Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports
Department of Computer Science

1987

Independent Checkpointing and Concurrent Rollback for Recovery in Distributed System—An Optimistic Approach

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Shy-Renn Lian

Report Number:

87-701

Bhargava, Bharat and Lian, Shy-Renn, "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed System—An Optimistic Approach" (1987). *Department of Computer Science Technical Reports.* Paper 607.
https://docs.lib.purdue.edu/cstech/607

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

INDEPENDENT CHECKPOINTING AND CONCURRENT ROLLBACK FOR RECOVERY IN DISTRIBUTED SYSTEMS — AN OPTIMISTIC APPROACH

Bharat Bhargava
Shy-Renn Lian

CSD-TR-701
August 1987
Revised April 1988

Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach*

Bharat Bhargava and Shy-Renn Lian

Computer Science Department
Purdue University
West Lafayette, Indiana-47907

ABSTRACT

Checkpointing in a distributed system is essential for recovery to a globally consistent state after failure. In this paper, we propose a solution that benifits from the research in concurrency control, commit protocols, and site recovery algorithms. A number of checkpointing processes, a number of rollback processes, and computations on operational processes can proceed concurrently while tolerating the failure of an arbitrary number of processes. In our approach, each process takes checkpoints independently. During recovery after a failure, a process invokes a two phase rollback algorithm. In the first phase, it collects information about relevent message exchanges in the system. In the second phase, this information is used to determine both the set of processes that must roll back and the set of checkpoints upto which rollback must occur. Concurrent rollbacks are completed in the order of the priorities of the recovering processes. The proposed solution is optimistic in the sense, it does well if failures are infrequent by minimizing overheads during normal processing.

* This research is supported by NASA and AIRMICS under grant number NAS 520-0392.

1. Introduction

In this section, we briefly introduce the approaches to the distributed checkpointing problem and summarize previous work.

1.1. Background

In a distributed system, there is no shared memory and processes communicate by exchanging messages. A local state of a process p is defined by p's initial state and the sequence of events that occurred at p. An event is the sending or receipt of a message, or a spontaneous state transition of a process. A checkpoint is a snapshot of a local state of a process. A set of checkpoints, one for each process in the system, called a global checkpoint, is consistent if all snapshots form a consistent global state. The definition of consistency requires that every message recorded as "received" in a checkpoint should also be recorded as "sent" in another checkpoint; not vice versa. Checkpointing and rollback/recovery techniques are used for consistent state restoration of all processes in the system[12].

There are two approaches to checkpointing and recovery[9,12]. One requires that the processes coordinate their checkpointing actions such that the current instance of the global checkpoint in the system is guaranteed to be consistent. When a failure occurs, the system restarts from these checkpoints. In the other approach, each process takes checkpoints independently and saves them on its stable storage. When a failure occurs, processes must coordinate to determine a consistent set of checkpoints. The disadvantages of the first approach are that it requires a number of communication messages between processes for each checkpoint and introduces synchronization delays during normal operation. Moreover, a process failure may block the synchronization of global checkpointing or the rollback. The main disadvantage of the second approach is the domino effect[13,14] while determining the global checkpoint.

1.2. Summary of previous work

The approaches to independent checkpointing have been proposed in [5,7,8,15]. The scheme in [5] is limited to a centralized database system and the crash recovery mechanism is based on the concept of transaction commitment. The concurrency controller keeps the interdependency relationship among operations issued by different transactions. The system periodically takes transaction checkpoints. After a crash, the system finds the optimal set of checkpoints to which uncommitted transactions must be rolled back according to the interdependency of operations. By rolling back to the optimal set of checkpoints the recovery cost is minimized. Transaction processing involves shared memories rather than exchange of messages. Thus it does not consider the problem of recognizing and discarding undone messages as other independent checkpointing schemes must do. The scheme proposed in [7,8] avoids domino effect by a so-called coordination-by-machine approach. It relies on an "intelligent" underlying processor system (or a virtual machine) to automatically establish appropriate checkpoints of interacting processes. The execution of a process is structured by recovery block[13] to provide it with error detection and recovery capabilities. This scheme allows independent and uncoordinated design of error detection and recovery capabilities for each process. The recovery mechanism [15], avoids domino effect by discarding old checkpoints and bounding the rollback by the earliest undiscarded checkpoint. It is quite possible that there are consistent global checkpoints between the earliest undiscarded checkpoint and the failure occurrence. The overhead caused by rolling back to the earliest undiscarded checkpoint is a drawback that we attempt to eliminate. The mechanism used in [15] requires processes to be deterministic. Since a recovering process always reexecutes logged input messages to restore a consistent state, the ordinal position of a message in the input message stream has to be always recoverable. This can cause unnecessary rollbacks. Another limitation of this mechanism is that it can only handle processor failures.

Several distributed checkpointing and rollback recovery mechanism requiring coordination among processes have been proposed in [1,9,11,16]. The common features among them are: processes that participate in checkpointing follow the two-phase commit approach to ensure the atomicity of an instance of global checkpointing; after rolling back, a process does not send or receive any normal system messages until other rollback processes have finished their rollbacks. The mechanism in [16] requires that all the processes in the system take checkpoints or roll back together. The mechanisms in [1,9,11] require that processes that have a exchanged message since their last checkpoints take checkpoints or roll back together. In [1,16], a process can not resume its normal execution between the instances that it makes an uncommitted checkpoint and the checkpoint is committed or aborted. In [9], a process after making an uncommitted checkpoint is not allowed to send out normal system messages until the checkpoint is either committed or aborted. In [11], concurrent execution of checkpointing and rollback initiated by multiple processes is allowed. Also, messages except those for coordination can be transmitted in any order. The mechanisms in [1,9,16] have none or limited resiliency to a process failure in the system. The requirement of coordinating the processes in checkpointing and recovery increases the response time and results in a lower system throughput. When the failures occur infrequently, the overhead in coordinating the checkpointing actions is undesirable.

We propose an algorithm that avoids the disadvantages resulting from creating globally consistent checkpoints during normal operations. It is for a distributed system, and is not restricted to a database system as in [5]. Unlike the scheme in [7,8], it is not a coordination-by-machine approach. It does not rely on an "intelligent" underlying system to take appropriate checkpoints. Instead of rolling back to an earlier global checkpoint as in [15], in our approach a recovering process computes the set of latest globally consistent checkpoints. Our algorithm does not rely on reexecuting logged input messages and causes rollback only when failure occurs in the system. Moreover,

besides processor failures, our algorithm can handle any process failures assuming there exist error detection algorithms for the failures.

1.3. Paper Organization

This paper is organized as follows. Section 2 contains the terminology and system model. In section 3 we present our algorithms for independent checkpointing and rollback recovery. In section 4, the rollback recovery algorithm is extended so that multiple failures and concurrent rollbacks are handled. The discussions and conclusions are stated in section 5 and 6, respectively. Appendix A contains two examples which are used in explaining the algorithm. The correctness proof of our algorithms is in Appendix B.

2. Terminology and System Model

2.1. Checkpoint Number and Checkpoint Interval

The checkpoints taken by a process are ordered by a sequence called "checkpoint" numbers. Each process p maintains a variable cn_p to indicate the sequence number of the last checkpoint taken by p.

A checkpoint with sequence number cn_p taken by process p is denoted by p^{cn_p}. Checkpoint p^i is smaller than p^j if i is smaller than j. The period between p^{cn_p-1} and p^{cn_p} is called the *checkpoint interval* p^{cn_p}. We introduce a *virtual checkpoint* p^f "taken" at the current instant. The period between the instant when process p takes the last checkpoint and the current time is considered to be the checkpoint interval p^f.

