Integrating a Checkpointing and Rollback-Recovery Algorithm with a Causal Order Protocol

Luis Moura Silva[†] João Gabriel Silva

Laboratório de Informática e Sistemas
Universidade de Coimbra
Urb. Boavista, Lt. 1-1
3000 - Coimbra - Portugal
e-mail : lams@pandora.uc.pt
fax : +351.39.701266

Abstract

The motivation of this paper results from an interesting observation that connects two algorithms with different purposes : a checkpointing and rollback-recovery algorithm with another algorithm to implement causal ordering of message delivery. Two of the causal ordering protocols existent in the literature [Schiper 89][Raynal 91] require a rollback mechanism to work properly in case of some subtle failures. We have already presented in a previous paper [Silva 92] a checkpointing and rollback algorithm that can be used to provide that. However, we have observed that while the causal order protocol needs some rollback support the rollback-recovery algorithm itself also wins the integration. Making use of the information provided by the causal ordering protocol we can obtain a domino-effect free checkpointing and an efficient rollback algorithm that forces a minimal number of processes to roll back in case of failures. This observation emphasizes even more the power of causal ordering in distributed systems.

Keywords : Fault-Tolerance; Distributed Algorithms; Causal Order; Checkpointing; Rollback Recovery.

Approximate word count : 6189

[†] Supported by JNICT under the "Programa Ciência" (BD-2083-92-IA).

1. Introduction

In the absence of global time and due to the asynchronism of the communication channels messages can be delivered in a way that violates the causal ordering. The first ISIS protocol [Birman 87] implements causal order of message delivery at the expense of a substantial information overhead in each application message. To prevent that shortcoming two interesting algorithms have been presented in the literature [Schiper 89][Raynal 91] that assure a bounded information overhead and are easy to implement[1]. However, as was explained in the first paper of [Schiper 89], some subtle site failures may lead to a deadlock situation in which messages are prevented to be delivered. This is a clear disadvantage when compared with the first ISIS protocol. To cope with that, some rollback mechanism is required. This means that a checkpointing and rollback-recovery algorithm should be incorporated in the system to provide the continuity of the application and the proper functioning of the protocol. If we include a rollback mechanism with the causal order protocol (chosen) we overcome the only drawback that such protocol has against the first ISIS implementation. The two algorithms can be running in separated, but as we will see, both have to win if they are integrated. To illustrate that we are going to use a previous checkpointing algorithm [Silva 92] and the causal order protocol presented in [Raynal 91], since their authors claim that it is more easy to understand. The same exercise could be made with the first protocol of Schiper and Sandoz. In our opinion the use of Mattern's Vector Time [Mattern 89] is at least as powerful as the sequence numbers used in Raynal,Schiper and Toueg's protocol, and an elegant solution could also be achieved, but it will not be presented here due to the lack of space.

2. A Brief Overview of the Causal Order Protocol

To turn the paper self-contained we are going to present in a brief way how the causal ordering is implemented in Raynal,Schiper and Toueg's protocol, but the reader is refered to the original paper [Raynal 91] for more details.
Every site in the distributed system has to manage two data structures (where N is the number of sites in the system) :

- DELIV : array[1..N] of integer; (Initially DELIV[i] $= 0, \forall\, i \in 1..N$)
- SENT : array [1..N][1..N] of integer; (Initially SENT[i][j] $= 0, \forall\, i,j \in 1..N$)

[1] There is a third different approach presented in [Peterson 87] that achieves the causal ordering through the use of the *conversation* abstraction.

On site S_i the vector $DELIV_i[j]$ represents the number of messages sent from S_j and delivered to S_i, while the matrix $SENT_i[k][l]$ represents the current knowledge of S_i of the number of messages sent from S_k to S_l. The causal ordering is implemented through the following rules :

(i) Emission of a message m from S_i to S_j :
- send(m,$SENT_i$) to S_j;
- $SENT_i[i][j] := SENT_i[i][j] + 1$;

(ii) Reception of (m,ST_m) sent from S_j to S_i [2]:
- wait $(\forall k \in 1..N\ ,\ DELIV_i[k] \geq ST_m[k][i])$;
- $DELIV_i[j] := DELIV_i[j] + 1$;
- $SENT_i[j][i] := SENT_i[j][i] + 1$;
- $(\forall k,l \in 1..N) : SENT_i[k][l] := \max(SENT_i[k,l], ST_m[k][l])$;

The problem with this and the other protocol has been pointed out by [Schiper 89] and it happens when some subtle site failures happen. Let us take a look at figure 1.

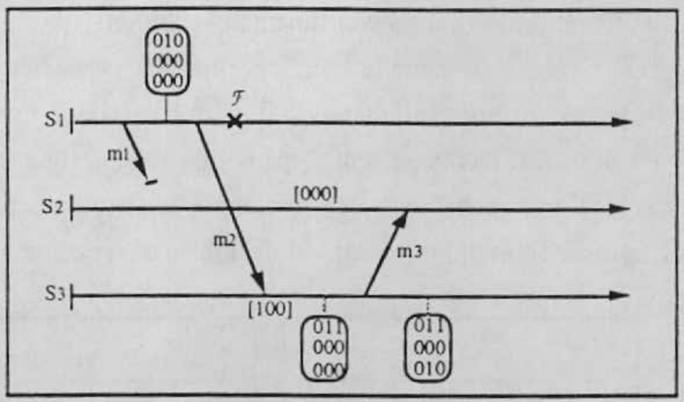

Figure 1 : The effect of the failure of site S_1.

