processes in C while $C \neq \emptyset$. Therefore, eventually $|C| = 0$ and the rollback algorithm is terminated. □

Thus, there occurs no livelock in the rollback recovery algorithm.

Theorem 3 The number of rollback request messages transmitted in a rollback class consisting of l channels is $O(l)$.

Proof As shown in the proof of the theorem 1, each process p_i does not receive a checkpoint message more than once from the same neighbor process p_j included in a rollback class C. Thus, since the rollback is initiated in C until C becomes empty, the rollback request is transmitted only once through each channel in the system. Therefore, the number of request messages transmitted in the rollback algorithm is $O(l)$. □

Next, we would like to evaluate the algorithm by comparing with the conventional one [7]. Table 1 represents the overhead for checkpointing and rollback recovery. The number of additional messages in [7] is $O(N)$ where N is the number of processes in the system and the required time is $O(D)$ where D is the diameter of the system because the two-phase commitment protocol is used. In our algorithm, no additional message is transmitted for checkpointing and $O(n)$ additional messages are transmitted and $O(d)$ time overhead is required for rollback recovery where n is the number of processes in a rollback class and d is the diameter of the rollback class. Therefore, our algorithm reduces the number of messages especially in a large-scale distributed system because $n \ll N$ and $d \ll D$ are satisfied.

Here, we would like to evaluate the availability of the system $\langle V, L \rangle$ executing a distributed application. The application is characterized by a communication pattern $\langle V, H \rangle$ where $V = \{p_1, \ldots, p_n\}$ is the set of processes and the subset $H \subseteq L$ of channels. $\langle p_i, p_j \rangle \in H$ if messages are transmitted from p_i to p_j in the application. Here, each channel in H is assumed to be bidirectional. The following types of communication patterns in applications are considered:

1) Linear type: $H = \{\langle p_i, p_{i+1} \rangle | i = 1, \ldots, n-1\}$, i.e., each p_i communicates only with p_{i-1} and p_{i+1}.
2) Star type: $H = \{\langle p_1, p_i \rangle | i = 2, \ldots, n\}$, i.e., each p_i only communicates with p_1.
3) Binary-tree type: $H = \{\langle p_i, p_{2i} \rangle, \langle p_i, p_{2i+1} \rangle | i = 1, \ldots, (n-1)/2\}$, i.e., p_1 is the root and each p_i communicates with p_{2i} and p_{2i+1}.

Let t be the time when p_1 fails and t^i be the time when p_i is restarted from the checkpoint. It takes $t^i - t$ to roll back p_i. Figures 5, 6 and 7 show the total recovery time $T = \sum_i (t^i - t)$ for the types 1), 2) and 3), respectively. The shorter T is, the more highly available the system is. Following the figures, these systems are made more highly available by using the proposed algorithm than the conventional algorithms.

In a large-scale distributed system, most of the elements in $m.clock$ are unknown, i.e., $m.cl_k = \perp$ for most of k. Thus, when the algorithm is implemented in a real system, each message m had better contain only pairs $\langle k, m.cl_k \rangle$ where $m.cl_k \neq \perp$ than the n-dimensional vector.

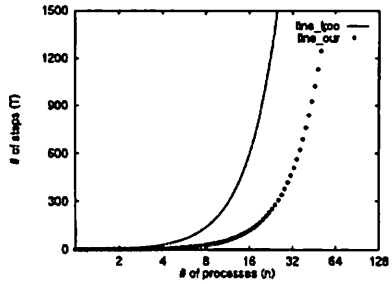

Figure 5: Overhead in linear-type application.

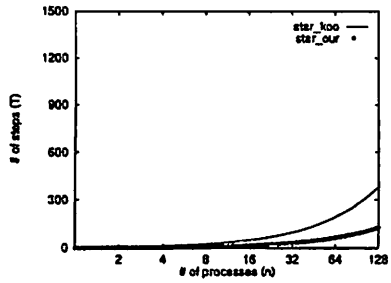

Figure 6: Overhead in star-type application.

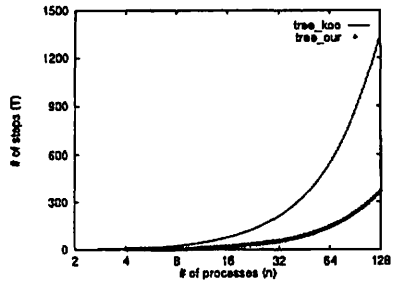

Figure 7: Overhead in binary-tree-type application.

7 Concluding Remarks

This paper has proposed the new algorithm for taking checkpoints and rolling back processes in asynchronous distributed systems. The minimum number of processes take checkpoints. The processes are rolled back asynchronously. Each process manipulates $O(n)$ information and each message contains $O(n)$ information for the algorithm. The rollback algorithm is terminated with $O(l)$ message transmissions where l is the number of channels. The algorithm realizes the more highly available system than the conventional one because the processes in the system can take the checkpoints without transmitting additional messages and and can be asynchronously rolled back without stopping the data transmission. Therefore, the algorithm will play an important role to develop the reliable and available large-scale distributed systems.

References

[1] Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, pp. 222–261 (1987).

[2] Bhargava, B. and Lian, S.R., "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems," *Proc. of the 7th International Symposium on Reliable Distributed Systems*, pp. 3–12 (1988).

[3] Chandy, K. M. and Lamport L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. on Computer Systems*, Vol. 3, No. 1, pp. 63–75 (1985).

[4] Dijkstra, E. W. and Scholten, C. S., "Termination Detection for Diffusing Computation," *Information Processing Letters*, Vol. 11, No. 1, pp. 1–4 (1980).

[5] Juang, T. T. Y. and Venkatesan, S., "Efficient Algorithms for Crash Recovery in Distributed Systems," *Proc. of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science (LNCS)*, pp. 349–361 (1990).

[6] Kim, J.L. and Park, T., "An Efficient Protocol for Checkpointing Recovery in Distributed Systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 8, pp. 955-960 (1993).

[7] Koo, R. and Toueg, S., "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Trans. on Software Engineering*, Vol. SE–13, No. 1, pp. 23–31 (1987).

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

[9] Manivannan, D. and Singhal, M., "A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing," *Proc. of the 16th International Conference on Distributed Computing Systems*, pp. 100–107 (1996).

[10] Perterson, S.L. and Kearns, P., "Rollback Based on Vector Time," *Proc. of the 12th International Symposium on Reliable Distributed Systems*, pp. 68–77 (1996).

[11] Randell, B., "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, Vol. SE–1, No. 2, pp. 220–232 (1975).

[12] Tong, Z., Kain, R. Y., and Tsai, W. T., "Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 2, pp. 246–251 (1992).

[13] Venkatesh, K., Radhakrishnan, T., and Li, H. F., "Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery," *Information Processing Letters*, Vol. 25, pp. 295–303 (1987).

[14] Wood, W. G., "A Decentralized Recovery Protocol," *Proc. of the 11th International Symposium on Fault Tolerant Computing Systems*, pp. 159–164 (1981).