2.2. A Consistent Set of Checkpoints and a Consistent Global State

In the literature[3,4,11], consistent sets of checkpoints and consistent global states have been defined. We briefly present the necessary definitions.

Let S={$p_1, p_2, ..., p_n$} be a system of n processes and C={ $p_1^{c_1}, p_2^{c_2}, ..., p_n^{c_n}$ } be the set of checkpoints, one from each of these processes. C is *globally consistent* if and only if for any message M sent from p_i to p_j, if the receipt of M is recorded in the checkpoint p_j^x of process p_j, the sending of M is also recorded in the checkpoint p_i^y of process p_i, where $p_i^y \le p_i^{c_i}$ and $p_j^x \le p_j^{c_j}$. The initial state of a system is assumed to be consistent. A global state of a system reachable from the initial state or from a set of consistent checkpoints before any failure occurred is also consistent.

2.3. Session Number

A process may roll back and then resume execution either as a result of its own failure or in response to failure of another process. Operational session[2] is the interval between the start of normal processing of a process and the instance of failure and resumption of rollback. Operational sessions are ordered by sequence numbers. Each process p maintains a session number sn_p for the current operational session. sn_p is initialized to 1. Each time p resumes execution after rolling back to an earlier checkpoint, p increments sn_p by one.

2.4. System Message

When process p sends a message M to process q, p attaches its current values of session number sn_p and checkpoint number cn_p to M. M is represented as <p,q,sn_p,cn_p,B>, where B is the message body.

M can be either a *normal system message* or a *control message*.

A normal system message is any message related to computation at hand sent by a process during its normal execution.

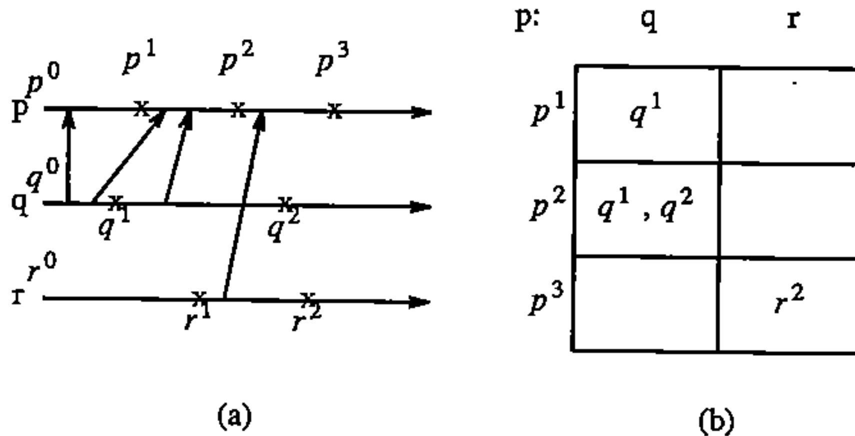
Control messages are used for failure recovery. We have three types of control messages: *rollback_initiating messages*, *rollback_request messages*, and *input_information messages*. The first two types are sent by a recovering process and the third type is sent by a nonfaulty process during failure recovery of the system.

p sends a rollback_initiating message to every other process q when p starts to recover from a failure. As q receives the rollback_initiating message from p, q sends an input_information to p. The input_information message contains the incremental message flow information that q records in its input information table (defined in section 2.5) since last such message was sent to p. After the recovering process p executes the rollback recovery algorithm, it includes the execution results in a rollback_request message and sends it to every other process q. The *rollback_request message* contains a set RS(p) of processes that need to rollback due to the failure of p, and a set of checkpoints, called *recovery points*, to which these processes should rollback.

2.5. Input Information Table

Each process p maintains an input information table (IIT). The usage and maintenance of input information table is further explained in section 3.3. The columns of the table are indexed by process identifiers of all processes in the system and the rows are indexed by p's checkpoint intervals. Each entry $IIT_p[p^c, q]$ of IIT contains the set of checkpoint intervals of q, and during these intervals q sent some messages received by p in checkpoint interval p^c. For every q, p maintains an "already sent to q" mark indicating the last row of the table whose input information has been sent to q by p. The mark is initialized to 0. The purpose of maintaining this mark is to eliminate sending duplicated input information to the same process. IIT is maintained in stable storage except the entries of the current checkpoint interval. The latter will be

incorporated to IIT when it takes a new checkpoint. Figure 1 gives an example for a IIT. With a global state shown in Figure 1.a, the IIT_p maintained by p is shown in Figure 1.b.



p:	q	r
p^1	q^1	
p^2	q^1 , q^2	
p^3		r^2

(b)

Figure 1. (a) A global state in a system. (b) Input information table maintained by p in the system in (a).

2.6. Local System Graph

A process p needs to construct a local system graph G_p when it is recovering from a failure. G_p contains the global message flow information of the system before the failure occurs. We present in this section the formulation and construction of G_p, which is used in the rollback recovery algorithm. Let the system graph constructed by process p be represented by G_p=(V,E_1, E_2). V is the set of nodes of G_p. A node q^i is in V if and only if q^i is a checkpoint perceived by p. A *checkpoint* q^i *is perceived by p* if the input information recorded during checkpoint interval q^i is sent to p under a request by p. To make the graph connected, we introduce a source node labeled s and a starting node q^0 for each process q. There is an edge from s to checkpoint q^0 for any

process q. E_1 is the set of *internal edges* which connects nodes representing checkpoints belonging to the same process to each other. E_1 is defined as $\{(s,q^0)\} \cup \{(q^i,q^j) | q^i \le q^j \le q^{max}$, q^{max} is the maximum checkpoint of q perceived by p and q is any process of the system$\}$. E_2 is the set of *interaction edges* showing the message flow information perceived by p, and is defined as $\{(q^i,r^j)|$ there is a message perceived by p sent in q^i and received in r^j , q and r are processes of the system$\}$. Once created, it is not necessary for p to always keep G_p "up-to-date". G_p will be dumped into stable storage after p is fully recovered. For subsequent failures of p, G_p will be updated and not created again and again.

2.7. System Model

The system consists of a finite collection of sites. Application processes assigned to particular sites may be created or destroyed dynamically. We use the terms site and process alternately. The processes do not share memory, but communicate by sending messages through communication channels. There is a channel between each pair of sites. We assume that the channels are FIFO[17](i.e., messages are delivered in the order of their arrival) and that whenever a process fails, all other processes are informed of the failure in a finite time. We assume that if an intended recipient of a rollback_request message fails, it is able to retrieve the rollback_request message after it recovers. This can be implemented by using multiple message spoolers[6]. We assume nothing about the arrival order of messages sent to a site from two different sources.

Each process p maintains an input information table IIT_p, and current checkpoint number cn_p and operational session number sn_p. Besides, for every other process q, process p maintains an expected session number esn_q^p, and a perceived checkpoint number pcn_q^p. esn_q^p is the largest session number of q known by p and is initialized to 1. Whenever p receives a rollback_request message from another process r and q is

in RS(r), p increments esn_q^p by 1. pcn_q^p is the starting checkpoint number of q's current operational session known by p and is initialized to 0. Whenever p receives a rollback_request message from any r and q is in RS(r), p updates pcn_q^p to q's recovery point. If process p has failed in the past, a local system graph G_p is created. In subsequent recoveries, this graph is updated and not created again and again.