Message m_1 was sent from site S_1 to site S_2 before the emission of message m_2. However, a communication failure might prevent message m_1 from arriving at site S_2, while message m_2 arrives at its destination (site S_3). Now, consider that site S_1 fails after sending message m_2 but before re-transmitting message m_1. If that happens, message m_3 will arrive at site S_2 but it will never be delivered since it must wait for the arrival of message m_1 : m_3 has to wait because $(\forall k \in 1..N,\ DELIV_2[k] \geq ST_{m3}[k][2])$ is not true. Worse than that, every site that receives a message from site S_3 will be prevented by the causal order protocol to communicate with site S_2. To solve that problem the protocol requires a rollback mechanism[3]. In this paper we are going to present a checkpointing and rollback algorithm that can avoid that problem.

[2] Where ST_m represents the control information $SENT_i$ carried with the message m.

[3] The implementation of ISIS didn't require it because a message carries with it every message that preceds it.

3. Coordinated Checkpointing and Rollback-Recovery

In [Silva 92] we have presented a distributed algorithm to implement a coordinated global checkpoint of a distributed application. There is one site which is called the *coordinator* that is responsible for periodically initiate a global checkpoint of the application. Every site should take a local checkpoint and the algorithm assures that the set of the local checkpoints must result in a consistent recovery line, to which processes can roll back in case of failure. Some authors have presented algorithms that are based on independent checkpointing [Bhargava 88][Wood 81]. In case of failure the rollback of processes may lead to the well-known domino-effect [Randell 75]. Other authors assume that processes are deterministic and use message logging to avoid that effect [Borg 83][Powell 83][Johnson 88][Strom 85]. Others assume that clocks are synchronized [Cristian 91][Tong 92], which is not our case. Our approach is not based on such simplistic assumptions, rather we assume that processes can be non-deterministic and there is no notion of global time. It is based on a distributed coordinated checkpoint, like the one presented in [Koo 87]. In short, processes must coordinate their checkpoints in a way that all the set form a consistent global state. According to [Chandy 85], a global state is consistent if no message is recorded as received before it has been sent. This notion is presented in figure 2. The set of all local checkpoints is designated by recovery line, since it is the place to which processes are rewound in case of failure. To avoid the domino-effect the recovery line must be consistent.

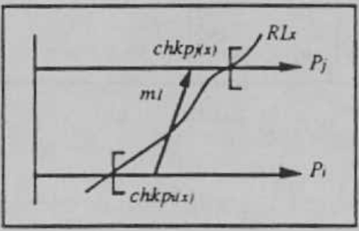

Figure 2a : orphan message.

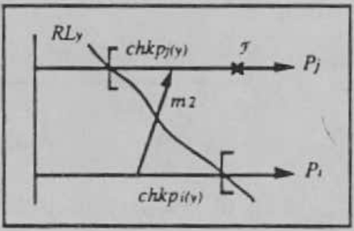

Figure 2b : missing message.

The recovery line in figure 2a (RL_x) is inconsistent since the associated global state (composed by the checkpoints $chkp_i(x)$ and $chkp_j(x)$) records the reception of message m_1 while that message has not yet been sent. That message that crosses the recovery line from the left to the right is designated by *orphan* message. Figure 2b represents a consistent recovery line (RL_y). The message m_2 was sent before $chkp_i(y)$ and received just after $chkp_j(y)$, but there is no inconsistency since it means the message was in transit and the state of the communication channels also belong to the global state of the system. However, in case of rollback of both processes that message will not be re-sent again from P_i to P_j. The message will be lost and for this reason we call it a *missing* message. A coordinated checkpointing algorithm should be able to prevent the loss of *missing* messages, and should not allow at all the occurrence of *orphan* messages. The way to achieve that will be discussed in section 5. In section 6 we will present a rollback algorithm that rolls back a minimal number of processes, and that is achieved thanks to the use of the information associated with the causal ordering protocol.

4. Model and Assumptions

The distributed system is composed by a number (N) of sites $\mathcal{DS} = \{S_1, S_2, ..., S_N\}$, that only communicate through asynchronous messages. Every site can communicate with every other site. Next, there are the assumptions made by our algorithm :

(i) messages can suffer arbitrary delays and nothing is assumed about the speed of execution of the processes (this also means that clocks do not need to be synchronized);

(ii) we assume that processes can be non-deterministic;

(iii) processes are fail-stop. We do not consider any kind of malicious behavior in case of failure;

(iv) there should be a membership [Ricciardi 91] or a diagnosis protocol [Silva 93] responsible for the detection of site failures.

(v) every process has access to stable storage [Lampson 79], that can be centralized or distributed;

(vi) the communication channels can be non-FIFO;

(vii) messages can be lost, but the communication protocol must assure a reliable delivery;

(viii) messages are tagged with sequence numbers in order to detect duplicates;

(ix) the causal order protocol of Raynal,Schiper and Toueg is assumed;

(x) each site is composed by at least one application process, and one thread called the Recovery Manager that implements the checkpointing and rollback-recovery algorithm.

(xii) one of the sites is called the *coordinator* and it is responsible for initiating the checkpointing algorithm. If it represents the weak point of the system there should be an election algorithm [Becker 91] that elects a new *coordinator* in case of a permanent failure.

(xiii) at last, we assume that there is no network partitioning.

