マルチメディア通信と分散処理 79－5
（1996. 11. 14）

Livelock-Free Asynchronous Recovery in Distributed Systems

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

This paper proposes a novel protocol for taking checkpoints and asynchronously rolling back the processes for recovery in asynchronous distributed systems. In the protocol, only the minimum number of processes take checkpoints. The amount of execution of application program wasted by the recovery is also the minimum. Moreover, each process can be rolled back and restarted asynchronously. Here, the livelocks might occur if the processes are asynchronously restarted. In the protocol proposed in this paper, each process is rolled back at most once to recover from a failure of process. Hence, the livelocks can be avoided. Only $O(l)$ messages are transmitted where l is the number of channels in the system. Therefore, the protocol makes the system highly available.

分散システムにおけるライブロックのない非同期リカバリ

桧垣 博章　田中 勝也　島 健司　立川 敬行　滝沢 誠
{hig, katsu, 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取得手法および非同期リカバリ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プロトコルを用いると、チェックポイントを取得するプロセスは最小数であり、リカバリによって失われるアプリケーションプログラムの実行時間も最小となる。各プロセスは他のプロセスに対して非同期に回復し、アプリケーションの実行を再開するが、これによってライブロックが発生してしまい、回復プロトコルが終了しない場合がある。そこで本論文では、ひとつのプロセス故障に対して各プロセスが一度だけしかロールバックされないプロトコルを設計し、ライブロックの発生を防いでいる。このリカバリに必要なメッセージはチャネル数 l に対して $O(l)$ である。

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. The distributed systems are designed and developed by using widely available products including free softwares rather than specially designed ones. These products are not always guaranteed to support enough reliability and availability for the applications. It is important to discuss the mechanism in the system softwares to make and keep the systems so reliable and available that even fault-tolerant applications could be implemented in the systems.

Distributed applications are realized by cooperation of multiple processes executed in multiple processors. Checkpointing and rollback recovery are well-known time-redundant techniques to allow processes to make progress even if some processes fail. The processes take local checkpoints by recording 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 restarted from the checkpoints. Hence, the system can tolerate *transient* failures, e.g. hardware errors, process crashes, transaction aborts, and communication deadlocks. These failures are unlikely to recur after the processes are restarted.

The system has to be kept consistent even if the processes are rolled back to the checkpoints. A global checkpoint is a set of local checkpoints taken by all the processes in the system. Many papers [1,6,7,9,10] have proposed so far protocols for taking a consistent global checkpoint among processes and ones for restarting the processes if one or more processes fail.

The conventional checkpointing protocols require all the processes in the system to take local checkpoints synchronously. However, each process rather communicates with only a subset of the processes than all the processes. Hence, a checkpointing protocol that allows the subset of the processes to take the local checkpoints is necessary. On the other hand, the processes are required to be synchronized to be rolled back and restarted during the recovery procedure. However, the recovery procedure may be slow down since it takes longer to synchronize the processes. If the checkpointing procedure is executed in a subset of the

processes to take a consistent checkpoint and the recovery procedure is asynchronously executed to restart the processes, the rollback recovery may continue forever, i.e. a livelock may occur. In this paper, we propose a protocol for a livelock-free asynchronous recovery where only a subset of the processes are restarted in the distributed system.

2 Checkpoint and Rollback

2.1 Consistent state

A distributed system is composed of multiple processes interconnected by channels, i.e. $S = \langle V, L \rangle$ where $V = \{p_1, \ldots, p_n\}$ is a set of processes and $L \subseteq V^2$ is a set of channels. $\langle p_i, p_j \rangle$ is a channel from p_i to p_j. In each process, three kinds of events occur: message-sending, message-receiving and local events. Among the events, the *happens before* relation is defined [8]. A local state of p_i is changed each time an event occurs in p_i. That is, 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 $\langle p_i, p_j \rangle \in L$. 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 global state of S is a collection of states of the processes in V.