3. Details of our Algorithms

We present the independent checkpointing and concurrent rollback scheme for recovery in distributed systems. In the scheme, a number of checkpointing processes, a number of rollback processes and normal execution are all executed concurrently. First we present a simple independent checkpointing algorithm in section 3.1. Since there is no synchronization among process execution, we have to prevent any action from resulting in an inconsistent state. The system will be in an inconsistent state if any process accepts a message and is not aware that the message has been actually *undone* by the sender. A message is undone by the sender if the sender, after sending out the message, rolls back to a checkpoint taken before the message is sent. A process must be able to recognize and discard incoming messages that have been undone by the sender. We present an algorithm for a process to recognize an undone message and discard it when it arrives in section 3.2. Section 3.3 gives a rollback recovery algorithm for single failure.

3.1. A Simple Independent Checkpointing Algorithm

Each process takes checkpoints independently according to its own needs. For example, a process may take a new checkpoint after, t local clock ticks elapsed, or after sending out k messages. However, in order to have better performance of the system, the strategy of taking checkpoints by each process should be governed by a common principle. For example, if in a system, the checkpoint interval of process p is

significantly larger than that of process q, p could receive a message from q near the end of its long checkpoint interval. If the message is later undone by q, process p would be requested to restore a snapshot taken unreasonably long time ago.

The pseudo-code in Figure 2 proposes a mechanism for a simple checkpointing algorithm.

```
For each process p in the system:
    initialize cn_p = 0;
    Loop
        reset and start counting local clock t_p;
        increment cn_p;
        while t_p < t_checkpointing do
            execute normal operations;
        take a checkpoint p^(cn_p);
    endloop
```

Figure 2. A simple checkpointing algorithm

3.2. Accepting/Rejecting Input Messages During Normal Execution

When process q receives a normal system message M = $<p,q,sn_p,cn_p,B>$ from p, it compares sn_p, cn_p with $esn_p{}^q$ and $pcn_p{}^q$, respectively, to determine whether M is an undone message. q accepts M if it is not an undone message and discards it otherwise. There are three cases to consider:

Case (i) $cn_p \geq pcn_p{}^q$ and $sn_p < esn_p{}^q$.

In this case, M is an undone message, hence q should discard M. Because M was sent by p in an earlier operational session after p^{cn_p} was taken, and p has rolled back or will roll back to the checkpoint with number $pcn_p{}^q$ smaller than cn_p. Example 1 shows how such a situation can occur and how a process can recognize an undone message.

Example 1: In Figure 3, suppose that r is a rollback initiator. In response to a rollback_initiating message(not shown in Figure 3) from process r, process p(q) sends an input_information message m_{pr} (m_{qr}) to process r. After executing the rollback recovery algorithm, process r determines that RS(r) contains process p and the recovery

point of p is p^1. Process r then sends a rollback_ request message m_{rp} (m_{rq}) to process p(q). Suppose that process p sent a normal system message m_{pq} to process q in its operational session s_p before it receives m_{rp} and that m_{pq} was received after m_{rq}. When q receives m_{pq}, it has already updated $pcn_p{}^q$ to 1 and incremented $esn_p{}^q$ to s_p+1. Since the checkpoint number associated with m_{pq} is greater than the perceived one (2 > 1), and the session number associated with m_{pq} is smaller than the expected one (s_p < s_p+1), process q determines that m_{pq} is an undone message and is to be discarded.□

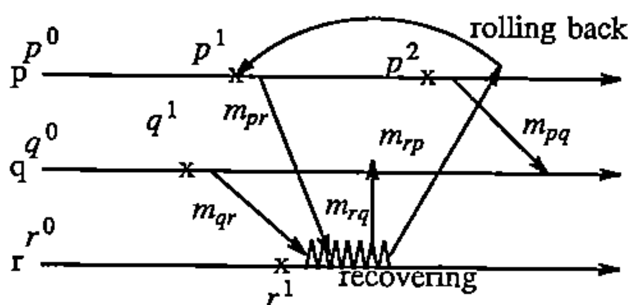


Figure 3. Example showing how a process can recognize an undone message.

Case (ii) $cn_p \geq pcn_p{}^q$ and $sn_p \geq esn_p{}^q$.

In this case, q should accept M since M is not an undone message. If $sn_p = esn_p{}^q$, M is sent in p's current operational session. If $sn_p > esn_p{}^q$, M is sent after p rolled back and restarted. We now give an example for case (ii) by referring to Figure 3 again. Suppose that process q updates $pcn_p{}^q$ to 1 and $esn_p{}^q$ to s_p+1 after receiving a rollback_request message m_{rq} from r (Figure 3). Further, suppose that p sends a message m_{pq} to q after it has rolled back and restarted, i.e. m_{pq}=<p,q,s_p+1, 2,B>. When q receives m_{pq}, since $p^2 > p^1$ and s_p+1 is equal to $esn_p{}^q$, q will accept the message m_{pq}.

Case (iii) $cn_p < pcn_p{}^q$.

In this case, M is not undone by p, and hence q can safely accept M. If $sn_p < esn_p{}^q$, the sender p has rolled back or will roll back to a checkpoint taken after M was sent. There is no possibility that sn_p equals $esn_p{}^q$. If $sn_p > esn_p{}^q$, M is sent after p rolled

back and restarted while q has not updated its knowledge about p's status. In any case, M is not an undone message. Example 2 illustrates a situation when $cn_p < pcn_p^q$ occurs.

Example 2: In Figure 4, suppose that r is a rollback initiator. Process p sends a message m_{pq} to process q after taking checkpoint p^1. In response to a rollback_initiating message(not shown in Figure 4) from r, process p(q) sends an input_ information message $m_{pr}(m_{qr})$ to process r. After r executes procedure rollback_compute(r), it determines that RS(r) contains process p and the recovery point of p is p^2. Suppose that the rollback_request message m_{rq} arrives at process q before message m_{pq} does. When q receives m_{pq}, it has already updated pcn_p^q to 2. Since $p^1 < p^2$, q is sure that m_{pq} would not be undone by p and m_{pq} can be safely processed.□

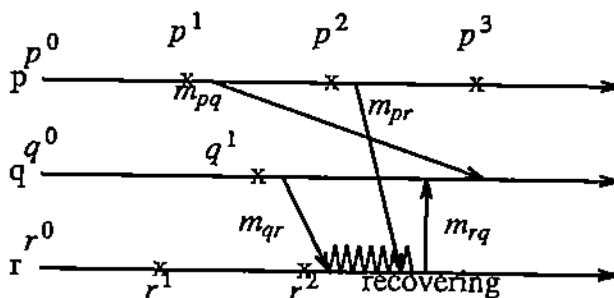


Figure 4. An example showing the case when $cn_p < pcn_p^q$ can occur.

The pseudo-code for the algorithm for a process to accept or reject a message is shown in Figure 5.

```
When receiving a message M=<p,q,sn_p,cn_p,B>, process q does:
    if (cn_p ≥ pcn_p^q) then
        if (sn_p < esn_p^q) then
            q discards M;
        else
            q accepts M;
    else
        q accepts M;
    endif;
```

Figure 5. Algorithm for a process to recognize an undone message during its normal execution

After process q decides to accept M, it first checks if p^{cn_p+1} is in IIT_q[current checkpoint interval of q, p] and adds p^{cn_p+1} to it if it is not already in the set. This is to update IIT_q so that q always maintains correct information in IIT_q. IIT_q will be used to help a recovering process build up a global view of message flow in the system.

3.3. Rollback Recovery after Single Failure

In this section, we consider the rollback recovery for a *single-failure* in the system. Single-failure means that there is at most one process fails at a time. Before the failed process fully recovers, no other process fails. An extension of the algorithm for recovering from multiple-failures is given in the next section. The algorithm for rollback recovery contains two parts: the first part is for a rollback initiator to recover from a failure; the second part is for other nonfaulty processes to coorporate in the failure recovery.

