Semantics of Recovery Lines for Backward Recovery in Distributed Systems

Jerzy Brzezinski, Jean-Michel Hélary, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Semantics of recovery lines for backward recovery in distributed systems

Jerzy Brzeziński,

Jean-Michel Helary, Michel Raynal

N° RR-2468

Mars 1995

PROGRAMME 1

ISSN 0249-6399

Semantics of recovery lines for backward recovery in distributed systems

Jerzy Brzeziński *,
Jean-Michel Helary, Michel Raynal **

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Algorithmes Distribués et Protocoles

Rapport de recherche n ° RR-2468 — Mars 1995 — 30 pages

Abstract: This paper addresses the definition of *recovery lines* in the context of backward recovery whose aim is to cope with failures in distributed sytems. A general framework that allows for several semantics of recovery lines is introduced. Key notions such as *missing messages* and *orphan messages* are precisely defined and their impact on the definition of consistency of recovery lines is carefully analyzed. Basic mechanisms such as local checkpointing, messages identification and (optimistic or pessimistic) messages logging are then discussed as an illustration of (coordinated or uncoordinated) checkpointing protocols.

Key-words: Distributed systems, fault tolerance, recovery, checkpointing, consistency

(Résumé : tsvp)

This work was supported by CNET projet CESAME (grant 92 1B 178)

*Poznań University of Technology, brzezinski@pozn1v.put.poznan.pl
**{helary}{raynal}@irisa.fr

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

Sémantique des états de reprise pour le rétablissement dans les systèmes distribués

Résumé : Cet article est consacré à la définition des *états de reprise* dans le cadre des techniques de rétablissement, utilisées pour traiter les défaillances dans les systèmes distribués. On introduit un cadre général permettant de considérer plusieurs sémantiques d'états de reprise. Des notions clefs telles que celles de *messages manquants* ou *messages orphelins* sont définies avec précision, et leur influence sur la définition de la cohérence d'un état de reprise est soigneusement analysée. Les outils de base, tels que les points de contrôle locaux, le stockage des messages (optimiste ou pessismiste) et la ré-exécution, sont introduits pour illustrer des algorithmes de reprise, coordonnés ou non.

Mots-clé : Systèmes répartis, tolérance aux fautes, reprise, points de contrôle, cohérence

1 Introduction

A distributed system may be thought of as a network of nodes (sites) interconnected by transmission channels. Each node consists of computing and storage facilities, and an interface to local users and to network channels. The nodes exchange informations with each other only by message passing as no common memory is available. Most of distributed systems, including computer networks and distributed memory massively parallel computers, are asynchronous, i.e., they do not use a common clock and do not impose any bounds on relative processor speeds or message transmission times. The basic motivation for development of distributed systems is the possibility to meet the ever increasing demand of computer applications, concerning mainly system performance and reliability (among other things). Enhanced performance is achieved executing distributed programs which incorporate many nodes, each node supporting one or several processes running simultaneously in realization of a common goal. High reliability, related to the success with which a system provides the service specified ([33]), potentially results from redundancy of compatible system components. In practice however, asynchrony and inherent complexity of distributed systems imply nondeterminism of distributed programs, which make coordination between distributed nodes difficult and increase the potential for system impairments to reliability.

Reliability issues are increasingly important for a large number of distributed systems, such as computer integrated manufacturing systems, time-critical systems of monitoring and control (e.g., aircrafts control, nuclear power-station monitoring and control), banking systems, etc. As stated in [23], impairments to reliability are *faults, errors* and *failures. Faults* may deviate the system state from its specification and cause *erroneous states.* In this context the term *error* is used to designate that part of the state which is incorrect. An error is liable to lead to subsequent *failures*, that occur when an erroneous state of the system is processed by a program. In other words, occurrence of a failure means that the delivered service no longer complies with the specifications.

Natural way for reliability improvement is *fault avoidance.* Unfortunately avoidance of all faults is not possible in practice both due to complexity of hardware and software of distributed systems as well as magnitude and variety of faults (permanent and transient faults, hardware faults, software faults, communication faults, system coordination faults, etc. ([9, 26, 32, 33]). Thus, to be realistic we have to accept that faults are inevitable attribute of real distributed systems. An alternative or complementary approach to fault avoidance is that of *fault tolerance.* In order to cope with failures, fault-tolerant systems are equipped with additional compo-

nents and programs (algorithms). These components and algorithms attempt to ensure that occurrences of erroneous states do not result in later system failures. Ideally, they remove these errors and restore systems to correct states from which normal processing can continue. There are many issues regarding to realization of fault-tolerant systems: error detection, damage confinement and assessment, error diagnosis, and error recovery, among other things ([26, 33]). Though all these issues are very important, in this paper we concentrate solely on error recovery. Other issues are orthogonal to this subject, thus can be considered separately and are not dealt with here.

In general *error recovery* in computer systems is a task that involves restoring an error-free state from an erroneous one. Error recovery schemes are usually classified into *forward* and *backward* error recovery ([33]). *Forward error recovery* is based on attempting to make further use of the state which has just been found to be in error. This is possible when the nature of the error and consequences of faults can be completely assessed. Then one can remove those errors and enable the system to move forward. Usually forward error recovery strategies are based on the use of error correcting techniques and codes, or error compensation providing supplementary information. *Backward error recovery* involves, first, backing up one or more of the processes of the system to a previous state, which is hoped to be error-free, before attempting to continue further operations. In this scheme it is not necessary to foresee the nature and consequences of all the faults, and to remove all the errors. Thus, it makes possible to recover from an arbitrary fault simply by restoring an appropriate previous state to the system. These features enable backward recovery to be considered as a general recovery mechanism to any type of fault. However, backward recovery needs *recovery points,* i.e., effective means by which a state of a process can be saved and later restored. For obtaining such recovery points two basic techniques can be applied: checkpointing and message logging (audit trial). *Checkpointing* is an operation that stores the correct state of a process on stable storage, so that, on recovery, the process can resume its computation from the checkpointed state. *Message logging* is on operation which saves messages on stable storage; thus, exactly the same sequence of states can be reproduced after a rollback by reprocessing those messages.

Backward recovery seems to be a simple, general and hence attractive approach for increasing distributed systems reliability. However, it turns out that this technique involves also many problems. First, the system performance penalty may be not negligible; three key performance issues in this context are: failure-free overhead, extent of rollback, and output commit latency. Second, there is danger that faults

are permanent and re-execution could be useless. Third, some states of the system may be unrecoverable. These problems are especially difficult in the context of asynchronous distributed systems due to their inherent nondeterminism.

This paper intends to provide a general framework for intrinsic analysis of backward recovery problems. It focuses on the concept of recovery lines, that is to say states of the system which are recoverable, and addresses very carefully their consistency issues, which are central in backward recovery techniques. For that purpose, it is shown that when a recovery semantics is associated with messages, the set of acceptable (or failure-free) computations is enlarged. This is achieved by defining the notions of *orphan* and *missing* messages. It must be emphasized that in most of the papers that can be found in the literature on this problem, these issues are addressed only partially and, as a consequence, consistency definitions are difficult to understand since they are based upon implicit hypotheses or upon not precise enough definitions.

In Section 2, models for distributed systems, programs and computations are described with detail, necessary for a good understanding of the following parts. Section 3, is devoted to the backward recovery problem and related concepts of recovery line and consistency are carefully defined. Sections 4 and 5 introduce the basic tools and strategies used throughout the literature for implementing backward recovery.

2 Model for Distributed Programs and Computations

2.1 The underlying system model

The *underlying distributed system*, supporting the execution of distributed programs, consists of a set of *nodes* that are connected by *communication links*.

2.1.1 Processors and local memories

Each node is equipped with processor, memory, interface to network links and, eventually, interface to local users. A processor executes computation at its own speed defined by its internal local clock. Local clocks of different processors are not synchronized, hence there is no bound on relative processors speeds. Processor states, programs and data are stored in memory which, in general, consists of volatile storage and stable storage.

