Extended Upgrading Protocol for Flexible Networks

Hiroaki Higaki, Motoki Sakai and Makoto Takizawa
Department of Computers and Systems Engineering
Tokyo Denki University

The network systems are required to be so *flexible* that the system adapts to the changeable user-requirements and environments. Hence, the method for upgrading the application processes in the operational networks is one of the most important and critical technologies. The authors have proposed a novel upgrading method named ***dynamic upgrading*** and designed a group communication protocol. By using the protocol, even while the application processes are being upgraded, the network system is kept highly available. However, the protocol cannot be applied to large-scale network systems. This paper proposes an extended protocol for achieving flexible large-scale network systems.

やわらかい大規模ネットワーク構築のための拡張動的改版プロトコル

桧垣 博章　酒井 基樹　滝沢 誠
{hig, sake, taki}@takilab.k.dendai.ac.jp
東京電機大学理工学部経営工学科

時々刻々と変化するシステム環境やユーザ要求に対応するために、やわらかいネットワークシステム構築技術が提案されている。我々はこれまでに、新しい要求を満足するネットワークアプリケーションをシステムに導入する際に、その導入手続きによるユーザに対するサービスの停止や遅延を最小限にする手法として動的改版手法を提案し、それを実現するためのグループ通信プロトコルおよびチェックポイント／リスタートプロトコルを設計してきた。ところが、このプロトコルを用いた場合、ひとつの改版手続きが終了するまで別の改版手続きを開始することができないため、大規模な分散システムへの適用が困難であるという問題がある。本論文では、複数の改版手続きを並行に実行することが可能となるように動的改版プロトコルの拡張を行なう。拡張プロトコルを用いた場合にも、プロトコルエラー検出時にリスタートする際にそのオーバヘッドが最小となるという性質は保たれている。

1 Introduction

The distributed systems are including a large number of computers interconnected by communication networks. Since it is expensive to newly design and implement the system each time the system environments and the requirements are changed in the system, the system is required to be flexible [12]. That is, the system has to absorb the changes of the requirements and the system environments. One way to make the system flexible is to change the application processes in the system so that the new requirements are satisfied in the new environments, i.e. ***upgrading*** the system. The system can be typically upgraded by replacing the old-version application processes with the new-version ones. However, the system has to be suspended in the upgrading procedure. Therefore, we discuss a mechanism to make the system so flexible that the system could continue to support the services for the users while the component application processes are being upgraded.

Even if the new-version application processes are sufficiently verified and tested, the upgrading procedure should be carefully designed in order not to cause such protocol errors as the unspecified receptions and the communication deadlocks. The protocol errors can be obviously avoided by temporarily suspending the whole system because there are only single versions of processes and there is no conflict between multiple versions of processes. However, the system becomes less available because the system is suspended. Moreover, if an application requires ***responsiveness*** [10], the system cannot be suspended. Therefore, the application processes are required to be upgraded without suspending the system. This paper discusses a highly available method for upgrading the system where multiple versions of processes can exist simultaneously. If both old and new versions of processes are simultaneously operational, the processes may receive unacceptable messages or the communication deadlocks may occur. In order to resolve such protocol errors, group communication protocol for taking checkpoints and detecting the protocol errors is proposed [4]. That is, if the protocol errors are detected, the new-version processes are removed and the old-version ones are restarted from the checkpoints. However, in the proposed group communication protocol,at most two versions of the application processes are simultaneously operational. Hence, multiple upgrading procedures cannot be done concurrently. Therefore, it is difficult to apply the protocol for realizing the large-scale flexible network systems. In this paper, we discuss a group communication protocol extended for solving this problem.

The rest of this paper is organized as follows. Section 2 reviews briefly the methods for upgrading and overviews the dynamic upgrading. In section 3, a checkpointing protocol is presented. In section 4, we evaluate the extended protocol by comparing with the original protocol.

2 Dynamic Upgrading

The system is upgraded by replacing the old-version processes with the new-version ones.

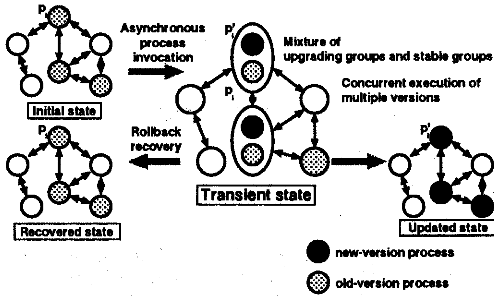

Figure 1: Overview of dynamic upgrading.