3.3.1. Algorithm for Rollback Initiator

When a process p just recovers from a failure, it executes a two-phase rollback recovery algorithm as presented in Figure 6.

```
For the recovering process p:
      increment the session number sn_p;
      restore to the last checkpoint number p^{cn_p};
      send a rollback_initiating message to every process q in the system;
      collect input_information messages from all other processes;
      augment(p);
      restart_point_determination(p);
      for any process q do
            send a rollback_request message from p to q;
      restore recovery point of p;
      resume execution;
endfor
```

Figure 6. Rollback recovery algorithm for a recovering process.

In the first phase, p increments its session number sn_p, restores to its last checkpoint number p^{cn_p}, and sends a rollback_initiating message to every other process q. After collecting responses(input_information messages) from all the processes, p begins the second phase of recovery. It executes procedure *augment(p)* and procedure *restart_point_determination(p)*. These two procedures are shown in Figure 7 and 8, respectively. At the end of phase two, p includes the execution results of procedure restart_point_determination(p) in the rollback_request message and sends such a message to every q. p then restores itself to its recovery point and continues its execution . During recovery, p queues all normal system messages.

Procedure *augment(p)* presents the algorithm for augmenting the local system graph. For each entry $IIT_q[q^i, r]$ contained in the input_information message sent from q, p creates an edge (r^c, q^i) in G_p to indicate that r sent a message in r^c and the message was received by q in q^i, if r^c is in $IIT_q[q^i, r]$.

```
procedure augment(p);
begin
    for any q do
        for any entry IIT_q[q^i,r] in the input_information message sent by q do
            let IIT_q[q^i, r] contain {r^{i1}, r^{i2},..., r^{ij}};
            for any 1 ≤ k ≤ j do
                if nodes r^{ik}, q^i are not already in G_p then create these nodes;
                add edge (r^{ik}, q^i) to the local system graph G_p;
    for any q do
        mark the last node of q "q^f";
end augment;
```

Figure 7. Algorithm for augmenting a local system graph G_p.

Procedure restart_point_determination(p) computes the set of processes that must roll back due to the failure of p. This set is called the rollback set of p and is denoted by RS(p). In addition, this procedure determines the respective recovery points for processes in this set.

```
procedure restart_point_determination(p)
    RS(p) = {p};
    mark p^f "visited";
    for any q do
        initialize the recovery point of q determined by p to q^f;
    dfsearch(p, p^f);
end restart_point_determination;

procedure dfsearch(p, r^c)
    for any edge (r^c,q^d) where r ≠ q and q^d is not "visited" do
        if (recovery point of q determined by p) > q^{d-1} then
            recovery point of q determined by p := q^{d-1};
        if (recovery point of r determined by p) > r^{c-1} then
            recovery point of r determined by p := r^{c-1};
        RS(p)= RS(p) ∪ {q};
        for any existing node q^i= q^f,..., q^{d+1},q^d do
            if q^i has no out edge that belongs to E_2 of G_p then discard q^i;
            else
                mark q^i "visited";
                dfsearch(p,q^i);
            end_if
    end_for
    discard r^c;
end dfsearch
```

Figure 8. Algorithm to determine restart points of global rollback.

To compute RS(p) and the set of recovery points, procedure restart_point_determination(p) uses a depth-first search algorithm to traverse G_p starting from the node p^f. p^f is the checkpoint interval indicating the period between the instants when p takes the last checkpoint and it fails. Any node that is reachable from p^f is "contaminated" by the failure since a message sent in p^f has been recorded in the node. During the traversal of G_p, whenever a node q^d is visited, q is inserted to RS(p) if it is not already present. The recovery point of q is set to the node previous to q^d. An example showing how procedure rollback_compute(p) works is given in Appendix A.

3.3.2. Algorithm for Rollback Participants

Now we present the algorithm for a nonfaulty process q to cooperate in the failure recovery. Upon receiving a rollback_initiating message from p, q sends an input_information message to p containing the entries of IIT_q from the mark "already sent to p" to the current checkpoint interval, and sets the mark to the current checkpoint interval in the table. q then invokes an interrupt handler presented in Figure 9.

```
procedure back_up(q);
    let q_p^c be the checkpoint last taken ;
    I_q^p = ∅;
    while not (receiving rollback_request message from p) do
        execute normal operations;
        if accepting a message from any process r then
            I_q^p = I_q^p ∪ {r};
    endwhile;
    process rollback_request message sent from p;
    if q is not in RS(p) then
        if I_q^p ∩ RS(p) ≠ ∅ then
            restore q_p^c;
    resume normal execution;
end back_up
```

Figure 9. Interrupt handler of a nonfaulty process for the arrival of a rollback_initiating message.

Between the instants when q sends out the input_information message to and receives a rollback_request message from p, q continues its normal execution without blocking. During that period, q receives messages as usual, but does not send any message. Also, during that period, q creates the set I_q^p of sites from which it receives normal system messages. When q receives the rollback_request message from p, q processes the message by the steps shown in Figure 10. q then compares set I_q^p with RS(p). If the intersection is not empty, q has to return to the point when it sent out the input_information message and continues its execution from then. This is because that the message q has received during that period would be undone by the sender. If the intersection of I_q^p and RS(p) is empty, q continues its execution from the instant it receives the rollback_request message.

When processing a rollback_request message M=<p,q,sn_p, cn_p,B>, q **does:**
begin (queues normal system messages until **end**)
    for each r in RS(p) **do**
        $esn_r^q = esn_r^q + 1$;
        pcn_r^q = recovery point of r;
        set in IIT_q the "already sent to r" mark to q^{c_1} where IIT_q $[q^{c_1},r]$ contains the largest checkpoint of r that is not greater than the recovery point of r;
    endfor
    if q is in RS(p) **then**
        if $cn_q \geq$ recovery point of q **then**
            roll back to the recovery point;
            discard $IIT_q[q^{c_2},t]$ from IIT for all q^{c_2} > recovery point of q;
        else
            discard M;
end

Figure 10. Steps that a process takes in processing a rollback_request message

4. Concurrent Rollback Recovery from Multiple Failures

In this section, the algorithm for rollback recovery is modified to tolerate multiple-failures and to allow concurrent rollbacks initiated by multiple processes. In Appendix A, we illustrate an example of multiple-failures in which the rollback recovery algorithm is not sufficient to handle the problem properly.

The approach to handle multiple failures in a system is described as follows: We assign a distinct priority to each process. This can be done, for example, by process identifiers. If there is no concurrent rollback, a process that recovers later executes rollback recovery algorithm based on the information obtained from the earlier recovered processes and the local information stored in its last checkpoint. If there are occurrences of k-concurrent rollbacks for k > 1, the concurrent rollback initiators finish execution of the rollback recovery algorithm sequentially according to their priorities. This strategy is used to avoid the inconsistency of message flow information obtained by a recovering process in a multiple-failed system.

4.1. Additional Terms and Definitions

We introduce a few terms (shown in italics) which are essential in describing the recovery algorithm for multiple-failures and in proving the correctness of the algorithm.

When a process recovers from a failure, it restores its state to the last checkpoint. This action may require other processes to back up to checkpoints that constitute a consistent global state. So the recovering process initiates a *global rollback*. An instance of *global rollback* is said to be *complete* if the rollback initiator finishes the execution of the rollback recovery algorithm. In this case, the rollback initiator is considered as *fully recovered.* A system is said to have *k-concurrent rollbacks* if a new global rollback is initiated while the other k-1 global rollbacks are not complete. A system is *k-failed*(has *k-failures*) if a process fails while there are k-1 failed processes, each of them either recovering (but not fully recovered) or still failed. The *l-restart* state is a global state S in a system that has k-failures after l out of the k failed processes fully recover from failures. S consists of a state s_i from each live process p_i. If p_i is in the rollback set of any of the l global rollbacks, then s_i is the state of the recovery point of p_i in the latest completed global rollback that requests p_i to rollback; otherwise, s_i is the state when p_i sends the input_information message to the l-th fully recovered rollback

initiator.