A local checkpoint c^i taken by p_i is the state of p_i recorded in the local log. A global checkpoint C is a set of local checkpoints taken by all the processes in V, i.e. $C = \{c^1, \ldots, c^n\}$. If the processes take the local checkpoints and are restarted from the local checkpoints independently of the other processes, there may exist inconsistent messages named *orphan messages* [1]. Hence, the global state of the system is defined *consistent* iff there is no orphan message [3].

Theorem 1 A global checkpoint C is consistent iff $\forall c^i, c^j \in C$ $c_i \not\rightarrow c_j$. □

2.2 Checkpointing

In the conventional synchronous checkpointing protocol [1, 5, 7, 9, 10], additional messages are transmitted to take a consistent global checkpoint and the processes are blocked while the checkpointing procedure is executed. Moreover, if some process takes a local checkpoint, all the other processes in the system are required to take the local checkpoints. However, each process rather communicates with only a subset of the processes than all the processes. Hence, all the processes are not always required to take the local checkpoints simultaneously. Here, we would like to define a *regional* checkpoint $\tilde{C}$ and a consistent regional checkpoint in the system $S = \langle V, L \rangle$.

Definition (regional checkpoint) Let W be a subset of the processes in S, i.e. $W \subseteq V$. Let c^i be the local checkpoint of p_i where $p_i \in W$. A *regional* checkpoint $\tilde{C}(W)$ of W is defined as a set $\{c^i | p_i \in W\}$ of local checkpoints. □

Definition (consistent regional checkpoint) A regional checkpoint $\tilde{C}(W)$ is consistent for a subset W of the processes in S iff there is no orphan message in every channel of every $p_i \in W$. □

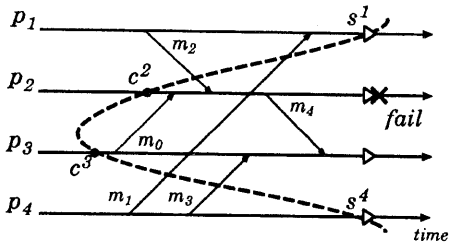

Figure 1: Consistent regional checkpoint.

Theorem 2 A regional checkpoint $\tilde{C}(W)$ is consistent for a subset W of the system iff $\forall c^i, c^j \in \tilde{C}(W)$ $c^i \not\rightarrow c^j$. □

Theorem 3 Let s^i be a state of an operational process $p_i \in V - W$, i.e. p_i is executing the application. A global state denoted by $\tilde{C}(W)$ and $\{s^i | p_i \in V - W\}$ is consistent if $\tilde{C}(W)$ is consistent. □

Figure 1 shows an example. If each p_i is at time marked ▷ and p_2 fails, only p_2 and p_3 have to be restarted from c^2 and c^3, respectively. Here, $\tilde{C}(W) = \{c^2, c^3\}$. $\{s^1, c^2, c^3, s^4\}$ denotes a consistent global state because there is no orphan message. That is, S can be recovered when a process in W fails iff $\tilde{C}(W)$ is consistent for a subset W of the processes and only and all the processes in W are restarted from the local checkpoints. Here, suppose that c^i is taken by $p_i \in W$. If m is the first message transmitted from p_i to p_j after taking c^i, m is referred to as a *checkpoint message* of c^i to p_j. If p_j receives m, p_j has to take c^j and has to be included in W. Hence, the following checkpointing method is introduced:

[Checkpointing method] If a message-receiving event e for a checkpoint message m occurs in p_j, p_j takes c^j just before e. □

Here, since m can carry the information on whether m is a checkpoint message or not, no additional message is transmitted for taking a consistent regional checkpoint.

2.3 Rollback recovery

By using *synchronous* recovery protocols [1, 5, 7, 9–12], all the processes are blocked and the additional messages are transmitted to synchronize all the processes in the system for the consistent recovery. Thus, the recovery procedure is slow down due to the synchronization overhead, i.e. the system becomes less available. Here, c^i_s represents the sth checkpoint taken by p_i and $r^i_{s,t}$ represents the possible recovery event occurring in p_i after c^i_t is taken by p_i where p_i is restarted from c^i_s $(s \leq t)$. c^i_s is *active* if c^i_s is taken but $r^i_{t,u}$ where $t \leq s \leq u$

has not yet occurred. p_i may have multiple active checkpoints.

