Rollback Recovery in Distributed Systems

Tamás Bartha
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
bartha@iei.pi.cnr.it

June 9, 1998

Categories and Subject Descriptors: D.4.5 [**Operating Systems**] Reliability— *Fault-tolerance; Checkpoint/restart.* D.4.7 [**Operating Systems**] Organization and Design—*Distributed Systems.* C.1.2 [**Processor Architectures**] Multiple Data Stream Architectures (Multiprocessors).

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: backward error recovery, coordinated checkpointing, message logging, optimistic and pessimistic algorithms, causal logging, message semantics, software architecture.

Abstract

Large distributed systems typically contain thousands of processing elements, and hence they are more liable to suffer individual node failures. Without a fault-tolerant design any single-point failure could cause the whole system to break down. Reset, the usual approach to failures in stand-alone applications, is usually not admissible in distributed computers due to the cost of lost work or catastrophic consequences. Therefore, error masking or error recovery must be employed as means to improve system dependability.

Checkpointing and rollback recovery are general-purpose techniques to provide fault tolerance in such computing environments. Message logging is another approach that facilitates recovery from failures. It can be used either in itself or in combination with checkpointing to avoid the domino effect. This survey classifies and presents the existing methods of checkpointing and message logging based recovery methods. The purpose of the survey is to help the system designer to find a suitable approach for a given application, or to aid in the development of new methods using existing experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel computing systems are nowadays widely used due to the rapid advancement in the design and production of VLSI electronic circuits. This trend continues and provides the designers of computing systems with high performance off-the-shelf components like fast processors, large capacity memories and complex controller units at low cost. The sophisticated manufacturing process in use made another beneficial contribution: increased reliability at the circuit level. However, parallel computers typically contain a large number of components, thus the probability of a failure at the system level is considerable even if dependable circuitry is used. *Fault-tolerance* is a design paradigm that assures the system will operate according to its specification even if a certain set of its components have failed.

Fault-tolerance has a fundamental connection to *redundancy.* On the one hand, adding external redundancy to the system in the form of spare units allows to either *eliminate* the effects of errors by switching off the failed component and substitute it with a fault-free one, or *mask* the errors by a majority voting of multiple identical components executing the same task in lockstep synchrony. This approach results in a relatively simple design, but still care must be taken about *synchronizing* the units, *matching* of inputs, self- or direct *testing* of the processors, excluding *common-mode* failures, avoiding *deadlocks*, and *reaching agreement* in the presence of unsuspected faulty units. Examples of error masking and elimination are the duplex and the n-modular redundancy systems. The use if spare units is expensive, but its major advantage is the fast response time which is practically unaffected by the presence of faults. This property makes the approach the preferred choice for responsive and mission critical applications such as transaction processing, database maintenance, process control, safety systems, and avionics.

On the other hand, parallel computers are made of many functionally identical processing elements that constitutes an inherent redundancy. This feature can be utilized to bring the system again into a valid state and *recover* from errors. The recovery approach requires little to none extra hardware, and the resulting system is less vulnerable to common-mode failures. Transient faults are handled automatically as the inconsistent system state is overwritten by a valid state during the recovery process. There are five phases that together provide general means by which recovery can be implemented:

1. *Error detection*: is the detection of an erroneous state that could lead to a system failure. While a fault cannot be directly detected by testing, the manifestation of the fault will create an error somewhere in the system. This error remains *latent* until it is detected by a test. Testing of a processing element can be done either by dedicated hardware (self-checking, comparator testing, watchdog processors), but in multiprocessor systems more often it is done by the neighboring processors. The success of any fault tolerant system critically depends on the effectiveness of the techniques used for error detection.

2. *Error confinement*: latent errors may *propagate* among interacting processors. The role of error confinement is to stop error propagation and limit the influence of latent errors by suspending the computation.

3. *Damage assessment*: before error recovery could be attempted, it is necessary to establish more precisely the extent to which the system state has been damaged. This includes the determination of the set of available and failed resources. However, test results are just as unreliable as the investigated resources themselves, therefore they can be affected by the errors and become *invalid*. System-level diagnosis aims at classifying the fault state of the components by analyzing the possibly invalidated test results.

4. *Reconfiguration and recovery*: a part of the system became unavailable as they were affected by the occurred errors. During reconfiguration the tasks previously executed by the failed units are reallocated to fault-free units, preferably in a way that distributes the computation load equally among the available resources. Recovery transforms the current erroneous state into a well-defines and error-free state from which normal operation can continue. This is accomplished by identifying a valid global system state and communicating it among the components. Tasks are then reinitialized according to the new global state.

5. *Fault treatment and continued system service*: as a final step, the fault can be located more accurately. This will make possible to distinguish between transient and permanent faults, and consequently, to decide if no action is needed or the faulty unit must be taken out of service and repaired.

There are two main alternatives of recovery respective to the strategy by which a consistent global system state is chosen:

Forward error recovery. This method constructs a new valid global state starting from the failed process using fault diagnosis information. The continuation of the currently executed and failed process is attempted from some further point with compensation for the corrupted or missed parts. Hence, the construction of the new global state and the compensation actions are completely application dependent and the design efforts resulted in domain-specific *ad hoc* solutions.

Backward error recovery. Also called rollback recovery, this method is more general and is similar to duplexing with a warm backup. It works by periodically saving the complete or a partial system state to a reliable medium (such an abstraction is called the *stable storage*). After an error occurrence the system is reconfigured and the tasks are *rolled back* to a previous state respective to the information stored on the stable storage. The normal operation resumes from this old state, so a part of the completed work is lost and must be redone.

The disadvantages of the recovery approach are obvious from the above discussion: it comprises the coordination of intricate procedures. System design becomes difficult due to the elaborate functional structure. There is an additional load on the system during both the normal operation and after a failure occurrence. Furthermore, it is hard to guarantee a short response time because of the delay represented by recovery actions and making up for the lost work. Still, the approach is becoming dominant in general-purpose application areas that do not have tight time restrictions and are very computation intensive, like modeling, simulation, verification, off-line transaction processing.

There is much literature on fault-tolerance in parallel systems (including both *distributed systems* and *multiprocessors*). General introduction to different aspects of fault-tolerance can be found in [6, 34, 38, 58, 61]. This paper gives a survey and a classification of existing backward error recovery techniques. It tries to cover most major areas of the field, serving as a base for finding the applicable methods for a given design, learning about the state-of-the-art of current approaches, or obtaining directions for further investigation of the literature.

The rest of this document is organized as follows. A general system model and the most frequently used assumptions are presented in Section 1.1. Section 1.2 describes the basic terminology and introduces the associated problems of checkpointing and message logging techniques. Different approaches to rollback recovery are discussed next: Section 2 describes checkpointing and Section 3 describes message logging. A comparison highlighting the trade-offs among these approaches is given in Section 4. Requirements and techniques particular to massively parallel systems are contained in Section 5. A summary and some general conclusion complete the paper in Section 6.

1.1 System model and assumptions

A distributed system consists of a finite set of processes and a finite set of channels. The system structure can be represented by labeled, strongly-connected directed graph in which the vertices represent processes and the edges represent channels [15, 42, 43]. Figure 1.1 presents an example distributed system with three processes and channels between them. Processes communicate with one another exclusively by sending and receiving messages via the channels.

Channels are assumed to be error-free and to deliver the messages in the order sent. The delay imposed by the channel on message delivery is arbitrary but finite. Consequently, the sequence of the messages received at the output of a channel during a certain time interval is the initial subsequence of the messages submitted to the input of the channel in the given time interval. The *state* of the channel is the sequence of messages sent along the channel, excluding the messages received along the channel.