4.2. Modification to the Algorithm of Rollback Recovery for Single Failure

We now give the informal description of the modifications to algorithms presented in section 3.3. When a faulty process p recovers, it immediately retrieves the rollback_request messages sent to but not received by it during its failure. We call this set of rollback_request messages the set of *orphan rollback_request messages*, and denote it by ORM_p. Let $OS = \{ o_1, o_2, ..., o_m \}$ be the set of the senders of messages in ORM_p. p increments sn_p by the number of times that it is in RS(o_i), for any o_i in OS. Then p executes the rollback recovery algorithm in the same way as described in section 5. p includes its priority $prio_p$ and $esn_q{}^p$ in the rollback_initiating message sent to every process q. If q is in RS(o_i), then p also includes q^s in the rollback_initiating message to q, where q^s is the smallest recovery point of q among those computed by all o_i's in OS.

4.2.1. Modification to Algorithm for Rollback Participants

In this section, we describe the modification to the algorithm for rollback participant presented in section 3.3.2. The only modification is as follows: instead of sending an input_information message to p, q executes the procedure *send_input_inform(q)* as presented in Figure 11. The intrrupt handler shown in Figure 9 need not be modified.

When q receives the rollback_initiating message from p, it compares sn_q with $esn_q{}^p$ contained in the message. If sn_q is equal to $esn_q{}^p$ (that is, q is in its normal execution during p's failure) q sends the input_information message to p containing IIT_q entries up to the current checkpoint interval cn_q. Since p sends out a rollback_initiating message to q right after p recovers from a failure, there is no possibility that sn_q is smaller than $esn_q{}^p$. If sn_q is greater than $esn_q{}^p$, q must have failed during p's failure, and is now either fully recovered or still recovering from the failure.

We now consider these two cases:

Case (1) q is fully recovered.

In this case, q has resumed its normal execution after a rollback. q sends an input_information message to p containing IIT_q up to q^s included in the rollback_initiating message.

Case (2) q is still recovering from a failure.

q compares $prio_q$ with $prio_p$. There are two possibilities:

(2.1) $prio_p > prio_q$: q stops its execution of the recovery algorithm, and sends an input_information message to p containing IIT_q up to cn_q. q then waits for a rollback_request message from p before it continues its recovering process.

(2.2) $prio_p < prio_q$: q queues p's rollback_initiating message and, continues the execution of its own rollback_recovery algorithm. When q finishes the execution of procedure restart_point_determination(q), it sends together the rollback_request message and an input_information message to p as the delayed reply to the rollback_initiating message from p.

```
procedure send_input_inform(q);
    if sn_q = esn_q^p then
        send an input_information message to p containing IIT_q up to cn_q;
    else
        if (q is in the normal execution) then
            send an input_information message to p containing IIT_q up to q^s;
        else
            if (prio_p > prio_q) then
                sends an input_information message to p containing IIT_q up to
                cn_q;
            else
                q queues p's rollback_initiating message;
end send_input_inform
```

Figure 11. A modification to the algorithm for rollback participants.

4.2.2. Modification to Algorithm for Rollback Initiator

Now we describe the modification to the rollback recovery algorithm for rollback initiator presented in section 3.3.1.

After process p sends out the rollback_initiating messages to every other process q, it collects reply from every q. A reply can be either an input_information message or an input_information message plus q's rollback_request message. The latter case occurs when q and p have concurrent rollbacks and q has the higher priority than p. Using all_input_information messages, p executes procedure augment(p) to update the local system graph G_p. There are two cases to consider:

Case (1)

There is no rollback_request message in the replies to p.

This case occurs when p is the only rollback initiator or when there are k-concurrent rollbacks and p is the initiator with the highest priority. Let C={$p_1^{c_1}, p_2^{c_2}, ..., p_t^{c_t}$} be a set of checkpoint, one from each live process. $p_i^{c_i}$, for $1 \leq i \leq t$, is the last checkpoint in p_i's input_information message to p, if p_i is in RS(o_i) for any o_i in OS. Otherwise, $p_i^{c_i}$ is the last checkpoint of p_i in G_p.

Case (2)

There are rollback_request messages in the replies to p.

Let R = {$r_1, r_2, ..., r_l$} be the set of senders of these messages. p increments esn_r^p for every r by the number of times r is in RS(r_i), where r_i is in R. Let C={$p_1^{c_1}, p_2^{c_2}, ..., p_t^{c_t}$} be a set of checkpoint, one from each live process. $p_i^{c_i}$, for $1 \leq i \leq t$, is the recovery point of p_i in the latest completed global rollback that requested p_i to rollback, if p_i is in RS(r_i) for any r_i in R. Otherwise, $p_i^{c_i}$ is the last checkpoint of p_i in G_p.

C, defined above, is the set of checkpoints that constitute the most recent consistent state before p recovers. If, during checkpoint interval p^l, p accepted any message from p_j sent after $p_j^{c_j}$, checkpoints taken after p^l have recorded the receipt of an undone message. Hence, p has to invoke procedure dfsearch from these points.

For handling multiple failures, procedure restart_point_determination(p) is modified to the version shown in Figure 12. Procedure augment and procedure dfsearch need no modification. The complete concurrent rollback recovery algorithm is presented in Figure 13.

We show the correctness of (concurrent) rollback recovery algorithms in Appendix B.

procedure restart_point_determination_m(p_k);
rollback_set(p_k)={p_k};
let $p_k^{rollback} = p_k^{c_k}$;
for any process p_i **do**
initialize the recovery point of p_i to $p_i^{c_i}$;
if there are any edges (p_i^j, p_k^l) and $p_i^j \geq p_i^{c_i}$ **then**
$p_k^{rollback} = \min\{p_k^{rollback}, p_k^l\}$;
endfor
let p_k^{last} be the last existing checkpoint of p_k;
for any node $p_k^l = p_k^{last}, \ldots, p_k^{rollback+1}, p_k^{rollback}$ **do**
mark p_k^l "visited";
dfsearch(p_k, p_k^l);
endfor
end restart_point_determination_m

Figure 12. Procedure restart_point_determination.

```
For the recovering process p:
    (* OS={o_1,...,o_m} is the set of senders of messages in ORM_p *)
    increment session number sn_p by m;
    restore to the last checkpoint p^{ch^p};
    for any process r in the system do
        if (r is in RS(o_i)) then
            let r^s be the smallest recovery point in any message in ORM;
            send rollback_initiating message to r containing prio_p, esn_r^p, and
            r^s;
        else
            send rollback_initiating message to r containing prio_p and esn_r^p;
    collect replies to p's rollback_initiating message from all other processes;
    augment(p);
    restart_point_determination_m(p);
    for any process q do
        if (p has queued a rollback_initiating message from q) then
            p sends together a rollback_request message and an
            input_information message to q;
        else
            p sends a rollback_request message to q;
    restore recovery point of p;
    resume execution;
endfor
```

Figure 13. Concurrent rollback recovery algorithm for a recovering process.

5. Discussions and Remarks

5.1. The Domino-Free Property

As in [8], the domino effect in this paper is given a narrow definition of a cyclic chain of rollback propagation. That is, the phenomenon in which a failure of a process causes any process to make two consecutive rollbacks without performing any useful computation between the two.

Proposition 1: The scheme of independent checkpointing and (concurrent) rollback recovery algorithms proposed in this paper is free of domino effect.

