Efficient Message Logging for Uncoordinated Checkpointing Protocols

Achour Mostefaoui, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Message Logging for Uncoordinated Checkpointing Protocols

Achour Mostefaoui , Michel Raynal

N ° 2972

Septembre 1996

THÈME 1

ISSN 0249-6399

INRIA
RENNES

Efficient Message Logging for Uncoordinated Checkpointing Protocols

Achour Mostefaoui , Michel Raynal

Thème 1 — Réseaux et systèmes
Projet Adp

Rapport de recherche n°2972 — Septembre 1996 — 11 pages

Abstract: A message is *in-transit* with respect to a global state if its sending is recorded in this global state, while its receipt is not. Checkpointing algorithms have to log such in-transit messages in order to restore the state of channels when a computation has to be resumed from a consistent global state after a failure has occurred. Coordinated checkpointing algorithms log those in-transit messages exactly on stable storage. Because of their lack of synchronization, uncoordinated checkpointing algorithms conservatively log more messages.

This paper presents an uncoordinated checkpointing protocol that logs all in-transit messages and the smallest possible number of non in-transit messages. As a consequence, the protocol saves stable storage space and enables quicker recoveries. An appropriate tracking of message causal dependencies constitutes the core of the protocol.

Key-words: Distributed Systems, Backward Recovery, Consistent Global Checkpoints, Optimistic Sender-Based Logging

(Résumé : tsvp)

IRISA, Campus de Beaulieu, Rennes Cedex, France. Email: {mostefaoui,raynal}irisa.fr

To appear in the Proceedings of EDCC2 (2nd European Dependable Computing Conference), october 1996.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Enregistrement sélectif de messages lors d'une définition non coordonnée de points de contrôle

Résumé : Les algorithmes de définition de points de reprise nécessitent l'enregistrement de certains messages dits *en transit* afin de pouvoir restaurer l'état des canaux lors de recouvrements arrières. Un message est *en transit* si son émission est enregistrée dans l' état global de l'application alors que sa réception de l'est pas. Les algorithmes coordonnés enregistrent exactement en mémoire stable les messages en transit relativement a l'état global calculé alors que les algorithmes non coordonnés enregistrent la totalité des messages. Cet article, d'une part, définit une classe de messages qui ne peuvent être en transit dans aucun état global cohérent et d'autre part, propose un algorithme qui n'enregistre pas ce type de messages et permet ainsi de réduire l'occupation de la mémoire stable.

Mots-clé : Enregistrement optimiste de messages, états globaux cohérents, points de reprise, recouvrement arrière

1 Introduction

Checkpointing and backward recovery are well-known techniques for providing fault-tolerance in asynchronous systems. During the execution of an application a checkpointing algorithm computes and saves global checkpoints (global states) of the computation on stable storage. When a fault occurs, a backward recovery algorithm restores the computation in the most up to date global checkpoint from which this computation can be resumed.

A global checkpoint is composed of two parts: (1) a local checkpoint (local state) for each application process, and (2) a set of messages (channel state) for each communication channel. Informally, a global checkpoint is consistent if the computation could have passed through it.

A lot of checkpointing algorithms have been defined. They can be divided into two classes according to the way local checkpoints are determined. In the class of *coordinated* algorithms [5, 9, 17], the determination of local checkpoints and the computation of corresponding channel states are synchronized in such a way that the resulting global checkpoints are always consistent.

The main disadvantage of this approach is the added synchronization requiring control messages that can slow down or even freeze the computation for a given duration [9]. Its main advantage is that the last global checkpoint computed (as it is consistent) is the only one that has to be kept in stable storage. Moreover, when a fault occurs, the only work of a backward recovery algorithm consists in installing this global checkpoint and resuming the computation from it.