Volatile storage offers very fast access and therefore is usually used for main memory (operating store) directly accessible by a processor. However, the content

of volatile storage is irretrievably lost whenever a failure occurs. The *stable storage* is relatively slow (its access time may be orders of magnitude higher than that of the volatile storage), and therefore it is mainly used as a secondary store. On the other hand stable storage persists accross any kind of failures and thus it can maintain information needed to reconstruct volatile storage.

We distinguish *fail-stop processors* ([35]). Their internal state and some predefined portion of the memory are assumed to be volatile. In contrast to real systems, a fail-stop processor never performs an erroneous state transition; instead, the processor simply halts. Thus, the only visible effects of a failure in a fail-stop processor are: stopping its execution as well as loss of its internal state and the contents of its volatile storage. In the fail-stop model, the failure of a process can be detected after a finite time by other processes, using some specific techniques (e.g., time out mechanisms).

2.1.2 Communication links

Nodes exchange information with each other only by messages passing through *communication links*. Links are bidirectional and are used to transmit messages in both direction. Each link has its own buffer whose capacity is assumed (for the sake of simplicity) to be infinite. It enables asynchronous communications, in which a sender processor hands over a message to a link, initiates sending, and immediately continues its further activity. Then, the link transfers the message to the destination node (receiver processor) and keeps the message within a buffer until it is retrieved by this receiver.

The *transmission delay* (time elapsed between the initiation of the transfer and the corresponding deposit of the message into the buffer), is finite but unpredictable. This is a characteristic of asynchronous links which, as opposed to synchronous ones, do not impose any bound on transmission delays. Usually links preserve (obey) first-in-first-out (FIFO) order: on the same link, messages are put into its buffer in the order they were sent. Links can also guarantee, in a way fully transparent for nodes, that no message is lost, duplicated or corrupted. In such a case, links are said to be *reliable (lossless, duplicate-free, error-free*, respectively).

2.2 Distributed programs

2.2.1 Local programs

A *distributed program* (or application) is a collection of n local programs $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, and each local program is the expression of a local algorithm. At run-time, each local

program will be executed by a processor of the underlying system. Each program is composed of atomic operations (also named statements). Depending on the granularity of the distributed program, different types of operations are considered. In general, these operations can be classified into two categories: internal and communication operations. An *internal operation* of a local program involves only its local data and control structures, whilst a *communication operation* involves either the sending or the receiving of a message and one or several channels.

2.2.2 Messages and channels

A *message* is a dynamic data structure from which it is possible to retrieve essentially two kinds of information:

- the identifier of the message, including the identifiers of the message creator (sender local program) and its consumer (receiver local program),
- a content (data).

A *channel* is a communication variable associated with an ordered pair of local programs, and the type of this variable is a set of messages. We denote by c_{ij} the channel associated with the ordered pair $(\mathcal{P}_i, \mathcal{P}_j)$. Every channel c_{ij} is implemented at run-time by a directed path of links of the underlying system. Two communication operations, namely *send* and *receive* can be used to exchange messages through channels.

- $send(m, \mathcal{P}_j)$. In program $\mathcal{P}_i$, this operation is parameterized by a single message m and the identity of another local program $\mathcal{P}_j$ (or the corresponding channel c_{ij}). As the effect of this operation execution, the message m is added into the channel c_{ij} : $c_{ij} := c_{ij} \cup \{m\}$.
- *receive(vm,S).* In program $\mathcal{P}_j$, this operation is parameterized by a single message variable vm and by a set S specifying the possible senders of the message $\mathcal{P}_j$ is waiting for (or equivalently S is a set of input channels of $\mathcal{P}_j$). If one of the input channels of S is not empty, let it be c_{ij}, a message m is removed from it ($c_{ij} := c_{ij} \setminus \{m\}$) and put into vm ($vm{:=}m$). If all channels specified in S are empty when the *receive* operation is invoked, two semantics can be considered: *blocking receive*, meaning that the operation will not be terminated until a message m belongs to one of channels c_{ij} specified in S, or *unblocking receive*, meaning that the operation has no effect.

2.2.3 Non-deterministic statements

In programs, some ad-hoc control structures allow operations to be combined into non-deterministic statements. Non-deterministic statements involve a choice between several possible operations, leaving the resolution of this choice completely undetermined. For example, the *alternative control structure* in languages such as CSP or ADA is a non-deterministic statement. Non-deterministic statements are inherent to distributed programs. The most characteristic is nondeterminism of receive statements. A receive operation *receive(vm,S)* is non-deterministic when the set S defining the possible senders for the message a process is waiting for, has more than one element; more specifically, the actual sender of the message, although belonging to S, is not defined statically and so not known in advance. Distinct executions of the same receive operation can then be associated with different senders; consequently they are not reproducible as far as senders are concerned, hence the name *non-deterministic* receive operations. This results from the fact that processors relative speeds are not known in advance, and transmission delays are unpredictable, although finite.

A program is *deterministic*, when all its statements are deterministic, otherwise, a program is said to be *nondeterministic*. The important case of non-deterministic programs is one in which all statements except *receive* are assumed to be deterministic. Such programs are called *piece-wise deterministic* or *quasi-determininistic*.

2.3 Distributed computations

2.3.1 Processes and events

A *distributed computation* is an execution of all the local programs of a distributed program on a distributed system. The execution of each local program $\mathcal{P}_i$ is a process, denoted by P_i. During an execution, processes produce *events*. An event produced by a process P_i is a particular execution of an atomic operation of the local program $\mathcal{P}_i$. Events are instantaneous and each has one of the following basic types *send*, *receive* or *internal*:

- The *send* event $send(P_i, P_j, m)$ is produced by process P_i when it executes the statement $send(m, c_{ij})$.

- The *receive* event $rec(P_j, P_i, m)$ is produced by process P_j when it executes the statement $receive(vm, S)$ and the message m received and consumed in vm has been sent by P_i.

- The *internal* event $int(P_i)$ is produced by process P_i when it executes an internal statement. We assume three particular internal events ([1]):
  - the *init* event $init(P_i)$ is produced by P_i when it starts.
  - the *fail* event $fail\ (P_i)$ occurs at process P_i, when the processor on which it is executing fails. The processors are assumed to be fail-stop.
  - the *stop* event $stop(P_i)$ terminates execution of P_i.

It is worth noting that a *send* event represents the execution of an asynchronous operation which physically puts a message into the buffer of a link. Then the underlying distributed system transmits the message to the destination node, which asynchronously captures incoming messages from the link and queues them for later retrieval by the intended recepient process (receiver). When a message is in a queue of the destination node we say that this message has *arrived*. In this context a *receive* event corresponds to the actual dequeuing of a message by the receiving process.

2.3.2 Partial orders on events

Processes of a distributed computation are *sequential*, in other words, each process produces a *sequence* of events. This sequence of events is called the *history* of P_i, and it is denoted by $h_i = e_i^0 e_i^1 \ldots e_i^s, \ldots$, where e_i^s, is s-th event executed by P_i (e_i^0 is the *init* event). Let h_i^s denote the *partial history* of P_i till the event e_i^s; $h_i^s = e_i^0 e_i^1 \ldots e_i^s$ is a prefix of h_i.

Local events at each process are totally ordered. However, as message transmission delays are unpredicatable, the set of all the events of the entire computation is only partially ordered. Precisely, let H be the set of all events that occurred in a distributed computation. Let "$\stackrel{e}{\rightarrow}$" denote the classical *causal precedence* (*happened-before*) partial order defined as follows ([22]):

$$e_i^x \stackrel{e}{\rightarrow} e_j^y \text{ if and only if :}$$

1. e_i^x and e_j^y are the same event, or
2. $i = j$ and $y = x + 1$, or
3. $i \neq j$ and e_i^x is the event $send(P_i, P_j, m)$ and e_j^y is the event $rec(P_j, P_i, m)$, or
4. there is an event e_k^r such that $e_i^x \stackrel{e}{\rightarrow} e_k^r \wedge e_k^r \stackrel{e}{\rightarrow} e_j^y$.

