マルチメディア通信と分散処理 76－8
（1996. 5. 16）

Checkpoint and Rollback in Asynchronous Distributed Systems

Hiroaki Higaki, Kenji Shima, Takayuki Tachikawa, and Makoto Takizawa
Dept. of Computers and Systems Engineering
Tokyo Denki University

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 checkpointing. (2) No additional message is transmitted for taking checkpoints. (3) A set of local checkpoints taken by the multiple processes denotes a consistent global state. (4) Multiple processes can initiate simultaneously rollback recovery. (5) The minimum number of processes are rolled back. (6) Each process is rolled back independently of the other processes. Therefore, the system is kept highly available by the algorithm.

非同期分散システムにおけるチェックポイント設定とロールバック回復

桧垣 博章　島 健司　立川 敬行　滝沢 誠
{hig, sima, tachi, taki}@takilab.k.dendai.ac.jp
東京電機大学理工学部経営工学科

非同期分散システムにおける、新しいチェックポイント設定手法およびロールバック回復手法について述べる。本論文で提案するチェックポイント取得手順は、任意のプロセスによる手順実行開始が可能であり、このために特別のメッセージを必要としない。また、本論文で提案するロールバック回復手順は、任意のプロセスによる手順実行開始が可能であり、一貫した広域状態を定めるチェックポイントへ、必要なプロセスのみをロールバックさせる。このロールバックは、各プロセスごとに非同期的に実行される。提案する手法は、この非同期的な実行によって発生し得るライブロックの問題を解決している。以上により、システムの可用性を保ったままでこれらの手順を実行することができる。

1 Introduction

Information systems are distributed and are getting larger by including many 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 used products including freewares and sharewares rather than specially designed hardwares and softwares. 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 in order 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 being executed. If the processes fail in the system, the processes are rolled back to the checkpoints by restoring the saved state information and then are restarted 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 [2]. Many papers [2, 4, 5, 7–10] have discussed so far how to take the consistent checkpoints among multiple processes. In addition, it is critical to discuss how to roll back the processes if the system suffers from the process faults. If each process is rolled back independently of the other processes, the system may be inconsistent. One idea [5] 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 methods are reviewed. In section 3, we show a basic algorithm for taking checkpoints and rolling back processes. In section 4, we make clear the problem of livelocks occurring 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

A distributed system is composed of multiple processes interconnected by channels, i.e., $\langle V, L \rangle$

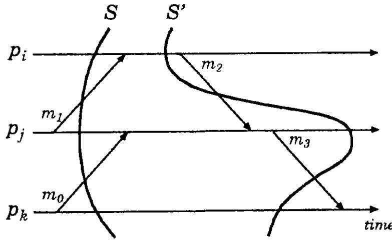

Figure 1: Consistent global state

where $V = \{p_1, \ldots, p_n\}$ is a set of processes and $L \subseteq V^2$ is a set of channels. In the distributed system, three kinds of events occur: message-sending, message-receiving and local events. The state of a process is changed when an event occurs. An event e_s *happens before* another event e_t ($e_s \rightarrow e_t$) iff one of the following conditions is satisfied [6]:

- 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 $[p_i, p_j]$. A state of $[p_i, p_j]$ is defined as a set of messages sent by p_i but not yet received by p_j. For a set P of processes, a global state $S(P)$ is a set of local states of the processes in P and of the channels with which the processes are connected.

If the processes take the checkpoints and are rolled back to the checkpoints independently, there exist two kinds of inconsistent messages: *lost messages* and *orphan messages*. A lost message means that a message-sending event occurs before the sender process takes a checkpoint and a message-receiving event occurs after the receiver process takes a checkpoint. An orphan message means that a message-sending event occurs after the sender process takes a checkpoint and a message-receiving event occurs before the receiver process takes a checkpoint. By recording the received messages in the log after taking a checkpoint, the lost messages can be received by taking them out of the log. However, if there exist orphan messages, the system becomes inconsistent. Even if the processes record messages in the log at message-sending events, the processes may not send the messages after the rollback recovery if the processes are not deterministic. Therefore, the global state of the system is consistent iff there is no orphan messages. Figure 1 shows three processes p_i, p_j and p_k. A global state S is consistent because there is no orphan message. However, another global state S' is inconsistent because m_2 is an orphan message.

There are two approaches to taking checkpoints among multiple processes. One is *asynchronous checkpointing* where the processes take the checkpoints without cooperating with the other processes [4, 9, 10]. This approach implies less overhead because there is no communication among the processes. However, *domino effects* may occur [7]. The other is *synchronous checkpointing* where multiple processes are coordinated to take the checkpoints [2,5,7,8]. By the synchronous one, the overhead for taking checkpoints is larger than the asynchronous approach while the overhead for rolling back is limited. This paper discusses the synchronous approach.