While the upgrading procedure is executed, there may exist some problems. First, each *upgrading* process, which is being replaced, is suspended during the process replacement. Thus, the services cannot be supported for the users. The shorter the suspension time is, the more highly available the system is. In addition, there may occur two kinds of the *protocol errors* in the inter-process communication; the *unspecified receptions* and the *communication deadlocks*. The unspecified reception occurs if a process receives a message which cannot be accepted. For example, suppose that a process p_i waits for receiving one of the messages included in a set of messages $M = \{a, b, c\}$. If p_i receives $x \notin M$, the unspecified reception occurs. On the other hand, the communication deadlock occurs if multiple processes wait for receiving messages from each other. Suppose that p_i waits for a message from p_j. If p_j also waits for a message from p_i, the communication deadlock occurs.

Several methods [8, 11] have been proposed so far for avoiding the unspecified receptions and the communication deadlocks. In these methods, the system is upgraded in a stable state where a certain subset of processes are suspended simultaneously. Hence, these methods can be applied to only the restricted applications where a set of processes to be suspended can be easily determined and is small. On the other hand, significant distributed applications as computer networks, multimedia communication systems, distributed control systems and multi-agent systems are classified as *partner type* [14]. Here, each process does autonomously the computation and communication. Thus, it is difficult to obtain a stable state where multiple processes are suspended simultaneously. Therefore, it is intrinsically difficult to apply these methods to the partner type applications [1].

A novel *dynamic upgrading* method is proposed [3, 6, 7]. Here, the system is allowed to temporarily include multiple versions of processes. That is, an old-version process p_i and a new-version process p'_i co-exist. A process group P_i is composed of p_i and p'_i. The protocol errors are detected and resolved instead of being avoided completely. In order to conceal the effects of the protocol errors from the execution of the application programs, the group communication protocol and the checkpoint-restart protocol are combined [4]. If the protocol error is detected, the system restarts the old-version processes from the checkpoints taken in the execution. After that, the upgrading procedure is restarted. Since multiple processes are not required to be suspended simultaneously in this method, the system is highly available even while the upgrading procedure is executed. Figure 1 depicts an overview of the dynamic upgrading procedure. Here, a collection of process groups $P_1, \ldots, P_n$ cooperate to execute the application programs.

- Initial state: Each process group P_i contains only an old-version process p_i. Application messages are transmitted among the old-version processes.
- Transient state: If p_i would be upgraded, a new-version process p'_i of p_i is invoked in the process group P_i independently of the other process groups. Here, the system is in a *transient* state. There are two kinds of process groups, i.e. *upgrading* process groups which contain multiple versions of processes, and *stable* process groups which contain only one process. p'_i starts to execute the application program while p_i still continues to execute the application program passively. That is, p'_i sends and receives application messages with the other process groups and p_i receives the messages forwarded by p'_i but does not send the messages. Moreover, p_i takes a local checkpoint from which p_i is restarted for the recovery from the protocol error.
- Upgraded state: If all the new-version processes are invoked, the old-version processes are removed. Here, the system consists of only the new-version processes. The upgrading procedure is finished.
- Recovered state: If the protocol error is detected in the transient state, the new-version processes are stopped and the old-version processes are restarted from the local checkpoints taken in the transient state.

Here, a collection of the local checkpoints taken by a subset of processes has to denote a *semi-consistent* global state [7].

Definition Let B be a subset of the processes in the system. A global state is *semi-consistent* for B iff there is no orphan message [2] for every communication channel of each process in B. □

Then, the upgrading procedure is restarted from the initial state after some interval[1].

3 Extended Protocol

3.1 System model

A process is assumed to consist of three layers as shown in Figure 2. The top layer is the application layer where an application program is executed by exchanging the application messages. The bottom layer is the process communication layer which supports a pair of two processes p_i and p_j at the group communication layer with the reliable point-to-point system message transmission. A unit of data exchanged among the group communication processes is a system message. Each system message is assigned one of the following type attributes; *intergroup*, *event-inform*, *detection* and *restart*. At the group communication layer, the group communication protocol and the

[1] The procedure for transition from the transient state to the upgraded state is discussed in [5].