5. The Checkpointing Algorithm

There are two main attributes that we take into account in the design of a checkpointing algorithm: first, it must be domino-effect free; second, it should be non-blocking. This last attribute means that processes should be blocked just while the system takes a snapshot of its state, but after that, processes may proceed their computations without being blocked by the algorithm itself. This is an important figure of merit in order to assure that the algorithm does not degrade too much the performance of the applications. In this point we depart from Koo and Toueg's proposal since their checkpointing algorithm is blocking. To avoid the domino-effect the algorithm must assure that the set of checkpoints belonging to the same recovery line forms a consistent global state. The same notion of Chandy and Lamport is here applied as follows :

Definition 1 : A global checkpoint (GC(n)) forms a consistent global state if :

(R1) For every message m_{ij} sent from P_i to P_j and consumed before $chkp_j(n)$, then the sending of that message must have happened before $chkp_i(n)$.

(R2) For every message m_{ij} sent before $chkp_i(n)$ and consumed after $chkp_j(n)$, then that message must be saved to be replayed if the processes have to roll back to that recovery line.

(R3) Processes can take another checkpoint (n+1) only if the previous global checkpoint is already complete.

This last rule means that a global checkpoint contains only one local checkpoint for each process, and each of those local checkpoints is identified by the same ordinal number.

In section 5.1 we are going to present our checkpointing algorithm, while in section 5.2 we show how to achieve the same by using the control information of the causal ordering protocol.

5.1 Algorithm I

The checkpoint event is triggered periodically by a local timer mechanism at the *coordinator* site. The associated Recovery Manager takes a checkpoint of the application process(es) running on its site and then sends a message *-chkp_req(n)* - to the other sites in the network. Checkpoints are assigned ordinal numbers from a monotonically increasing number (CN), which is incremented every time a global checkpoint is created. That message - *chkp_req(n)* - contains the new value of the Checkpoint Number, and that number (CN) is also piggybacked in each outgoing application message that is sent after taking the local checkpoint. When each of the remaining sites receives the - *chkp_req(n)* - message it takes a *tentative* checkpoint [4] of its application processes and sends an acknowledge message to the coordinator site - *chkp_ack (n)* . The application processes are only suspended during the time of taking a checkpoint of its state. Then, they proceed normally while the checkpointing algorithm is still running. When the *coordinator* site receives the acknowledgments of all the other sites it transforms the *tentative* checkpoints into *permanent* checkpoints, by broadcasting a message *-commit_chkp(n)* - to the network. This means that every process needs two keep two checkpoints in stable storage, and the need for this was already demonstrated elsewhere [Koo 87]. Without losing generality, let us assume for the rest of the paper that each site S_i only runs one application process P_i. Now let us see how the algorithm avoids the so called *orphan* messages and identify the *missing* messages, in order to be saved. *Orphan* messages may happen since we admit that communication channels are non-FIFO and due

[4] The checkpoint also includes some system information, like the state of the causal ordering protocol.

to the asynchronism of the network that may violate the causal order between messages. *Missing* messages are simply in transit during the checkpointing algorithm. Let us take a look to figure 3.

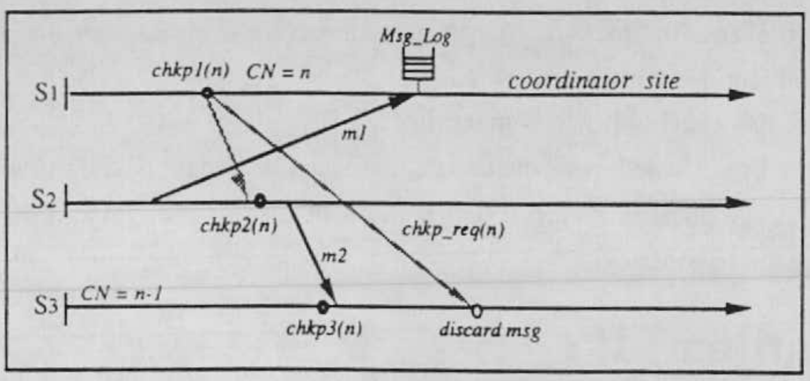

Figure 3 : Avoiding *orphan* messages and identifying *missing* messages.

Message m_1 was sent by process P_2 before taking checkpoint (n) and it is consumed by P_1 after its correspondent checkpoint. This is an example of a *missing* message, and it is easily identified since the message carries with it the CN of the sender process. In that case $m_1.CN = (n-1)$ while $P_1.CN = (n)$. In order to replayed in case of rollback, that message is logged in stable storage at the receiver in a placed called the *Msg_Log*. The other message, m_2, is an example of a message that violates rule R1 of definition 1. It is an *orphan* message since it was sent by a process after taking its checkpoint ($P_2.CN = n$) and it arrives at site S_3 before taking the n^{th} checkpoint ($P_3.CN = (n-1)$). Since message m_2 has a piggybacked CN higher than the CN of the receiver site, it means that the checkpointing algorithm is already running, process P_2 has been taking its checkpoint (n) but the message *chkp_req(n)* destined to site S_3 did not arrived yet. So, the way to avoid that *orphan* message is to take a checkpoint at site S_3 before consuming it. Later, when the proper message - *chkp_req(n)* - arrives at S_3 it will be discarded since the n^{th} checkpoint was already taken and the message become redundant. The CN counter forbids the occurrence of *orphan* messages and facilitates the task of detecting *missing* messages. Upon rollback, those messages that were saved in the *Msg_Log* are introduced on the message queue of the application process.

Lemma 1 : The global checkpoint achieved by algorithm I corresponds to a consistent system state.

Proof : derives directly from definition 1 and the explanation of the algorithm.