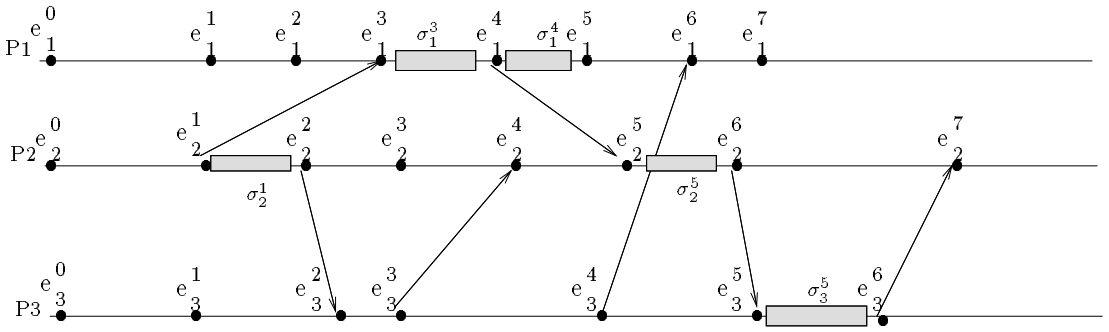


Figure 1: A sample distributed computation.

If $e_i^x \xrightarrow{e} e_j^y$ these events are called *causally dependent*. Otherwise, they are called *concurrent* (causally independent). Concurrent events are denoted by $e_i^x \parallel_e e_j^y$. A sample distributed computation involving three processes is depicted in Figure 1 using the classical space-time diagram. In that figure horizontal axes represent time progress of each process, black points represent events, and arrows from one process time line to another represent messages communication. Note, for example, that $e_2^1 \xrightarrow{e} e_3^5$, $e_2^2 \xrightarrow{e} e_1^7$, but $e_2^2 \parallel_e e_1^3$, $e_1^3 \parallel_e e_3^2$.

2.3.3 Local states of processes

As events are instantaneous, any two successive events e_i^s and e_i^{s+1} occurring at a process P_i define an interval of local time. The *local state* of a process P_i at any time belonging to this interval is defined by its partial history h_i^s; a local state is thus defined by a sequence of local events. Let us denote by σ_i^s the local state of process P_i produced by the event e_i^s. Then, by definition, an event e_j^x *belongs to* local state σ_i^s if and only if $i = j$ and $x \leq s$. Moreover, we can define a partial ordering on process local states as follows:

$$\sigma_i^s \xrightarrow{\sigma} \sigma_j^t \Leftrightarrow i = j \text{ and } s = t, \text{ or } e_i^{s+1} \xrightarrow{e} e_j^t$$

This relation means that one process state precedes another if and only if both states are the same, or the event giving rise to the latter causally depends on the event terminating the former. Process states that are incomparable under this relation are said to be *concurrent* (denoted $\parallel_\sigma$). In Figure 1, intervals between successive events, depicted by rectangular boxes, represent processes local states. Note, for example,

that $\sigma_2^1 \stackrel{\sigma}{\rightarrow} \sigma_3^5$ (since $e_2^2 \stackrel{e}{\rightarrow} e_3^5$); similarly, $\sigma_1^3 \stackrel{\sigma}{\rightarrow} \sigma_2^5$ (since $e_1^4 \stackrel{e}{\rightarrow} e_2^5$), but $\sigma_1^4 \parallel_\sigma \sigma_2^5$ (since the event e_1^5 terminating the state σ_1^4 is concurrent with the event e_2^5 producing the state σ_2^5).

2.3.4 Global states

Definition

A *global state* Σ of a distributed computation (in brief a state) is a set of local states, one for each process, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. As each local state σ_i is defined by a sequence of events that occurred at P_i, a global state Σ includes a set of *events*. More precisely, an event e *belongs to* a global state $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ (denoted by $e \in \Sigma$) if, and only if, there exists $i, 1 \leq i \leq n$, such that e belongs to σ_i.

With a global state $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ can be associated the set of *channel states*, where the state of the channel c_{ij} is the set of messages m in-transit on channel c_{ij} in the global state Σ, that is to say the set of messages m such that $send(P_i, P_j, m) \in \Sigma \wedge rec(P_j, P_i, m) \notin \Sigma$.

Consistency of global states

Note, that histories used in the definition of a global state Σ are not correlated or related one to the other. Thus, in general, a state Σ can include an event e_j^t which causally depends on an event e_i^s not belonging to Σ. However, this state is inconsistent as it cannot occur in an actual execution, as demonstrated by the causal precedence relation. Indeed, if such a global state could occur at a time moment, say τ, the event e_i^s would not have occurred at time τ, whereas the event e_j^t would have already occurred at that time; but this is a contradiction with the supposition that e_i^s must occur before e_j^t, as $e_i^s \stackrel{e}{\rightarrow} e_j^t$. Intuitevely, a state Σ is *consistent* with respect to a distributed computation if the computation might have passed through this state ([7, 36]).

More formally, a state $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is *consistent* if and only if:

$$(\forall e)\ (\forall e')\ ((e' \in \Sigma \wedge e \stackrel{e}{\rightarrow} e') \Rightarrow (e \in \Sigma))$$

This definition implies that if an event e' belongs to the consistent state Σ, and if e' causally depends on e, then e belongs also to the state Σ. In other words, a consistent state Σ containing the event e' contains also all events on which e' causally depends. One can note that a consistent global state is one in which the component

local states are concurrent:

$$\Sigma = \{\sigma_1, \ldots, \sigma_n\} \text{ is consistent } \Leftrightarrow (\forall i)(\forall j)((i \neq j) \Rightarrow (\sigma_i \parallel_\sigma \sigma_j))$$

Moreover, in such a state channels need not be empty, i.e., some messages can be sent but not yet received. A special type of consistent state is $\Sigma(\tau)$, the state at time τ, which is the collection of local states of all the processes at the same global time moment τ. The $\Sigma(\tau)$ states are more of theoretical constructs since they require some outside ideal observer to instantaneously and simultaneously capture the local states of all the processes. The Figure 2 shows two global states, indicated by boxes linked by a thick line. The first one is $\Sigma_1 = \{\sigma_1^2, \sigma_2^1, \sigma_3^1\}$ and is consistent; the second one is $\Sigma_2 = \{\sigma_1^5, \sigma_2^5, \sigma_3^5\}$ and is not consistent since, for example, $e_3^5 \in \Sigma_2, e_2^6 \notin \Sigma_2$ and $e_3^5 \xrightarrow{e} e_2^6$.

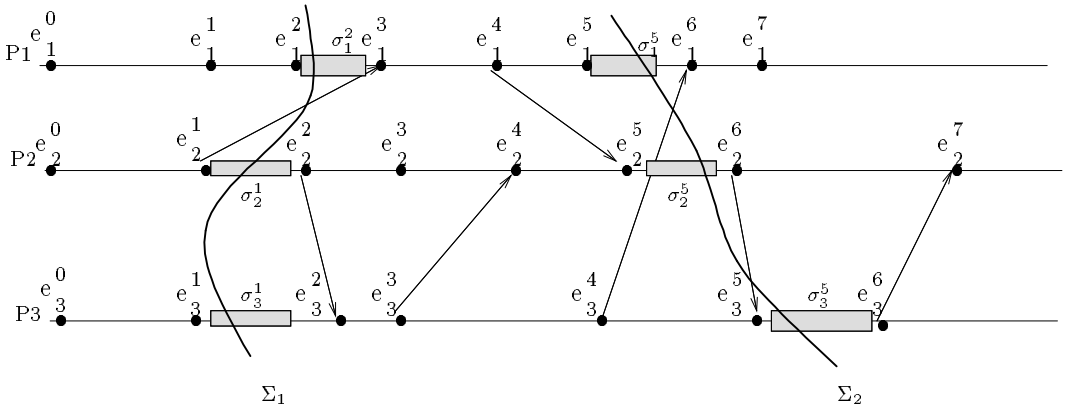


Figure 2: Two global states.