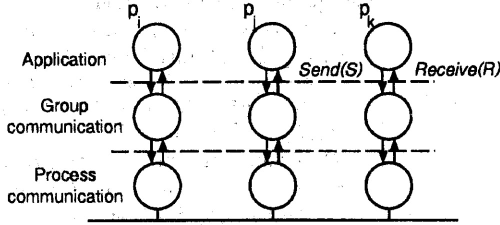

Figure 2: Three-layer system.

checkpoint-restart protocol are implemented. The following primitives in the process p_i are defined for the application layer:

- A message sending event $send(S)$ where $S = \cup_j\{(p_j, m_j)\}$: This means that p_i sends a message m_j to p_j.
- A message receiving event $receive(R)$ where $R = \cup_j\{(p_j, m_j)\}$: This means that p_i can accept a message m_j from p_j.

Here, we make the following assumptions:

A1 The process communication layer supports reliable FIFO (first-in first-out) communication channels. Thus, any system message is neither lost, duplicated nor contaminated.

A2 State transitions in the application layer are caused by the communication events, i.e. message sending and receiving events.

A3 No protocol error occurs in a system consisting of only single-version processes.

A4 In every process groups, new version processes are created in the same order.

Suppose that p_i^l and p_i^m are lth and mth versions of p_i and p_j^l and p_j^m are lth and mth versions of p_j. A4 means that if $l < m$, p_i^l is created before p_i^m and p_j^l is created before p_j^m. On the other hand, suppose that p_i^l and p_i^m are lth and mth versions of p_i, p_j^m and p_j^n are mth and nth versions of p_j, and p_k^l and p_k^n are lth and nth version of p_k. If p_i^l is created before p_i^m and p_j^m is created p_j^n, p_k^l has to be created before p_k^n.

3.2 Requirement

The protocol for the dynamic upgrading is required to satisfy the following requirements. Here, a process group P_i consists of processes p_i^l.

R1 While multiple versions of processes p_i^l are executing the application programs concurrently in P_i, the same events occur in the same order in every p_i^l.

R2 The protocol errors, i.e. the unspecified receptions and the communication deadlocks are detected and resolved in finite time.

R3 A global state denoted by a set of local checkpoints taken are consistent.

3.3 Checkpointing protocol

In this subsection, we discuss the group communication protocol that satisfies the requirements in the previous subsection. The *happened before* relation "$\rightarrow$" among the events is defined as follows [9]:

Definition An event e_i *happens before* an event e_j ($e_i \rightarrow e_j$) iff one of the following conditions is satisfied:

- e_i happens before e_j in the same process.
- e_i is an event for sending a message m and e_j is one for receiving m.
- $e_i \rightarrow e_k \rightarrow e_j$ for some event e_k. □

In order to reduce the overhead for the recovery from the protocol errors, each local checkpoint should be taken as late as possible. Here, suppose that a process group P_i consists of processes $\{p_i^l, \dots, p_i^m\}$ where p_i^m is the newest version in P_i. Our strategy is that each old-version process p_i^n ($\neq p_i^m$) takes a local checkpoint c_i^n immediately before the first event $e_r^{n,i}$ that satisfies one of the following conditions:

CP1 $e_r^{n,i} \neq e_r^{m,i}$ and $e_s^{n,i} = e_s^{m,i} (r < s)$ where event sequences $\langle e_0^{n,i}, e_1^{n,i}, e_2^{n,i}, \dots\rangle$ and $\langle e_0^{m,i}, e_1^{m,i}, e_2^{m,i}, \dots\rangle$ occur in each process p_i^n and p_i^m, respectively.

CP2 $c_j^o \rightarrow e_r^{n,i}$ where c_j^o is a local checkpoint of p_j^o in a process group P_j and $o \geq n$.

CP3 $c_i^q \rightarrow e_r^{n,i}$ where c_i^q is a local checkpoint of p_i^q which is newer version process than p_i^n in P_i, i.e. $q \geq n$.