A process is defined by a set of state intervals, an initial state (from this set), and a set of events. Process execution within the same state interval is completely

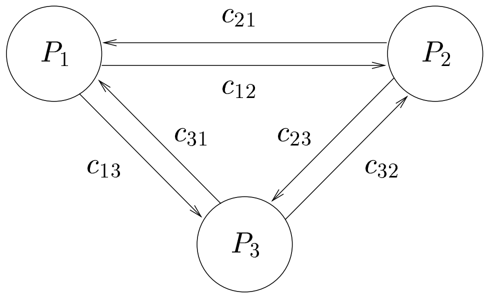

Figure 1.1: Structure of a simple distributed system

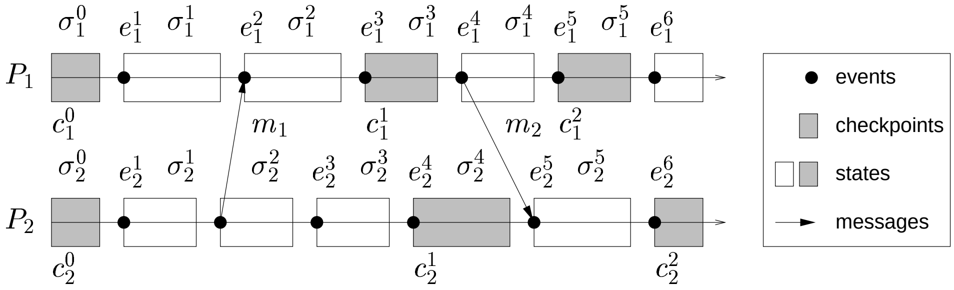

Figure 1.2: Model of a distributed computation

deterministic. An event e_x^i in a process P_x is an atomic action that starts a new state interval σ_x^i and may change the state of *at most one* channel incident on P_x. The state of a channel c_{xy} or c_{yx} may be altered by the sending of a message along channel c_{xy} (directed away from P_x), or the receipt of a message along channel c_{yx} (directed towards P_x). Therefore, any event e_x^i is defined by the 5-tuple $e_x^i = \langle P_x, \sigma_x^{i-1}, \sigma_x^i, m_i, c\rangle$, where P is the process in which it occurs, σ_x^{i-1} is the state interval immidiately before the event, σ_x^i is the state interval immidiately after the event, and m_i is the message (if any, m_i and c are `null` symbols if no message is involved in the event) sent or received along the channel c incident on P_x.

A global state S of a distributed system is a set of process state intervals and channel states (on for each process and channel). The *initial* global state is one in which each process is in its initial state interval and the state of each channel is the empty sequence. A function $\text{next}(S_i, e_x^{i+1})$ can be defined as the global state immediately after the occurrence of event e_x^{i+1} in global state S_i. The value of $\text{next}(S_i, e_x^{i+1})$ in this case is identical to the global state S_i except that the state interval of process P_x became σ_x^{i+1}, and if c is a channel directed towards P_x, then the state of c in $\text{next}(S_i, e_x^{i+1})$ is the same as in S_i with message m_i deleted from its head; or if c is a channel directed away from P_x, then the state of c in $\text{next}(S_i, e_x^{i+1})$ is the same as in S_i with message m_i added to its tail. Let $E = (e_1, \ldots, e_m)$ be a sequence of events in component processes of a distributed system, then E is a *computation* of the system if and only if event e_i can occur in global state S_i, where S_0 is the initial global state and $S_{i+1} = \text{next}(S_i, e_i)$ for $0 < i \leq m$. Figure 1.2 is an example for this computation model.

Computations can also be viewed as partially ordered set of events and state intervals [44]. An event e_x^i of process P_x is happened before an event e_y^j of process P_y (denoted as $e_x^i \rightarrow e_y^j$) if

1. $x = y$ and e_x^i occurred before e_y^j, or
2. e_x^i is a send event of message m_i along channel c_{xy} and e_y^j is the receive event of message m_i, or
3. $\exists e_z^k : e_x^i \rightarrow e_z^k$ and $e_z^k \rightarrow e_y^j$ (transitive closure).

The *happened before* relation creates a partial temporal ordering above events. It can be extended over process state intervals by observing that

$$\sigma_x^i \rightarrow \sigma_y^j \text{ if and only if } e_x^i \rightarrow e_y^j$$

where event e_x^i changes the state interval of process P_x from σ_x^{i-1} to σ_x^i and event e_y^j changes the state interval of process P_y from σ_y^{j-1} to σ_y^j. Events and state intervals not related to each other by the 'happened before' relation are called *concurrent*.

To illustrate the definition of a distributed system, consider a simple system consisting of two processes P_1 and P_2 connected by two channels in both directions. Initially each process owns a token, process P_1 a white and process P_2 a black one, respectively. They have two events: e_x^0 and $e_x^1, x \in \{1, 2\}$, where e_x^0 denotes sending the own token of process P_x and e_x^1 denotes receiving the token of the other process. A system computation corresponds to a path in the global state-transition diagram (Figure 1.3) starting at the initial global state S_0. Examples of system computations are

- the empty sequence,
- E_1 = (P_1 sends white token, P_2 sends black token, P_1 receives black token, P_2 receives white token),
- E_2 = (P_1 sends white token, P_2 receives white token, P_2 sends black token, P_1 receives black token), etc.

The following sequence is not a computation in the system: (P_1 sends white token, P_1 receives black token), because the event 'P_1 receives black token' cannot occur in the global state S_1. The reader should observe that there may be more than one transition allowable from a global state. For instance, events 'P_1 sends white token' and 'P_2 sends black token' may occur in the initial global state, and the next states after these events are different. If the determinism of a computation depends only on the sned and receive event of processes, it is called *piecewise deterministic.*

The above general model has already demonstrated some of the assumptions of distributed systems. In the following we summarize and comment the assumptions that serve as a common basis of many rollback recovery methods:

- Processes communicate and synchronize by exchanging messages. Message exchange is the only way of error propagation between processes, yet it is the only mean of testing another process as well. Therefore, the reliability of a certain process becomes *dependent* on the fault state of the other processes it has communicated with. A process never sends a message to itself.

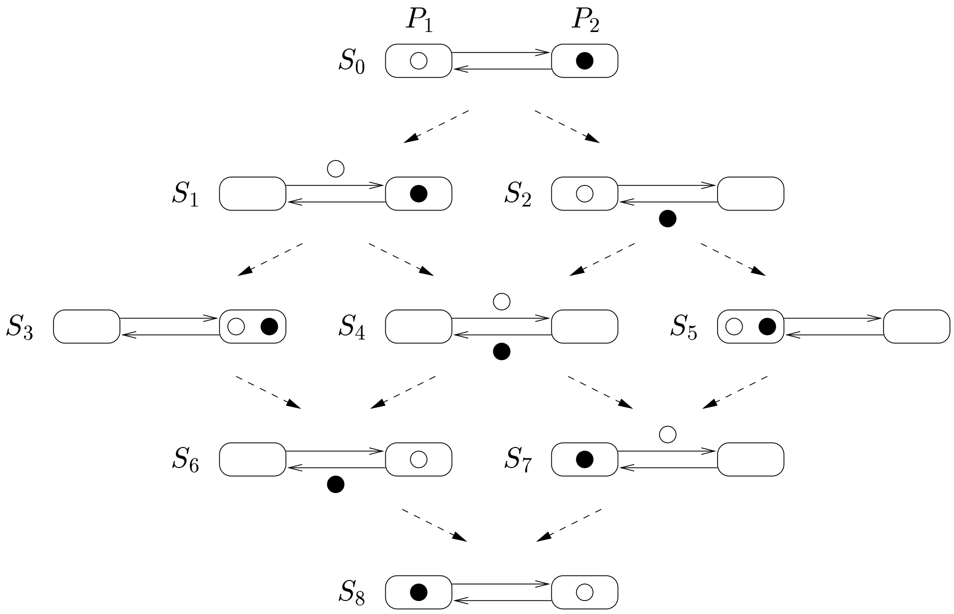

Figure 1.3: State-transition diagram of a nondeterministic computation

- Processes are sequential. They have internal, send, and receive statements, which are represented by events. Event e_x^i entails the change of local state interval σ_x^i to σ_y^{i+1} at process P_x. Only nondeterministic events are of interest from the recovery viewpoint. Usually process execution is assumed to be piecewise deterministic, that is there are no nondeterministic internal statesments.