2.3.5 Equivalent executions and non-determinism

As stated in Section 2.2.3, distributed programs can be non-deterministic. As a consequence, different executions of the *same* distributed program can produce *different* process histories, even if the program is executed with the same input data ([2]). In other words, several distributed computations associated with a given distributed program and set of input data can produce different sets of events and different partial orders of the events, leading to different results. In particular, it is possible that some events occurring during a given computation *do not* occur in

another computation, or they occur in different orders; as a consequence, some messages exchanged during a computation might not exist during another one. This fact will be of primary importance in the context of backward recovery, since this technique may imply a re-execution of a part of the program, and thus, a re-execution possibly different from the previous one. Two distributed computations of the same distributed program with the same input data are said *equivalent* if they produce the same set of partially ordered events ([24, 29]). As far as causal precedence is concerned, both computations are indistinguishable.

Note however that, in the context of backward recovery, we are not interested in producing equivalent executions. This contrasts with the case of distributed debugging ([24, 12]), for example, where *replaying* the same execution is necessary. The aim of backward recovery is rather to restore a computation in a global state from which it can proceed and then possibly produce another partially ordered set of events (due to possible non-deterministic statements).

3 The backward recovery problem

3.1 Backward recovery and recovery lines

3.1.1 Backward recovery

Consider distributed computations of a given distributed program, in which fail-stop events can occur. Such an event is a consequence of a system crash implied by an error in a fail-stop processor on which a process runs. Recall that, when an error occurs in a fail-stop processor, it is halted and the content of its volatile storage is lost. To tolerate such transient faults, *backward error recovery* can be applied. Generally, this technique refers to restoring processes of a distributed computation to previous local states from which this computation can restart leading to consistent global states of an error-free computation, not necessarily equivalent to the previous one. Thus, recovery requires the ability to preserve information (e.g., some previous states) accross a crash, and the ability to construct appropriate states, from which computation can be resumed. To preserve required information, some states of processes and some messages are saved in stable storage at certain times. This information may then be used to reconstruct some previous states which have not been saved explicitely.

3.1.2 The concept of recovery line

A local state σ_i which can be reconstructed after a fail event occurrence is called a *recovery point* and is denoted by r_i. In this context, a *recovery line* R is defined as a set of recovery points, one for each process: $R = \{r_1, r_2, \ldots, r_n\}$, where r_i is a recovery point of P_i. Let us note that a recovery line constitutes a global state of the computation. In general, different recovery lines are available. In the context of recovery, we are only interested in *consistent recovery lines*, whose a precise definition will be given in the following Sections.

3.1.3 Faulty, resumed and new computations

In order to address more precisely the issue of recovery line consistency, which is central to the backward recovery technique, let us introduce some terms. The computation in which a fail-stop event occurs will be called the *faulty computation*; this computation stops in a global state which will be denoted by $S = \{s_1, s_2, \ldots, s_n\}$. The backward recovery technique implies that processes are rolled back to recovery points forming a recovery line $R = \{r_1, r_2, \ldots, r_n\}$ such that, for each $i, 1 \leq i \leq n$ we have $r_i \stackrel{\sigma}{\rightarrow} s_i$; the computation resumed from this recovery line will be called the *resumed computation* with respect to R. Finally, the *new computation* with respect to R is the concatenation of the faulty computation up to the recovery line R and of the resumed computation (see Figure 3).

Let $\mathcal{L}(\mathcal{P})$ be the set of all the failure-free distributed computations of a distributed program $\mathcal{P}$ (we suppose that $\mathcal{P}$ includes its input data). A recovery line R is *consistent* if resuming the execution from R, and if no failure occurs, the new computation belongs to $\mathcal{L}(\mathcal{P})$. Note, that usually, in a failure-free computation a couple of *send* and *receive* events can be associated with each message. In the following sub-sections (3.2 and 3.3) we show that the notion of failure-free computation can be extended in such a manner that there can be messages for which either *send* or *receive* events are missing. Such a possibility, based upon a notion of *recovery semantics for the messages*, enlarges the set of failure-free computations. Consequently, a more precise definition of consistency for recovery lines, taking into account this recovery semantics for messages, will be given in Section 3.3.

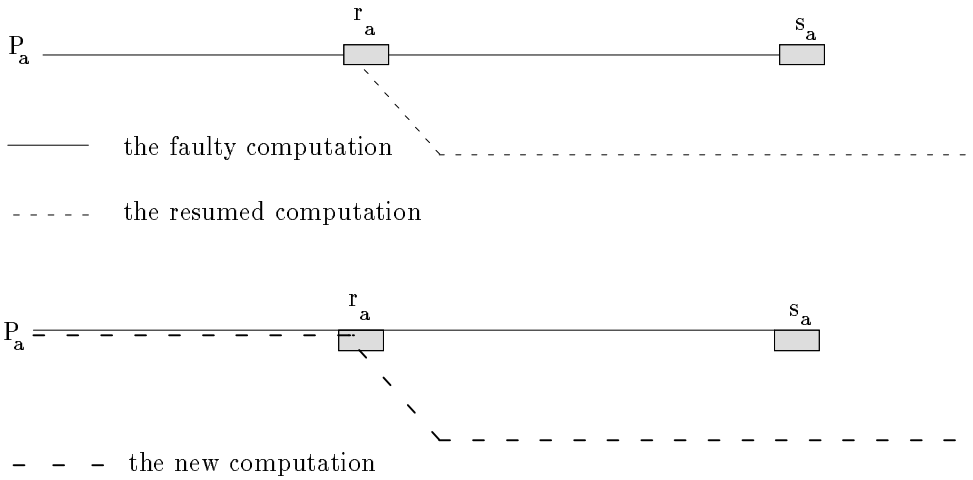


Figure 3: Faulty, resumed and new computations for a process.

3.2 Orphan and missing messages

3.2.1 Definition

In the resumed computation, the sequence of events that occur on each process P_a may be different from the sequence in the faulty computation, as a consequence of the non-determinism of the distributed program executed by these computations. Of particular importance, as far as consistency is considered, is the situation of messages involved in the resumed computation. According to the occurrence and position of messages in the faulty computation, some *send* or *receive* events that occurred in this computation may be missing in the resumed one, and conversely. In this context, one can distinguish two important situations, concerning *orphan* and *missing* messages:

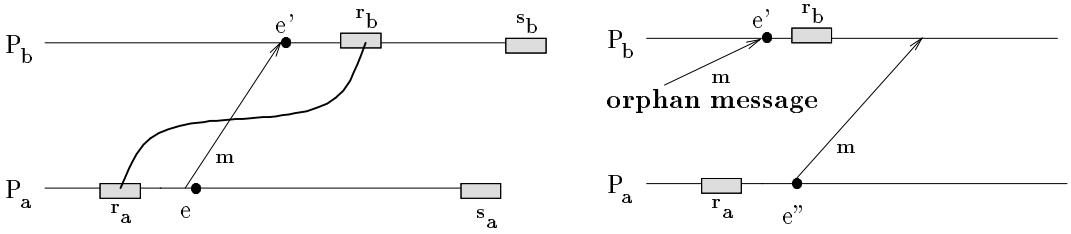
- A message is *orphan* with respect to a given computation if the *receive* event of this message occurs during the computation, but not the corresponding *send* event.
- A message is *missing* with respect to a given computation if the *send* event of this message occurs during the computation, but not the corresponding *receive* event.