CP1 means that each old-version process p_i^n takes a local checkpoint c_i^n just before the first event that does not occur in the newest version process p_i^m in P_i. By applying CP2, there is no inconsistent message in the global checkpoint denoted by a set of the local checkpoints. If CP3 is not applied, the system cannot be restarted consistently. In Figure 3, three versions of processes p_i^2, p_i^1 and p_i^0 is in a process group P_i. Suppose that $e_0^{2,i} = e_0^{1,i} = e_0^{0,i}$, $e_1^{2,i} = e_1^{0,i} \neq e_1^{1,i}$ and $e_2^{2,i} \neq e_2^{0,i} \neq e_2^{1,i}$. According to CP1 and CP2, p_i^1 and p_i^0 take local checkpoints c_i^1 and c_i^0 just before $e_1^{1,i}$ and $e_2^{0,i}$, respectively. If some protocol error is detected and p_i^1 is restarted from c_i^1, p_i^0 has to take a local checkpoint c_i^0 just before $e_1^{0,i}$ for CP1. However, p_i^0 has already taken c_i^0 just before $e_2^{0,i}$. Hence, if another protocol error is detected and p_i^0 is restarted, the system becomes inconsistent. By using CP3, since p_i^0 takes c_i^0 just before $e_1^{0,i}$, the system is kept consistent even in the above restart scenario. In order to realize the minimum overhead recovery, if p_i^n takes c_i^n according to CP3 and p_i^{n+1} is restarted, p_i^n is also restarted and executes the application program passively. The modified recovery protocol is discussed in the following subsection.

When p_i^n takes a local checkpoint c_i^n in an upgrading process group P_i, p_i^n is suspended until the protocol error is detected and p_i^n is restarted from c_i^n. On the other hand, if p_i^n is in a stable process group, i.e. P_i consists of only p_i^n, p_i^n records the current state information in a stable log and continues to execute the application

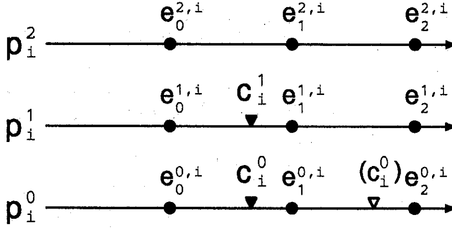

Figure 3: Checkpoint by CP3.

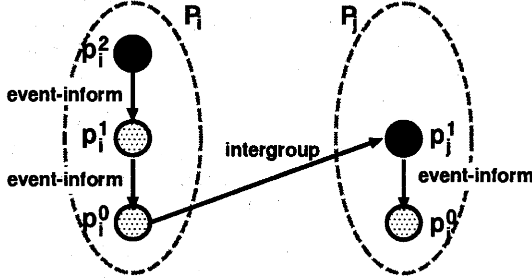

Figure 4: Group communication protocol.

program. The group communication protocol for transmitting an application message m from P_i and to P_j is as follows (Figure 4). Here, P_i and P_j consist of $\{p_i^0, \ldots, p_i^{n_i}\}$ and $\{p_j^0, \ldots, p_j^{n_j}\}$, respectively. Each system message $m_s(P_k, m, flag, ver)$ carries a process group identifier P_k, an application message m, a binary flag $flag$ and a version information ver which is used for checking CP2 [2]. On receiving a system message m_s from another process through the process communication layer, a process p_k enqueues m_s to the system message queue.

[Communication protocol]

1) When a message sending event $Send(S)$ where $S = \{(P_j, m)\}$ occurs in the newest version process $p_i^{n_i}$ in P_i, $p_i^{n_i}$ sends an event-inform type system message $m_s(P_j, m, F, \perp)$ to $p_i^{n_i-1}$ in P_i.

2) On receiving an event-inform type system message $m_s(P_j, m, flag, ver)$, $p_i^k (0 < k < n_i)$ in P_i enqueues it to the system message queue in FIFO order.

 2–1) If p_i^k has a local checkpoint c_i^k, p_i^k dequeues $m_s(P_j, m, flag, ver)$ from the message queue in FIFO order and sends an event-inform type system message $m_s(P_j, m, T, max(ver, k))$ to p_i^{k-1}.

 2–2) Otherwise, i.e. if p_i^k does not have a local checkpoint c_i^k and a message sending event $Send(S)$ where $S = \{P_l, m_l\}$ occurs in p_i^k, p_i^k dequeues an event-inform type system message $m_s(P_j, m, flag, ver)$ from the system message queue in FIFO order.

  2–2–1) If $flag = T$, p_i^k takes a local checkpoint c_i^k just before $Send(S)$ according to CP3 and sends an event-inform type system message $m_s(P_j, m, T, ver)$ to p_i^{k-1}.