In the class of *uncoordinated* checkpointing algorithms [2, 6, 7, 8, 19, 21], processes save local checkpoints on stable storage in an independent way. Furthermore, they also log messages on stable storage so that channel states can be determined. When a failure occurs, it is up to the backward recovery algorithm to compute a consistent global state from local checkpoints [18] and messages saved on stable storage. The message logging technique is called *sender-based* (respt. *receiver-based*) if messages are logged by their senders (respt. receivers). Two logging techniques are possible. In the case of *pessimistic* logging, messages are logged on stable storage at the time they are sent (respt. received). This can incur high overhead in failure-free executions as each sending (respt. receiving) entails an additional input/output. This is why optimistic message logging techniques have been developed. In that case, when a message is sent (respt. received), it is saved on a volatile log and this log is saved on stable storage when the process takes a local checkpoint (as all messages are saved, the stable storage contains all in-transit messages).

In this paper we are interested in uncoordinated checkpointing algorithms with optimistic sender-based message logging. We develop a technique allowing a process to save the smallest possible number of volatile log messages on stable storage. This subset is consistent in the sense that it includes all the messages which are in-transit with respect to any consistent global checkpoint from which the computation could be resumed.

This paper is composed of two main parts. Section 2 introduces the model of distributed computations and formally defines the consistency of a global checkpoint. Section 3 presents the checkpointing algorithm.

2 Consistent Global Checkpoints

2.1 Distributed Computations

A distributed program is made up of n sequential local programs which, when executed, can communicate and synchronize only by exchanging messages. A distributed computation describes the execution of a distributed program. The execution of a local program gives rise to a sequential process. Let $P_1, P_2, \ldots, P_n$ be this finite set of processes. We assume that, at run-time, each ordered pair of communicating processes (P_i, P_j) is connected by a reliable and FIFO channel c_{ij} through which P_i can send messages to P_j. Message transmission delays are finite but unpredictable. Process speeds are

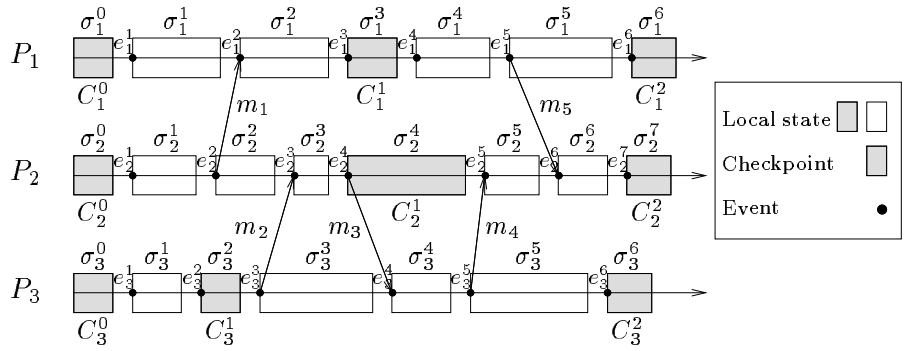

Figure 1: Local states of a distributed computation.

positive but arbitrary. In other words, the underlying computation model is asynchronous. Processes can fail by crashing, i.e. halting prematurely. The local program associated with P_i can include send, receive and internal statements; some statements can be non-deterministic[1].

Execution of an internal, send or receive statement produces an internal, send or receive event. Let e_i^x be the x^{th} event produced by process P_i. The sequence $h_i = e_i^1 e_i^2 \ldots e_i^x \ldots$ constitutes the history of P_i. h_i^x denotes the partial history $e_i^1 \ldots e_i^x$ of process P_i.

Let H be the set of events produced by a distributed computation. This set is structured as a partial order by Lamport's causal precedence relation [10], denoted "$\rightarrow$" and defined as follows:

$e_i^x \rightarrow e_j^y$ if, and only if:

1. $i = j$ and $x \leq y$, or
2. e_i^x is the sending of a message whose receiving is e_j^y, or
3. $\exists e_k^z : e_i^x \rightarrow e_k^z$ and $e_k^z \rightarrow e_j^y$ (transitive closure).

The partial order $\widehat{H} = (H, \rightarrow)$ constitutes a model of the distributed execution it is associated with.

