Vol. 37 No. 5 Transactions of Information Processing Society of Japan May 1996

Regular Paper

Group Communication for Upgrading Distributed Programs

HIROAKI HIGAKI† and YUTAKA HIRAKAWA†

Large-scale distributed systems are not always stable, because the environments and the user requirements are changeable. The authors previously discussed a "receptive platform" that makes distributed systems flexible and reliable. To realize such a platform, it is essential to consider how to upgrade distributed programs. Conventional upgrading methods require that multiple processes be suspended simultaneously. Thus, the availability of the system becomes low. The authors also proposed a new method called "dynamic upgrading" whose key ideas are that multiple versions of processes are allowed to co-exist temporarily and that the effects of protocol errors caused by the co-existence are concealed by rollback recovery. An algorithm was designed for treating unspecified receptions. However, it cannot treat communication deadlocks. This paper proposes an extended group communication algorithm that can detect and resolve communication deadlocks. By using the algorithm, dynamic upgrading can be applied to a wider range of distributed application programs.

1. Introduction

The development of computer and communication technology has recently led to the development of large-scale distributed systems. These systems are normally used continuously for a long period after their construction. However, these are not always stable, because the environments and the user requirements are changeable. We previously proposed a *receptive platform* that can support stable services in the presence of various kinds of changes [1]. To realize a receptive platform, it is important to provide procedure for upgrading distributed programs. One typical method for upgrading the system is to replace processes with new ones.

Even if the modified application programs are sufficiently verified and tested, the upgrading procedure should be carefully designed, because it may cause serious error execution. For protocol errors such as unspecified receptions and communication deadlocks, it is obviously safe to suspend the whole system temporarily, because conflicts between multiple versions of processes can be avoided. However, suspending the whole system reduces its availability, because a large number of processes have to be suspended simultaneously. Moreover, some critical applications require responsiveness and cannot accept such suspension. Therefore, a new method for upgrading the system is needed.

The rest of this paper is organized as follows: Section 2 describes related work. Section 3 gives an overview of dynamic upgrading. In Section 4, a novel group communication algorithm is proposed and its properties are discussed.

† NTT Software Laboratories

2. Related Work

A distributed system is upgraded by replacing its processes. An upgrading process is one that executes the upgrading procedure. The effects of the procedure can be classified into two types. One is a *direct* type that affects only the upgrading processes. An upgrading process is suspended temporarily during the process replacement. The other is an *indirect* type, in which the effects spread to other processes. This is a problem peculiar to distributed systems. Unspecified receptions and communication deadlocks are typical indirect effects of changes in protocol specifications. An unspecified reception occurs when a process receives a message that is not acceptable. Suppose that a process p waits for a message included in a set of messages $M = \{a, b, c\}$. If p receives a message $x \notin M$, an unspecified reception occurs. On the other hand, a communication deadlock occurs when a set of processes wait for messages from one another. Suppose a process p waits for a message from a process q. If q also waits for a message from p, a communication deadlock occurs.

Several methods have been proposed for completely avoiding unspecified receptions and communication deadlocks [2),3)]. A certain set of processes are suspended simultaneously when the following conditions are satisfied:

- The processes are free from processing requests from other processes.
- The communication channels between the processes are empty.

However, these methods can only be applied to applications in which the set of processes to be suspended is determined and is sufficiently small. Applications based on remote procedure call and client-server model are examples.

On the other hand, recent distributed applications executed in such systems as computer networks, multimedia communication systems, distributed control systems and multi-agent systems are classified as *partner-type* applications[4]. Each process computes and communicates autonomously, and is related to others in a complicated way because of the following properties:

- A process is simultaneously related to multiple processes.
- The relationships change dynamically.
- The related processes are on an equal footing.

Thus, it is difficult to obtain a stable state by suspending multiple processes simultaneously. This is because it is difficult to determine a set of processes to be suspended; even if the set is determined, it usually contains a large number of processes. Therefore, it is intrinsically difficult to apply conventional methods to partner-type applications[5].