  2–2–2) If $flag = F$ and $(P_j, m) \in S$, p_i^k sends an event-inform type system message $m_s(P_j, m, F, ver)$ to p_i^{k-1}.

  2–2–3) Otherwise, i.e. if $flag = F$ and $(P_j, m) \notin S$, p_i^k takes a local checkpoint c_i^k just before $Send(S)$ according to CP1 and sends an event-inform type system message $m_s(P_j, m, T, k)$ to p_i^{k-1}.

3) On receiving an event-inform type system message $m_s(P_j, m, flag, ver)$, p_0^k in P_i enqueues it to the system message queue in FIFO order.

 3–1) If p_i^0 has a local checkpoint c_i^0, p_i^0 dequeues $m_s(P_j, m, flag, ver)$ from the message queue in FIFO order and sends an intergroup type system message $m_s(P_i, m, T, max(ver, 0))$ to $p_j^{n_j}$ in P_j.

 3–2) Otherwise, if p_i^0 does not have a local checkpoint c_i^0 and a message sending event $Send(S)$ where $S = \{P_l, m_l\}$ occurs in p_i^0, p_i^0 dequeues an event-inform type system message $m_s(P_j, m, flag, ver)$ from the system message queue in FIFO order.

  3–2–1) If $flag = T$, p_i^0 takes a local checkpoint c_i^0 just before $Send(S)$ according to CP3 and sends an intergroup type system message $m_s(P_i, m, T, ver)$ to $p_j^{n_j}$ in P_j.

  3–2–2) If $flag = F$ and $(P_j, m) \in S$, p_i^0 sends an intergroup type system message $m_s(P_i, m, F, ver)$ to $p_j^{n_j}$ in P_j.

  3–2–3) Otherwise, i.e. if $flag = F$ and $(P_j, m) \notin S$, p_i^0 takes a local checkpoint c_i^0 just before $Send(S)$ according to CP1 and sends an intergroup type system message $m_s(P_i, m, T, 0)$ to $p_j^{n_j}$ in P_j.

4) When a message receiving event $Receive(R)$ where $R = \cup_l\{(P_l, m_l)\}$ occurs in the newest version process $p_j^{n_j}$ in P_j, $p_j^{n_j}$ dequeues an intergroup type system message $m_s(P_i, m, flag, ver)$ from the system message queue in FIFO order. If $flag = T$, $ver \geq n_j$ and $p_j^{n_j}$ does not have a local checkpoint $c_j^{n_j}$, $p_j^{n_j}$ takes $c_j^{n_j}$ just before $Receive(R)$ according to CP2.

 4–1) If $(P_i, m) \in R$, m is delivered to the application layer. At the same time, $p_j^{n_j}$ sends an event-inform type system message $m_s(P_i, m, F, ver)$ to $p_j^{n_j-1}$ in P_j.

[2] The value of v is an integer or $\perp$. We define $\perp < 0$.

4–2) Otherwise, the unspecified reception is detected in P_j.

5) On receiving an event-inform type system message $m_s(P_i, m, flag, ver)$, $p_j^k (0 \le k < n_j)$ in P_j enqueues it to the system message queue in FIFO order.

5–1) If p_j^k has a local checkpoint c_j^k, p_j^k dequeues $m_s(P_i, m, flag, ver)$ from the message queue in FIFO order. If $k \neq 0$, p_j^k sends an event-inform type system message $m_s(P_i, m, T, max(ver, k))$ to p_j^{k-1}.

5–2) Otherwise, i.e. if p_j^k does not have a local checkpoint c_j^k and a message receiving event $Receive(R)$ where $R = \cup_l \{P_l, m_l\}$ occurs in p_j^k, p_j^k dequeues an event-inform type system message $m_s(P_i, m, flag, ver)$ from the system message queue in FIFO order.

5–2–1) p_j^k takes a local checkpoint c_j^k just before $Receive(R)$ according to CP3 if $flag = T$. If $k \neq 0$, p_j^k sends an event-inform type system message $m_s(P_i, m, T, ver)$ to p_j^{k-1}.

5–2–2) If $flag = F$ and $(P_i, m) \in R$, p_j^k sends an event-inform type system message $m_s(P_i, m, F, ver)$ to p_j^{k-1} where $k \neq 0$

5–2–3) Otherwise, i.e. if $flag = F$ and $(P_i, m) \notin R$, p_j^k takes a local checkpoint c_j^k just before $Receive(R)$ according to CP1. If $k \neq 0$, p_j^k sends an event-inform type system message $m_s(P_i, m, T, k)$ to p_j^{k-1}.