The three control messages used by the checkpointing algorithm are not included in the causal order protocol, but every *chkp_ack(n)* message should carry the SENT matrix of the sender process at the time of its checkpoint to allow the construction by the *coordinator* site of the system state associated to the global checkpoint. That information is represented by the following data structure :

- δ : array [1..N][1..N] of integer;

That matrix represents the messages that were sent before the recovery line formed by the nth global checkpoint, and is constructed as follows :

- $(\forall\ i,k,l \in 1..N) : \delta[k][l] := \max(SENT_i[k,l]);$

Matrix δ will be broadcasted to all the other sites since it is piggybacked in the *commit_chkp(n)* message, and as we shall see later, it will be used in case of recovery to determine the set of processes that need to rollback.

5.2 Algorithm II

In this sub-section we will present another way of achieving a consistent global checkpoint, but instead of using the CN counter as before, it just relies on the control information provided by the causal order protocol. In this case, the *chkp_req(n)*, *chkp_ack(n)* and *commit_chkp(n)* messages are also included in the causal order protocol, i.e., their emission and reception follows the rules presented in section 2. The checkpointing algorithm is also made non-blocking, by the same reasons presented before. To facilitate the explanation of the algorithm let us take a look at figure 4.

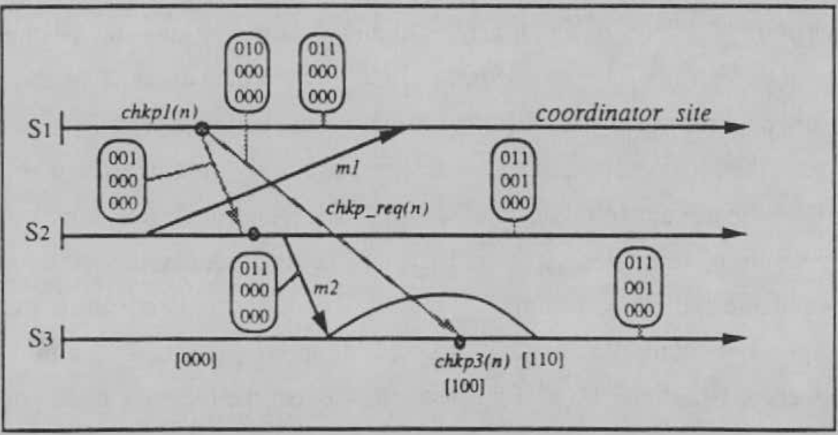

Figure 4 : avoiding *orphan* messages in algorithm II.

To understand that figure the reader should know that a broadcast message using Raynal's protocol should follow the next rule :

(i) Emission of a broadcast message m from S_i to a set of sites (DEST) :
- for all $j \in$ DEST : $SENT_i[i][j] := SENT_i[i][j] + 1;$

```
• for all j ∈ DEST :   • $SENT_i[i][j] := SENT_i[i][j] - 1$;
                       • send($m, SENT_i$) to $S_j$;
                       • $SENT_i[i][j] := SENT_i[i][j] + 1$;
```

In the figure, we can see that message m_2 was a potential *orphan* message but it has to wait in site S_3 since ($\forall$ k $\in$ 1..N, $DELIV_3[k] \geq ST_{m2}[k][3]$). The causal ordering protocol assures that it will only be delivered to the application just after the arrival of the *chkp_req(n)* message from site S_1. This leads to the following interesting observation :

Observation 1 : every potential *orphan* message that violates the consistency of the global checkpoint is at the same time a message that would violate de causal ordering.

So, in this simple way the potential inconsistency caused by the *orphan* messages is eliminated. Now, let us see what happen with the *missing* messages. There are two different cases to be considered : (1) *missing* messages at the *coordinator* site; (2) *missing* messages at a normal site. In the previous figure, message m_1 is one of such messages that according to rule R2 of definition 1 should be saved in order to be replayed in case of recovery. When the *coordinator* site receives message m_1 the Recovery Manager has to know if the message it is or not delivered to the application before the commit operation. If m_1 is delivered before receiving the *chkp_ack(n)* message of the same sender, it means that the message was sent before the nth checkpoint of the sender. In this case it should be saved on the *Msg_Log*. If any other message arrives but it has to wait until the arrival of the correspondent *chkp_ack(n)* message this means that such message has been sent after the nth checkpoint of the sender. This is not a *missing* message and do not need to be logged. In figure 5 is presented another example[5]. We can observe that message *chkp_req(n)* is not delivered as soon as it arrives at site S_3, since there is a previous message (m_1) from site S_1 to site S_3 still in transit in the communication channels. This leads us to the conclusion that the *coordinator* site does not generate any *missing* message at any other site. Now let us consider the second case of a *missing* message. In the same figure, there are two messages m_2 and m_3 that were sent from site S_3 to site S_2. Both messages are delivered to site S_2 and until the *commit_chkp(n)* arrives there is no way for site S_2 to determine if the messages were sent before or after the nth checkpoint of the sender. One solution is to keep them in volatile memory of the receiver site (S_2). Then, when the *commit_chkp(n)* message arrives it brings with it the global state of the system associated to that recovery line, represented by the matrix δ mentioned before. So, a simple way to determine if a message m from site S_i that had arrived at site S_k between the

[5] For the sake of clarity, the evolution of the SENT matrix in each site is not shown. It is an exercise left to the reader.

last checkpoint of that site and the delivery of the *commit_chkp(n)* message is a *missing* message or not is as follows :

- if $(\delta[i][k] > ST_m[i][k]) \rightarrow$ *missing* message.

Missing messages are saved in the *Msg_Log*, that is kept in stable storage.