At the underlying system level, orphan (resp. missing) messages can exist, for example as a result of message duplication (resp. loss) due to non-reliable links. But usually, the underlying communication system makes these failures transparent to the computation. However, backward recovery can create orphan or missing messages in the new computation. Even if links supporting channels are made reliable, messages sent in the faulty computation, whose *send* or *receive* event or both do not belong to the corresponding recovery points, can be either orphan or missing in the new computation. These facts explain why, although a recovery line is a global state, the two concepts of consistency for a recovery line and for a global state are essentially distinct, and are thus defined differently one from the other. The creation of orphan or missing messages in such circumstances is described hereafter.

3.2.2 Problems due to orphan and missing messages

Let m denote a message sent from P_a to P_b in the faulty computation, and r_a, r_b, s_a, s_b denote respectively recovery and stop points (local states) of P_a and P_b.

Case 1 : Orphan message in the recovery pair $\{r_a, r_b\}$.

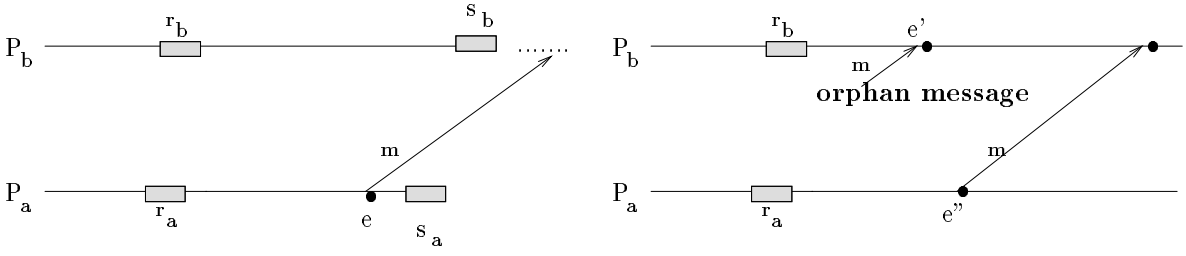


a) Faulty computation

b) new computation, with occurrence of $e'' = send(P_a, P_b, m)$

Figure 4: Orphan messages in a recovery pair.

Both events $e = send(P_a, P_b, m)$ and $e' = rec(P_b, P_a, m)$ occur in the faulty computation, but, only e' belongs to the new one. So, the message m can be orphan or duplicated (or both) in the new computation (Figure 4). Let us consider the two following situations, which differ in the behaviors of P_a and P_b in the resumed computation, according to the non-determinism of the program:



a) Faulty computation

b) New computation with occurrence of $e'' = send(P_a, P_b, m)$

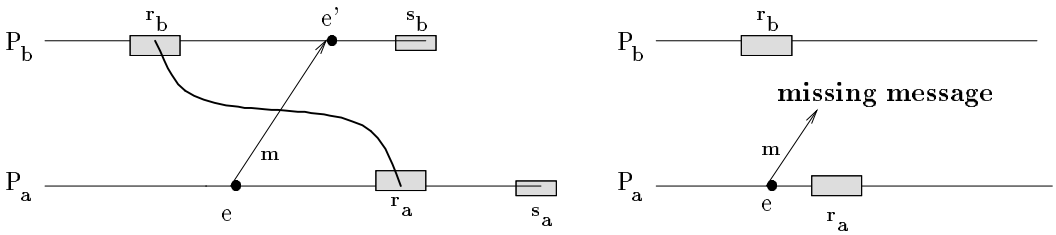
Figure 5: Orphan message due to non-empty channels.

- If P_a replays the $send(m, c_{ab})$ statement, another event $e'' = send(P_a, P_b, m)$ occurs in the resumed computation; in that case, the duplication of m takes place. The copy of m sent in the faulty computation is orphan if and only if P_b replays the corresponding *receive* statement in the new computation; in the other case, e'' matches with e'.
- If P_a does not replay the $send(m, c_{ab})$ statement, no $send(P_a, P_b, m)$ event exist in the new computation, which has only one copy of m, and this copy of m remains orphan in the new computation.

However, if, at the computation level, the receipt of m does not affect P_b[1] or if, at the system level, an additional underlying mechanism designed to tolerate message duplication is available[2], the new computation is failure-free and thus the recovery pair $\{r_a, r_b\}$ is consistent.

Case 2 : Orphan messages due to non-empty channels.
The event $e = send(P_a, P_b, m)$ occurs in the faulty computation, but not the event $e' = rec(P_b, P_a, m)$ since the process P_b has been stopped before the receipt of m (Figure 5). However, e' may belong to the resumed computation: that will be the case if the message m has been kept in the channel c_{ab}, and if the process P_b executes the $receive(vm, \{c_{ab}\})$ statement. Additionally, P_a may replay the $send(m, c_{ab})$ statement (event e''). Depending on these behaviors, the message m can be orphan or

[1] e.g., m is insignificant due to its semantics and by definition can be ignored ([27]).
[2] e.g., by not transmitting duplicated messages to the application ([11, 15, 37]).



a) Faulty computation

b) New computation

Figure 6: Missing message in a recovery pair.

duplicated or both, exactly like in the previous case. But it is worth noting a feature belonging specifically to the present case: if the duplication of this message is neither acceptable at the computation level nor avoidable at the system level, the only way to consistently restore the computation involves a mechanism to "flush" the channel c_{ab} in order that the event e' does not occur in any resumed computation unless as a consequence of a corresponding *send* event[3].

Case 3 : Missing message in the recovery pair $\{r_a, r_b\}$.
Both events $e = send(P_a, P_b, m)$ and $e' = rec(P_b, P_a, m)$ occur in the faulty computation; but e belongs to r_a while e' does not belong to r_b (Figure 6). Thus, e belongs to the new computation, but, as far as e' is considered, two cases are to be considered:

- P_b does not replay the *receive* statement whose execution in the faulty computation had produced e'. In that case, the message m is missing in the new computation.
- P_b replays the *receive* statement whose execution in the faulty computation had produced e'. If the message m is recorded by the underlying system, it can be delivered again: in that case, the message m is no longer missing. Otherwise, the message m is lost and the new computation is not failure-free.

The previous discussion shows that the set of failure-free computations can be enlarged by considering a *recovery semantics* for the messages exchanged. If a message

[3] Indeed, if the duplication of m takes place and cannot be tolerated, no pair of local states $\{\sigma_a, \sigma_b\}$ can be consistent, even the initial one.

m is orphan (resp. missing) in a new computation resulting from resumption from a recovery line, then this new computation can be considered as failure-free (and thus the recovery line is consistent) provided that:

- either m can be accepted as orphan (resp. missing) in this new computation, if the semantics of the application allows it,
- or m can be recovered into a "normal" message, by appropriate underlying mechanisms.

The following Section gives a more precise definition of recovery line consistency, by taking into account the semantics given by the application to messages, as far as recovery is considered.

3.3 Analyzing consistency of recovery lines

3.3.1 Consistency of a recovery pair

First, let us analyze the consistency of a *recovery pair*, i.e. the recovery line composed of two recovery points r_a and r_b belonging respectiveley to processes P_a and P_b. As mentioned previously, some orphan or missing messages may be compatible with the consistency of a recovery pair, depending on many elements, including: program characteristics (deterministic or nondeterministic), message recovery semantics as well as availability of additional mechanisms (to ignore duplicate and in-transit messages, or to reproduce lost messages). Hence, recovery pair consistency is affected by both system and program features, and therefore, in general, it must be established specifically for each pair of recovery points.

3.3.2 Tagging of messages

These features are abstracted by tags associated with messages of a distributed program. Tags associate a recovery semantics with each message, indicating whether it can be orphan or missing without preventing the consistency of a recovery line. Each message can be tagged *by the application* with a pair of tags (*cb_o, cb_m*), leading to four different status : *orphan or missing*, *orphan but not missing*, *missing but not orphan*, *neither orphan nor missing*.

