Checkpoint and Rollback in Asynchronous Distributed Systems *

4 0 − 5

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

1 Introduction

Distributed applications are realized by cooperation of multiple processes executed in multiple computers. These processes are not always reliable and available. Checkpointing and rollback 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 executing 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. In the conventional methods, all the processes are synchronized by using such a protocol as the two-phase commitment protocol [1]. In this paper, we would like to discuss a new method where the processes are allowed to be asynchronously rolled back and restarted.

2 Checkpoint and Rollback

An asynchronous 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. $\langle p_i, p_j\rangle$ indicates a channel from p_i to p_j. 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.

2.1 Checkpoint

c^i_s represents the sth checkpoint taken by p_i. r^i_s represents a rollback where p_i is rolled back to c^i_s. If p_i fails at r^i_s, p_i is rolled back and restarted from c^i_s. c^i_s is *active* if p_i takes c^i_s and r^i_s does not occur. If p_i has an active checkpoint and p_i sends a message m to p_j, m is referred to as a *checkpoint message* of c^i to p_j. A global checkpoint is a set of checkpoints taken by all the processes in V, i.e., $\{c^1, \ldots, c^n\}$. If the processes take the checkpoints and are rolled back to the checkpoints independently of the other processes, there exist two kinds of inconsistent messages: *lost messages* and *orphan messages*. If p_i records the received messages in the log, lost messages can be received by taking them out of the log after p_i is rolled back. Hence, the global state of the system can be defined to be consistent iff there is no orphan message.

In the conventional checkpointing [2], if some process takes a checkpoint, all the processes are required to take checkpoints. Moreover, additional messages are transmitted and the processes are suspended during the checkpointing. However, all the processes are not always needed to take checkpoints. Here, we define a *semi-consistent* global state.

Definition (semi-consistent) 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 iff a global state S is semi-consistent for a set P of processes and only and all the processes in P are rolled back.

2.2 Rollback

In the conventional rollback [2], the processes have to be synchronized to be restarted. One of the disadvantages is that all the processes are suspended and additional messages are transmitted. 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, we would like to discuss a method where the processes are asynchronously restarted from the checkpoints.

Here, we would like to define the precedence relation among the active checkpoints and the rollback domain.

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

Definition (rollback domain) A *rollback domain* D^i contains only and all the following processes:

1) $p_i \in D^i$ if there is an active checkpoint c^i_s in p_i. Otherwise, $D^i = \emptyset$.
2) $p_j \in D^i$ if c^j_t is active in p_j and $c^j_t \Rightarrow c^k_u$ or $c^k_u \Rightarrow c^j_t$ where c^k_u is active in $p_k \in D^i$.

For each $p_j \in D^i$, $p_i \in D^j$ and $D^i = D^j$. A set $C = D^i$ of processes is referred to as a *rollback class*.

For every state S of $\langle V, L\rangle$ and a subset $V' \subseteq V$, let $S_{V'}$ denote a projection of S into V', i.e., a set of the local states of the processes in V'. Let $S_{V'}$ be a set of local states denoted by the active checkpoints taken by the processes in V'.

Theorem 1 For every C at every system state S, $S_{V-V'} \cup S_{V'}$ is semi-consistent for V' iff $C \subseteq V'$. □

If p_i in$C$ fails, the system state is semi-consistent for C if all the processes in C are rolled back. This also means that C is the minimum set of processes to be rolled back for keeping the system semi-consistent after the rollback.

Definition (rollback view) p_j is included in a p_i's *rollback view* W^i of D^i if p_i knows $p_j \in D^i$. □

p_i does not have the complete information on which processes are included in C. Thus, $W^i \subseteq D^i$. Based on the view W^i of p_i, p_i can be rolled back and restarted from the active checkpoint c^i_s.