We previously proposed a novel upgrading method called *dynamic upgrading*[6]. The basic concept is that it is sufficient to detect and resolve indirect effects. Thus, since multiple processes are not required to be suspended simultaneously, the availability of the system is kept high. We also designed the first version of a group communication algorithm in order to implement dynamic upgrading[7]. By using the algorithm, it is possible to detect and resolve unspecified receptions. However, it is required that the application programs satisfy the following restriction:

Restriction. An old-version process p and a new-version process p' execute sequences of events $\{e_0, e_1, e_2, \ldots\}$ and $\{e'_0, e'_1, e'_2, \ldots\}$, respectively. If $e_i \neq e'_i$ and $e_j = e'_j$ ($j = 0, \ldots, i-1$), one of the following should be satisfied:

- Both e_i and e'_i are message-sending events.
- Both e_i and e'_i are message-receiving events.

Because of the restriction, communication deadlocks are never caused by the co-existence of multiple versions of processes. However, only a few application programs satisfy the restriction when the existing functions are modified or when synchronous messages are added or removed. We would like to design the algorithm so that it can be applied to more general applications.

3. Dynamic Upgrading

3.1 Overview

The key idea of dynamic upgrading is that a system is allowed temporarily to consist of multiple versions of processes. If an unspecified reception or a communication deadlock is detected, the system restarts old-version processes from the checkpoints taken in the execution. The upgrading procedure is restarted later. **Figure 1** gives an overview.

- (Initial state → Transient state) Initially, each process group contains only an old-version process. For upgrading, a new-version process is invoked in the same process group. Each new-version process is invoked independently of the others. The old-version process continues as a backup process. Therefore, the system is in a transient state. There are two kinds of process groups: *upgrading* process groups, each of which contains multiple versions of processes, and *stable* process groups, each of which contains only one process.
- (Transient state) Messages are transmitted between process groups. In an upgrading process group, a new-version process transmits and receives signals containing a message. An old-version process observes these transmissions and receptions and takes a checkpoint for rollback recovery.
- (Transient state → Upgraded state) If all new-version processes are invoked without unspecified receptions or communica-

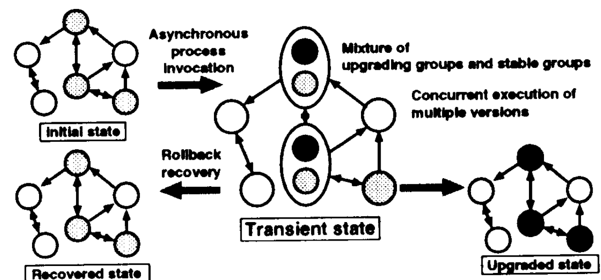

Fig. 1 Overview of dynamic upgrading.

tion deadlocks, the old-version processes are stopped and deleted. The upgrading procedure is finished☆.

- (Transient state → Recovered state) If an unspecified reception or a communication deadlock is detected, the upgrading procedure quits upgrading the system. The new-version processes are stopped and the old-version processes are restarted from the checkpoints taken in the execution. After some interval, the upgrading procedure is restarted.

3.2 System Model

This subsection describes the system model and some assumptions. A process consists of three layers. The top one is the application layer: this executes an application program that contains communication events. The bottom layer is the process communication layer, which supports point-to-point transmissions of signals between processes. Each signal is assigned one of the following type attributes: intergroup, eventinform, detection, and rollback.

The middle layer is the group communication layer, in which a process executes the group communication algorithm. It consists of tasks invoked by requirements for processing communication events from the application layer or by requirements for processing signals from the process communication layer. These tasks transform requirements for transmissions or receptions of messages into transmissions or receptions of signals, and vice versa. The following two primitive interfaces are defined for the application layer: a message-sending event $send(S)$, where $S = \cup_i(p_i, m_i)$, and a message-receiving event $receive(R)$, where $R = \cup_i(p_i, m_i)$. The meaning is that a message m_i is transmitted to a process p_i and m_i from p_i is acceptable, respectively.