When considering that all messages are systematically tagged by the application in the same way, four basic types of message deliveries are possible:

- *Exactly once* message delivery (all messages are tagged (_, _), i.e., failure-free computations do not allow orphan nor missing messages),

- *At least once* message delivery (all messages are tagged (*cb_o*, _), i.e., failure-free computations allow orphan messages but not missing ones),
- *At most once* message delivery (all messages are tagged (_, *cb_m*), i.e., failure-free computations allow missing messages but not orphan ones),
- *No constraint* on message delivery (all messages are tagged (*cb_o*,*cb_m*), i.e., failure-free computations allow orphan as well as missing messages).

Thus, tags can be attributed by the application either at the level of each message or at the global level of the application.

3.3.3 Consistency of a recovery line

As indicated previously, the definition of consistency of a recovery line can be precised when considering messages with recovery semantics. Let $R = \{r_a, r_b\}$ denote a recovery pair and let m be an application message sent from P_a to P_b. We consider a resumed computation from a recovery line containing the recovery pair R. Then, R is *consistent with respect to* m if and only if one of the following conditions holds:

- m is neither orphan nor missing in the resumed computation,
- m is orphan in the resumed computation, and it is tagged *cb_o* (for example, in Figure 4.a, if m is tagged $(cb\_o, *)$ then $\{r_a, r_b\}$ is consistent).
- m is missing in the resumed computation and it is tagged *cb_m* (for example, in Figure 6.a, if m is tagged $(*, cb\_m)$ then $\{r_a, r_b\}$ is consistent).

More generally, a recovery pair $R = \{r_a, r_b\}$ is consistent if it is consistent with respect to all the messages exchanged between P_a and P_b.

From this definition, a binary, symmetric relation over a set $\mathcal{R}$ of recovery points can be considered. This relation, denoted by D, is the subset of $\mathcal{R} \times \mathcal{R}$ comprising all the consistent recovery pairs in $\mathcal{R}$ (according to the consistency definition of each pair).

Finally, a recovery line $\{r_1, r_2, \ldots, r_n\}$ is said to be *consistent* with respect to D if and only if, for every (i, j) such that $1 \leq i \leq n, 1 \leq j \leq n$ and $i \neq j$, recovery pair $(r_i, r_j) \in D$.

As we can see, this definition of recovery line consistency takes into account the recovery semantics of messages enlarging the set of failure-free computations.

4 Basic tools for implementing backward recovery

In order to implement backward recovery, it is necessary to construct recovery points from which consistent recovery lines can be drawn. To this end, some basic tools can be used, some of them allowing storage of local states and/or messages on volatile or stable memories, others allowing to control the replay of application processes starting from an available state until an appropriate state is reached. All these tools can be used by a protocol implementing the recovery.

4.1 Recovery protocol

Consider a distributed program called henceforth, *application program*. A computation of this program will be called *application computation* and its messages *application messages*. Eventual backward recovery of an application computation is a task which may involve some messages and events not belonging to this computation. This task is performed by a distributed computation different from the application one, which is called *recovery computation*. It is governed by a distributed program called *recovery protocol*. A recovery computation is composed of *processes* $C_1, C_2, \ldots, C_n$, called *controllers*. A controller C_i is associated with each application process P_i. Its role is to observe the behavior of P_i (events occurring at P_i) and to cooperate with other controllers in the execution of the recovery protocol[4]. This involves local and cooperative actions. Examples of local actions are: save, at certain time, the local state of P_i in stable storage, construct recovery points, resume local process P_i and control it if necessary for some recovery period. Cooperative actions try to find distributively optimal ([41]) recovery line when fail event occurs. Since the recovery protocol is executed on the same distributed system than the application computation, cooperation between controllers is realized only by message passing, and these messages are called *recovery* messages. Recovery protocols are usually evaluated with respect to the optimality of the recovery line they find, and with respect to messages and time complexities.

4.2 Checkpointing

Checkpointing is the execution of an atomic operation realized by a controller C_i, to save the correct current local state (history) of the application process P_i on stable storage, such that the application process will be able to resume its normal

[4] A formal framework well suited to this situation is often referred to as a *superimposition* model ([8]).

computation from the checkpointed state instead of restarting computation from the initial state. Checkpointing should be transparent to application processes. The local states stored as a result of checkpointing are called *checkpoints*.

4.3 Logging

Logging is an activity of a controller C_i to record an ordered sequence of events produced by non-deterministic statements of the local program $\mathcal{P}_i$. Note the behavior of P_i up to a given point is completely determined by its initial state and the sequence of events which have been produced in the process up to that point by the execution of non-deterministic statements. The logged information (*log*) can be stored in volatile storage (*volatile log*) or in stable storage (*stable log*). Stable log of $\mathcal{P}_i$ is available after the process failure, but volatile log is then lost. When asynchronous message exchange is the only cause of nondeterminism, logging can be restricted to message logging. It this case the log (*message log*) is composed of a sequence of features of received (or sent) application messages (e.g., identities, tags, and so on). Message logs can be saved either in the sender storage (*sender-based logging*) or in the receiver storage (*receiver-based logging*).

4.4 Replaying

Replaying is an activity initialized and controlled by C_i, in which an application process P_i is allowed to run forward from a checkpoint, replaying in sequence events from its log, until it is eventually suspended by C_i when the appropriate state is reached. Usually, replaying concerns only application messages. This is completely sufficient when all the internal events are produced by the execution of deterministic statements.

4.5 Recovery points

To construct recovery points, checkpoints, logs and replaying can be applied. In the simplest case, a checkpoint directly constitutes the corresponding recovery point. However, other recovery points can be obtained by restoring a process to the state saved as checkpoint preceding the required recovery point, and then allowing the process to run forward, replaying the appropriate sequence of events (mostly those produced by the execution of non-deterministic statements). When the appropriate state is reached it constitutes the *recovery point*.

5 Basic strategies for implementing backward recovery

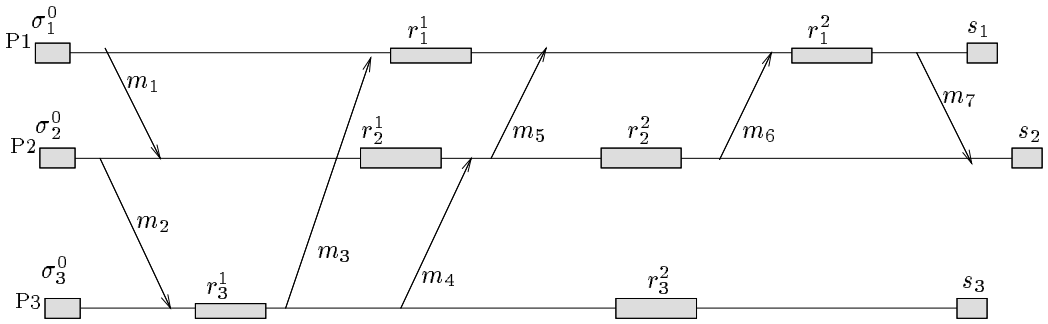
For the sake of simplicity, we assume henceforth, that asynchronous message exchange is the only cause of nondeterminism, and thus only receive statements can be nondeterministic in the distributed application. There are three basic strategies of backward recovery: coordinated, independent, and adaptive checkpointing. The underlying principles are presented in the next three sections.

5.1 Coordinated checkpointing

In *coordinated checkpointing* (also called *synchronous checkpointing*), all controllers coordinate their checkpointing activity considering checkpoints as recovery points ([7, 18, 20, 21, 42]). This checkpointing requires to take into account the relation D (consistency of recovery pairs), to obtain eventually a consistent recovery line. This recovery line (set of checkpoints) is maintained in the system until a next checkpointing action is successfully finished. Hence, only one consistent recovery line is available at a time. The storage requirement in this approach is limited (only one checkpoint per process is sufficient) and recovery becomes straightforward: when a fail event occurs in the application computation, this computation is rolled back to the available recovery line (i.e. each application process is rolled back to its last checkpoint) and then the application computation is restarted. However, this approach may incur relatively high overhead during failure-free computations. Indeed, if there are no process failure, recovery messages and checkpointing activity may delay the application computation.