If p_i fails, the system has to be restarted from $\tilde{C}(D_i)$ where D_i is a subset of the processes including p_i. That is, only the processes in D_i are restarted. We name such a subset D_i a *rollback domain* of p_i.

Definition (checkpoint precedence) Let c_s^i and c_t^j be active checkpoints taken by p_i and p_j, respectively. c_s^i *precedes* c_t^j ($c_s^i \Rightarrow c_t^j$) if there exist e^i in p_i and e^j in p_j where $c_s^i \rightarrow e^i$, $c_t^j \rightarrow e^j$, and $e^i \rightarrow e^j$. □

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

1) $p_i \in D_i$ if there is an active checkpoint in p_i.
2) $p_j \in D_i$ if $c_s^i \Rightarrow c_t^j$ where c_s^i is the most recent active checkpoint in p_i and c_t^j is an active checkpoint in p_j.
3) Only the processes satisfying 1) and 2) are included in D_i. □

From the definition of the rollback domain and Theorem 2, the following theorem is induced.

Theorem 4 A regional checkpoint $\tilde{C}(D_i)$ is consistent for every rollback domain D_i. □

That is, a rollback domain is a region for the rollback recovery. Suppose that p_i fails and is restarted from the most recent checkpoint c_s^i. If every $p_j \in D_i$ is restarted from c_t^j where $c_s^i \Rightarrow c_t^j$, the system is kept consistent after the recovery. However, each process cannot know completely which processes are included in D_i. Though each process cannot know D_i completely, the process knows of a subset of the processes in D_i. That is, if p_j sends a message to p_k after taking c_t^j where c_s^i is the most recent local checkpoint in p_i and $c_s^i \Rightarrow c_t^j$, p_j knows that p_k is included in D_i.

Definition (rollback view) A *rollback view* W_i^j is a subset $\{p_k | p_j \text{ knows } p_k \in D_i\}$ of the processes. □

Based on the rollback views, if p_i fails, the system can be restarted *asynchronously* from $\tilde{C}(D_i)$ by applying the message diffusion protocol [2].

[Recovery method]

1) If p_i fails, p_i sends a restart request message $m_{restart}$ to every $p_j \in W_i^i$ and is restarted from the most recent local checkpoint c_s^i.
2) On receipt of $m_{restart}$ from p_k, each p_j also sends $m_{restart}$ to every process included in W_i^j except p_k and is restarted from c_t^j where $c_s^i \Rightarrow c_t^j$.

2.4 Livelock

By taking the regional checkpointing and restarting the processes asynchronously, livelocks might occur in the system [4]. Thus, the proposed checkpointing method has to be modified to avoid the occurrence of the livelocks in the recovery. Here, we introduce a *generation* concept.

Definition (generation of process) A generation $g(p_i)$ of p_i is assigned to s, i.e. $g(p_i) = s$ if the local checkpoint most recently taken by p_i is c_s^i. □

Definition (generation of event) A generation $g(e)$ of e in p_i is assigned to s, i.e. $g(e) = s$ if $g(p_i) = s$ when e occurs in p_i. □

We assume that each p_i takes c_1^i at the initial state. Each time p_i takes c_s^i, $g(p_i)$ is incremented by one. If $r_{s,t}^i$ occurs in p_i, p_i takes c_{t+1}^i just before p_i is restarted to execute the application. Each time a message sending event $s(m)$ occurs in p_i, $g(s(m))$ is piggied back by m. Suppose that e^i occurs in p_i and p_i receives m sent by p_j at $s(m)$ where $e^i \rightarrow s(m)$. If $r_{s,t}^i$ where $s \leq g(e^i) \leq t$ occurs in p_i, p_i has to discard m. If p_i accepts m, the livelock occurs. This is realized by using the generations of events. If $g(e)$ is piggied back by m and there has occurred $r_{s,t}^i$ where $s \leq g(e) \leq t$ in p_i, p_i has to discard m for the livelock-free recovery. In order to implement the livelock-free asynchronous recovery, generation vectors have to be piggied back by the messages transmitted among the processes. In the succeeding section, we would like to present the detailed protocol using the generation vectors.