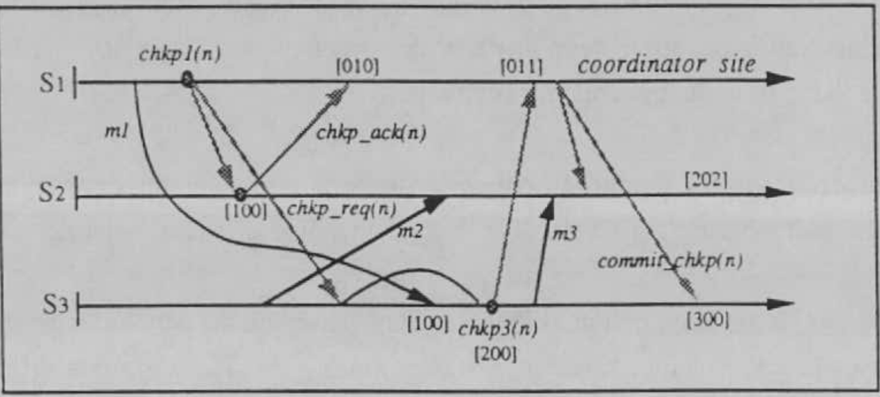

Figure 5 : identifying *missing* messages in algorithm II.

In short, we have two different rules to detect missing messages : one for normal sites, and another for the *coordinator* site.

Lemma 2 : The global checkpoint achieved by algorithm II corresponds to a consistent system state.

Proof : Based on observation 1 we conclude that if the coordinator site includes the *chkp_req* message in the causal order protocol it may be sure that there is no *orphan* message. Also, by the two rules explained before, the algorithm is capable of identifying the *missing* messages that are saved in stable storage during the commit operation. The next checkpoint is only started after the *coordinator* commits the current one. Since the 3 rules of Definition 1 are followed we conclude that the global checkpoint obtained by algorithm II also corresponds to a consistent global state.

Theorem 1 : Any rollback algorithm that uses any of the checkpointing algorithms (I or II) is domino-effect free.

Proof : by lemma 1 and 2.

We said before that *missing* messages arrive between the local checkpoint and the delivery of the *commit_chkp(n)* message, but we have to justify that.

Theorem 2 : *Missing* messages are never delivered after the delivery of the *commit_chkp(n)* message.

Proof : suppose a message m_x that was the last message sent before the nth checkpoint of site S_i and is received in site S_j after its nth checkpoint. As was told before, site S_i cannot be the *coordinator* site since it does not originate *missing* messages. Then, the sending of message m_x causally precedes the sending of the *chkp_ack(n)* message at site S_i : $\text{send}(m_x) \rightarrow \text{send}(\text{chkp\_ack}(n))$. At the same time, the message *commit_chkp(n)* is only sent by the *coordinator* just after receiving all the acknowledgments. This means that $\text{send}(\text{chkp\_ack}(n)) \rightarrow \text{send}(\text{commit\_chkp}(n))$ and since the relation of causality is transitive we have that $\text{send}(m_x) \rightarrow \text{send}(\text{commit\_chkp}(n))$. Since both the messages have the same destination site (S_j) we conclude that the protocol assures their delivery in the same order : $\text{deliver}(m_x) \rightarrow \text{deliver}(\text{commit\_chkp}(n))$. This means that all the missing messages are delivered before the *commit_chkp(n)* message. QED

A straightforward rollback algorithm to be used together with algorithm II is one that in case of failure rolls back all the application processes to the last recovery line. Since that line represents a consistent global state that algorithm is domino-effect free (by theorem 1). Just using the information provided by the causal order protocol we have obtained a domino-effect free checkpointing algorithm. The message complexity of both algorithms (I and II) is the same : $O(3n)$. This is an advantage when compared with Koo and Toueg's algorithm, which has a message complexity of $O(2n^2)$. Furthermore, Koo and Toueg's algorithm is blocking while the algorithms presented here are non-blocking.

6. The Rollback Algorithm

In some cases it is not really necessary to rollback all the application processes in case of a failure to reach a consistent system state. It depends on the communication patterns. In this section, we are going to present a rollback-recovery algorithm that minimizes the number of processes that are forced to roll back. In fact, it is the optimum algorithm since it rolls back the minimal number of processes. To achieve that, we need a way to keep track of the dependencies between processes, and the information provided by the causal ordering protocol is just what we need. First of all we define the relation of dependency that we represent by the operator (-»):

Definition 2 : A process P_i is dependent on process P_j (P_j -» P_i) if : (a) it has received a message from P_j; (b) or it has received a message from P_k and P_j -» P_k.

If a process P_j rolls back to a state before the sending of a message m_x consumed by process P_i, then we say that the event $\text{send}(m_x)$ is backed out due to the rollback recovery. If $\text{send}(m_x)$ is backed out the message become an *orphan* and the correspondent event - $\text{deliver}(m_x)$ - should also be rolled back. This means that process P_i is dependent on process P_j, and if P_j rolls back process P_i must do the same. We have also to think on those messages that were sent by processes that do not have dependencies on processes that are forced to roll back. For instance, what happen if a process P_k has sent a message m_y that is consumed by process P_i and this process fails and has to roll back ? (considering that (P_i -» P_k) is not true). If that message is not replayed again it becomes a *missing* message during the replay of process P_i . There are two approaches to solve that problem : (1) we alter definition 2 in order to include the reception of messages also as a source of dependency; (2) or we keep those messages in order to be replayed in case of recovery. If we choose solution 1, as has been done in [Tong 92], we are increasing the dependencies between processes and this means that we are not going to obtain the optimum rollback algorithm. If we choose solution 2, we must decide where the messages are kept. If they are kept on the stable storage of the receiver site it certainly introduces a high performance degradation during failure-free execution. So, the best way is to keep those messages on the volatile memory of the sender process [Johnson 87]. However, we do not need to keep all the messages that have been sent, otherwise the processors would run out-of-memory. There is a bound which is established by the following lemma.