2.2 Local States and Local Checkpoints

Let σ_i^0 be the initial state of process P_i. Event e_i^x entails P_i's local state change from σ_i^{x-1} to σ_i^x: σ_i^x is the local state of P_i resulting from its local history h_i^x; we say that e_i^y *belongs to* σ_i^x if $y \leq x$. Figure 1 depicts a distributed computation where events are denoted by black points and local states by (white or grey) rectangular boxes.

Let Σ be the set of all local states associated with a distributed computation $\widehat{H}$. Lamport's precedence relation can be extended to local states in the following way:

$$\sigma_i^x \rightarrow \sigma_j^y \Leftrightarrow e_i^{x+1} \rightarrow e_j^y$$

Local states not related by "$\rightarrow$" are said to be *concurrent*, (denoted $\|$); more formally:

$$\sigma_i^x \| \sigma_j^y \Leftrightarrow \neg(\sigma_i^x \rightarrow \sigma_j^y) \text{ and } \neg(\sigma_j^y \rightarrow \sigma_i^x)$$

In Figure 1, we have $\sigma_3^2 \rightarrow \sigma_2^4$ and $\sigma_1^3 \| \sigma_3^6$.

A local checkpoint of process P_i is a local state of P_i saved on stable storage. So, the set of all local checkpoints is thus a subset of Σ. C_i^t will denote the t-th local checkpoint of P_i; it corresponds

[1] When the only non-deterministic statements are "*receive*" statements, the program is *piece-wise* deterministic. If statements other than "*receive*" are non-deterministic the program is *fully* non-deterministic.

to some local state σ_i^x with $t \leq x$. We assume that each process takes an initial local checkpoint C_i^0 corresponding to σ_i^0. Grey rectangular boxes on Figure 1 depict local states that correspond to local checkpoints.

Determining which local states of a process are selected as local checkpoints, and consequently which are not, is the job of a checkpointing algorithm superimposed on the computation. When this algorithm defines the current local state of a process P_i as a local checkpoint, it saves it on stable storage, and P_i is said to "take a checkpoint".

2.3 In-transit Messages

A message m is *in-transit* with respect to an ordered pair (C_i^s, C_j^t) if its sending event belongs to C_i^s while its receiving event does not belong to C_j^t. In Figure 1, m_4 is in transit with respect to (C_3^2, C_2^1). In-transit messages actually constitute channel states which have to be considered to define consistency of global checkpoints.

2.4 Consistent Global Checkpoints

Informally, a consistent global checkpoint is a global state of a computation through which this computation could have passed. It consists of: (1) a set of concurrent local checkpoints (one per process), and (2) for each channel c_{ij}, the sequence[2] of messages that are in transit with respect to the ordered pair of local checkpoints of P_i and P_j. Formally, let C_i be a local checkpoint of process P_i and c_s_{ij} a state of the channel c_{ij}, i.e. a sequence of messages sent by P_i to P_j. A global checkpoint G is defined as:

$$G = \cup_i \{C_i\} \cup_{i,j} \{c\_s_{ij}\}$$

G is *consistent* if for any ordered pair (P_i, P_j):

- $\neg(C_i \rightarrow C_j)$ (C_j does not depend on C_i).
- c_s_{ij} is the sequence of messages in transit with respect to (C_i, C_j).

As an example, let us consider Figure 1. C_3^1 and C_2^1 cannot belong to a same consistent global checkpoint as $C_3^1 \rightarrow C_2^1$. C_1^1, C_2^1 and C_3^2 can belong to a same consistent global checkpoint when considering all channel states empty except $c\_s_{32} = \{m_4\}$.

This consistency definition is the classic one used to define global checkpoints. The algorithms described in [5, 9] are classic examples of coordinated checkpointing algorithms. Due to the synchronization they use to define local checkpoints and associated channels states, coordinated checkpointing algorithms ensure the a priori consistency of the global checkpoints they determine.