- Channels deliver the messages in FIFO order. They do not lose or alter any messages. This requirement is inherent to every recovery procedure that obtains dependency information on the basis of the 'happened before' relation. The requirement can be enforced by using a fault tolerant communication protocol like [11, 12].

- Message transmission delays are finite but unpredictable. The message reception order over multiple channels is nondeterministic. Furthermore, some methods assume that identical cannels are assigned for the subsequent communication of the same processes. Routing techniques with such properties are described in [22, 20].

- Processes have fail-stop failure semantics.

1.2 Concepts and problems

Backward error recovery attempts to maintain the *consistency* of a computation by returning the system to a previous state upon an error occurrence. Two basic services are used in this process [15]: *state recording* which stores global system states to a reliable medium, and *state restoration* which assembles the stored information and recreates a consistent global restored state. These services can be either *semi-automatic* if they are executed upon the invocation of the application programmer, or *user-transparent* if the necessary actions are carried out automatically, without any user interaction. Recovery techniques can be classified in five major classes according to the characteristics of the above two services:

Checkpointing. A global state is stored as the collection of local process states. A stored local process state is called a *checkpoint*. *Independent* checkpointing methods take the checkpoints process-wise. In this case, there is no guarantee that a consistent global state can be restored from the stored set of process states. The state restoration service must be able to determine which the most recent recovery line (also called as *maximum recoverable global state*) included among the set of stored local checkpoints. This makes the recovery process more complicated, and there is the possibility of *domino effect* (described below). *Consistent* checkpointing methods avoid these disadvantages by coordinating the checkpointing actions among the processes to guarantee that the stored set of local checkpoints is always a recovery line.

Message logging. In piecewise deterministic systems, a global state of computation can alternatively be recomputed instead of restarted from a set of checkpoints. Since the only source of nondeterminism in this class of systems is communication, only message transmissions need to be stored for replay during recovery. Recording of the relevant information about messages is called *logging*. The recovery protocol must recreate the exact order and content of each message transmitted before the failure occurred. The data and causal order of messages determines the dependency relationship among the interacting processes, and so the stored dependency information can be used to compute the latest global state that can be recovered, or detect and eliminate inconsistencies in the restored system state. The maintenance and analysis of the dependency information is called *dependency tracking*.

Hybrid techniques. A composition of the above two techniques which combines their advantages. Although it is possible to implement backward error recovery solely based on message logging, the stored log data could grow unacceptably large over a long time span. Therefore it is customary to combine logging with checkpointing: when a global state becomes stable among the recorded checkpoints, log entries related to messages preceding the global state are no longer meaningful, thus can be discarded. Note, that message logging already guarantees the consistency of recovery, so even independent checkpointing can be used for the purpose without risking the domino-effect (described later in this section). Yet, it is argued that using coordinated checkpointing with message logging some optimizations become available and the resulting protocol outperforms traditional ones [31].

In the rest of this section we define the main terms of backward error recovery and identify the significant problems concerning them.

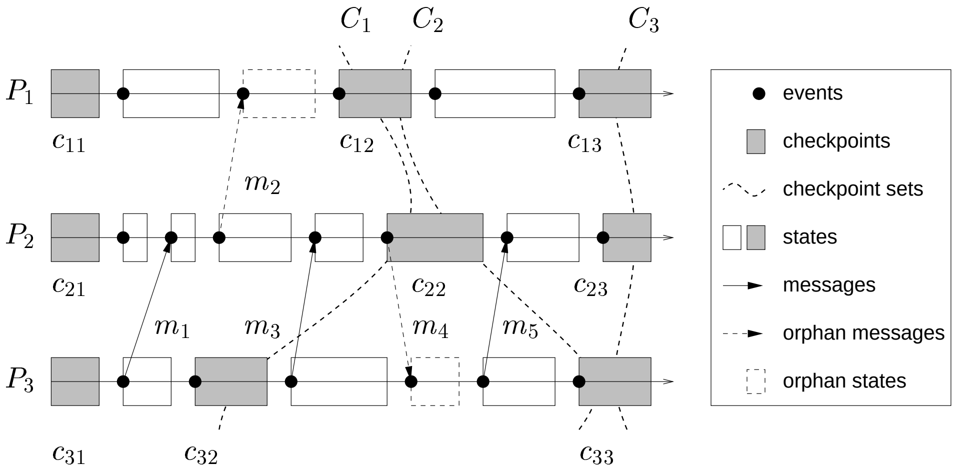

Figure 1.4: Distributed computation example for the main concepts

Checkpoint, recovery line

A *checkpoint* is a copy of the application state (and optionally of the state of its communication channels). It must contain the value of all variables and data structures that describe the actual state of the application process. The checkpoint can be used to restore a previous, failure-free state of a failed process. It must be continuously available despite of errors occurring in the system, therefore generally it is stored on stable storage. The *stable storage* is an abstraction of a reliable backup medium which is unaffected by errors occurring in the device, at the user interface, or coming from the environment. The retrieval of data recorded in stable storage is guaranteed. A checkpoint in itself is consistent if describes the application state at a certain point of time, as if the process would have been stopped and a *snapshot* of the relevant variables and data structures would have been made.

A computation in a system involves several independent but cooperating application processes. The global state of a system is a collection of local states, one per each application process. A *consistent* global state is set of application states that are concurrent (mutually independent). A *recovery line* is a globally consistent set of checkpoints. Two kind of inconsistencies may arise in a global system state: (1) one of its member states may depend on a future application state, or (2) a future application state may depend on one of its member states. In a piecewise deterministic system such inconsistencies can be expressed in terms of messages. The former kind of inconsistency is a message received but not yet sent by the time the global system state was recorded, also called an *inconsistent* message. The latter is a message sent but not yet received by the time the global system state was recorded, also called an *in-transit* message. Note, that in-transit messages generally are not considered to violate system consystency, because they are assumed to be handled by an underlying fault-tolerant communication protocol, such as [1, 2]. Refer to Figure 1.4 for examples: message m_3 is an inconsistent message with respect to checkpoint set C_1, while message m_5 is an in-transit message with respect to checkpoint set C_2.

Message log, orphan processes

The message log is a record of the identifiers (and optionally the contents) of process interactions within the system. The message log provides a partial causal ordering over the set of deterministically replayable communication events. Like checkpoints , the message log must be retrieved reliably, thus it is also stored on stable storage. When a volatile storage is used to record the message log, it might happen that some log entries are destroyed by the errors occurring in the system. As a consequence, the corresponding message will not be replayed during recovery, and became so called *lost* messages. States that are dependent on lost messages cannot be deterministically recreated, and hence might themselves be lost. In this case the course of distributed computation takes a new route.