Lemma 3 : if $\text{DELIV}_i[k] > \delta[k][i]$ then site S_i does not need to keep any more message that it sends or has sent to S_k during the current checkpoint interval.

Proof : A message logged in the sender can only be used during recovery if the receiving process P_k rolls back but the sender P_i does not. If both processes have to roll back those logged messages will be discarded. The relation $\text{DELIV}_i[k] > \delta[k][i]$ means that after the last checkpoint site S_k has already delivered at least one message from S_i to process P_k. Therefore, P_k -» P_i , and process P_i will roll back too if process P_k has to roll back. For this reason, messages sent from P_i to P_k do not need to be logged at the sender. QED

Before describing the algorithm lets us describe what are the data structures needed for that :

- δ : array [1..N][1..N] of integer;
- Ψ : array [1..N][1..N] of integer;
- CC : array [1..N] of integer; (Initially $CC[i] = 0, \forall i \in 1..N$)

The array δ is received by each site with the *commit_chkp(n)* message sent by the coordinator site when the last checkpoint was committed and it represents the state of the system associated to the last recovery line (n). The array Ψ will be used and constructed during the rollback algorithm, as will be seen later. The vector CC, which is kept in stable storage, contains the crashcount value for each process of the system. Every time a process P_i crashes or has to roll back due to the rollback propagation increments the CC[i] value. Every application message sent by a site S_i is piggybacked with the current $CC_i[i]$ value. We assume algorithm I as the checkpointing algorithm since algorithm II includes the control messages in the causal ordering protocol and that introduces dependencies between processes. Nevertheless, we assume the implementation of causal order protocol, and its data structures will be used to determine the set of processes that need to rollback. That set is calculated by the rollback algorithm and is represented by $\mathfrak{R}$. Let us consider that a site S_i fails and then restarts[6]. After restarting it has to run the following procedure :

(1) It inspects its stable storage to see what is its the last permanent checkpoint (LPC) in stable storage. Then, it sends a broadcast message to the other sites in the network - *roll_req(i,LPC)*. The purpose of this message is twofold : it notifies the other sites that have to execute the rollback algorithm, and at the same time it is used to determine the last committed recovery line (LCRL). In the meanwhile, site S_i waits for the answers of the other nodes.

(2) Any of the other sites S_j after receiving that message also inspects its stable storage to see which is its last permanent checkpoint and sends a *roll_ack(j,LPC, DELIV$_j$)* [7] message that includes the current value of the DELIV vector on that site. This message is also broadcasted to all the other sites. If there happens a failure that affects more than one site the other restarted sites should send instead a *roll_req(i,LPC)* message, as in the previous case.

(3) Each site after receiving a message from each one of the remaining sites (i.e. (N-1) messages) construct the array in the following way : $\Psi[l][k] := DELIV_l[k]$ ($\forall$ l,k $\in$ 1..N). It should be noted that the contents of $DELIV_i$ were lost with the failure, and it means that $\Psi[i][k] := (?)$ ($\forall$ k $\in$ 1..N). However, those values are not needed for the execution of the algorithm. After that step, it determines what is the LCRL. If all the processes have committed the last checkpoint then the LCRL corresponds to the last global checkpoint. However, if there is at least one process that had not committed its last checkpoint it means that the failure happened during the checkpointing algorithm. In this case, the LCRL corresponds to the old global checkpoint that remains in the stable storage. Every site S_i discards its nth checkpoint, and those messages in the *Msg_Log* associated to that checkpoint. It also puts CN_i equal to $(CN_i - 1)$. After that, and in both cases, every site executes the algorithm presented in figure 6.

```
1. $\mathfrak{R} := \{failed\}$;
2. If there is any process $P_k$ such that : ($P_k \notin \mathfrak{R}$) and ($\Psi[k][i] > \delta[i][k]$, for any $P_i \in \mathfrak{R}$)
        then $\mathfrak{R} := \mathfrak{R} \cup \{P_k\}$;
   otherwise goto step 4;
3. goto step 2.
4. if (LCRL = old) then $CC[i] := CC[i] + 1$, ($\forall$ i $\in$ 1..N)
   else $CC[i] := CC[i] + 1$, ($\forall P_i \in \mathfrak{R}$)
5. end.
```

Figure 6 : determination of the rollback set.

[6] We are here considering just transient failures. The same algorithm of rollback can be used in case of permanent failures but the system should be subjected to a reconfiguration strategy.

[7] The control messages *roll_req* and *roll_ack* are not included in the causal order protocol.

At the end of it, the vector $\mathfrak{R}$ contains the set of processes that need to roll back. If site $P_k \notin \mathfrak{R}$ it does not need to rollback. It re-sends the messages logged in its volatile memory to those processes that are forced to roll back. Those messages are piggybacked with the updated CN and CC values of the sender. Then it may proceed its computation normally. However, if $P_k \in \mathfrak{R}$ then it should rollback to the checkpoint associated to the LCRL. Its execution restarts from that point, the messages contained in the associated *Msg_Log* are inserted in the message queue of the process and the data structures of the causal order protocol ($SENT_k$ and $DELIV_k$) are rewound to the values they had when that checkpoint was taken.