5.2 Independent checkpointing

In *independent checkpointing*, controllers save checkpoints asynchronously regardless of consistency ([5, 19, 31, 34, 38, 40, 46]). The aim is to reduce interference of the checkpointing with respect to application computation. However, as a consequence of this independence the last checkpoints might not constitute a consistent recovery line (with respect to the relation D). Hence, after a failure controllers must cooperate by exchanging recovery consistency information to find consistent (optimal) recovery line. Usually, this searching is realized rolling back one application process to an earlier recovery point in order to eliminate inconsistency of the current recovery line. Thus, the so-called *domino effect* can occur ([32, 33]), in which the application processes are rolled back recursively, while determining a consistent recovery line.



tags of the messages : $m_1(\_, cb\_m)$: cannot be orphan, can be missing
$m_2(\_, cb\_m)$: cannot be orphan, can be missing
$m_3(\_, cb\_m)$: cannot be orphan, can be missing
$m_4(cb\_o, \_)$: can be orphan, cannot be missing
$m_5(cb\_o, \_)$: can be orphan, cannot be missing
$m_6(\_, cb\_m)$: cannot be orphan, can be missing
$m_7(cb\_o, \_)$: can be orphan, cannot be missing

Figure 7: Domino effect.

Figure 7 depicts an example of domino effect. The example shows that such an effect can happen even if some messages can be orphan or missing (i.e. if some tags of messages are different from $(\_, \_)$). Suppose P_2 stops in local state s_2, because of a fail event. To recover, P_2 could resume from recovery point r_2^2. But, since message m_7 cannot be missing, P_1 has to roll back to recovery point r_1^2 and since m_6 cannot be orphan, P_1 must further roll back to recovery point r_1^1. But then m_5 is missing in the recovery pair $\{r_2^2, r_1^1\}$ and thus P_2 must roll back to recovery point r_2^1. As m_4 cannot be missing, P_3 has also to roll back to recovery point r_3^1. Now, m_3 cannot be orphan, and thus P_1 has to roll-back to its initial state. Message m_1 is orphan in the pair $\{\sigma_1^0, r_2^1\}$ and thus P_2 has to roll-back to its initial state, and finally, since m_2 cannot be orphan, P_3 has also to roll-back to its initial state. This is an example of the worst case, where all the application processes must roll back to their initial states, i.e., a replay of the entire application computation is required due to inconsistency of later recovery lines.

The domino effect decreases performance of backward recovery techniques. To cope with this effect, pessimistic or optimistic message logging and replaying may be applied, in addition to independent checkpointing ([3]).

5.2.1 Pessimistic message logging

Pessimistic (also called *synchronous*) *message logging* refers to the stable logging of all the application messages before they are processed ([6, 25, 30, 45]). Thus, assuming a piece-wise deterministic program, recovery points can be constructed by rolling back application processes to their last checkpoints, and replaying sequentially all appropriate application messages. Of course, it is only possible if logs have been put in stable storage. It means that all messages have to be saved in stable but slow storage before they are available to application computation for processing. This synchronization between logging and processing of messages slows down application computations. The synchronization delay is introduced even if there are no failures. This is the main disadvantage of pessimistic message logging.

5.2.2 Optimistic message logging

In the *optimistic* (also called *asynchronous*) *message logging* approach, messages are processed by application computation independently of and concurrently to their stable logging ([4, 10, 13, 15, 16, 17, 27, 37, 39, 43, 44]). Sent or received messages are, first, logged in volatile storage with negligeable interference with respect to application computations. Then periodically, or when an application process is idle, its controller independently saves volatile logs in stable storage to make them stable, and clears the volatile logs. During the normal error-free computation, controllers do not communicate with each other. Thus, in the absence of failures, the only overhead is volatile logging and overhead due to stable logging realized in background by controllers. However, when a fail event occurs, all volatile logs of failed processes are lost, and therefore recovery point can be only constructed using stable logs. But as stable logs have been saved independently, available recovery points are, in general, not consistent. Hence, after a failure, controllers must cooperate to find optimal recovery line and domino effect is possible, unless careful additional control is exercised.

5.3 Adaptive checkpointing

In the absence of failures, the coordinated checkpointing introduces some overhead due to exchange of recovery information and possible application delay, but when a fail event occurs, recovery is simple and quite efficient in terms of rollback (no risk of domino effect). Independent checkpointing with pessimistic message logging avoids recovery message overhead and cascading rollback, however it is achieved at the expense of blocking each application message processing till the message is saved

in stable storage. On the other hand, independent checkpointing with optimistic message logging actually eliminates the overhead when there are no failures, but it incurs the danger of an available rollback of process, when searching for a consistent recovery line.

Adaptive checkpointing refers to the approach which joins some elements of the previous approaches to achieve both the low failure-free overhead and limited extend of rollback in case of failure ([49]). The general idea of this approach is to supply independent checkpointing and optimistic message logging with additional mechanism that determines when each controller should occasionally save extra checkpoints or logs in stable storage to prevent cascading rollback.

6 Conclusion

Reliability issues are increasingly important for a large number of distributed systems, and, as avoidance of all the faults is practically impossible, systems must be equipped with additional components and programs ensuring that occurrences of erroneous states do not result in later system failures. Amongst the many issues regarding to realization of fault-tolerant systems, this paper has been devoted to error recovery techniques, and more precisely to backward recovery. Indeed, this technique makes possible to recover from an arbitrary fault simply by restoring an appropriate previous state to the system, and thus can be considered as a general recovery mechanism to any type of fault as far as the system is not concerned by real-time constraints.

A detailed description of the distributed system, program and computation models has been carried out, in order to address as precisely and rigorously as possible fondamental concepts related to backward revovery. Although widely used throughout the variety of papers that exist in the literature, these concepts are often difficult to apprehend. This paper has been intended to provide a better understanding, clearer definition and synthetic presentation of these concepts, as well as basic tools and strategies used for their implementation. In that context, original contributions are related to recovery lines and re-execution with the notions of *faulty, resumed* and *new* computations, recovery semantics of messages with a clear definition of *orphan* and *missing* messages (abstracted through the notion of *tags*), consistency of recovery line, defined differently from the consistency of global states, and based upon *relationship* between recovery line and tagged messages.

Basic strategies for backward recovery in message passing distributed systems have been explained with respect to basic tools. An important and up-to-date set

of references has been matched with the classification into these various strategies. Additionaly, the reader interested in backward recovery in distributed memeory systems can consult [14, 28, 47, 48]).

Acknowledgements The authors would like to thank R. Baldoni for interesting discussions about recovery problems and a careful reading of a previous draft of this paper.

References

[1] M. Ahuja, S. Mishra, A basic unit of computation in fault-tolerant distributed systems, *Proc. 14th Int Conf Distributed Computing Systems*, Poznan, 1994, pp.626-633.

[2] S. Alalgar, S. Venkatesan, Hierarchy in testing distributed programs, *Proc. Int. Workshop AADEBUG'93*, Springer Verlag LNCS, 1993, pp. 101-116

[3] L. Alvisi, B. Hoppe, K. Marzullo, Nonblocking and orphan-free message logging protocols, *Proc. Fault Tolerant Computing Systems*, Toulouse, 1993, pp 145-154.

[4] L. Alvisi, K. Marzullo, Optimistic message logging protocols, *Proc. Workshop on Unifying Theory and Practice in Distributed Systems*, Dagstuhl, Germany, 4-9 Sept. 1994

[5] B. Bhargava, S-R. Lian, Independent checkpointing and concurrent rollback for recovery in distributed system – an optimistic approach, *Symp. Reliable Distributed Systems* SRDS'88, 1988 pp.

[6] A. Borg, J. Baumback, S. Glazer, A message system supporting fault tolerance, *Proc. ACM Symp. on Operating Sstems Principles*, 1993, pp. 90-99.