However, some processes may have recorded messages that were sent before the failure from the now lost states. These messages, called *orphan messages*, are inconsistent with the new global system state. Conventionally, the processes and process states that depend on orphan messages are referred to as orphan processes and orphan states, respectively. Figure 1.4 displays a situation leading to orphan processes. Suppose the log entry of message m_1 is destroyed by the failure of process P_2 occurring before checkpoint c_{22}. Due to the failure process P_2 will be restarted from checkpoint c_{21}, but message m_1 is lost and will not be replayed. Consequently, process P_2 might not enter the states from which messages m_2 and m_4 are sent. These messages then are orphaned and process P_1 and P_3 will be orphans.

Domino effect, livelock problem

When the checkpoints are taken independently and in a user-transparent manner, it is the task of the recovery procedure to find the latest consistent recovery line among the set of stored checkpoints. However, there is no guarantee that there exists a recoded consistent global state at all without coordination. If a recovery line cannot be found on stable storage, each process must be rolled back to the initial state, causing the whole computation completed up to the time of failure to be wasted as the system must restart from scratch. The problem is illustrated by Figure 1.5(a). When a failure occurs in process P_1, then it must roll back to its latest checkpoint: c_{13}. When P_1 resumes from that checkpoint, it expects message m_4 from P_2 to arrive, therefore P_2 should also roll back to checkpoint: c_{22}. Similarly, in its currently restored state P_2 expects message m_3 from P_1, so P_1 should roll back, etc. Finally, both P_1 and P_2 will return to their initial state. This is called the *domino-effect* or *cascading rollback*.

When the recovery manager tries to compose a recovery line, it begins with the most recent checkpoint of the failed process. If the process interacted with other processes since the last checkpoint, and these activities were not recorded or are not available any more, the set of interacting processes must also be rolled back. There are three reasons for this: if only the failed process was rolled back, other processes could continue using erroneous information (the error has already propagated), information could get lost (the sender processes do not re-send those messages transmitted since the last checkpoint), or duplicate messages could be sent (by the recovering process). In such a situation liable leading to domino effect three classes of techniques avoids cascad-

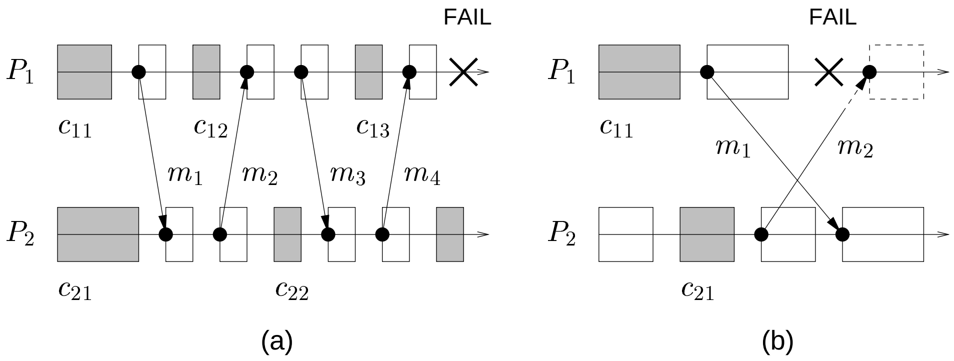

Figure 1.5: Checkpoint and rollback problems: (a) domino effect, (b) livelock

ing rollback: (1) the application programmer specifies a consistent recovery line with semi-automatic checkpointing, (2) checkpoints of the processes are coordinated, or (3) interprocess communication is logged (or a combination of the previous two techniques).

A *livelock* (also called *cyclic restoration*) occurs when a single process causes infinite number of roll-backs, preventing the system from making progress. Suppose a process rolls back to its last checkpoint and it notifies other processes about this action. The others might in turn decide to also roll back to their last checkpoint, and so on. However, this simple method is prone to livelock if the rollback of the processes is not synchronized. See the system of two processes shown in Figure 1.5(b). Checkpoints c_{11} and c_{21} constitute a recovery line, so it seems to be safe returning to it. Assume a failure occurs in process P_1 before message m_2 is received. Process P_1 then rolls back to checkpoint c_{11}, informs P_2 and resumes operation: it sends m_1' (a message identical to m_1, but from the recovered state) and receives m_2 that was under way. To undo the receiving of message m_1 (of which process P_1 does not have knowledge anymore) process P_2 must also roll back and then it resumes its operation: it sends m_2'. Again, the state is not consistent any more because process P_1 has received m_2 while P_2 does not "remember" sending it. Consequently, process P_1 should roll back a second time, forcing P_2 to roll back as well, and so forth. Due to a single failure the system will try to restore its consistency in an endless loop and become unavailable for the user. This livelock problem can be avoided when messages are signed by incarnation numbers (counting how many times the recovery procedure was invoked) or if a two-phase commit protocol is used to synchronize the recovery of processes.

Chapter 2

Checkpointing

Checkpointing is a straightforward approach for (semi- or completely) transparently adding fault tolerance to distributed applications without requiring (considerable) additional programmer effort. With checkpointing, the state of each process is saved separately on stable storage as a local checkpoint. The checkpointing of distinct processes can be independent, or can be synchronized such that the collection of checkpoints represents a consistent state of the whole system. After a failure, failed processes are restarted on any available machine and their state is restored from their most recent checkpoint on stable storage. Surviving processes may have to roll back to their latest checkpoint as well in order to remain consistent with the recovering process.

A checkpoint contains the complete state of an application process. It is up to the system or application programmer to decide what must be included in the set of checkpointed information. There are important system as well as application parameters. The system parameters comprise workspace pointers, program counters, registers, queues, other operating system information, etc. They are determined by the operating system and/or the system software. The application parameters cover the different ranges address space such as heap, stack, static variables, etc. Problems arise when memory is dynamically allocated at run-time: the newly allocated area should also be checkpointed.

Avoiding to store code within the checkpoint is the preferable practice, therefore it is usually forbidden to write code that alters itself during run-time. Relocatable code guarantees that it can be loaded to a different address space during the restart procedure (reconfiguration might necessitate such relocation is the application is to be run on different hardware than prior to the failure). Also the state-variables of the shared libraries should be handled with caution. In the ultimate case, the whole memory should be checkpointed. There are methods that reduce the extent of later checkpoints, provided the state of the process changes "moderately", such as the*dirty bit* (a flag is assigned to each unit of memory to indicate if that part has been changed since the previous checkpoint) or *recovery cache* (write operations between two checkpoints are recorded). Recording only the changed information in subsequent checkpoints is called *incremental checkpointing.*

The checkpointing frequency (i.e. the length of the checkpoint interval) affects both the overhead related to fault tolerance during failure-free operation and the amount of work lost upon a failure occurrence. The optimal checkpointing interval mainly depends on the failure-rate, the failure-free overhead, and some other characteristics of the system. Young gave a first order approximation to an optimal checkpoint-interval with respect to checkpointing duration and MTBF [73]. Later Vaidya generalized these results to the case of more effective checkpointing methods (see Section 5.1) where the checkpoint overhead (increase in the execution time of the application because of checkpointing) and the checkpoint latency (duration of time required to save the checkpoint) are not equal [69]. Several methods for adapting the length of the checkpoint interval (i.e. adjust the checkpointing frequency) exist [74], the most important ones include:

- invoking the checkpointing routines (in semi-automatic checkpointing) in a programmer defined way,
- taking checkpoints periodically after t clock ticks of a local hardware timer,
- taking checkpoints after k messages have been sent,
- activating the checkpointing routines just before the process terminates, so that the output can be committed.