The rollback algorithm will force to recover all the processes that have some causal dependencies with the failed one, and those dependencies are determined from the first message sent after its previous valid checkpoint. Those processes that have dependencies with the failed one may force the rollback of others, as well. This is the rollback propagation phenomenon. To determine if a process P_k is dependent on any process P_i (with $P_i \in \mathfrak{R}$) we just need to make a comparison of the values of Ψ and δ : if ($\Psi[k][i] > \delta[i][k]$) then P_k is also included in the rollback set. To better understand the algorithm let us look to figure 7.

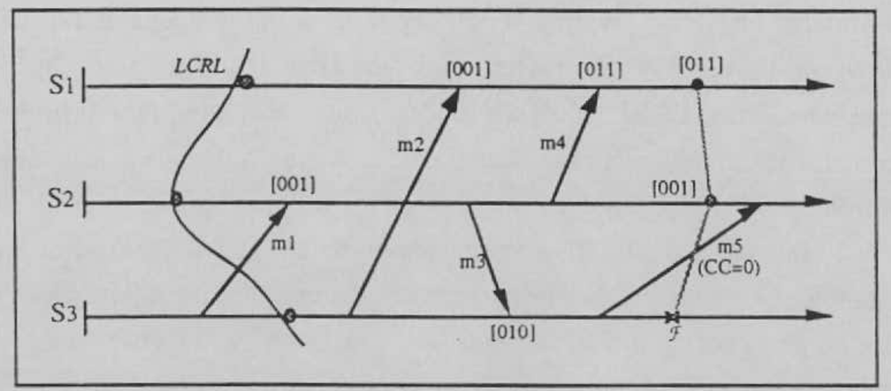

Figure 7 : A rollback scenario.

The failure of site S_3 forces the rollback of the process that was running on site S_1 (($\Psi[1][3] = 1$) > ($\delta[3][1] = 0$)). Message m_1 is a *missing* message and does not create any dependency between process P_3 and P_2. In result, $\mathfrak{R} = \{P_1, P_3\}$. Process P_2 does not need to roll back. It re-sends messages m_3 and m_4 to the others, and then proceeds its computation. Message m_5 was a message that was sent by site S_3 before it fails and that was in transit during the rollback algorithm. It carries $CC = 0$, and when it arrives at site S_2 that site already have $CC[3] = 1$. So, site S_2 discards that message. This rule can be stated as follows :

- if site S_i receives a message m_x from S_k and $M_x.CC < CC_i[k]$ then m_x is discarded.

This rule is needed to avoid the *livelock-effect*, described in [Koo 87]. Those messages that are in the queue waiting to be delivered are also subjected to that filter rule.

The distributed nature of the rollback algorithm adapts well to the case of multiple failures, better than a centralized approach. In the case of simultaneous failures, there are more than one *roll_req* message. Those processes that have sent a *roll_req* instead of a *roll_ack* are also included on the rollback set $\Re$, before running the algorithm of figure 6. Other case to be considered is the interference between the two algorithms : checkpointing and rollback. If some site fails during the checkpointing algorithm some sites that were waiting for the *commit_chkp* message may heard instead a *roll_req* message. Since that message has a higher priority the sites will immediately abort the checkpointing operation and start the rollback algorithm.

Correctness of the Algorithm :

Theorem 3 : The re-execution after rollback achieves a consistent system state.

Proof : By lemma 1 there is at least one global checkpoint in stable storage which corresponds to a consistent global state. However, we need to prove that the rollback operation will not introduce any *orphan* or *missing* message. Four cases are considered : (1) the undoing of messages sent from a process P_k ($P_k \in \Re$) to other process P_i ($P_i \in \Re$) does not cause any *orphan* message since both sender and receiver are forced to roll back. The only *missing* messages are those kept in the *Msg_Log* that are replayed by the receiver; (2) messages sent from a process P_k ($P_k \notin \Re$) to a process P_i ($P_i \in \Re$) are *missing* messages. During the recovery phase, those messages will be replayed since they were kept in the volatile memory of the sender process (P_k). However, the replaying of those messages may generate some duplicates if the originals are still in transit in the communication channels, but we have assumed that duplicates are detected by the communication protocol; (3) messages sent by a process P_k ($P_k \in \Re$) to another process P_i ($P_i \notin \Re$) would be *orphan* messages, but by definition 2 and the execution of algorithm this situation is impossible; (4) messages sent from P_k ($P_k \notin \Re$) to P_i ($P_i \notin \Re$) are not replayed but none of those processes are forced to rollback. QED

Theorem 4 : The rollback algorithm does not introduce any deadlock in the causal order protocol.

Proof : (by contradiction). The rollback operation undoes messages when processes are forced to roll back and discards messages that were sent by rolled back processes and were in transit during the execution of the algorithm. Assume that process $P_i \notin \Re$ and that there is a message m_y sent by P_i in the queue of a surviving site P_j. That message cannot be delivered because it is waiting for a message m_x (sent by $P_k \in \Re$) that was discarded by the algorithm. This may lead to a deadlock on the protocol. However, if m_y cannot be delivered it means that deliver(m_x) $\rightarrow$ deliver(m_y), and by the causality definition sent(m_x) $\rightarrow$ sent(m_y) as well. Then we may conclude that process P_k has sent at least one message to process P_i or to other process P_l and (P_l -» P_i). In both cases, P_k -» P_i. By definition 1, process P_i is also forced to roll back ($P_i \in \Re$) which contradicts our assumption. Thus, it is assured that the rollback algorithm does not introduce any deadlock on the causal ordering protocol. QED

Theorem 5 : This rollback-recovery algorithm achieves the optimum set of processes that are forced to roll back.