3 Efficient Optimistic Sender-Based Message Logging

Uncoordinated checkpointing algorithms replace the previous a priori synchronization by tracking dependence on local checkpoints and by logging enough messages so that all in-transit messages can be retrieved to determine correct channel states. In the rest of this paper, we are interested in optimistic sender-based message logging.

[2] As we consider FIFO channels, their states are sequences of messages. If channels were not FIFO, their states would be sets of messages.

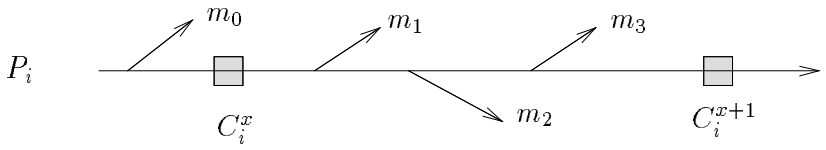

Figure 2: Sender-based message logging.

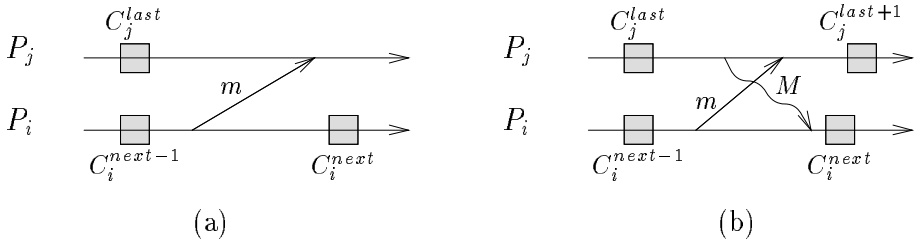

Figure 3: To log or not to log.

3.1 Optimistic Sender-Based Logging

The principle of optimistic sender-based message logging is illustrated on Figure 2. When a message is sent, it is momentarily logged by its sender on a volatile log. More precisely, when P_i takes a local checkpoint C_i^x, (1) it saves C_i^x and the current content of the volatile log on the stable storage and (2) it re-initializes the volatile log to empty. Then P_i logs on its volatile log all the messages it sends till its next local checkpoint C_i^{x+1} (m_1, m_2 and m_3 in the example depicted in Figure 2). In this way, messages are saved on stable storage "by batch" and not independently. This decreases the volume of input/output and, consequently, the overhead associated with message logging.

3.2 To Log or not to Log on Stable Storage?

The question we want to answer is the following one: "given a message m stored in its volatile log, should P_i log it on stable storage when it takes its next local checkpoint C_i^{next} ?" Basically, a message has not to be saved on stable storage if it cannot be in transit in any consistent global checkpoint. To answer the previous question more precisely, let us consider the message m in Figures 3.a and 3.b; C_j^{last} denotes the last local checkpoint taken by P_j before receiving m.

In Figure 3.a, C_j^{last} and C_i^{next} are concurrent and can therefore belong to a same consistent global checkpoint in which $c\_s_{ij} = \{m\}$. In this case, m, kept in the volatile log of P_i, must be saved on stable storage when C_i^{next} is taken. In that way, this in-transit message can be retrieved if a consistent global checkpoint including C_j^{last} and C_i^{next} has to be restored.

Let us now consider Figure 3.b where M denotes a causal chain[3] of messages starting after C_j^{last} and arriving at P_i before C_i^{next}. We have then $C_j^{last} \rightarrow C_i^{next}$ from which we conclude that C_j^{last} and C_i^{next} cannot belong to the same consistent global checkpoint.

Let C_j^{last+1} be the first local checkpoint taken by P_j after receiving m. When considering any global checkpoint including C_i^x and C_j^y:

- if $x \leq next - 1$ and $y \leq last$: m has been neither sent nor received.
- if $x \geq next$ and $y \geq last + 1$: m has been sent and received.

[3] A causal chain of messages is characterized by the following property: every message of the chain (but the first one) is received, according to "$\rightarrow$", before the sending of the next one.