We made the following assumptions:

A1. Functions in the process communication layer offer fully connected, reliable (no omission, no duplication, and no contamination), and FIFO (first-in first-out) communication channels.

A2. Process behavior in the application layer is modeled to be a deterministic finite state machine. State transitions are caused only by communication events that do not depend on timing.

☆ Higaki and Hirakawa[8] discuss the conditions for finishing the upgrading procedure.

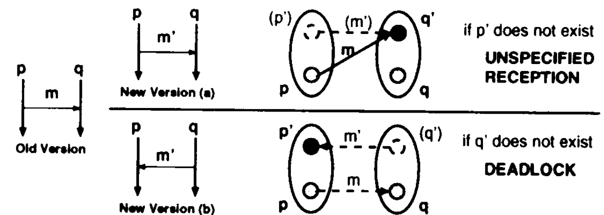

Fig. 2 Unspecified reception and communication deadlock.

A3. Neither unspecified receptions nor communication deadlocks occur in a system consisting of a single version of processes.

A4. At most two versions of processes coexist.

4. Group Communication

4.1 Requirements

The algorithm should satisfy the following three requirements:

R1. While multiple versions of processes are executing the application programs concurrently within a process group, the processes execute the same events in the same order.

R2. Every unspecified reception or communication deadlock is detected in finite time.

For example, **Fig. 2** shows specifications of a two-process system. In the old-version specification, a process p transmits a message m to a process q. In the new-version specification (a), p transmits a message m' to q. Suppose that a new-version process q' of q is invoked. An unspecified reception should be detected when q' receives m from p. In the new-version specification (b), q transmits m' to p☆☆. Suppose that a new-version process p' of p is invoked. A communication deadlock should be detected in finite time.

R3. A global state denoted by a set of checkpoints is consistent[9].

To ensure correct execution after rollback recovery, there should be no inconsistent message that a process transmits after taking a checkpoint and another process receives before taking a checkpoint.

4.2 Algorithm

This subsection explains the group communication algorithm. In order to reduce the overhead for rollback recovery, checkpoints should be taken as late as possible. The strategy is to take a checkpoint immediately before the first

☆☆ This is out of the restriction in Section 2.

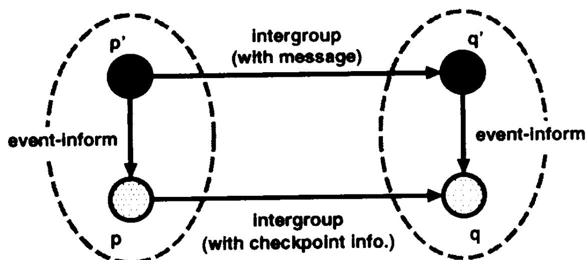

Fig. 3 Group communication algorithm.

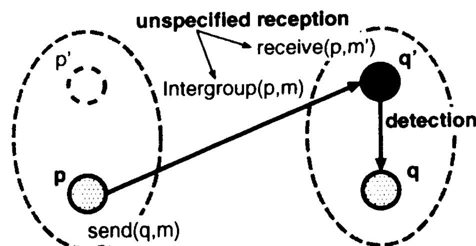

Fig. 4 Unspecified reception detection.

event e that satisfies one of the following conditions:

CP1. $e = e_i \neq e'_i$ and $e_j = e'_j$ ($j = 0, \ldots, i-1$) where an old-version process p and a new-version process p' execute sequences of events $\{e_0, e_1, e_2, \ldots\}$ and $\{e'_0, e'_1, e'_2, \ldots\}$, respectively.

CP2. $c_p \rightarrow e$ where c_p is a checkpoint and '$\rightarrow$' represents the causal relationship among events [10].

To take a checkpoint, an old-version process suspends the execution of the application program. On the other hand, a process in a stable process group records the current state and continues the execution of the application program. The algorithm for checking CP1 is CPAL1 and **Fig. 3**☆.