Proof : (By contradiction). As was stated in theorem 3, the re-execution state achieved by roll backing the set of processes $\mathfrak{R}$ is a consistent system state. Let us assume that it is also possible to achieve a consistent system state by forcing the rollback of a sub-set of $\mathfrak{R}$, that we designate by $\mathfrak{R}'$ ($\mathfrak{R} \supset \mathfrak{R}'$). This means that there is at least one process P_j which is not forced to roll back in this new case. However, P_j has consumed at least one message M sent by one process P_i that belongs to the rollback set. Since P_i undoes the sending of that message and P_j does not undo the delivery of M, it means that M is an *orphan* message and by definition 1 the result state is inconsistent. This is a contradiction to our assumption. QED

At last, we should say that the message complexity of our rollback algorithm is $\mathcal{O}(n^2)$ while the equivalent algorithm of Koo & Toueg is $\mathcal{O}(2n^2)$. Moreover, their algorithm also considers the acknowledge messages as source of dependencies between processes which means that the rollback set achieved is not the optimum.

7. Conclusions

In this paper we have presented an interesting combination of two algorithms that have different purposes. The causal order protocol needs a rollback mechanism, and we have shown that with such protocol we could design an elegant checkpointing algorithm, which is non-blocking and domino-effect free. At the same time, if we use the information of that protocol within the rollback algorithm we could obtain the minimum set of set processes that are forced to roll back, in case of failure. In our opinion, the success of the integration of those algorithms emphasizes even more the need to have causal ordering in distributed systems.

References

[Becker 91] T.Becker. "Keeping Processes under Surveillance", Proc. 10th Symposium on Reliable Distributed Systems, pp. 198-205, 1991

[Bhargava 88] B.Bhargava, S.R.Lian. "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems", Proc. 7th Symposium on Reliable Distributed Systems pp. 3-12, 1988

[Birman 87] K.Birman, T.Joseph. "Reliable Communication in Presence of Failures", ACM Transactions on Computer Systems, Vol. 5, No.1, pp. 47-76, February 1987

[Borg 83] A.Borg, J.Baumbach, S.Glazer. "A Message System Supporting Fault-Tolerance" Proc. 9th ACM Symp. on Operating Systems Principles, pp. 90-99, 1983

[Chandy 85] K.M.Chandy, L.Lamport. "Distributed Snapshots : Determining Global States of Distributed Systems", ACM Trans. on Computer Systems, pp. 63-75, Feb. 1985

[Cristian 91] F.Cristian, F.Jahanian. "A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Applications", Proc. 10th Symposium on Reliable Distributed Systems, pp. 12-20, 1991

[Johnson 87] D.B.Johnson, W.Zwaenepoel. "Sender-Based Message Logging", Proc. 17th Int. Symposium on Fault-Tolerant Computing, pp. 14-19, June 1987

[Johnson 88] D.B.Johnson, W.Zwaenepoel. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing", Proc. 7th ACM Symp. on Principles of Distributed Computing, pp. 171-181, August 1988

[Koo 87] R.Koo, S.Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems" IEEE Trans. on Software Engineering, Vol. SE-13, No. 1, pp. 23-31, January 1987

[Lampson 79] B.W.Lampson, H.E.Sturgis. "Crash Recovery in a Distributed Data Storage System", Technical Report XEROX PARC, April 1979

[Mattern89] F.Mattern. "Virtual Time and Global States of Distributed Systems", in Parallel and Distributed Algorithms, ed. by M.Cosnard et al., pp. 215-226, North-Holland, 1989

[Peterson89] L.L.Peterson, N.C.Buchholz, R.D.Schlichting."Preserving Context Information in an IPC Abstraction", ACM Trans. on Computer Systems, pp. 217-246, August 1989

[Powell 83] M.L.Powell, D.L.Presoto. "Publishing : A Reliable Broadcast Communication Mechanism", Proc. 9th ACM Symp. on Operating Systems Principles, pp. 100-109, 1983

[Randell 75] B.Randell. "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol.SE-1, pp. 220-232, June 1975

[Raynal 91] M.Raynal, A.Schiper, S.Toueg."The Causal Ordering Abstraction and a Simple Way to Implement It", Information Processing Letters, 39, pp. 343-350, 1991

[Ricciardi 91] A.Ricciardi, K.Birman. "Using Process Groups to Implement Failure Detection in Asynchronous Environments", Proc. 10th ACM Symposium on Principles of Distributed Computing, pp. 341-152, August 1991

[Schiper 89] A.Schiper, J.Eggli, A.Sandoz. "A New Algorithm to Implement Causal Ordering", Distributed Algorithms, Lecture Notes in Computer Science, 392, pp. 219-232, 1989

[Silva 92] L.Moura Silva, J.G.Silva. "Global Checkpointing for Distributed Programs", Proc. 11th Symposium on Reliable Distributed Systems, Houston, pp. 155-162, 1992

[Silva 93] L.Moura Silva, J.G. Silva. "DIP : Distributed Diagnosis Protocol", Proc. EUROMICRO'93, Barcelona Spain, pp. 171-178 ,1993

[Strom 85] R.Strom, S.Yemini. "Optimistic Recovery in Distributed Systems", ACM Transactions on Computer Systems, Vol. 3, No. 3, pp. 204-226, August 1985

[Tong 92] Z.Tong, R.Y.Kain, W.T.Tsai. "Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks", IEEE Trans. on Parallel and Distributed Systems, Vol. 3 No. 2, pp. 246-251, March 1992

[Wood 81] W.G.Wood. "A Decentralized Recovery Control Protocol". Digest of Papers FTCS-11, pp. 159-164, 1981