2.1 Independent checkpointing

In *independent* checkpointing the state of processes is recorded process- or processor-wise according to a local schedule. The checkpointing activity is separate from the rest of the system, i.e. there is no form of coordination in taking a checkpoint (in contrast with *coordinated* checkpointing, to be introduced in the Section 2.2). There is no need for a particular *checkpoint* protocol, therefore no additional overhead is created during the failure-free operation (see also Section 5.1 on how to create checkpoints efficiently). However, a high price must be paid for the relative simplicity of this approach:

- The set of recorded local process states residing on the stable storage is not guaranteed to have any global properties. In particular, it might not contain a consistent global state, so any attempt of rolling back to a recovery line results in a complete restart: a major drawback.
- The recovery procedure is complicated: it must determine the maximum recoverable global state which is consistent and avoid livelocks at the same time (such a recovery protocol is presented below).
- Several checkpoints need to be maintained to facilitate possible large *rollback distances* (the number of stored local states rolled back). Since the stable storage is finite, eventually a *garbage collection* algorithm will be necessary to reclaim old and redundant checkpoints.

The paper [9] by Bhargava and Lian is an implementation of independent checkpointing. The technique used allows for multiple failures and concurrent recovery sessions. During failure-free operation, every process maintains an *input information table* (IIT) to record message exchanges. Each entry $IIT_x[i, y]$ at process P_x contains the set of checkpoint intervals of process P_y in which P_y sent some messages received by P_x in its checkpoint interval $c_{x,i}$. The input information table is stored on stable storage except the entries of the current checkpoint interval. The approach uses a two-phase recovery protocol to restore a consistent global state in the system:

1. In the first phase, the rollback initiator process P_c broadcasts a 'rollback request' message to all other processes. In response, each process P_x sends its IIT_x to the initiator, who uses the obtained IITs to build a local *system graph*. (Further communication of application messages by all rollback participant is suspended until the recovery is complete.) The system graph represents the interactions among the processes of the system in the different checkpoint intervals, as perceived by P_c. Then, the initiator process determines the restart point as the maximum recoverable global state. This is achieved by traversing the local system graph and finding each earliest checkpoint interval $c_{y,j}$ of the respective process P_y whose receipt of messages would have been undone by the recovery. The restart point for process P_y will be the checkpoint interval $c_{y,j-1}$ previous to $c_{y,j}$.

2. In the second phase, the initiator instructs the processes whose state is inconsistent with the found global state to roll back. Each rollback participant P_x restores the checkpoint interval $c_{x,i}$ accordingly to the restart point, and normal execution resumes.

Multiple failures and concurrent recovery sessions are handled *sequentially* using a priority assigned to every process to decide the processing order of sessions.

This technique is adequate to avoid the livelock problem and restore a consistent global state if it exists among the local states recorded on stable storage, yet it does not avoid the domino effect. However, the problem of cascading rollback is not intrinsic to independent checkpointing. We present two solutions, called *adaptive checkpointing* and *local recording* that avoid the domino effect without external coordination. The basic idea of both methods is to insert additional local checkpoints at certain points of the computation, so that any local checkpoint belongs to at least one set of n concurrent local checkpoints.

Adaptive checkpointing, introduced by Russell in [59], concentrates on the sequence of local events to determine when it is necessary to take a checkpoint. Let $ckpt_x$, $recv_x$, $send_x$ denote the respective events of taking a local checkpoint, receiving and sending a message in process P_x. If the sequence of local events for every process P_x in the system obey the following (regular expression) pattern:

$$(ckpt_x\ [recv_x]*\ \ [send_x]*)*$$

then it is guaranteed that each checkpoint will belong to a consistent global state and consequently the domino effect cannot occur [7]. It follows that forcing each process P_x to insert an additional local checkpoint between each sequence of $send_x$ events and each sequence of $recv_x$ is sufficient to prevent cascading rollback.

Local recording by Venkatesh et al. uses a more sophisticated approach in order to insert only a minimum number of additional local checkpoints while forming a recovery line [71, 8]. A *checkpoint vector* (CV) is associated with every process P_x that contains the sequence number of the last local checkpoint taken by process P_y (as perceived by P_x) in its $CV_x[y]$ entry. Every set of recorded global states is tagged with a tuple $\langle P_z, i\rangle$ where P_z is the process identifier, and i is the sequence number of a local checkpoint $c_{z,i}$. The sender processes piggyback their current CV to the messages they transmit. The recipient of the message, say process P_x, strips the CV from the message (called RCV) and compares it with its own copy (called CCV). There are three operations to be performed by the recipient based on the result of checkpoint vector comparison:

1. If $(RCV_y[z] = j) > (CCV_x[z] = i)$ then P_x finds that some process P_z has a checkpoint with a sequence number j greater than i what is currently known in the CV. Therefore,) P_x records its local state (before processing the corresponding message) and tags the new checkpoint with $\langle P_z, j\rangle$. This checkpoint is now a part of the recovery line $\langle P_z, j\rangle$, and regarded as "owned" by P_z.

2. If $(RCV_y[z] = j) < (CCV_x[z] = i$, then P_x "dates back" this message, that is the message is appended to all previous local checkpoints whose identifier $\langle P_z, k\rangle$ satisfies the condition $k > j$.

3. If there is no local checkpoint tagged as $\langle P_z, (RCV_y[z] = j)\rangle$, then such a checkpoint is created so the two recovery lines do not cross each other. This is achieved by copying the local state contained in checkpoint $\langle P_z, i\rangle$ (where i is the smallest sequence number greater than j for which a checkpoint of P_x owned by P_z exists) excluding the message itself.

Thus, a recovery line for each local checkpoint is dynamically established and the messages flowing into the recovery line are identified. Since the checkpointing actions are triggered by communication, a recovery line is confined to the set of interacting processes. The number of recovery lines depends on the frequency of taking individual local checkpoints and the subsequent mutual interactions.

2.2 Consistent checkpointing

The idea of coordinating the checkpointing activities comes from Chandy and Lamport. In [15] they present an algorithm by which a process can determine a global state of a distributed system. The technique, called *distributed snapshots*, is useful for detecting stable property of a computation, as well as for checkpointing. A stable property is one that persists: once a stable property became true it remains true thereafter. Examples for stable properties are "computation has terminated," "the process is deadlocked," and "a message is lost." The basic operation of the distributed snapshot algorithm is quite simple. First, every process records its own local state, then each pair of adjacent processes cooperate in recording the state of the channel they are mutually incident on. Channel states are recorded in three steps:

1. Every process P_x sends a *marker* message along each of its outgoing channels c_{xy} immediately after taking the local snapshot. It does not send any non-marker messages along the channels between taking the snapshot and sending markers.

2. Upon receiving a marker along a channel c_{zx}, if process P_x has not yet taken the local snapshot, then it records its local state and the state of channel c_{zx} as the empty sequence, then sends a marker message along each of its outgoing channels. Otherwise, if it has already taken the local snapshot, then it records the state of channel c_{zx} as the sequence of messages received along c_{zx} in the period between the recording of its state and the receipt of the marker.

3. The set $SN(t_i) = \bigcup_{x=1}^{n} SN_x(t_x)$, where $t_i = \max\{t_1, \ldots, t_n\}$ is a meaningful global snapshot.