3 Protocol

3.1 Assumptions

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 reliable.
A2 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.
A3 S is asynchronous, i.e. the maximum message transmission delay is unbounded.
A4 Each process has a stable storage.

c_u^i is *active* unless there occurs $r_{s,t}^i$ where $s \leq u \leq t$ in p_i. c_u^i becomes *inactive* if there occurs $r_{s,t}^i$ where $s \leq u \leq t$ in p_i. c_u^i is *obsolete* if c_u^i is inactive or p_i will never be restarted from c_u^i. If p_i knows that c_u^i is obsolete, p_i can remove c_u^i.

An application message m contains the data $m.data$ and a generation vector $m.G = \langle m.g_1, \ldots, m.g_n \rangle$. A rollback request message m_r contains a generation vector $m_r.G = \langle m_r.g_1, \ldots, m_r.g_n \rangle$.

p_i manipulates the following variables. Here, let $Neighbor^i$ be a set of neighbor processes of p_i.

- A generation vector $G^i = \langle g_1^i, \ldots, g_n^i \rangle$ named a *checkpoint generation vector* in p_i. Every e^i that will occur in p_i causally depends on e^j that has occurred in p_j where $g(e^j) = g_j^i$, i.e. $e^j \rightarrow e^i$. Initially, $g_i^i = 1$ and $g_j^i = 0 (j \neq i)$.

- A set of generation $Inactive^i$. $u \in Inactive^i$ iff c_u^i is inactive, i.e. $r_{s,t}^i$ occurs in p_i, where $s \leq u \leq t$.
- A subset $Flow^i \subseteq Neighbor^i$ of the neighbor processes of p_i. $p_j \in Flow^i$ iff p_i sends an application message to p_j after taking the most recent checkpoint. Each time p_i takes a local checkpoint, $Flow^i = \emptyset$.
- A sequence $Receive^i$ of messages received after taking the most recent checkpoint in p_i: Each time p_i takes a local checkpoint, $Receive^i = \emptyset$.

Each time p_i takes c_s^i, $Flow^i$ and $Receive^i$ are recorded in the log with the state information as $Flow_s^i$ and $Receive_s^i$, respectively.

3.2 Failure-free execution

The protocol for taking consistent regional checkpoints is invoked each time a message-sending or message-receiving event occurs in a process. Here, the generation vector is manipulated as follows:

1) Each time a message-sending event occurs in p_i, $G^i = \langle g_1^i, \ldots, g_n^i \rangle$ is piggied back by m as $m.G$, i.e. $m.g_j \leftarrow g_j^i$ for every j.
2) Each time a message receiving event for m occurs in p_i, G^i is updated to $max(G^i, m.G)$. Here, $G' = max(G^i, G^j)$ is defined as $g'_k = max(g_k^i, g_k^j)$ for every k.

If $r(m)$ occurs in p_i where $g(p_i) = s$ and m is a checkpoint message transmitted from p_j, p_i takes c_{s+1}^i just before $r(m)$. Suppose that c_t^j is the most recent local checkpoint in p_i. If $m.g_k \leq g_k^i$ for every p_k, p_i is not required to take a local checkpoint. p_i is also restarted from c_s^i if p_j is restarted from c_t^j. On the other hand, if $g_k^i < m.g_k$ for some p_k, p_i has to take c_{s+1}^i. Unless p_i takes c_{s+1}^i, m becomes an orphan message if p_j is restarted from c_t^j.

In order to assure that no message be lost after the recovery, if p_i receives m from p_j, m is added to $Receive^i$. Each time p_i takes c_{s+1}^i, the messages in $Receive^i$ are moved to $Receive_s^i$ and $Receive^i$ becomes empty. That is, each $m \in Receive_s^i$ is received while $g(p_i) = s$. If $r_{s,t}^i$ occurs in p_i, every m satisfying the following conditions is received by p_i:

1) $m \in Receive_u^i (s \leq u < t)$ or $m \in Receive^i$
2) $m.g_j < (g_j^i)_s$ for some p_j where the generation vector $G_s^i = \langle (g_1^i)_s, \ldots, (g_n^i)_s \rangle$ is assigned to c_s^i.