CPAL1. When a new-version process p' of p executes an event $send(S)$, where $S = \{(q, m)\}$, p' transmits a signal $signal(intergroup, p, m)$ to a new-version process q' of q. On receiving the signal, q' enqueues it to the signal queue. When q' executes an event $receive(R)$, where $R = \cup_i(p_i, m_i)$, q' dequeues $signal(intergroup, p, m)$ from the signal queue in FIFO order. If $(p, m) \in R$, m is delivered to the application layer. On executing an event $send(e)$ or $receive(e)$, where $e = (p, m)$, a new-version process transmits a signal $signal(eventinform, p, m)$ to an old-version process before the control returns to the application layer. On receiving the signal, the old-version process enqueues it in the signal queue. When the old-version process executes an event $send(S)$ or $receive(R)$, it dequeues a signal $signal(eventinform, p, m)$ from the signal queue in FIFO order. If $(p, m) \in S$ or $(p, m) \in R$, the control returns to the application layer after m is delivered to the application layer in the case of $receive(R)$. Otherwise, CP1 is satisfied. The old-version process takes a checkpoint immediately before this event.

Even if CPAL1 is executed, R3 may not be satisfied. Thus, CP2 is required to be checked [7]. When a message m is transmitted from a process p that has already taken a checkpoint to a process q, p informs q that m might become an inconsistent message, and q has to take a checkpoint.

CPAL2. A new-version process p' of p transmits an eventinform-type signal to p when p' executes an event $send(S)$, where $S = \{(q, m)\}$. On receiving the signal, p transmits an intergroup-type signal that does not contain any messages to q. By means of the signal, p informs q of whether or not p has taken a checkpoint.
When q executes a message-receiving event, q checks whether or not p has taken a checkpoint. If so, q also takes one immediately before the event.

Next, we would like to explain the method for detecting an unspecified reception and a communication deadlock. When one of the following conditions is satisfied, the system can no longer continue the upgrading procedure:

DT1. A new-version process in an upgrading process group or a process in a stable process group receives an unacceptable message.

DT2. A process in a stable process group receives no message before the timer expires.

If DT1 is satisfied, the unspecified reception occurs and the system should invoke rollback recovery. DTAL1, represented in **Fig. 4**, describes this algorithm.

DTAL1 When a process p that is a new-version process in an upgrading process group or a process in a stable process group executes an event $receive(R)$, where $R = \cup_i(p_i, m_i)$, p dequeues a signal

☆ For simplicity, we explain only the algorithm in an upgrading process group and omit the one in a stable process group.

$signal(intergroup, p, m)$ from the signal queue in FIFO order. If $(p, m) \notin R$, p transmits a detection-type signal to a corresponding old-version process and stops.

Figure 5 shows the specifications of a three-process system by means of time-space diagrams. Consider a case in which a new-version process p' of p is invoked and each process is at the $\otimes$ mark. Process p has already taken a checkpoint. At this moment, p' and q wait for messages from one another. Thus, the system is involved in a communication deadlock. The communication deadlock is detected by means of the timeout mechanism. As described later, at least one stable process group is involved in a communication deadlock. Thus, it is sufficient to use the timer when a process in a stable process group executes a message-receiving event. In the example, when the timer expires, q detects a communication deadlock. In order to resolve a communication deadlock, q informs p that p' should be stopped and p should restart from checkpoint c_p. DTAL2, represented in **Fig. 6**, describes this algorithm.

DTAL2. When a process p in a stable process group executes an event $receive(R)$, where $R = \cup_i(p_i, m_i)$, p starts the timer. If p dequeues an intergroup-type signal from the signal queue, p resets the timer and processes the signal. Otherwise, p transmits a detection-type signal to an old-version process in each p_i.

By means of DTAL2, a process r in Fig. 5 also transmits a detection-type signal to q. However, q ignores the signal, because it eventually detects a communication deadlock.

Consider a case in which an old-version process p is restarted from a checkpoint c_p. Old processes should be restarted from a consistent state. Thus, a new-version process q' of q that executes an event e satisfying $c_p \rightarrow e$ should be stopped and q should be restarted. The detailed rollback recovery algorithm using the message diffusion method [11] is described in our previous paper [7].