It cannot be ensured, that the the states of all processes and channels will be recorded at the same instant, because there is no form of global synchronization. However, the recorded process and channel states will be meaningful. A global snapshot $SN(t_i)$ taken between times $\tau_i \leq t_i \leq \tau_{i+1}$ is defined to be *meaningful* if and only if $SN(t_i)$ is reachable from the global state $S(\tau_i)$, i.e. there is a computation to transform the state of the system from $S(\tau_i)$ to $SN(t_i)$; and the global state $S(\tau_{i+1})$ is reachable from $SN(t_i)$. Denote the set of sent and received messages along channel c_{xy} up to the time t_i as $\text{sent}_{xy}(t_i)$ and $\text{received}_{xy}(t_i)$, respectively. A global snapshot is *feasible*, if at each channel c_{xy} the set of messages $\text{received}_{xy}(t_y) \subseteq \text{received}_{xy}(t_x)$, where t_x and t_y indicate the time corresponding to the creation of local snapshots $SN_x(t_x)$ by process P_x and $SN_z(t_z)$ by process P_y. Note, that a marker sent along a channel c_{xy} serves two purposes: (1) it ensures that the global snapshot will be feasible, and (2) it implicitly informs the recipient process P_y about the set of messages $\text{sent}_{xy}(t_x)$. More discussion on and interesting derivatives of the Chandy-Lamport algorithm can be found in [25, 51, 64, 70, 47].

Lai and Yang derived a distributed snapshot algorithm that uses no markers and does not require the channels to be FIFO [42]. Their idea was to let the distributed snapshot be transformed into a global snapshot *at one node*, thereby eliminating Purpose 2 of the marker. Purpose 1 is achieved without markers by piggybacking the information about the actuality of the message directly on it. The same idea was later revisited in the paper [48] by Li et al.. We present the operation of the snapshot algorithm using the terms of the recent paper. According to the algorithm, each message includes an extra bit, called the *tag bit*. The purpose of the tag bit is to differentiate between messages that are from the current snapshot interval or the previous snapshot interval. The algorithm guarantees that there are no other possibilities.

Suppose all processes have completed the global snapshot $SN(t_{k-1})$, but have not yet started the global snapshot $SN(t_k)$. Furthermore, assume without loss of generality that the tag bit value for the messages is currently zero. At some point a special process P_c, called the *coordinator process*, decides that a checkpoint needs to be taken. First it checkpoints itself, then it broadcasts a 'take local snapshot' message to all the other processes. Each process then observes the following protocol to ensure the above two snapshot consistency properties:

- If process P_x sends a message m_i before it records its local state for snapshot $SN(t_k)$, then it sets the tag bit of m_i to 0. If it sends m_i after recording its local state, then it sets the tag bit to 1.

- If process P_x has not yet recorded its local state for snapshot $SN(t_k)$, and it receives a message m_j with a tag bit of 1, then it must take its local snapshot before processing the message.
- If process P_x has already recorded its local state for snapshot $SN(t_k)$, and it receives a message m_j with a tag bit of 0 along channel c_{yx}, then it includes m_j in the state of c_{yx}.

When the coordinator process P_c determines that each process has finished taking its local snapshots and determining the state its channels, then it broadcasts a 'global snapshot complete' message to all the other processes. The function of the tag bit in the new snapshot interval is flipped.

This last step needs some further explanation. The problem to be solved here is the following: how can a process P_x make sure of there are no more messages from the previous snapshot interval that are bound for another process P_y, but have not yet been received by P_y. The solution to this problem first given by Chandy and Lamport in [15] requires $O(n^2)$ messages. Li et al. improved on this result: their technique takes $O(n \log n)$ messages in the hypercube connected multicomputers, and takes only $O(n)$ messages in the mesh connected multicomputers. The solution assumes the multicomputer interconnection network to be *well-behaved*, that is to satisfy certain requirements [48]. Furthermore, it employs a non-adaptive, memoryless routing function $R(c_{xy}, P_d)$ as described in [21, 22]. The routing function maps the current channel c_{xy} on which the message currently resides, end its destination node P_d to the next channel c_{yz} that it will take on its way to P_d. It is non-adaptive because it always yields the same path from one node to another, and it is memoryless because the function works from channel to channel, retaining no knowledge of the source node. The routing function defines a *channel dependency graph* $G_C = \langle V_C, E_C \rangle$ where V_C is the set of channels, and E_C is the set of edges determined by $R(c_{xy}, P_d)$:

$$E_C = \{(c_{xy}, c_{yz}) : R(c_{xy}, P_d) = c_{yz} \text{ for some } P_d\}$$

Suppose that process P_x has recorded its local state, then the algorithm by Li et al. proceeds as follows:

- For all outgoing channels c_{xy} from P_x, if there is no incoming channel c_{zx} such that $(c_{zx}, c_{xy}) \in E_C$, then P_x sends a 'local state snapshot complete' (LSSC) message on channel c_{xy}.
- For all outgoing channels c_{xy} from P_x, when LSSC messages have been received from all incoming channels c_{zx} such that $(c_{zx}, c_{xy}) \in E_C$, then P_x sends an LSSC message on channel c_{xy}.
- When P_x has sent LSSC messages on all of its outgoing channels, and received LSSC messages on all of its incoming channels, then it can send a 'channel state snapshot complete' (CSSC) message to the coordinator process P_c.

Typical realizations of consistent checkpointing protocols use a distributed consistency protocol, such as two- or three-phase commit, for coordination between the processes [8]. We describe the method developed by Elnozahy et al. [27] both for its simplicity, and because its performance was compared to other approaches of rollback recovery (see Section 4 for the results). The checkpointing procedure is basically fashioned after a two-phase commit protocol. One distinguished process acts as the coordinator and sends messages to the other processes to synchronize the consistent global checkpoint. Each process maintains the most recent consistent checkpoint as a permanent checkpoint. On each invocation of the protocol, every process takes a tentative checkpoint. The tentative checkpoint replaces the permanent one only if the global checkpoint creation terminates successfully. Each global checkpoint is identified by a monotonically increasing *consistent checkpoint number* (CCN). Every application message is tagged with the CCN of its sender, enabling the protocol to run in the presence of message re-ordering or loss. The protocol proceeds as follows:

1. The coordinator starts a new consistent global checkpoint interval by incrementing the CCN and sending 'marker' messages, that contain the new CCN, to each process in the system.

2. Upon receiving a marker message, a process takes a tentative checkpoint by recording its local state and writing the modified pages of the address space to stable storage. The tentative checkpoint is written concurrently with the execution of the application (see also Section 5.1 for a detailed description of these techniques). A process also starts a tentative checkpoint if it receives a message whose appended CCN is greater than the local CCN. Such a message was transmitted after its sender had taken its local checkpoint, therefore the recipient must record its state before delivering it to maintain the feasibility of the global checkpoint.

3. The process sends a 'success' message to the coordinator after the tentative checkpoint has been completely written to stable storage.

4. The coordinator collects the responses from all processes. If all the tentative checkpoints have been recorded successfully, then it sends a 'commit' message to each process; otherwise, it sends an 'abort' message. When a process receives a 'commit' message, it makes the tentative checkpoint permanent and discards the previous permanent checkpoint. Otherwise, if it receives an abort message, then it discards its tentative checkpoint and retains the previous consistent checkpoint as permanent.