3.4 Protocol error detection

The unspecified reception is detected in step 4–2 in the group communication protocol discussed in the previous subsection. On detecting the unspecified reception in the communication between process groups P_i and P_j, it is required to negotiate which version of the processes are restarted according the following rule:

E1 If the newest version of the processes in P_i and P_j are $p_i^{n_i}$ and $p_j^{n_j}$, respectively, $min(p_i^{n_i}, p_j^{n_j})$th version of the processes are restarted from the local checkpoints. That is, kth version of the processes in P_h where $min(p_i^{n_i}, p_j^{n_j}) < k < p^{(}n_h)_h$ are removed.

On the other hand, the communication deadlock is detected by using the timeout mechanism as the original protocol. In the extended protocol, more than two versions of the processes are concurrently operational, it is required to determine which version of the processes are restarted. In order to realize the optimal recovery, we apply the following rule:

E2 The newest version of the processes in the deadlock cycle is restarted from the checkpoint.

The protocol for detecting the communication deadlock and invoking the recovery is as follows:

1) If a message receiving event $Receive(R)$ where $R = \cup_l \{(p_l, m_l)\}$ occurs in the newest version process $p_i^{n_i}$ in P_i, $p_i^{n_i}$ start the timer.
2) If the timer expires without receiving an intergroup type system message, $p_i^{n_i}$ sends a detection type system message to each $p_l^{n_l}$.
3) When $p_j^{n_j}$ in P_j receives the message sent in 2), $p_j^{n_j}$ sends back an acknowledgment if $n_j \ge n_i$. Otherwise, $p_j^{n_j}$ sends back a negative acknowledgment.
4) If $p_i^{n_i}$ receives acknowledgments from all $p_l^{n_l}$ and has not yet received any intergroup type system message, $p_i^{n_i}$ is removed and the recovery procedure is invoked. That is, the recovery protocol is executed and $p_i^{n_i-1}$ is restarted from the local checkpoint $c_{n_i-1}^i$.

3.5 Restarting protocol

For resolving the protocol errors, i.e. the unspecified receptions and the communication deadlocks, detected in the previous subsection, the recovery protocol is executed in a collection of process groups. This is because the system has to be restarted consistently. Based on the message diffusion protocol, we have designed the recovery protocol which is applied to the process group containing two versions of processes [6]. In order to apply to the process groups containing more than two versions, a version information ver is included in the restart type system message where the recovery is invoked in a process group P_i and p_i^{ver} is restarted. On receiving the system message in P_j , the newest version of the process whose version is less or equal to ver is restarted from the local checkpoint. Here, in order to realize the recovery with the minimum overhead, i.e. the local checkpoint is taken as late as possible for reducing the amount of the wasted computation time caused by the recovery, we add the following restart rule:

E3 If a process p_i^k in a process group P_i takes a local checkpoint c_i^k by CP3 and p_i^{k+1} is restarted, p_i^k is also restarted from c_i^k.

In Figure 3, p_i^0 takes c_i^0 by CP3. If p_i^1 is restarted from c_i^1, p_i^0 is also restarted simultaneously. After that, p_i^0 newly takes local checkpoint c_i^0 just before $e_2^{i,0}$ by CP1. Thus, if the protocol error is detected in a process group P_i, p_i^0 is restarted from $e_2^{i,0}$ not $e_1^{i,0}$. Therefore, the overhead is reduced by E3.

4 Evaluation

The extended upgrading protocol proposed in the previous section has the following properties:

P1 The set of local checkpoints from which the processes are restarted denotes the most recent semi-consistent global state.

P2 The number of the restarted processes is the minimum.

P3 In each cluster, the minimum number of processes are removed for restarting the system, i.e. the newest-version process is restarted.

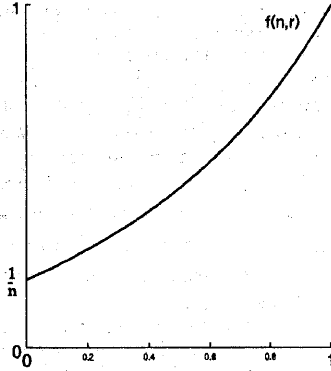

Figure 5: Time overhead.