4.3 Example

This subsection gives a simple example to show that the processes take consistent checkpoints and detect communication deadlocks. **Figure 7** shows time-space diagrams of a duplicated sensor system. The system consists of three processes. One is a controller process (p_c). The others are sensor processes that are duplicated for reliability. One is a main sensor process (p_{sm}) and the other is a backup sensor process (p_{sb}). The system is usually in the main mode. Sometimes the sensor processes exchange roles and the system goes into the backup mode. The controller process first transmits start-type messages (m_s) to the sensor processes. The main sensor process informs the controller process of the current mode by means of a main-type message (m_m) or a backup-type message (m_b). Then, one of the sensor processes, namely, the main sensor process in the main mode or the backup sensor process in the backup mode, transmits data-type messages (m_d) to the other sensor process and

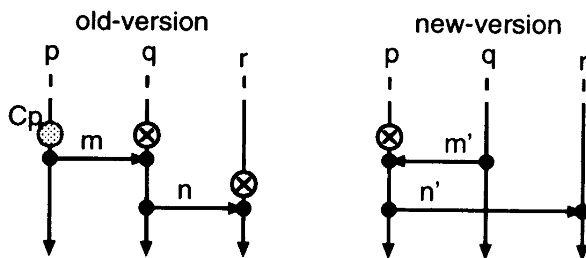

Fig. 5 Example of deadlock.

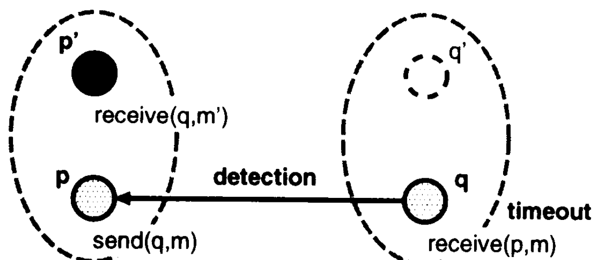

Fig. 6 Deadlock detection.

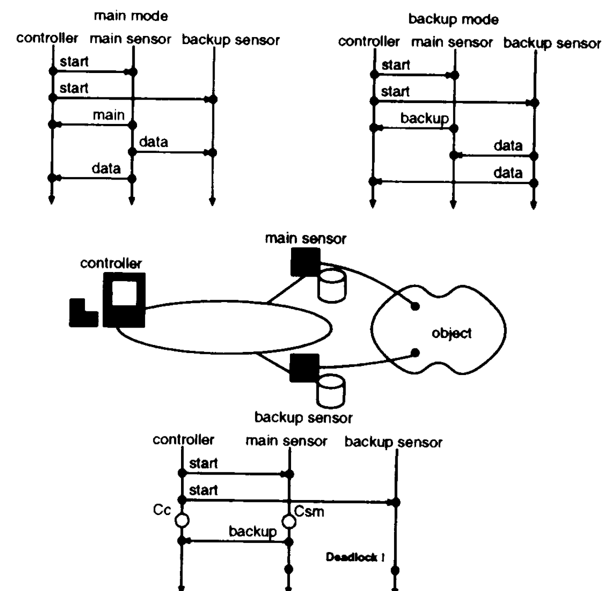

Fig. 7 Duplicated sensor system example.

the controller process. Consider a case in which the system is going to change from the main mode to the backup mode. The controller process and the main sensor process invoke new-version processes; however, the backup sensor process does not. The following execution sequence is then observed:

(1) Both p_c and p'_c execute $send(p_{sm}, m_s)$ and $send(p_{sb}, m_s)$.
(2) Both p_{sm} and p'_{sm} execute $receive(p_c, m_s)$.
(3) p_{sb} executes $receive(p_c, m_s)$.
(4) p'_{sm} executes $send(p_c, m_b)$.
(5) p_{sm} takes a checkpoint c_{sm} before $send(p_c, m_m)$ by CP1.
(6) p'_c executes $receive(p_{sm}, m_b)$.
(7) p_c takes a checkpoint c_c before $receive(p_{sm}, m_m)$ by CP1.
(8) p_{sb} executes $receive(p_{sm}, m_d)$ and the timer expires, because p'_{sm} executes $receive(p_{sb}, m_d)$. Thus, p_{sb} detects a communication deadlock by DT2 and requires p_{sm} to invoke rollback recovery.
(9) p_c and p_{sm} restart from consistent checkpoints c_c and c_{sm}, respectively.