Koo and Toueg refined the above approach to guarantee a minimal number of processes to take checkpoints [41]. In their protocol any process P_x can act as the coordinator and initiate a new global checkpoint interval. They modified the operation so a process P_y takes a tentative checkpoint *only if* it receives a 'marker' message from the initiator P_x and the tentative checkpoint of P_x records the receipt of a message m_i from P_y, but the last permanent checkpoint of P_y does not records the sending of m_i. Process P_y determines whether this condition is true using a label piggybacked on the 'marker' message from P_x.

Labels maintained by the Koo-Toueg algorithm are message sequence numbers instead of CCN numbers. Each outgoing application data message is assigned a monotonically increasing *sender sequence number* (SSN). A process P_x stores two of these numbers locally for every other process P_y in the system: the sequence number $LRM_x(P_y) =$

SSN_y^i of the last message m_i received from P_y, and the sequence number $FSM_x(P_y) = SSN_x^j$ of the first message m_j sent to P_y, after P_x took its last permanent or tentative checkpoint. If no such messages exists in the current checkpoint interval, then the value of $LRM_x(P_y)$ and $FSM_x(P_y)$ are $\perp$.

Process P_x *depends on* the state of process P_y if P_x has taken a tentative checkpoint and $LRM_x(P_y) > \perp$ before the tentative checkpoint was recorded. The set of processes whose states P_x is depending on is written as parent(P_x). Every process P_x also keeps a variable called $COMMIT_x$ which represents its ability to participate in the current global checkpoint. Whenever P_x cannot take a local checkpoint (for any reason) $COMMIT_x$ is 'abort'. Since both the Koo-Toueg and the Elnozahy et al. methods are patterned on two-phase commit protocols, their steps have a one-to-one correspondence:

1. The initiator P_c starts the checkpoint protocol by recording its local state and sending 'marker' messages with the value of $LRM_c(P_x)$ piggybacked on them to every $P_x \in$ parent(P_c).

2. A process P_x *inherits* this request if $COMMIT_x$ has the value of 'commit' and $LRM_c(P_x) \geq FSM_x(P_c) > \perp$. If P_x inherits a request, it takes a tentative checkpoint and disseminates 'marker' messages with the value of $LRM_x(P_y)$ piggybacked on it to every $P_y \in$ parent(P_x). (Figure 2.1 illustrates the propagation of 'marker' messages induced by an example conversation in the given checkpoint interval.) Otherwise, if P_x receives but does not inherit a checkpoint request, P_x replies the value of $COMMIT_x$ to P_c.

3. After sending 'marker' messages, process P_x waits for replies that can be either 'commit' or 'abort' from parent(P_x), indicating the acceptance or rejection of the checkpoint request. If any of the replies is 'abort', then $COMMIT_x$ is also changed to 'abort'. When all the replies of parent(P_x) have arrived, P_x sends the value of $COMMIT_x$ to the process whose request it has inherited.

4. Eventually, all replies traveling backward the dependency graph reach their destination and the initiator obtains the values of $COMMIT_x$, for all $P_x \in$ parent(P_c). If every reply was 'commit', the initiator decides to make all tentative checkpoints permanent; otherwise the tentative checkpoints are undone. This decision is propagated to each process which received 'marker' messages.

In the time interval between taking the tentative checkpoint and receiving the decision from the initiator, a process does not send any application messages.

The recovery protocol was designed to meet two requirements: (1) it forces a minimal number of processes to roll back, and (2) it avoids the livelock problem (see Section 1.2). The two requirements can again be satisfied using a two-phase commit protocol. In the first phase, the initiator process P_c requests the dependent processes to restart from their checkpoints. The rollback requests and the local commit/abort decisions traverse the dependency graph in a similar way to the checkpoint requests and commit/abort decisions, but in the opposite direction. The roles of certain data structures change as well: the $COMMIT_x$ variable now represents the ability of process P_x to roll back, and instead of $FSM_x(P_y)$ processes store the sequence number $LSM_x(P_y) = SSN_x^i$ of the

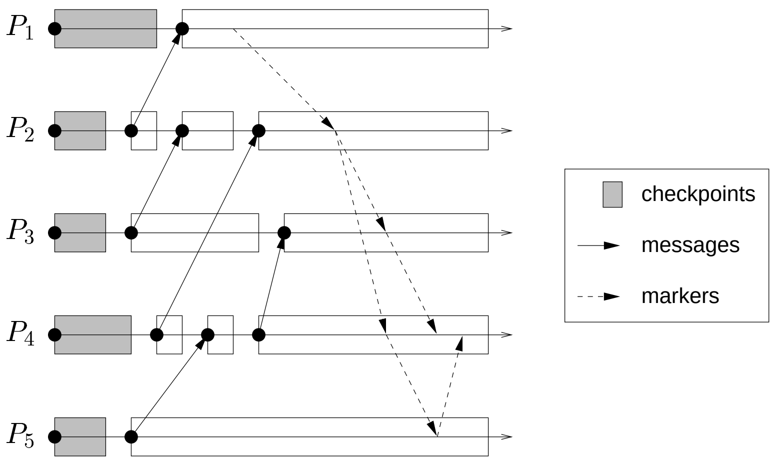

Figure 2.1: Dissemination of markers after a conversation

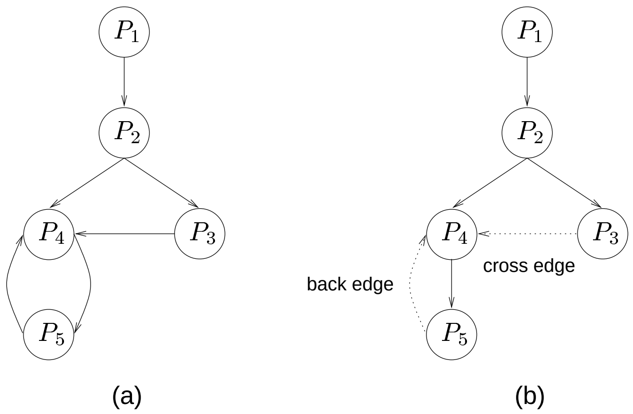

Figure 2.2: Transitive dependency among processes: (a) Graph, (b) Tree

last message m_i sent to P_y. The value of $LSM_x(P_y)$ is also piggybacked on rollback request 'marker' messages. A process P_x inherits a rollback request, if the rollback of P_c *undoes* (rolls back to a state before the event of) the sending of a message to P_x and P_x has not inherited another request to roll back, i.e. $COMMIT_x$ is 'commit' and $LRM_c(P_x) \geq LSM_x(P_c)$. When all local commit/abort decisions from the dependent processes were received by the initiator process P_c, it decides whether or not to roll back the system. Provided, that all of the processes have agreed upon rollback, this decision is broadcasted and carried out in the second phase. Since every process follows the command of the initiator, the global state after the termination of the protocol is consistent.

Kim and Park omitted the need for a second phase of coordination, by assigning the checkpoint-commitment decision to each process participating in checkpointing, rather than waiting for the decision of the coordinator [39]. As the initiator process P_c transmits 'marker' messages, the checkpointing requests propagate along the *dependency graph* $G_D = \langle V_D, E_D \rangle$, where V_D is the set of dependent processes and E_D is the set of directed edges representing the dependency between two related processes. Figure

2.2(a) shows the dependency graph derived from the example computation of Figure 2.1. Let reach(P_x) denote the set of processes are *transitively* dependent, that is

$$\text{reach}(P_x) = \{P_y : \exists P_x \xrightarrow{e_1} \cdots \xrightarrow{e_k} P_y\}, \text{ where } \{e_1, \ldots, e_k\} \subseteq E_D$$

Upon receiving a checkpointing request, a process P_x inherits it only for the first time, redundant copies of the 'marker' message are rejected.