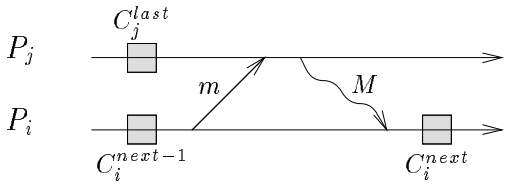

Figure 4: P_i can decide not to log on stable storage.

As, additionally, $C_j^{last} \rightarrow C_i^{next}$ (because of M) and $C_i^{next-1} \rightarrow C_j^{last+1}$ (because of m), we can conclude that m cannot be in transit in any consistent global checkpoint. It follows that m need not be logged in stable storage when P_i takes C_i^{next}.

Figure 3.b exhibits the pattern of local checkpoints and message exchanges in which a message m has not to be logged on stable storage. The problem is now to answer the following question: "how can a process P_i know that a message m it sent to P_j will never be in transit with respect to any pair of local checkpoints (C_i^s, C_j^t)?" The only way[4] for P_i to learn this is on receiving a message piggybacking the information "m has been received by P_j". This situation is described in Figure 4 where the causal chain M conveys this information to P_i. On the contrary, in Figure 3.b, the chain M missed the information "P_j receives m".

The uncoordinated checkpointing algorithm described in the next section exploits the flow of messages exchanged by a distributed computation to convey control information used to reduce the number of messages logged on stable storage by their senders.

3.3 A Checkpointing Algorithm

We assume an uncoordinated checkpointing algorithm[5] that uses an optimistic sender-based message logging strategy. We augment it, according to the previous discussion (Section 3.2), in order to reduce the number of messages transferred from a volatile log to stable storage.

Let CKT_i be the checkpointing algorithm associated with P_i. CKT_i is augmented with the following data structures:

- vol_log_i: a volatile log initialized to empty.
- $sn_i[1..n]$: an array of sequence number generators.
  $sn_i[j] = \alpha \Leftrightarrow P_i$ sent α messages to P_j.
- $known\_rec_i[1..n, 1..n]$: a matrix of sequence numbers.
  $known\_rec_i[j,k] = \beta \Leftrightarrow P_i$ knows P_j has received β messages from P_k.

These three data structures are managed[6] and used by CKT_i. The following one is managed by CKT_i but used (with the previous ones) only by the recovery algorithm (Section 3.4) to compute a set of concurrent local checkpoints.

- $ckpt\_ts_i[1..n]$: a vector clock for local checkpoints.
  $ckpt\_ts_i[j] = \gamma \Leftrightarrow CKT_i$ knows P_j has taken γ local checkpoints.

[4]Remember we are in the context of uncoordinated checkpointing algorithms and, consequently, no additional control messages are used to convey control information.

[5]The most encountered uncoordinated checkpointing algorithm is the periodic one: at regular time intervals, each process individually takes local checkpoints.

[6]If we suppose processes do not send messages to themselves, entries $known\_rec_i[x,x]$ remain equal to zero. These entries can therefore be used to store the array sn_i and consequently save space.

Vector $ckpt\_ts_i$ is a vector clock [11] representing P_i's knowledge of the number of local checkpoints taken by each process.

When P_i sends or receives a message or when it takes a local checkpoint, CKT_i is required to execute the following statements atomically:

when P_i **sends** m **to** P_j
begin $sn_i[j] := sn_i[j] + 1$;
  append $(m, sn_i[j], j)$ **to** vol_log_i;
  send $(m, known\_rec_i, ckpt\_ts_i)$ **to** P_j
end
when P_i **receives** (m, k_r, c_ts) **from** P_j
begin **deliver** m **to** P_j;
  $known\_rec_i[i, j] := known\_rec_i[i, j] + 1$;
  $\forall (x, y) : known\_rec_i[x, y] := \max(known\_rec_i[x, y], k\_r[x, y])$;
  $\forall x : ckpt\_ts_i[x] := \max(ckpt\_ts_i[x], c\_ts[x])$
end
when a local checkpoint is taken by P_i
begin $ckpt\_ts_i[i] := ckpt\_ts_i[i] + 1$;
  let $x = ckpt\_ts_i[i]$;
  let $C_i^x = P_i$'s current local state; /* last local checkpoint */
  $\forall (m, sn_m, dest_m) \in vol\_log_i$:
    do **if** $sn_m \leq known\_rec_i[dest_m, i]$
      then suppress $(m, sn_m, dest_m)$ **from** vol_log_i
      fi
    od
  save on stable storage $(C_i^x, vol\_log_i, ckpt\_ts_i, known\_rec_i[i, *], sn_i)$;
/* $known\_rec_i[i, *]$ is a vector whose value is the i-th line of the matrix $known\_rec_i$ */
  $vol\_log_i := \emptyset$
end

3.4 A Recovery Algorithm

A recovery algorithm, similar to the one described in [21], can be associated with the previous checkpointing algorithm. Upon a fault occurrence, the recovery algorithm executes the following steps.

1. First, non-faulty processes are required to take a local checkpoint (this allows as much correct computation as possible to be saved).

2. Then, the most recent set of n concurrent local checkpoints is determined. This determination is done in the following way:

   - construct a set $\{C_1, C_2, \ldots, C_n\}$ by taking the last local checkpoint C_i of each process P_i; let c_ts_i be its timestamp (value of $ckpt\_ts_i$ when C_i is saved on stable storage)[7].
   - **while** $\exists (i, j) : C_i \rightarrow C_j$
     do let C'_j be the first predecessor of C_j such that $\neg(C_i \rightarrow C'_j)$
     $C_j := C'_j$
     od

   The set $\{C_1, \ldots, C_i, \ldots, C_n\}$ obtained contains the latest local local checkpoints which are concurrent.

[7] Due to the vector clock properties, we have: $C_i \rightarrow C_j \Leftrightarrow \forall x : c\_ts_i[x] \leq c\_ts_j[x]$ and $c\_ts_i \neq c\_ts_j$.

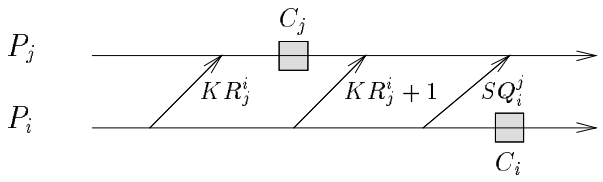

Figure 5: Determining in-transit messages.

3. The state of each channel c_{ij} (messages in transit with respect to the ordered pair (C_i, C_j)) is extracted from the stable storage of P_i in the following way. Let SQ_i^j and KR_j^i be the values of $sn_i[j]$ and $known\_rec_j[j,i]$ saved on stable storage with C_i and C_j, respectively. As channels are FIFO, messages sent by P_i to P_j which are in transit with respect to (C_i, C_j) have sequence numbers x such that $KR_j^i < x \leq SQ_i^j$ (Figure 5).

   This set of messages is contained in P_i's stable storage (see the discussion of Section 3.2) from which they can be extracted to constitute a consistent global checkpoint.

3.5 Discussion

Unlike coordinated checkpointing algorithms, "pure" uncoordinated checkpointing algorithm cannot a priori prevent the occurrence of the *domino effect* [13] when a fault occurs. In the worst case, it is not possible to construct a set of n concurrent local checkpoints distinct from the initial set of local checkpoints. To prevent the occurrence of the domino effect, uncoordinated checkpointing algorithms can be made "adaptive": in that case, processes are required to take additional local checkpoints so that any local checkpoint belongs to at least one set of n concurrent local checkpoints. Contrary to coordinated algorithms, the implementation of this "adaptiveness" does not require the addition of control messages. [1, 3, 16] present such "adaptive" uncoordinated checkpointing algorithms.