4.4 Properties

The proposed algorithm satisfies the following properties:

P1. The set of checkpoints denotes the most recent consistent global state where no further rollback recovery is needed.

P2. The minimum number of processes must invoke rollback recovery.

P3. A system that is not involved in a communication deadlock might be considered to be in a communication deadlock.

The system must not become inconsistent even in the presence of premature rollback recovery. Thus, if a process q executes an event $receive(R)$, where $R = \cup_i(p_i, m_i)$, the algorithm should support the following cases:

- When the timer expires, an intergroup-type signal is in the communication channel between p_i and q.
- When p_i receives a detection-type signal, p_i has not taken a checkpoint.

Therefore, p_i and q should negotiate before rollback recovery.

(1) On receiving a detection-type signal, p_i transmits an acceptance-type signal to q if p_i has taken a checkpoint. Otherwise, p_i transmits a rejection-type signal to q.
(2) If q does not receive any intergroup-type signal and q receives an acceptance-type signal from p_i, q transmits an acknowledgment-type signal to p_i. Otherwise, q transmits a negative acknowledgment-type signal to p_i.
(3) q restarts the timer.
(4) On receiving an acknowledgment-type signal, p_i invokes rollback recovery.

We would like to prove that the algorithm satisfies R2 and P2. It has been proved that the algorithm satisfies the other requirements and properties in Ref. 7).

Theorem 1 Each deadlocked process p, which is a new-version process in an upgrading process group or a process in a stable process group, executes a message-receiving event.

Proof. A set of processes are involved in a communication deadlock if and only if the processes wait for signals from one another. Thus, p waits for an intergroup-type signal containing a message m_i from p_i. That is, p executes an event $receive(R)$, where $R = \cup_i(p_i, m_i)$. Therefore, the theorem is proved. □

Theorem 2 At least one deadlocked process is in a stable process group.

Proof. Suppose that all the deadlocked processes are in upgrading process groups. Form Theorem 1, all the new-version processes execute message-receiving events and wait for messages from one another. This contradicts A3. Thus, at least one process is a stable process. □

Theorem 3 (R2) Any communication deadlock is detected eventually.

Proof. A communication deadlock is detected when the timer expires. An old-version process starts the timer when it executes a message-receiving event. From Theorems 1 and 2, at least one deadlocked process is in a stable process group and executes a message-receiving event. Therefore, the stable process starts the timer and eventually detects a communication deadlock. □

In the following theorems, a deadlocked process q is in a stable process group and waits for a message from a process p.

Theorem 4 In order to resolve a communication deadlock, it is necessary that p be restarted from the checkpoint.

Proof. q should receive a message from p to resolve the communication deadlock. That is, p should execute an event $send(S)$, where $(q, m) \in S$. However, p is also involved in the communication deadlock, and

executes a message-receiving event. Thus, if p is not restarted from the checkpoint, the message-sending event is never executed and the communication deadlock is not resolved. Therefore, the theorem is proved.□

Theorem 5 In order to resolve a communication deadlock, it is sufficient that p be restarted from the checkpoint.

Proof. Suppose that the communication deadlock is not resolved. Since the old-version process of p restarts from the checkpoint, an event $send(S)$, where $(q,m) \in S$, is executed and q is out of the communication deadlock. Thus, the number of process groups involved in the communication deadlock is reduced. Therefore, the communication deadlock is eventually resolved. □

Theorem 6 (P2) To resolve a communication deadlock, the minimum number of processes must be restarted from the checkpoints.