[7] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, *ACM TOCS*, Vol. 3, No. 1, 1985, pp.63-75.

[8] K.M. Chandy, J. Misra, *Parallel program design: a foundation*, Addison Wesley, New York, 1988.

[9] F. Cristian, Understanding fault - tolerant distributed systems, *Commun. ACM*, Vol. 34, No. 2, 1991, pp. 56-78.

[10] E.N. Elnohazy, W. Zwaenepoel, Manetho– transparent rollback-recovery with low overhead, limited rollback and fast output commit, *IEEE Trans. Comp.* Vol. 41, no. 5, 1992, pp.526-531.

[11] A.P. Goldberg, A. Gopal, A. Lowry, R. Strom, Restoring consistent global states of distributed computation, *ACM Sigplan*, 26, No. 12, 1991, pp. 144-154.

[12] M. Hurfin, N. Plouzeau, M. Raynal, A debugging tool for distributed Estelle programs, *Journal of Computer Comminications*, vol. 16, 5, 1993, pp. 328-333.

[13] P. Jalote, Fault tolerant processes, *Distributed Computing*, No. 3, 1989, pp.187-195.

[14] B. Janssesns, W.K. Fuchs, Relaxing consistency in recoverable distributed shared memory, *Proc. Fault Tolerant Computing Systems*, Toulouse, 1993, pp.155-163.

[15] D.B. Johnson, W. Zwaenepoel, Recovery in distributed systems using optimistic message logging and checkpointing, *Journal of Algorithms*, Vol. 11, No. 3, 1990, pp. 462-491.

[16] D.B. Johnson, W. Zwaenepoel, Sender-based message logging, *Proc. Fault Tolerant Computing Systems*, 1987, pp.14-19.

[17] T.T-Y. Juang, S. Venkatesan, Crash recovery with little overhead, *Proc. 11th Int Conf Distributed Computing Systems*, 1991, pp. 454-461.

[18] J.L. Kim, T. Park, An efficient protocol for checkpointing recovery in distributed systems, *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 8, 1993, pp. 955-960.

[19] K.H. Kim, J.H. You, A. Abouelnaga, A scheme for coordinated execution of independently designed recoverable distributed processes, *Proc. 16th IEEE Symp. Fault-Tolerant Comput.*, 1986, pp. 130-135.

[20] K.H. Kim, Programmer-transparent coordination of recovering concurrent processes: philosophy and rules for efficient implementation, *IEEE Trans. on Software Eng.*, Vol. 14, No. 6, 1988, pp. 810-821.

[21] R. Koo, S. Toueg, Checkpointing and rollback- recovery for distributed systems, *IEEE Trans. Software Eng.*, Vol. 13, No. 1, 1987, pp. 23-31.

[22] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Commun. of the ACM*, Vol. 21, No. 7, 1978, pp. 558-565.

[23] J.C. Laprie (ed.), *Dependability : Basic Concepts and Terminology*, IFIP WG 10.4, Series in Dependable Computing and Fault-Tolerant Systems, vol.5, Springer-Verlag, 1992.

[24] T.J. Leblanc, J.M. Mellar-Crummey, Debugging parallel programs with instant replay, *IEEE Trans. on Comp.*, Vol. 36, No. 6, 1987, pp. 471-482.

[25] P. L'Ecuyer, J. Malenfant, Computing optimal checkpointing strategies for rollback and recovery system, *IEEE Trans. on Computers*, Vol. 37, No.4, 1988, pp. 491-496.

[26] P.A. Lee, T. Anderson, *Fault Tolerance Principles and Practice*, Springer Verlag, Wien, 1990.

[27] H.V. Leong, D. Agrawal, Using message semantics to reduce rollback in optimistic message logging recovery scheme, *Proc. 14th Int Conf Distributed Computing Systems*, Poznan, 1994, pp. 227-234.

[28] L. Lin, M. Ahamad, Checkpointing and rollback- recovery in distributed object based systems, *Proc. Fault Tolerant Computing Systems*, 1990, pp. 97-104.

[29] R.H.B. Netzer, B.P. Miller, Optimal tracing and replay for debugging message passing parallel programs, *Proc. Supercomputing*, Nov. 1992.

[30] M.L. Powell, D.L. Presotto, PUBLISHING: A reliable broadcast communication mechanism, *Proc. ACM Symp. on Operating Systems Principles*, 1983, pp. 100-109.

[31] P. Ramanathan, K.G. Shin, Checkpointing and rollback recovery in a distributed system using common time base, *Proc. Symp. Reliable Distributed Systems*, 1988, pp. 13-21.

[32] B. Randel, System structure for software fault tolerance, *IEEE Trans. Soft. Eng*, SE-1, No. 2, 1975, pp. 220-232.

[33] B. Randel, P.A. Lee, P.C., Treleaven, Reliability issues in computing system design, *Computing Surveys*, Vol. 10, No. 2, 1978, pp. 123-165.

[34] D.L. Russell, State restoration in systems of communicating processes, *IEEE Trans. on Software Eng.*, Vol. 6, No. 2, 1980, pp.183-194.

[35] R.D. Schlichting, F.B., Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, *ACM Trans. on Computing Systems*, Vol. 1, 1983, no. 3, pp. 222-238.

[36] R. Schwarz, F. Mattern, Detecting causal relationship in distributed computations: in search of the holy grail, *Distributed Computing*, Vol. 7, 1994, pp. 149-174.

[37] A.P. Sistla, J.L. Welch, Efficient distributed recovery using message logging, *Proc. ACM Symp. on Principles Of Distributed Ccomputing*, 1989, pp. 223-238.

[38] R.E. Strom, S. Yemini, Optimistic recovery in distributed systems, *ACM Trans. on Computer Systems* Vol. 3, No. 2, 1985, pp.204-226.

[39] R.E. Strom, D.F. Bacon, S.A. Yemini, Volatile logging in n-fault-tolerant distributed systems, *Proc. Fault Tolerant Computing Systems*, 1988, pp. 44-49.

[40] 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, 1992, pp. 246-251.

[41] S. Toueg, Ö. Babaoğlu, On the optimum checkpointing selection problem, *SIAM J. Comput.*, Vol. 13, No. 3, 1984, pp. 630-649.

[42] K. Venkateshan, T. Radhakrishnan, H.F. Li, Optimal checkpointing and local recording for domino-free rollback recovery, *Information Processing Letters*, Vol. 25, No. 5, 1987, pp. 295-303.

[43] S. Venkateshan, T.T-Y. Juang, Efficient optimistic crash recovery in distributed systems. *RR-Nov'93, University of Texas at Dallas*, 1993, pp. 1-28.

[44] Y-M. Wang, W.K. Fuchs, Optimistic message logging for independent checkpointing in message-passing systems, *Proc. Symp. on Reliable Distributed Systems*, 1992, pp. 147-154.

[45] Y-M. Wang, Y. Huang, W.K. Fuchs, Progressive retry for software error recovery in distributed systems, *Proc. Fault Tolerant Computing Systems*, 1993, pp. 138-144.

[46] Y-M. Wang, W.K. Fuchs, Scheduling message processing for reducing rollback propagation, *Proc. Fault Tolerant Computing Systems*, 1992, pp.204-211.

[47] K-L. Wu, W.K. Fuchs, J.H. Patel, Error recovery in shared memory multiprocessors using private caches, *IEEE Trans. on Parallel and Distributed Systems* Vol. 1, No. 2, 1990, pp. 231-240.

[48] K-L. Wu, W.K. Fuchs, Recoverable distributed shared virtual memory, *IEEE Trans. on Computers* Vol. 39, No. 4, 1990, pp. 460-469.

[49] J. Xu, R.H.B. Netzer, Adaptive independent checkpointing for reducing rollback propagation, *IEEE Symp. on Parallel and Distributed Processing*, 1993, pp.754-761.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité 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