Next, we evaluate the extended protocol with the original protocol by comparing the time-overhead for n time upgrading. Suppose the following conditions are satisfied:

1) The system is upgraded from 0th version to nth version.
2) For each upgrading procedure, it takes δ time units from the initial state to the upgraded state.
3) The probability that the protocol error occurs and the system is restarted from the checkpoint in the Δ time units is r.

By using the original upgrading protocol and the extended one, it takes $T_o(n,r)$ and $T_e(n,r)$ time units, respectively.

$$T_o(n,r) = \sum_{i=1}^{\infty}(1-r)^n r^{i-1}\frac{2n+i-1}{2}C_{n-1}^{n-1+i}\Delta = \frac{1}{2(1-r)}n(2-r)\Delta$$

$$T_e(n,r) = \sum_{i=1}^{\infty}(1-r)^n r^{i-1}\frac{i+1}{2}C_{n-1}^{n-2+i}\Delta = \frac{1}{2(1-r)}(nr-2r+2)\Delta$$

Figure 5 shows $f(n,r) = T_e(n,r)/T_o(n,r)$. For any $r \in [0,1]$, $0 < f(n,r) \leq 1$, i.e. the time overhead is reduced by applying the extended group communication protocol.

5 Concluding Remarks

This paper proposes a group communication protocol for upgrading the application processes that is extended for applying to large-scale distributed systems. The protocol realizes the lowest overhead recovery, i.e. the minimum number of processes are restarted from the most recent local checkpoints that denote a semi-consistent global state, if the protocol error is detected. Moreover, we evaluate the extended protocol by comparing with the original one. In the future work, we will combine this protocol with another extended upgrading protocol for dynamic configuration changes [13].

References

[1] Barbacci, M. R., Doubleday, D. L. and Weinstock, C. B., "Application Level Programming," *Proc. of the 9th IEEE ICDCS*, pp. 458–465 (1990).

[2] Chandy, K. M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. on Computer Systems*, Vol. 3, 1985, pp. 63–75.

[3] Higaki, H., "Dynamically Updating in Distributed Systems," *Proc. of the 46th Annual Conventions IPS Japan (1)*, pp. 195–196 (1993).

[4] Higaki, H., "Group Communications Algorithm for Dynamically Updating in Distributed Systems," *Proc. of ICPADS*, pp. 56–62 (1994).

[5] Higaki, H. and Hirakawa, Y., "Dynamically Updating Technique in Distributed Systems," *Proc. of the 49th Annual Conventions IPS Japan (1)*, pp. 289–290 (1994).

[6] Higaki, H. and Hirakawa, Y., "Group Communication for Upgrading Distributed Programs," *Proc. of the 16th IEEE ICDCS*, pp. 420–427 (1996).

[7] Higaki, H. and Takizawa, M., "Group Communication Protocol for Flexible Distributed Systems," *Proc. of the 4th IEEE ICNP*, pp. 48–55 (1996).

[8] Kramer, J. and Magee, J., "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Trans. Software Engineering*, Vol. 16, No. 11, pp. 1293–1306 (1990).

[9] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Comm. ACM*, Vol. 21, No. 7, pp. 558–565 (1978).

[10] Malek, M., "Responsive Systems," *Microprocessing and Microprogramming*, Vol. 30, pp. 9–16 (1990).

[11] Segal, M. E. and Frieder, O., "Dynamically Updating Distributed Software: Supporting Change in Uncertain and Mistrustful Environments," *Proc. of IEEE Conf. on Software Maintenance*, pp. 254–261 (1989).

[12] Shiratori, N., Sugawara, K., Kinoshita, T. and Chakraborty, G., "Flexible Networks: Basic Concepts and Architecture," *IEICE Trans. on Communication*, Vol. E77–B, No. 11, pp. 1287–1294 (1994).

[13] Yasuda, M., Shima, K., Higaki, H. and Takizawa, M., "Flexible Inter-Cluster Communication Protocol to Absorb Configuration Changes," *IPSJ Technical Report*, vol. 97, No. 20, pp. 117–122 (1997).

[14] Yoshida, N., "Towards Next-Generation Parallel/Distributed System Development," *Journal of Computer Science*, Vol. 2, No. 4, pp. 300–305 (1992).