Proof. From Theorems 4 and 5, a communication deadlock is resolved if and only if p is restarted from the checkpoint. It is proved in Higaki[7] that the algorithm requires the minimum number of processes to be restarted from the checkpoints in order for p to be restarted consistently. Therefore, this theorem is proved. □

5. Conclusion

This paper has described a method for upgrading distributed programs, called dynamic upgrading. It does not require multiple processes to be suspended simultaneously. Therefore, the availability of the system is kept high. We also designed an extended group communication algorithm. By means of the algorithm, unspecified receptions and communication deadlocks are detected and resolved. Therefore, the method can be applied to a wide range of distributed applications. The dynamic upgrading algorithm will play an important role in the construction of flexible and reliable distributed systems.

Acknowledgments The authors greatly appreciate the encouragement and suggestions provided by Dr. Haruhisa Ichikawa of NTT Multimedia Business Department, Mr. Atsushi Terauchi of NTT Software Laboratories, Professor Makoto Takizawa of Tokyo Denki University, and Professor Yoshitaka Shibata of Toyo University.

References

1) Higaki, H., Moriyasu, K., Okuyama, H., Hirakawa, Y. and Ichikawa, H.: Receptive Platform - A System Evolution Environment, *Proc. 47th Annual Conventions IPS Japan (1)*, pp. 201-202 (1993).
2) Segal, M.E. and Frieder, O.: Dynamically Updating Distributed Software: Supporting Change in Uncertain and Mistrustful Environments, *Proc. IEEE Conf. on Software Maintenance*, pp. 254-261 (1989).
3) Kramer, J. and Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management, *IEEE Trans. Softw. Eng.*, Vol.16, No.11, pp.1293-1306 (1990).
4) Yoshida, N.: Towards Next-Generation Parallel/Distributed System Development, *Journal of Computer Science*, Vol.2, No.4, pp.300-305 (1992).
5) Barbacci, M.R., Doubleday, D.L. and Weinstock, C.B.: Application Level Programming, *Proc. 9th ICDCS*, pp.458-465 (1990).
6) Higaki, H.: Dynamically Updating in Distributed Systems, *Proc. 46th Annual Conventions IPS Japan (1)*, pp.195-196 (1993).
7) Higaki, H.: Group Communications Algorithm for Dynamically Updating in Distributed Systems, *IEICE Trans. on Information and Systems*, Vol.E78-D, No.4, pp.444-454 (1995).
8) Higaki, H. and Hirakawa, Y.: Dynamically Updating Technique in Distributed Systems, *Proc. 49th Annual Conventions IPS Japan (1)*, pp.289-290 (1994).
9) Chandy, K.M. and Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. on Computer Systems*, Vol.3, pp.63-75 (1985).
10) Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM*, Vol.21, No.7, pp.558-565 (1978).
11) Moses, Y. and Roth, G.: On Reliable Message Diffusion, *Proc. 8th PODC*, pp.119-127 (1989).

Appendix

The algorithm is now described in detail. Each process stores the following information structures: three FIFO signal queues (E-queue, R-queue, and RB-queue), one signal buffer (P-buffer), two sets of process group identifiers (CP and RB) and a variable for version information (Vop). Each signal contains the following four header fields and a message field: the message identifier field, the signal type field, the checkpoint flag field and the event type field. The value of the signal type field is one of the fol-