Moreover, if p_i sends an application message to p_j, p_j is added to $Flow^i$. Each time p_i takes c_{s+1}^i, the processes in $Flow^i$ are moved to $Flow_s^i$ and $Flow^i$ becomes empty. If $r_{s,t}^i$ occurs in p_i, p_i sends a recovery request message m_r to every p_j where $p_j \in \cup_{s \leq u < t} Flow_u^i$ or $p_j \in Flow^i$.

The system has to prevent from the livelocks caused by the asynchronous recovery. By using the generation vector, p_i never accepts phantom checkpoint messages. Each time $r_{s,t}^i$ occurs in p_i, $Inactive^i$ is updated to $Inactive^i \cup \{u | s \leq u \leq t\}$. Let m be a message taken out of the channel $\langle p_j, p_i \rangle$ where $m.G = \langle m.g_1, \ldots, m.g_n \rangle$ is piggied back.

[Livelock-free message receipt] On receipt of m from p_j, p_i discards m if $m.g_i \in Inactive^i$. Otherwise, p_i accepts m. □

If $m.g_i \in Inactive^i$, there exists e^i such that $g(e^i) \in Inactive^i$ and $e^i \rightarrow s(m)$ where $s(m)$ is the message-sending event. Since e^i is canceled by one of the recovery events which occurred in p_i, $s(m)$ also has to be canceled, i.e. m is a phantom checkpoint message. Thus, m is discarded, i.e. the recovery becomes livelock-free.

The following procedures Send(m) and Receive() are executed when a message-sending event and a message-receiving event occur in p_i, respectively:

Send(m)
  $m.G \leftarrow G^i$;
  $Flow^i \leftarrow Flow^i \cup \{m.receiver\}$;
  send m *to* $m.receiver$;

Receive()
  take m *out of* $\langle m.sender, p_i \rangle$;
  if $m.g_i \in Inactive^i$
   then
    discard m;
   else
    if $m.g_{m.sender} > g_{m.sender}^i$
     then
      $Receive_{g_i^i}^i \leftarrow Receive^i$; $Receive^i \leftarrow \{m\}$;
      $Flow_{g_i^i}^i \leftarrow Flow^i$; $Flow^i \leftarrow \emptyset$;
      $G^i \leftarrow max(G^i, m.G)$; $g_i^i \leftarrow g_i^i + 1$;
      $G_{g_i^i}^i \leftarrow G^i$;
      take $c_{g_i^i}^i$;
     else
      $Receive^i \leftarrow Receive^i \cup \{m\}$;
    fi
    accept m *from* $m.sender$;
  fi

3.3 Recovery

If p_i fails, the recovery procedure is initiated. p_i is restarted from the most recent local checkpoint c_s^i where $s = g(p_i)$. In order to keep the system consistent, every p_j which includes e^j where $c_s^i \rightarrow e^j$ also has to be restarted. This is realized by using $Flow^i$ and the message diffusion protocol [2]. p_i has to send recovery request messages to every p_j to which p_i sends m after taking c_s^i. Here, p_j has to be restarted from the local checkpoint taken before the message-receiving event for m. In our protocol, only and all the processes to be restarted are recorded in $Flow^i$. Thus, p_i sends a recovery request message to each $p_j \in Flow^i$. Moreover, every m received after taking c_s^i has

to be accepted by the application again. Since $s(m)$ satisfies $c_s^i \not\rightarrow s(m)$, m becomes a lost message unless p_i accepts m again after the recovery. Thus, all the messages in $Receive^i$ are put into the buffer in p_i. p_i accepts the messages again in the procedure Receive(). Here, the failed process p_i is restarted asynchronously with the other processes.