A global checkpoint is a set of local checkpoints taken by all the processes in the system [2, 4, 5, 7–10]. Additional messages are transmitted and processes are suspended during the checkpointing procedure. However, all the processes are not always needed to take checkpoints to keep the system consistent after the rollback recovery.

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 in the channels connected with the processes in P. □

In our checkpointing algorithm, a message m contains the information on whether the sender process of m has taken a checkpoint or not. If the sender process of m had taken a checkpoint, m is referred to as a *checkpoint message*.

Checkpoint If a message-receiving event e for a checkpoint message occurs in a process p which has not yet taken a checkpoint, p takes a checkpoint just before e.

The minimum number of processes take checkpoints and no additional message is transmitted to take checkpoints by using the protocol.

In the conventional methods, the processes have to be synchronized to be restarted by the following procedure after they are rolled back:

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

One of the disadvantages of the method is that the processes are suspended and additional messages are transmitted to synchronize 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 processes are less synchronized to be restarted from the checkpoints.

3 Basic Algorithm

Here, we would like to show a basic algorithm for taking checkpoints and rolling back processes by using an example in Figure 2. The system

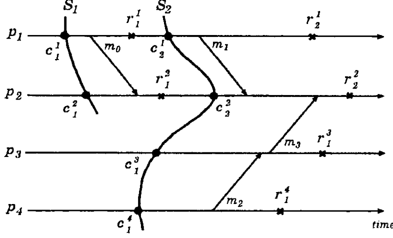

Figure 2: Semi-consistent global state.

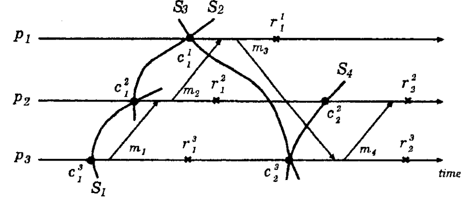

Figure 3: Livelock in rollback recovery.

consists of processes $\{p_1, p_2, p_3, p_4\}$ and channels $\{[p_1, p_2], [p_2, p_3], [p_3, p_4]\}$. Here, $[p_i, p_j]$ denotes a bidirectional channel between p_i and p_j. Each process p_i takes a checkpoint c^i. If some failure occurs, p_i is rolled back and restarted from c^i. c^i_s represents the sth checkpoint taken by p_i. r^i_s shows a possible rollback event where p_i is rolled back to c^i_s. If c^i_s occurs and r^i_s does not occur in p_i, c^i_s is *active*. For example, c^1_2 is active when p_1 sends m_1. If p_1 is rolled back from r^1_2 and is restarted from c^1_2, c^1_2 is not active.

If p_i fails, it is not sufficient to roll back p_i to the checkpoint 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 realize a semi-consistent global state after the rollback recovery, p_2 has to be restarted from c^2_1. That is, if p_i is restarted from c^i_s, an event e^j_t in p_j where $c^i_s \rightarrow e^j_t$ 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^2_1, respectively, p_3 and p_4 are not required to be rolled back. That is, if no event e^j_t where $c^i_s \rightarrow e^j_t$ 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 rolled back.

Definition (rollback-domain) Let c^i and c^j be active checkpoints taken by processes p_i and p_j, respectively. Let e^i and e^j be events such that 1) $c^i \rightarrow e^i$ and $c^j \rightarrow e^j$, and 2) there is no event e where $c^i \rightarrow e \rightarrow e^i$ or $c^j \rightarrow e \rightarrow e^j$. Here, $c^i \Rightarrow c^j$ is defined iff $e^i \rightarrow e^j$. A *rollback-domain* $D(p_i)$ of p_i is defined to be a following set of processes:

- $p_i \in D(p_i)$ if there is an active checkpoint in p_i. Otherwise, $D(p_i) = \emptyset$.
- $p_j \in D(p_i)$ if c^j is active in p_j and $c^j \Rightarrow c^k$ or $c^k \Rightarrow c^j$ where c_k is active in $p_k \in D(p_i)$. □

For example, when p_1 takes a checkpoint c^1_1, $D(p_1) = \{p_1\}$. When m_0 is transmitted from p_1 to p_2, p_2 takes a checkpoint c^2_1 before accepting m_0. Here, $D(p_1) = D(p_2) = \{p_1, p_2\}$.