lowing: I (intergroup), E (eventinform), D (detection), or R (rollback). The checkpoint flag field contains T (true) or F (false) and the event type field contains S (send) or R (receive).

```
Send(S): S=(q,m)
if Vop=single then
if CP= ∅ then
send(all(q),(mid,I,F,,p,m));
else
send(all(q),(mid,I,T,,p,m));
append(CP,q);
fi
else if Vop=new then
send(new(q),(mid,I,,,p,m));
send(old(p),(mid,I,,S,q,m));
else
s:=dequeue(E-queue);
if event_s =S then
if (group_s,message_s)=(q,m) then
send(old(q),(mid_s,I,F,,p,m));
else
append(CP,{p,group_s});
send(old(group_s),(mid_s,I,T,,group_s,message_s));
p takes a checkpoint;
fi
else p takes a checkpoint;
fi fi
Receive(R): R= ∪_i(q_i,m_i)
if Vop=single then
if the timer expires then
send(old(∀q_i),(,D,,,,));
else
p restarts the timer;
s:=dequeue(E-queue);
if checkpoint_s =T then
if CP= ∅ then
append(CP,p);
p takes a checkpoint;
fi
append(CP,group_s);
if (group_s,message_s)∈R then
deliver(group_s,message_s);
else
send(old(∀CP),(,R,,,,));
append(RB,p);
fi
else
deliver(group_s,message_s);
if CP≠ ∅ then
enqueue(RB-queue,s);
fi fi fi
else if Vop=new then
s:=dequeue(E-queue);
if (group_s,message_s)∈R then
deliver(group_s,message_s);
send(old(p),(mid_s,E,,R,group_s,message_s));
else
send(old(p),(,D,,,,));
p stops;
fi
else
s:=dequeue(E-queue);
if event_s =R then
if checkpoint_s = ∅ then
p is suspended until receiving ws where mid_ws = mid_s;
remove(P-buffer,ws);
checkpoint_s :=checkpoint_ws;
fi
if checkpoint_s =T then
append(CP,{p,group_s});
p takes a checkpoint;
else
deliver(group_s,message_s);
fi
else
append(CP,{p,group_s});
send(old(group_s),(mid_s,I,T,,p,message_s));
p takes a checkpoint;
fi fi
SignalReception(s){intergroup}
if Vop=single then
if checkpoint_s ≠ ∅ then
enqueue(E-queue,s);
fi
else if Vop=new then
enqueue(R-queue,s);
else
if checkpoint_s = ∅ then
send(new(p),s);
else if (∃qs∈R-queue) mid_qs =mid_s then
checkpoint_qs :=checkpoint_s;
else if CP= ∅ then
append(P-buffer,s);
else if checkpoint_s =T then
append(CP,group_s);
else
enqueue(RB-queue,s);
fi fi
SignalReception(s){eventinform}
if CP= ∅ then
enqueue(E-queue,s);
if (∃bs∈P-buffer) mid_bs =mid_s then
checkpoint_s :=checkpoint_bs;
fi
remove(P-buffer,bs);
else if event_s =S then
append(CP,group_s);
```

send(old(group$_s$),(mid$_s$,I,T,,p,message$_s$));
fi
SignalReception(s){detection}
if CP= ∅ **then**
append(RB,p);
send(old(∀CP−p),(,R,,,,));
if RB=CP **then**
p moves all the signals in RB-queue to R-queue;
p is restarted;
fi
else *p moves all the signals in RB-queue to R-queue*;
p is restarted;
fi
SignalReception(s){rollback}
if Vop=single **then**
if RB= ∅ **then**
send(old(∀CP−p),(,R,,,,));
append(RB,p);
fi
append(RB,group$_s$);
if RB=CP **then**
p moves all the signals in RB-queue to R-queue;
p is restarted;
fi
else if Vop=new **then**
send(old(p),(,D,,,,));
p is stopped;
else
if RB= ∅ **then**
append(RB,group$_s$);
send(new(p),(,R,,,,));
else
append(RB,group$_s$);
if RB=CP **then**
p moves all the signals in RB-queue to R-queue;
p is restarted;
fi fi fi

(Received September 22, 1995)
(Accepted February 7, 1996)

Hiroaki Higaki was born in Tokyo, Japan, on April 6, 1967. He received the B.E. degree from the Department of Mathematical Engineering and Information Physics, the University of Tokyo in 1990. Since joining NTT Software Laboratories in 1990, he has been engaged in research regarding distributed algorithms and distributed operating systems. He received IPSJ Convention Award in 1995. He is a member of ACM and IEICE.

Yutaka Hirakawa was born in Hamamatsu, Japan, on February 28, 1956. He received the B.E., M.E., Ph.D. degrees in electrical engineering from Kobe University in 1978, 1980, and 1991, respectively. He is currently a research group leader in NTT Software Laboratories. His research interests includes distributed systems, specification description techniques and their applications to multimedia systems. He is a member of IEEE Computer Society and IEICE.