On the other hand, if an operational process p_i receives a recovery request message, p_i also has to be rolled back. At first, p_i finds the most recent local checkpoint c_s^i to keep the system consistent after the recovery. This is realized by piggying back a generation vector $m_r.G = \langle m_r.g_1, \ldots, m_r.g_n \rangle$ named a *recovery vector* by each recovery request message m_r. If p_j is restarted from c_t^j, the value t is assigned to $m_r.g_j$ and p_j sends m_r to all the processes in $Flow^j$. Now, p_i finds the oldest local checkpoint c_s^i such that $m_r.g_j \leq (g_j^i)_s$ for every p_j. After that, p_i sends the recovery request messages and puts the received application messages to be accepted again after the recovery into the buffer as the failed process. Here, p_i that does not fail is also restarted from c_s^i asynchronously with the other processes.

Each p_i in which $r_{s,t}^i$ occurs adds a collection $\{s, \ldots, t\}$ to $Inactive^i$. If p_i receives m where $m.g_i \in \{s, \ldots, t\}$, p_i discards m. By using the procedure, no livelock occurs in the asynchronous recovery procedure.

The
following procedures Recovery-from-failure() and Recovery-for-consistency() are executed when p_i is recovered from the failure and p_i receives a recovery request message m_r, respectively:

Recovery-from-failure()
  $m_r.g_i \leftarrow g_i^i$; $m_r.g_j \leftarrow 0$ where $j \neq i$;
  foreach $p \in Flow^i$ **do**
    send m_r *to* p;
  od
  $Inactive^i \leftarrow Inactive^i \cup \{g_i^i\}$;
  put every $m \in Receive^i$ *to buffer*;
  $Receive^i \leftarrow \emptyset$; $Flow^i \leftarrow \emptyset$;
  $G^i \leftarrow G^i_{g_i^i}$; $g_i^i \leftarrow g_i^i + 1$; $G^i_{g_i^i} \leftarrow G^i$;
  take $c^i_{g_i^i}$; *restart* p_i;

Recovery-for-consistency()
  take m_r *out of* $\langle m_r.sender, p_i \rangle$;
  find the oldest G_s^i *where* $m_r.g_j \leq (g_j^i)_s$ *for every* j;
  $m_r.g_i \leftarrow s$;
  $Flow \leftarrow \{p | p \in Flow_u^i (s \leq u < g_i^i)$ or $p \in Flow^i\}$;
  foreach $p_j \in Flow$ **do**
    if $m_r.g_j = 0$
    then
      send m_r *to* p_j;
    fi
  od
  $Inactive^i \leftarrow Inactive^i \cup \{s, \ldots, g_i^i\}$;

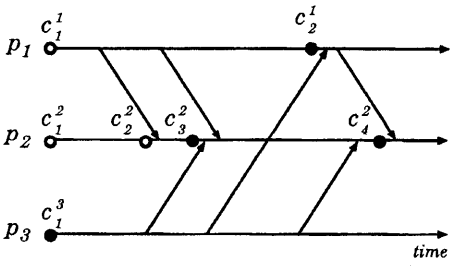

Figure 2: Garbage collection.

  put every $m \in Receive_u^i (s \leq u < g_i^i)$ *to buffer*;
  put every $m \in Receive^i$ *to buffer*;
  $Receive^i \leftarrow \emptyset$; $Flow^i \leftarrow \emptyset$;
  $g_i^i \leftarrow g_i^i + 1$; $g_j^i \leftarrow (g_j^i)_s$ where $j \neq i$; $G^i_{g_i^i} \leftarrow G^i$;
  take $c^i_{g_i^i}$; *restart*;

3.4 Garbage collection

In order to reduce the storage used for the information on the local checkpoints, a garbage collection procedure has to be executed to remove the obsolete checkpoints. If p_i would never be restarted from c_s^i both for recovering from the failure of p_i and for keeping the system consistent even in the failure of any $p_j \neq p_i$, c_s^i has to be discarded. Here, we also use the generation vector. Let $G_s^i = \langle (g_1^i)_s, \ldots, (g_n^i)_s \rangle$ be the checkpoint generation vector of c_s^i and $G^i = \langle g_1^i, \ldots, g_n^i \rangle$ be the generation vector manipulated in p_i. If $s \leq t$ for every t where $(g_j^i)_t = g_j^i \neq 0$ for some j, p_i is restarted from c_s^i when p_j fails, i.e. c_s^i is an active checkpoint. Otherwise, p_i will never be restarted from c_s^i. c_s^i is obsolete and p_i discards c_s^i.