It is clear that $p_i \in D(p_j)$ and $D(p_i) = D(p_j)$ for $p_j \in D(p_i)$, and $D(p_i) \cap D(p_j) = \emptyset$ if $p_j \notin D(p_i)$. A set $C = D(p_i)$ of processes is referred to as a *rollback-class*. If p_i in a rollback-class C sends a message m and p_k in another rollback-class C' receives m, C and C' are merged into $C'' = C \cup C'$ according to the definition. For example, before m_3 is transmitted, p_1 and p_2 are in $C = \{p_1, p_2\}$ and p_3 and p_4 are in $C' = \{p_3, p_4\}$. When m_3 is transmitted from p_3 to p_2, C and C' are merged into $C'' = C \cup C' = \{p_1, p_2, p_3, p_4\}$.

Theorem If a process in a rollback-class C fails, the system state is semi-consistent if only and all the processes in C are rolled back to the active checkpoints. □

That is, C is the minimum set of processes to be rolled back for keeping the system semi-consistent. However, p_i has no information on which processes are included in C. Suppose that p_1, p_2, p_3, p_4 are at r^1_2, r^2_2, r^3_1, r^4_1, respectively. p_1 does not know that p_4 is in $D(p_1)$ while knowing that p_2 is in $D(p_1)$. That is, p_i has only the information on whether the neighbor processes are in the same rollback-class or not. Let $W(p_i)$ denote a p_i's view of $D(p_i)$. $W(p_i)$ is a set of processes which p_i knows in $D(p_i)$, i.e., $W(p_i) \subseteq D(p_i)$.

Based on the view of p_i, p_i can be rolled back and restarted from the checkpoint by using the message diffusion protocol [3].

1) If p_i fails, p_i sends request r to all the processes in $W(p)$.
2) On receipt r, p_i also sends r to all the processes in $W(p)$.
3) If p_i receives r from all the processes in $W(p)$, p is rolled back and restarted from the checkpoint.

4 Livelock in Rollback Recovery

Consider the following scenario (Figure 3).

1) p_3 takes c^3_1 and sends m_1 to p_2. m_1 is a checkpoint message. Here, c^3_1 is active and $D(p_3) = \{p_2, p_3\}$.
2) p_2 receives m_1 and takes c^2_1 where the local state of p_2 before receiving m_1 is recorded in the log. Here, c^2_1 is active and $D(p_2) = D(p_3) = \{p_2, p_3\}$. $S_1 = \{c^2_1, c^3_1\}$ is semi-consistent.
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 if 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^3_1 to c^3_1. Now, since c^3_1 is not active, 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.

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 the 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.
7) p_3 sends m_4 to p_2. p_2 receives m_4 and takes c_2^2. $D(p_1) = D(p_2) = D(p_3) = \{p_1, p_2, p_3\}$.
8) p_1 is rolled back from r_1^1 to c_1^1. Here, $D(p_2) = D(p_3) = \{p_2, p_3\}$. $S_4 = \{c_2^2, c_2^3\}$ is semi-consistent.

Thus, the rollback recovery can be continued forever, i.e., livelock occurs. It is noted that p_3 does not take c_2^3 on receipt of m_3 if p_3 is not rolled back before receiving m_3.

Suppose that a rollback r_s^i has occurred in p_i and then p_i receives a message m from p_j. Let e^j be a message-sending event of m in p_j. If an event e^i has occurred in p_i where $e^i \rightarrow e^j$ and $c_s^i \rightarrow e^i \rightarrow r_s^i$, p_i cannot receive m. That is, p_i is sure that p_j would be rolled back by the rollback of p_i. The message-receiving event for m has to be canceled. If p_i receives m, the livelock may occur. Hence, p_3 ignores m_3.

Definition (generation) If an event e occurs in a process p_i after a checkpoint c_s^i is taken and before a rollback r_s^i occurs, the generation $g(e)$ of e is s. Otherwise $g(e)$ is $\perp$ (unknown). □

The generation $g(e)$ of a message-sending event e for a message m is piggied back by m. On receipt of m, p_i rejects m if p_j knows by $g(e)$ that m is canceled by the rollback-recovery. By the method, the livelock is prevented.

5 Algorithm

5.1 Assumptions and Definitions

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

A1 The channels are bidirectional.
A2 The channels are reliable.
A3 For each channel, messages are transmitted in the first-in-first-out order.
A4 S is asynchronous, i.e., a maximum message transmission delay is unbounded and finite.

If $[p_i, p_j] \in L$, p_j is a *neighbor process* of p_i. N^i is a set of neighbor processes of p_i.