Proof: In the rollback recovery algorithm(Figure 6), when a process p is recovering from a failure, instead of just rolling back itself to the last checkpoint, p collects the input_information messages to construct the local system graph G_p, then invokes procedure restart_point_determination(p) to determine the set of recovery points for

processes that need to back up due to the interaction with p. By lemma B.2(shown in Appendix B), this set of recovery points is globally consistent, since the processes that need to rollback will back up to these consistent points in one step, there is no domino effect.□

In the worse case, though no consecutive rollbacks will occur, a process may need to roll back to the beginning of execution in one step. Since in most distributed systems, failures are very rare, we expect that the chances for the worse case to occur are especially slim. Moreover, we argue that even in the worse case, the performance of independent checkpointing approach may still not be all that bad compared with the coordinated checkpointing approach.

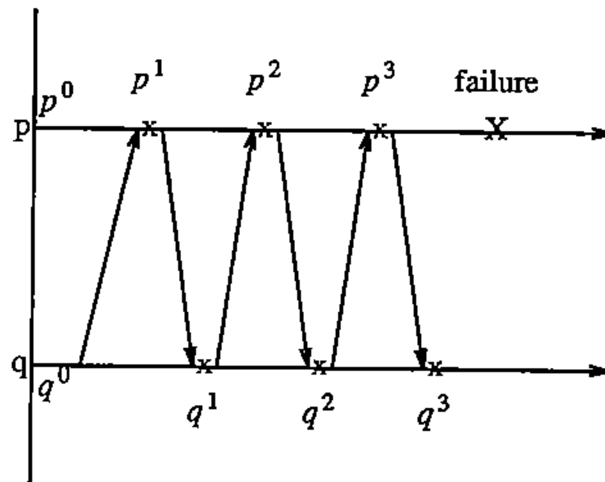


Figure 14. The worse case for independent checkpointing approach.

For example: Consider the exchange of messages as shown in Figure 14. Suppose that this situation occurs in two systems running the same application. One system uses the independent checkpointing approach, the other uses the coordinated approach. In independent checkpointing approach, p and q each takes three local checkpoints and then p fails. However, in coordinated checkpointing approach[1,8,10,15], p and q may have to take six global checkpoints and then p fails. The reason that p and q may take

six global checkpoints is the follows: when p takes p^1, p initiates a global checkpoint, q has to take a checkpoint in response to p's initiation since there is a message exchange between p and q after q took q^0. As q takes q^1, q also initiates a global checkpoint. Since there is a message exchange between p and q after the last global checkpoint, p has to take a checkpoint in response to q's initiation, etc. For each global checkpoint in coordinated approach, the initiator has to invoke a two-phase commit protocol. In the first phase, the initiator sends to other processes the request of taking a local checkpoint. Every other process response with the message if it could take a checkpoint or not. In the second phase, the initiator collects all the replies, makes a decision whether to take it or not, and sends the decision to other processes. The two-phase protocol can be blocked due to a failure. On the other hand, if we choose a nonblocking three-phase commit protocol, very large delay and message traffic will be involved. It is possible that, in the system using independent approach, when p and q restart from the beginning and reach the execution point before failure, the system that uses the coordinated approach has not reached the same point because of the synchronization delay in taking six global checkpoints.

5.2. Recovery with Minimum-Distance Rollbacks

The set of recovery points computed by a recovering process is optimal. That is, each process that needs to roll back will back up to the latest possible checkpoint so that the rollback distance is minimized.

Proposition 2: The set of recovery points computed by procedure restart_point_determination is optimal.

Proof: Let C = { $p_1^c, p_2^c, ..., p_t^c$ } be the set of recovery points computed by procedure restart_point_determination(p_k). Suppose C is not optimal, and C_1 = { $p_1^{c_1}, p_2^{c_2}, ..., p_t^{c_t}$ } is the optimal set of recovery points. Then $p_i^{c_i} \geq p_i^c$ for each p_i and there is at least one p_j such that $p_j^{c_j} > p_j^c$. Since for each point p_i^c in C, there is a

path from p_k^f to p_i^{c+1} in G_{p_k}, the effect of messages sent by p_k in p_k^f is propagated to p_i^{c+1}, p_i has to roll back at least to p_i^c in order to undo these messages. But since C_1 contains some point $p_j^{c_j} > p_j^c$, C_1 can not be consistent, and thus can not be the optimal set of recovery points. Therefore, C must be optimal.□

5.3. Discarding Obsolete Checkpoints

Old checkpoints whose states do not depend on the state of q^f for every process q in the system can be discarded. Recall that q^f is the virtual checkpoint indicating the period between the last checkpoint and the current time of q. We can use an approach similar to the algorithm for rollback recovery in Figure 8 for discarding obsolete checkpoints: any process can be an initiator to discard obsolete checkpoints whenever there is no failure in the system. The initiator p, collects input_information messages from all other processes and uses these messages to augment the local system graph G_p. It traverses G_p by a depth-first search from q^f for every q, and marks the nodes visited by the depth-first search. After all depth-first searches terminated, those nodes that are not marked can be discarded since no process will be asked to roll back to any of the unmarked nodes. The initiator p then notifies every process q of the least checkpoint number q^c that q should keep. Any checkpoint smaller than q^c can be discarded from its stable storage. Whenever a checkpoint is discarded, all the corresponding entries in the input information table and the local system graph are discarded.

5.4. Overhead of Various Approaches

In discussing the performance of the algorithm, we need to compare our approach with other approaches in terms of the time and space overhead. We classify the overhead in two classes: (1) during normal execution. (2) during recovery after failure.

5.4.1. Overhead in Normal Execution

The independent checkpointing approach requires every process to keep multiple checkpoints in stable storage. Stable storage can be constructed by introducing some abstraction on top of disk storage to eliminate errors[9]. Since disk storage is considered inexpensive, we expect that the cost of this storage is not of a major concern.

The overhead in our approach is as follows:

(1) Each process maintains its own checkpoint number and session number.

(2) The checkpoint number and the session number have to be attached to each message sent between processes and checked upon message arrival.

(3) Each process maintains an input information table to record the checkpoint numbers associated with input messages.

The input information table is kept in stable storage except the entry of current checkpoint interval for efficiency purpose. The size of the entry is the total number of different checkpoint numbers associated with input messages received by the process in its current checkpoint interval. In summary, during failure-free execution the overhead is: updating a variable when taking a checkpoint, appending two numbers when sending out a message, checking two variables, and inserting a number to a table entry if it is not yet existent when receiving a message.

Our algorithm has less overhead than the optimistic approach proposed in [14] where the overhead is listed as follows.

(1) A session number is appended to each message and checking it upon arrival.

(2) A dependency vector is maintained and appended to each message sent. Also, it is updated whenever a message is received. The size of a dependency vector is the number of recovery units in the system, where a recovery unit is a basic unit of the system for error recovery.

(3) Input messages are logged.

(4) A log vector which records the status of logging progress of input messages in the system is periodically transmitted and updates. The log vector is of the same size as the dependency vector.

Note that in [14] more variables have to be appended to message sent and checked on message arrival due to (1) and (2). In addition, extra costs are introduced due to the delays caused by logging and communications.

In the coordinated checkpointing approach, whenever a process initiates a global checkpoint, it invokes a two-phase commit protocol or a three-phase nonblocking protocol. Compared with the communication delay resulted from invoking the commit protocol, the time overhead of our independent checkpointing approach is negligible. This is because that communication delay usually takes a few orders longer than operations executed in local site.