Remarks: About strategies for preventing the domino effect.
In [12] Netzer and Xu state and prove a necessary and sufficient condition for a local checkpoint to be useless; a local checkpoint is *useless* if it cannot belong to any consistent global checkpoint. Useless local checkpoints are *the* cause of domino effects.
In [4] we formally studied the modeling of consistent global checkpoints and the domino effect in distributed systems. We extended the following result due to Russell [16] (this result states a sufficient condition to eliminate the domino effect).

- Russell's adaptive algorithm. Let P_i be a process and $ckpt_i$, $receive_i$ and $send_i$ denote the event "P_i takes a local checkpoint", "P_i receives a message" and "P_i sends a message", respectively. If, for any P_i, the sequence of events produced by P_i obeys the following (regular language) pattern:

  $$(ckpt_i \ receive_i^* \ send_i^*)^*$$

  then no local checkpoint will be useless, and consequently the domino effect can not occur. It follows that the domino effect will be prevented by forcing each process P_i to take an additional local checkpoint between each sequence of $send_i$ events and each sequence of $receive_i$ events.

- Russell's adaptive algorithm follows (in some sense) a "brute force" strategy as the decision to force a process P_i to take an additional local checkpoint is only based on P_i's local behaviour.

  A more sophisticated algorithm preventing the domino effect is presented in [3]. This adaptive algorithm uses global information (piggybacked by application messages) to force processes to take "as few as possible" additional local checkpoints in order no local checkpoint be useless. When there is no look-ahead on the future of the execution (a realistic assumption!) this algorithm is nearly optimal with respect to the number of additional local ckeckpoints that are taken.
  End of remarks

Local checkpoints and messages logged on stable storage constitute a space overhead. A space reclamation algorithm is described in [22]. Basically, as soon as a consistent global checkpoint has been determined, all local checkpoints and messages that belong to its "past" can be discarded.

Sender-based message logging algorithms have been investigated for a long time [7, 19]. They usually either consider executions of piece-wise deterministic distributed programs [6, 8, 19] or log all messages. In the context of receiver-based message logging, [21] characterizes a particular set of messages that are not in transit with respect to any pair of local checkpoints; [23] adapts [21] to the case of sender-based logging. Expressed in our framework, [23] actually considers causal chains composed of only one message; this reduces the size of control information carried by messages but increases the number of messages that are logged on stable storage though they can not belong to a consistent global checkpoint.

A cost of the algorithm we presented lies in the matrix $known\_rec_i$, messages have to piggyback. Actually only entries of the matrix corresponding to edges of the communication graph have to be considered. In the case this graph is a directed ring, the matrix shrinks and becomes a vector. Matrices with similar semantics are used (1) in [20] to solve the distributed dictionary problem and (2) in [14] to implement causal message delivery in point-to-point communication networks. As shown in [15] the size of such matrices can be reduced by appropriate techniques when considering the actual communication graph.

4 Conclusion

This paper has studied the logging of messages in the context of optimistic sender-based uncoordinated checkpointing algorithms. A global checkpoint is consistent if it is composed of a set of n concurrent local checkpoints and a set of channel states recording all messages sent but not received (*in-transit* messages) with respect to these local checkpoints.

Coordinated checkpointing algorithms log those messages exactly. Because of their lack of synchronization, uncoordinated algorithms can be more efficient in failure-free computation but conservatively log more messages. This paper has presented a general technique to reduce the number of messages logged by uncoordinated checkpointing algorithms. An appropriate tracking of the causal dependencies of the underlying computation message exchanges constitutes the core of the protocol.

References

[1] A. Acharya, B.R. Badrinath, Checkpointing Distributed Applications on Mobile Computers, *Proc. 3rd Int. Conf. on Par. and Dist. Information Systems*, 1994.

[2] L. Alvisi, K. Marzullo, Message Logging: Pessimistic, Optimistic, and Causal, *Proc. 15th IEEE Int. Conf. on Distributed Computing Systems*, 1995, pp. 229-236.