The following events occur in a process p_i:

- Message-receiving event: p_i takes out a message m from a channel $[p_j, p_i]$ and accepts m.
- Message-sending event: p_i puts a message m to a channel $[p_i, p_j]$.
- Checkpoint: p_i records the local state information in the log. The sth checkpoint taken by p_i is denoted by c_s^i.
- Rollback: p_i restores the local state information recorded at the checkpoint c_s^i and is restarted from c_s^i. Here, the rollback is denoted by r_s^i.

c_s^i is *active* from the time when c_s^i is taken to the time when r_s^i occurs.

p_i manipulates the following variables:

- A vector clock $GL^i = \langle gl_1^i, \dots, gl_n^i\rangle$. Each gl_i^i shows the current generation of p_i. g_i^i is incremented by one each time p_i is rolled back. Initially, $gl_i^i = 0$ and $gl_j^i = \perp$ for $j \neq i$.
- A flag $flag^i$. If c_s^i is active, $flag^i = True$. Otherwise, $flag^i = False$.
- A subset $W^i \subseteq N^i$ of the neighbor processes included in the rollback-domain $D(p_i)$ of p_i, 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, $M^i = \emptyset$.

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

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

For a pair of vector clocks $v^i = \langle v_1^i, \dots, v_n^i\rangle$ and $v^j = \langle v_1^j, \dots, v_n^j\rangle$, $max(v^i, v^j)$ is defined to be $\langle v_1, \dots, v_n\rangle$ 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 algorithm has to satisfy the following requirements:

R1 A process which has an active checkpoint is included in exactly one rollback-class.
R2 A global checkpoint in a rollback-class is semi-consistent.
R3 Every process p_i has the information W^i on which neighbor processes are included in the same rollback-class.

A process p_i takes a checkpoint if one of the following conditions is satisfied:

C1 If a checkpoint event occurs in p_i, p_i takes a checkpoint.
C2 If a message-receiving event e occurs in p_i and p_i receives a message m transmitted from a neighbor process p_j that has taken a checkpoint, p_i takes a checkpoint just before e.

C1 means that checkpointing can be initiated independently by multiple processes. C2 means that there is no orphan message in $[p_i, p_j]$. Thus, R2 is satisfied. Suppose that p_i sends a message m after taking a checkpoint c_s^i. If c_s^i is active, $flag^i = True$. Thus, $m.flag = True$, i.e., m is a *checkpoint* message. Suppose that p_j receives m after taking c_s^j, p_j does not take another checkpoint even if $m.flag = True$. As presented in the previous section, if p_i in a rollback-class C sends a checkpoint message to p_j in another rollback-class C', C and C' are merged into one rollback-class C''. Thus, R1 is satisfied.

The system has to prevent from livelock caused by a rollback recovery. Here, we would like to present the checkpointing algorithm for the livelock-free rollback recovery. Suppose that p_i receives a checkpoint message m from p_k, i.e., $m.flag = True$. Let e^i denote the message-receiving event of m in p_i and e^k denote the message-sending event of m in p_k. If r^i_s has occurred in p_i where $c^i_s \rightarrow e^k$ and $g(c^i_s) < g(e^i)$, e^i is canceled, i.e., m is not received. Otherwise, p_i takes a checkpoint $c^i_{g(e^i_s)+1}$. p_i has the local vector clock $GL^i = \langle gl^i_1, \dots, gl^i_n \rangle$. Each time p_i sends m, $m.clock = \langle m.cl_1, \dots, m.cl_n \rangle$ where $cl_k = gl^i_k (k = 1, \dots, n)$. Here, on receipt of a message m from p_j, m is discarded if $gl^i_i > m.cl_i$.

p_i records an identifier of a neighbor process p_j after taking a checkpoint if one of the following conditions is satisfied:

- p_i receives a checkpoint message from p_j.
- p_i sends a message to p_j.

Hence, R3 holds.

Moreover, in order to assure that no message is lost after the rollback recovery, if p_i receives a message m when p_i has an active checkpoint c^i_s, p_i records m in M^i. If p_i is rolled back to c^i_s and is restarted, p_i receives m from M^i before receiving from the channels.