If processes are rolled back independently of the other processes, C may not become empty and the rollback may be continued forever, i.e., the livelock occurs. Suppose that p_i sends a checkpoint message m^i at e^i after taking c^i_s and p_j sends a message m^j at e^j after receiving m^i as shown in Figure 1. Here suppose

---

[*] 非同期分散システムにおけるチェックポイントとロールバック
[†] 桧垣 博章　島 健司　立川 敬行　滝沢 誠
[‡] 東京電機大学

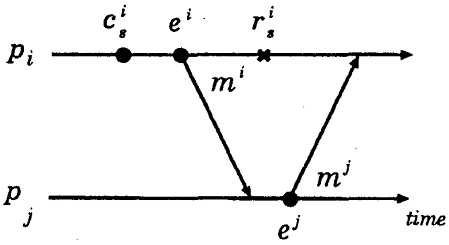

Figure 1: Livelock.

that 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 in p_j. Thus, livelock may occur. In order to identify the messages to be discarded, we define the *generation* of each process and each event as follows:

Definition (generation) The generation $g(p_i)$ is s between r_{s-1}^i and r_s^i. Before r_1^i occurs, $g(p_i) = 0$. If c_s^i is active and an event e occurs in p_i, the generation $g(e)$ is s. Otherwise, $g(e)$ is $\perp$ (unknown). □

3 Algorithm

In this section, we would like to present the algorithm using the vector clock of the generations for preventing the livelock. A message m contains the data $m.data$ and a vector clock $m.clock = \langle m.cl_1, \ldots, m.cl_n \rangle$ in the header. p_i manipulates the following variables. Here, let N^i be a set of neighbor processes of p_i.

- A *checkpoint clock* $c.CL^i = \langle c.cl_1^i, \ldots, c.cl_n^i \rangle$: Each $c.cl_j^i$ shows the generation of the active checkpoint in p_j that p_i knows. $c.cl_i^i$ is incremented by one each time p_i takes a checkpoint. Initially, $c.cl_i^i = 0$ and $c.cl_j^i = \perp$ for $j \neq i$.
- A *rollback clock* $r.CL^i = \langle r.cl_1^i, \ldots, r.cl_n^i \rangle$: Each $r.cl_j^i$ shows the generation of the rollback most recently occurred in p_j that p_i knows. That is, if p_i receives a message m where 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. $r.cl_j^i$ 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 rollback view W^i: p_i records an identifier of a neighbor process p_j in W^i if p_i has an active checkpoint c_s^i and p_i communicates with p_j. Initially, $W^i = \emptyset$.

p_i takes c_s^i if one of the following conditions is satisfied:

C1 If p_i decides to take a 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 from a neighbor process p_j of p_i, p_i takes c_s^i just before e.

By C1, checkpointing can be initiated by multiple processes. By C2, there is no orphan message in $\langle p_i, p_j \rangle$ and $\langle p_j, p_i \rangle$.

Table 1: Overhead.

	Checkpointing		Rollback	
	Message	Time	Message	Time
Koo & Toueg [2]	$O(N)$	$O(D)$	$O(N)$	$O(D)$
Ours	0	0	$O(n)$	$O(d)$

Here, we would like to present the checkpointing algorithm. Suppose that p_i and p_j are in C where p_i and p_j have active checkpoints c_s^i and c_t^j, respectively. Consider a case that 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 $e_j(m)$ would be canceled. 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 1) p_i has no active checkpoint and 2) $m.cl_k \neq \perp$ and $m.cl_k \leq r.cl_k^i$ for some k. □

If p_i fails, a rollback is initiated. The rollback is finished if C of p_i becomes empty. This is realized by using the message diffusion protocol. 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 is restarted from c_s^i. $r.CL^i$ is updated to be $c.CL^i$.

4 Evaluation

The algorithm has the following properties:

Theorem 2 The rollback is terminated. □

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

Here, we would like to evaluate the overhead for checkpointing and rollback by comparing with [2]. In Table 1, N is the number of processes in the system, D is the diameter of the system, n is the number of processes included in C and d is the diameter of C. Our algorithm reduces the overhead especially in a large-scale distributed system because $n \ll N$ and $d \ll D$ are satisfied.

5 Concluding Remarks

This paper has proposed the new algorithm for taking checkpoints and rolling back processes in asynchronous distributed systems. Each process manipulates $O(n)$ information and each message contains $O(n)$ information. 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.

References

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

[2] 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).