5.4.2. Overhead During Recovery after Failure

After failures occur, the independent checkpointing approach recovers the system somewhat more slowly than most coordinated checkpointing approaches do. In the coordinated approaches, the initiator invokes a two-phase commit protocol to request other processes to roll back to the previous checkpoints. In the independent approach, the recovering process does more computation. It collects information, constructs a local system graph, and invokes procedure restart_point_determination which uses a depth-first search algorithm to determine the optimal set of recovery points. The size of the local system graph is the total number of local checkpoints that are not discarded. The time of executing procedure restart_point_determination is O(number of nodes visited in the local system graph) as seen in the algorithm in Figure 9.

The independent checkpointing approach is designed to be an optimistic approach with low overhead for normal execution. Since in most distributed systems, failures are

very rare, we expect that overall the independent checkpointing approach would perform significantly better than other recovery schemes.

6. Conclusion

We have proposed an independent checkpointing scheme which provides better throughput and response time since it eliminates the synchronization overhead of creating globally consistent checkpoints. It is beneficial especially when failures are rare because of the low checkpointing overhead. The scheme is free of domino effect. Checkpointing operations, rollback operations, and normal computations on multiple processes can proceed concurrently while tolerating the failure of an arbitrary number of processes. In the failure recovery, processes are rolled back to an optimal set of recovery points with minimum rollback distance. Obsolete checkpoints can be discarded to save stable storage.

7. References

[1]. G. Barigazzi and L. Strigini, "Application-transparent setting of recovery points," in Proc. 13th IEEE Symp. Fault-Tolerant Computing, June 1983.

[2]. B. Bhargava and Z. Ruan, "Site recovery in replicated distributed database systems", Sixth IEEE Intl. Conf. on Distributed Computing Systems, May 1986.

[3]. K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems", ACM Trans. Comput. Syst., vol. 3, no. 1, pp.63-75, Feb. 1985.

[4]. M. Fischer, N. Griffeth, and N. Lynch, "Global states of a distributed system", IEEE Trans. Software Eng., vol.SE-85, pp.198-202, May 1982.

[5]. V. Hadzilacos, "An algorithm for minimizing roll back cost,", ACM Symposium on Principles of Database Systems, pp.93-97, 1982.

[6]. M. M. Hammer and D. W. Shipman, "Reliability mechanism for SDD-1: A system for distributed databases," ACM Trans. Database Syst., vol.5,no. 4,431-466, Dec. 1980.

[7]. K.H. Kim, "An approach to programmer-transparent coordination of recovering parallel processes and its efficient implementation rules," Proc. 1978 Int. Conf. on Parallel Processing, pp.58-68, Aug. 1978.

[8]. K.H. Kim, J.H. You and A. Abouelnaga, "A scheme for coordinated execution of independently designed recoverable distributed processes," Proceedings of the 16th Annual Symposium on Fault Tolerant Computer Systems, pp.130-135, 1986.

[9]. R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems" in IEEE Trans. Software Eng., vol.SE-13, NO.1, Jan. 1987.

[10]. B.W. Lampson, "Atomic transactions," In Distributed Systems - Architecture and Implementation, ED.Springer-Verlag, Berlin and New York, pp.246-265.

[11]. P. Leu and B. Bhargava, "Concurrent Robust checkpointing and recovery in Distributed systems," Proc. 4th IEEE Int. Conf. on Data Engineering, Feb., 1988.

[12]. B. Randell, P. A. Lee, and P.C. Treleaven, "Reliability issues in computing system design," ACM Comput. Surveys, Vol.10, no.2, pp. 123-166, June 1978.

[13]. B. Randell, "System structures for software fault tolerance," IEEE Trans. on Software Engineering, pp.220-232, June 1975.

[14]. D. L. Russell, "State restoration in systems of communicating processes," IEEE Trans. Software Eng., SE-6, no.2, pp.183-194, Mar.1980.

[15]. R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems", ACM Trans. Comput. Syst., vol. 3, no. 3, pp.204-226, Aug. 1985.

[16]. Y. Tamir and C.H. Sequin, "Error recovery in multicomputers using global checkpoints," in Proc. 13th IEEE Int. Conf. Parallel Processing, Aug. 1984.

[17]. A. S. Tannenbaum, Computer Networks, Prentice-Hall, Englewood Cliffs, N.J., 1981.

Appendix A

In this appendix, we present two examples: Example A.1 shows how procedure restart_point_determination(p) works and Example A.2 shows how problems may occur due to multiple failures.

Example A.1: Suppose process p failed and is now recovering(with the local system graph in Figure A.1). Initially, procedure dfsearch(p, p^f) is invoked and RS(p) contains p only. There is an edge (p^f, q^2) and q^2 is not visited yet, so it needs to execute the for-loop of procedure dfsearch(p, p^f). Recovery points of q and p are set to q^1 and p^2, respectively. RS(p) now contains p and q. For the existing nodes q^f and q^2, it executes dfsearch(p, q^f) and discards q^2 since q^f has an out edge that belongs to E_2 of G_p while q^2 has not. Since there is an edge from q^f to r^2, the invocation of dfsearch(p, q^f) sets the recovery point of r to r^1 and does not change the recovery point of q because that $q^1 < q^2$. The rollback set of p now contains p,q, and r. The execution of procedure restart_point_determination is then terminated since none of the r^f and r^2 has any out edge that belongs to E_2 of G_p. □

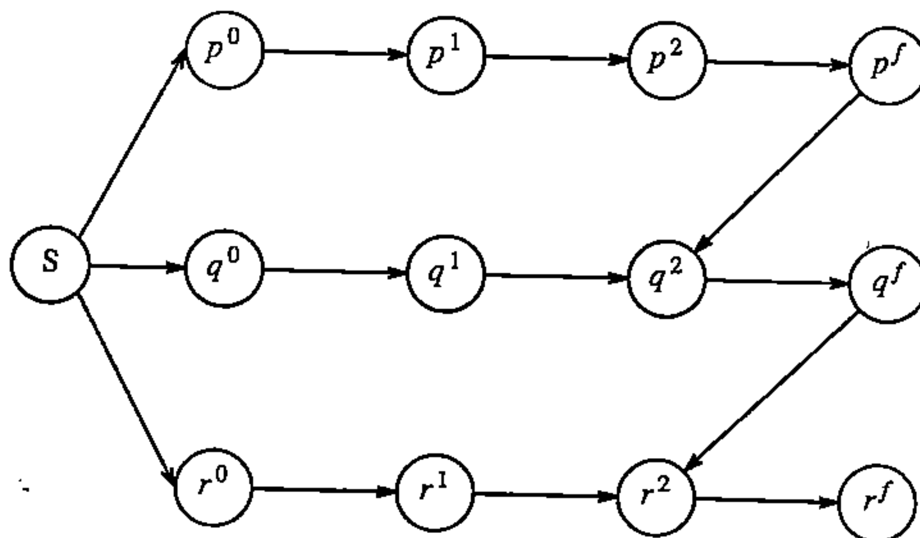


Figure A.1. The local system graph G_p.

Example A.2: Process p sends a message M to q after taking checkpoint p^1 and then fails(Figure A.2). Process q receives M and takes checkpoint q^2, but then fails. Suppose that p starts its recovery before q. Process p collects input_information messages from all live processes and executes the rollback recovery algorithm. Since q is failed and unable to reply p's rollback_initiating message, p does not know that q has recorded the receipt of M and will not request q to rollback. When q recovers, according to the rollback recovery algorithm in Figure 6, q first restores to its last checkpoint q^2 and does not know that M was undone by p.□

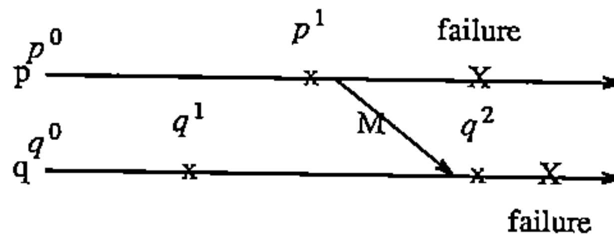


Figure A.2. Example for the case of multiple failures.