The procedure Send(m) is executed when a message-sending event occurs in p_i. p_i sends a message m to a process $m.receiver$.

```
Send(m)
    m.flag ← flag^i;
    m.clock ← GL^i;
    send m to m.receiver;
```

The procedure Receive(m) is executed when a message-receiving event occurs in p_i. p_i receives a message m from a process $m.sender$.

```
Receive(m)
    if flag^i = True
        if m.flag = True
            add m.sender to W^i.
            GL^i ← max(GL^i, m.clock);
            accept m;
        else
            add m to M^i;
            accept m;
        fi
    else
        if m.flag = True
            if m.cl_i ≠⊥ and m.cl_i < gl^i_i
                discard m;
            else
                checkpoint;
                add m.sender to W^i;
                GL^i ← max(GL^i, m.clock);
                accept m;
            fi
        else
            accept m;
        fi
```

5.3 Rollback recovery

If a process p_i fails, a rollback recovery procedure is executed. The procedure is finished if the rollback-class C of p_i becomes empty. This is realized by using the message diffusion protocol [3]. If p_i receives the request r for rollback recovery from p_j, p_i sends r to all the processes in W^i. On receiving r from all the processes in W^i, p_i restores the state information recorded at the active checkpoint c^i taken by p_i and is restarted from c^i. Thus, p_i can be restarted as soon as p_i is rolled back to the checkpoint while p_i has to wait for rollback recoveries of other processes in the other algorithms [4, 5, 7–10].

When p_i is restarted, gl^i_i is incremented by one. If p_i receives a message m from a process in W^i after p_i receives the request r for the rollback recovery and before p_i is restarted, m is discarded. This is because the message-sending events at which m is sent is eventually canceled by the rollback recovery. If p_i receives a message m from a process $p \in N^i - W^i$ after p_i receives r and before p_i is restarted, m is recorded in M^i and is accepted when p_i is restarted.

The procedure Rollback() is executed when a process p_i is recovered or p_i receives a request message r for a rollback recovery.

```
Rollback()
    foreach p ∈ W^i do send r to p; od
    do
        if receive m from some p ∈ W^i
            discard m;
        else if receive m from some p ∈ N^i − W^i
            add m to M^i;
        else if receive r from some p ∈ W^i
            discard r;
        fi
    until receive r from every p ∈ W^i
    restart;
```

6 Evaluation

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

Theorem The rollback algorithm is terminated in finite time. □

Next, we would like to evaluate the algorithm by comparing with the conventional one [5]. The following kinds of distributed systems $S = \langle V, L \rangle$ are considered where $V = \{p_1, \dots, p_n\}$:

1) Linear system: $L = \{[p_i, p_{i+1}] | i = 1, \dots, n-1\}$.
2) Star system: $L = \{[p_1, p_i] | i = 2, \dots, n\}$.
3) Binary-tree system: $L = \{[p_i, p_{2i}], [p_i, p_{2i+1}] | i = 1, \dots, (n-1)/2\}$.

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 4, 5 and 6 show $T = \sum_i (t^i - t)$ for the three systems. The shorter T is, the more highly available the system is. Fol-

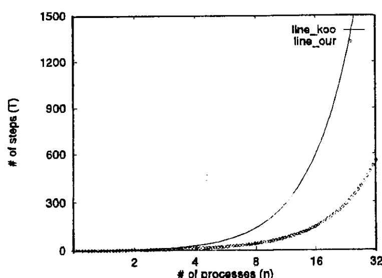

Figure 4: Overhead in linear-network.

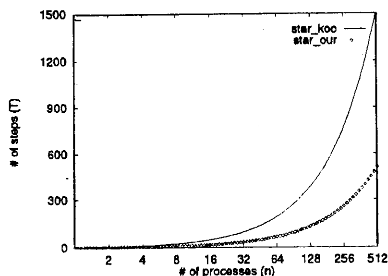

Figure 5: Overhead in star-network.

lowing the figures, these systems are made more highly available than the conventional algorithm by using the proposed algorithm.

The view $W(p_i)$ of p_i includes only the neighbor processes with which p_i exchanges messages. If each message from p_i to p_j contains the information $W(p_i)$, p_j can get a larger $W(p_j)$ by adding $W(p_i)$ to $W(p_j)$. By using the method, the rollback time can be reduced. We would like to discuss the optimization methods in another paper.

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. The algorithm realizes the more highly available system than the conventional one. 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] 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).

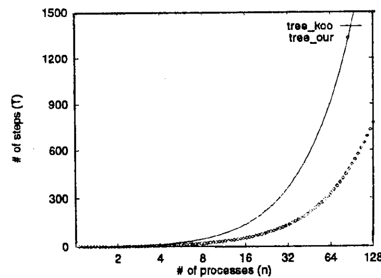

Figure 6: Overhead in binary-tree-network.

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

[4] 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).

[5] 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).

[6] 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).

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

[8] 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).

[9] 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).

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