[3] R. Baldoni, J. M. Hélary, A. Mostefaoui, M. Raynal, Consistent Checkpointing in Distributed systems, INRIA *Research Report* 2564, June 1995, 25 p.

[4] R. Baldoni, J. Brzezinski, J.M. Hélary, A. Mostefaoui, M. Raynal, Characterization of Consistent Checkpoints in Large Scale Distributed Systems. *Proc. 6th IEEE Int. Workshop on Future Trends of Dist. Comp. Sys.*, Korea, pp. 314–323, August 1995.

[5] K.M. Chandy, L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. on Comp. Sys.*, Vol. 3(1), 1985, pp. 63-75.

[6] E.N. Elnozahy, W. Zwaenepoel, Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit, *IEEE Trans. on Computers*, Vol. 41(5), 1992, pp. 526-531.

[7] D.B. Johnson, W. Zwaenepoel, Sender-Based Message Logging, *Proc. 17th IEEE Conf. on Fault-Tolerant Computing Systems*, 1987, pp. 14-19.

[8] D.B. Johnson, W. Zwaenepoel, Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing, *Journal of Algorithms*, Vol. 11(3), 1990, pp. 462-491.

[9] R. Koo, S. Toueg, Checkpointing and Rollback-Recovery for Distributed Systems, *IEEE Trans. on Software Engineering*, Vol. 13(1), 1987, pp. 23-31.

[10] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol. 21(7), 1978, pp. 558-565.

[11] F. Mattern, Virtual Time and Global States of Distributed Systems. In Cosnard, Quinton, Raynal, and Robert, Editors, *Proc. Int. Workshop on Dist. Alg.*, France, October 1988, pp. 215-226, 1989.

[12] R.H.B. Netzer, J. Xu, Necessary and Sufficient Conditions for Consistent Global Snapshots, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6(2), 1995, pp. 165-169.

[13] B. Randell, System Structure for Software Fault-Tolerance, *IEEE Trans. on Software Engineering*, Vol. 1(2), 1975, pp. 220-232.

[14] M. Raynal, A. Schiper, S. Toueg, The Causal Ordering Abstraction and a Simple Way to Implement it, *Inf. Processing Letters*, Vol. 39, 1991, pp. 343-350.

[15] F. Ruget, Cheaper Matrix Clocks, *Proc. 8th Int. Workshop on Distributed Algorithms*, Springer Verlag, LNCS 857, pp. 340-354, 1994.

[16] D.L. Russell, State Restoration in Systems of Communicating Processes, *IEEE Trans. on Software Engineering*, Vol. 6, 1980, pp. 183-194.

[17] L.M. Silva, J.G. Silva, Global Checkpointing for Distributed Programs, *Proc. 11th IEEE Symp. on Reliable Distributed Systems*, Houston, TX, 1992, pp. 155-162.

[18] M. Singhal, F. Mattern, An Optimality Proof for Asynchronous Recovery Algorithms in Distributed Systems, *Inf. Processing Letters*, Vol. 55, 1995, pp. 117-121.

[19] R.E. Strom, S. Yemini, Optimistic Recovery in Distributed Systems, *ACM Transactions on Computer Systems*, Vol. 3(3), 1985, pp. 204-226.

[20] G.T. Wuu, A.J. Bernstein, Efficient Solutions to the Replicated Log and Dictionary Problems, *Proc. 3rd ACM Symp. on Principles of Dist. Comp.*, 1984, pp. 233–242.

[21] Y.M. Wang, W.K. Fuchs, Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems, *Proc. 11th IEEE Symp. Reliable Distributed Systems*, 1992, pp. 147-154.

[22] Y.M. Wang, P.Y. Chung, I.J. Lin, W.K. Fuchs, Checkpointing Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6(5), 1995, pp. 546-554.

[23] J. Xu, R.H.B. Netzer, M. Mackey, Sender-Based Message Logging for Reducing Rollback Propagation, *Proc. 7th IEEE Symp. on Parallel and Distributed Processing*, 1995, pp. 602-609, San Antonio.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399