Figure 2 shows an example. The checkpoint generation vectors for each c_s^i are as follows: $c_1^1 = \langle 1, 0, 0 \rangle$, $c_2^1 = \langle 2, 0, 1 \rangle$, $c_1^2 = \langle 0, 1, 0 \rangle$, $c_2^2 = \langle 1, 2, 0 \rangle$, $c_3^2 = \langle 1, 3, 0 \rangle$, $c_4^2 = \langle 2, 4, 1 \rangle$, $c_1^3 = \langle 0, 0, 1 \rangle$. If p_1 fails, p_1 and p_2 are restarted from c_2^1 and c_2^2, respectively. If p_2 fails, p_2 is restarted from c_4^2. p_1, p_2 and p_3 are restarted from c_2^1, c_2^2 and c_1^3, respectively, if p_3 fails. Thus, since c_1^1, c_1^2 and c_2^2 are obsolete, they have to be discarded.

By using the garbage collection procedure, p_i has only one checkpoint from which p_i is restarted if p_j fails and p_i and p_j are the same rollback domain.

Theorem 5 There are at most n active local checkpoints in each p_i. □

4 Evaluation

In the proposed protocol, since p_i discards any phantom checkpoint messages, p_i is restarted from the local checkpoint at most once for a failure of process. Hence, the protocol realizes the asynchronous livelock-free recovery. Here, $O(l)$ recovery request messages are transmitted for the recovery from the failure of p_i where l is the number

Table 1: Overhead.

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

of processes included in D_i.

Now, we would like to evaluate the overhead for the checkpointing procedure and the recovery procedure in the proposed protocol and the conventional synchronous protocol [7]. Table 1 shows the results. The conventional protocol is based on the two-phase commitment protocol both in the checkpointing procedure and the recovery procedure. Thus, the number of the additional messages 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. In our protocol, no additional message is transmitted in the checkpointing procedure. In the asynchronous recovery procedure, only $O(n)$ additional messages are transmitted where n is the number of processes included in a rollback domain of a failed process. The required time is $O(d)$ where d is the diameter of the rollback domain. Since $n \ll N$ and $d \ll D$ especially in a large-scale distributed system, our protocol reduces the overhead.

5 Concluding remarks

This paper has proposed the protocol for taking consistent regional checkpoints and recovering the processes asynchronously in distributed systems. For taking a consistent regional checkpoint, no additional messages are transmitted and no process is blocked. The number of processes required to take a local checkpoint is the minimum. Each process keeps at most n local checkpoints where n is the number of processes in the system. If some process fails, the minimum number of processes are restarted from the local checkpoints asynchronously with the other processes. The amount of execution of the application program wasted by the recovery is the minimum. Since the process is restarted at most once for a failure of process by discarding phantom checkpoint messages, the recovery protocol is livelock-free. To realize the recovery, each message contains a n-size vector of generation and only $O(l)$ messages are transmitted where l is the number of channels. Therefore, the reliable and available large-scale distributed systems can be easily and effectively developed and operated by using the protocol proposed in this paper.

References

[1] 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).

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

[3] Higaki, H. and Takizawa, M., "Group Communication Protocol for Flexible Distributed Systems," *Proc. of the 3rd International Conference on Network Protocols*, (1996).

[4] Higaki, H., Shima, K., Tachikawa, T. and Takizawa, M., "Checkpoint and Rollback in Asynchronous Distributed Systems," *IPSJ Technical Reports*, Vol. 96, No. 40, pp. 43–48 (1996).

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

[10] 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).

[11] 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).

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