Appendix B

In this appendix, the correctness proof of (concurrent) rollback recovery algorithms is given.

Lemma B.1. The rollback set of p computed by procedure restart_point_determination(p) contains exactly the set of processes that must rollback.

Proof: Suppose that process q should rollback but is not in the rollback set of p. Since q should rollback, q must have violated the definition of a consistent global state, i.e., q must have recorded the receipt of a message M from r in checkpoint q^c but r does not record the sending of M. There are two cases to consider:

case (i) r never sent M. This is not possible in our model.

case (ii) r sent M in checkpoint r^{c_1} and will have to back up to r^{c_2} according to the rollback_compute algorithm. Then there is an edge (r^{c_1}, q^c) in G_p. Since r

is in the rollback set of p, and $r^{c_1} > r^{c_2}$, node r^{c_1} is visited by procedure dfsearch(p,p^f). Hence q^c will be visited by procedure dfsearch(p,p^f) too, i.e., q is in the rollback set of p. This contradicts the hypothesis that q should rollback but is not in the rollback set of p.

Moreover, since the rollback set of p is initialized to empty, it is obvious that only the processes that must roll back will be in the set at the end of execution.□

Lemma B.2. The set of recovery points computed by procedure restart_point_determination(p) is globally consistent.

Proof: Let C={p^c, q^c, r^c,...., t^c} be the set of recovery points computed by procedure restart_point_determination(r). Suppose that C is not globally consistent. Then according to the definition of the consistent global state, there is some message M=<p,q,s_p,p^{i-1},B> recorded as received by q in checkpoint q^j and not recorded as sent by p in p^i, where $p^i \le p^c$ and $q^j \le q^c$. There are two cases to consider:

case(i) p never sent M. This is not possible in our model.

case(ii) p has rolled back to a checkpoint p^l preceding checkpoint p^i. Then, the recovery point p^c of p would be set by the algorithm to be no greater than p^l. Hence $p^c \le p^l < p^i$. This contradicts the fact that $p^i \le p^c$.□

Theorem B.1. If a system is in a consistent global state, the effect of any undone message will not persist in the system after an invocation of the rollback recovery algorithm.

Proof: Suppose that process q is not a rollback initiator and it receives and accepts a message sent but later undone by process p. Let the message be M=<p,q,s_p,p^{c_1},B>. Process p is rolled back to a checkpoint p^{c_2}, where $p^{c_2} < p^{c_1}$. There are two possible cases:

case (i) rollback_request message that has undone M in p arrived at q before M.

When q received the rollback_request message it updated the perceived

checkpoint number of p to p^{c_2} and the expected session number of p to s_p+1. When q receives M, according to the algorithm in Section 4.2., process q will discard M since $p^{c_1} > p^{c_2}$ and $s_p < s_p+1$.

case (ii) rollback_request message that has undone M arrived at q after M.

(a) M arrived before q sent input_information message to the rollback_initiator r whose failure caused p to undo M. The receipt of M by q is known to r when r computes its rollback set. By Lemmas 1 and 2, q will be requested to undo the receipt of M.

(b) M arrived between the instants when q sent input_information message to r and received the rollback_request message from r. Since p is recorded in the set I_q^r of q, q will be forced to undo the receipt of M.

If process q was the rollback initiator, case(i) can be applied. □

__Theorem__ __B.2.__ The 1-restart state of a system that has a single failure is globally consistent.

Proof: It immediately follows from Lemmas B.1 and B.2, since the set of recovery points for the rollback set of p computed by procedure restart_point_determination(p) is globally consistent.□

__Lemma__ __B.3.__ The *l-restart* state of a system that has k-failures and the k failed processes invokes k-concurrent rollbacks is globally consistent, for any l, $1 \leq l \leq$ k.

Proof: We prove this lemma by induction.

For l=1, the statement is true by Theorem B.2 since the recovery processing for the process with highest priority in a multiple-concurrent rollback is the same as for a faulty process in a single failure.

Suppose the statement is true for l = m, i.e. m-restart state is globally consistent.

For l = m+1, let p_{m+1} be the recovering process with (m+1)-th highest priority, and let $p_1, p_2, \ldots, p_m$ be the fully recovered rollback initiators in the order of the completion of global rollbacks. We shall prove that the statement is true for l=m+1 by contradiction.

Let C={p_1^c, ..., p_t^c} be a set of checkpoints, one from each live process, where p_r^c, 1 ≤ r ≤ t, is the recovery point of p_r if p_r is in the rollback set of p_j, j ≤ m+1, and p_j is the latest completed rollback initiator that contains p_r in its rollback set; otherwise, p_r^c is the last checkpoint of p_r recorded in $G_{p_{m+1}}$. Suppose that the (m+1)-restart state is not globally consistent, i.e., C is not globally consistent. Then there exists some message M=< p_i, p_j, s_{p_i}, c_1-1, B> recorded as received by p_j in checkpoint $p_j^{c_2}$ and not recorded as sent by p_i in $p_i^{c_1}$, where $p_i^{c_1} \leq p_i^c$ and $p_j^{c_2} \leq p_j^c$. Since by the induction hypothesis, before integrating p_{m+1} to the system the m-restart state is globally consistent, the only possible cases are:

case(i) p_i is some process with i ≤ m and p_j is the process p_{m+1} and $p_{m+1}^{c_2}$ is a checkpoint ≤ the last existing checkpoint saved in stable storage of p_{m+1}.

Since C is not consistent, p_i must have rolled back to some checkpoint $p_i^{c_3}$ preceding $p_i^{c_1}$. Then p_i^c must have been set to no greater than $p_i^{c_3}$. In the execution of procedure rollback_compute_2(p_{m+1}) (Figure 13), since there is an edge ($p_i^{c_1}$, $p_{m+1}^{c_2}$) and $p_i^{c_1} > p_i^c$, $p_{m+1}^{rollback}$ is set to a checkpoint ≤ $p_{m+1}^{c_2}$. For any checkpoint $p_{m+1}^l \geq p_{m+1}^{rollback}$, procedure dfsearch($p_{m+1}$, p_{m+1}^l) is invoked. Hence p_{m+1}^c is set to some checkpoint < $p_{m+1}^{c_2}$. This contradicts the assumption that $p_{m+1}^{c_2} \leq p_{m+1}^c$.

case(ii) p_i is the process p_{m+1} and $p_{m+1}^{c_1-1}$ is the last existing checkpoint of p_{m+1} saved in stable storage and p_j is any live process.

In this case, $p_{m+1}^{rollback}$ is set to p_{m+1}^f and procedure dfsearch(p_{m+1}, p_{m+1}^f) is invoked(Figure 13). p_{m+1}^c is then set to be no greater than $p_{m+1}^{c_1-1}$, i.e., $p_{m+1}^c \leq p_{m+1}^{c_1-1}$ and so $p_{m+1}^c < p_{m+1}^{c_1}$. This contradicts the assumption that $p_{m+1}^{c_1} \leq p_{m+1}^c$.

From both case(1) and (2), we show that C is globally consistent, i.e., the (m+1)-restart state is globally consistent.□

Theorem B.3. The l-restart state of a system that has k-failures is globally consistent, for any $1 \leq l \leq k$.

Proof: It immediately follows from Theorem B.2 and Lemma B.3 since for multiple failures there can be either a process recovers at a time or processes initiate rollback recovery concurrently.□