When all of the processes in G_D received a 'marker' message and recorded their local state, the decision of checkpoint commit/abort is made upwards to the coordinator P_c. Those processes make their decisions first, that do not have processes depending on them (called *leaf* processes). As an optimization, commit/abort decisions are transmitted only via the edges of the dependency tree (which is a spanning tree of G_D), as illustrated in Figure 2.2(b). Since decisions are not collected along the back- or cross-edges, a process P_x cannot make a decision by itself if it has a process $P_y \in \text{reach}(P_x)$ to which no path exists in the dependency tree (i.e. P_x and P_y are only connected by back- and/or cross-edges). Therefore, instead of sending either a 'commit' or 'abort', it responds to the checkpointing request with an 'unsure' message to ask its parent process to make a decision for him. The responsibility of the checkpoint-commit decision may be subsequently handed over until there is a process P_z which is able to decide on commit (if every process $P_w \in \text{reach}(P_z)$ responded with either 'commit' or 'unsure'). In this case, P_z makes its tentative checkpoint to be permanent and sends 'commit' messages to its parent and unsure children processes. If any of the processes $P_w \in \text{reach}(P_z)$ has aborted its checkpoint, then P_z discards its tentative checkpoint and sends 'abort' messages to its parents and unsure children.

In order to check if all processes $P_y \in \text{reach}(P_x)$ are descendants of P_x in the dependency tree as well, process P_x maintains two n-bit arrays. The entry $DES_x[y]$ is set to one when process P_y inherits the checkpointing request of P_x, and the entry $SENT_x[y]$ is set to one when the 'marker' message was sent to P_y. A child process piggybacks these arrays on the checkpoint-commit decision it sends in reply to its parent. When a process receives a decision from its child process, it merges the two arrays with its own arrays:

$$\begin{aligned} DES_x &= (\textstyle\bigcup_{P_y \in \text{reach}(P_x)} DES_y) \cup DES_x \\ SENT_x &= (\textstyle\bigcup_{P_y \in \text{reach}(P_x)} SENT_y) \cup SENT_x \end{aligned}$$

If $DES_x = SENT_x$, then all processes $P_y \in \text{reach}(P_x)$ are descendants of P_x in the dependency tree, therefore P_x can make its local checkpoint-commit decision.

Another advantage of localizing the decision about checkpoint commit/abort comes when multiple checkpoint request are disseminated simultaneously in the system. In order to differentiate between independent checkpointing sessions, the 'marker' and decision messages carry the identifier of the coordinator. When a process P_x is participating in a checkpointing session initiated by P_c, and receives a 'marker' message tagged with the identifier of P_d, it can use its already existing tentative checkpoint for the new checkpointing session, since P_x does not send any application messages after taking the tentative checkpoint. The delivery order of checkpointing requests is nondeterministic, thus different subtrees rooted at P_x (including the same set of $P_y \in \text{reach}(P_x)$ processes) may be formed by various sessions. Therefore, 'marker' messages must be disseminated once again in each new session to form a complete tree even if the dependency graphs contain cycles. If $DES_x = SENT_x$, then P_x can make its local checkpoint-commit decision for all of the checkpointing sessions it is involved in, and reply immediately to its parents in each dependency tree. Otherwise, P_x is unsure and delegates the responsibility of decision-making to its parents. When a parent in any of the sessions decides to send 'commit', P_x will commit immediately since it is guaranteed that every process $P_y \in \text{reach}(P_x)$ has reached the same decision. On the other side, an 'abort' message from a parent may be due to a failure of a process $P_z \notin \text{reach}(P_x)$, and so P_x waits for the decision of its parents in other checkpointing sessions. Only a unanimous 'abort' reply from all of the parents will make P_x abort and discard its tentative checkpoint.

Chapter 3

Message Logging

Message logging refers to the backward error recovery technique in which the interactions between processes are saved instead of the complete global state of the system. Process interactions are denoted as events. Events drive the transition of a process from one state interval to another, thus determine the course of computation. In a piecewise deterministic system (see Section 1.1) events are limited to the sending and receiving of messages. Therefore the distributed computation can be restarted from any earlier state and replayed identically to the previous run by creating a so called *message log.*

The message log contains the complete history of the interactions among application processes. The main components of this history are the log entries that identify the transmitted messages, together with a dependency information providing a partial causal ordering over the logged messages. These two components enable the recovery manager to recreate the communication that took place before it was invoked. Then, due to the determinism each process will go through the same state transitions as earlier, hence the computation is reproduced. Storing the contents of messages in the log speeds up the recovery process. Yet, it is not strictly necessary for a successful rollback, since the messages can be regenerated together with the other constituents of the distributed computation.

The messages must be uniquely identified in order to record their processing order for subsequent replay. The following items of system data are necessary to maintain for each entry corresponding to a message m_i in the log:

- Identifier of the sender process, P_s^i: each process P_s is assumed to have a uniquely assigned number that identifies it. The number of the sender process is included in each message to record its source.

- Identifier of the recipient process, P_r^i: the number of the destination process P_r for a message m_i.

- Sender sequence number, SSN_s^i: a monotonically increasing number counting the amount of messages sent by the process P_s. It is included in the message to suppress the reception of duplicates copies of the same information during recovery.

- Recipient sequence number, RSN_r^i: the number of messages received by the process P_r.

Each process also keeps track of the highest SSN and RSN numbers received so far in two tables. These tables together with the other data enable the recovering process to recreate the correct causal order of messages that entered the log.

Because of network delay, a message m_i that originates at process P_s^i may arrive to its destination P_r^i after P_s^i has failed. If the recipient process P_r^i has been notified of the failure of P_s^i before receiving m_i, P_r^i will not be able to determine whether the message was started before the failure or after the recovery of the sender process P_s^i. This is a problem of ordering the perception of failures with respect to messages. To solve the problem, each process starts a new incarnation [68] at the beginning of each recovery. An incarnation is identified by a monotonically increasing incarnation number, and every message m_j is tagged with the current IN_s^j incarnation number of the sender P_s^j. When a process enters the recovery protocol, it reliably informs the other processes of the computation about its new incarnation number. Messages tagged with old incarnation numbers are rejected. The tuple of the above five fields $\langle P_s^i, SSN_s^i, P_r^i, RSN_r^i, IN_s^i \rangle$ unequivocally determines the message m_i and the order in which m_i was transmitted from P_s and delivered by P_r, therefore this tuple is often referred to as the *determinant* of message m_i. Note, that if the assumption of piecewise determinism holds, it is sufficient to log the determinants of messages without the contents for further replay. Naturally, incorporating the contents of messages in the log speeds up the recovery process and decreases the communication load.

Two main message logging techniques can be distinguished with respect to the place where the logging action is carried out:

Receiver-based message logging. With receiver-based logging, the processes participating in a distributed computation log the messages upon receipt to stable storage during failure-free operation. When recovering from a failure, the failed process restarts from a previous checkpoint and replays the messages in the log. This way the execution is restored to a consistent state that occurred before the failure. However, the recovery protocol is more complicated if not the so-called *pessimistic logging policy* is employed, i.e. messages are not written immediately to stable storage after receiving them. The other approach, asynchronous logging (also called *optimistic logging policy*) gives a better failure-free performance, but introduces a window of time when the received messages are only buffered in the volatile memory of the recipient process. This temporary buffer is periodically flushed to stable storage. Still, there is a significant probability of errors occurring when the temporary buffer is not empty, in this case the unstable message log entries will be lost.

Lost messages will eventually result in orphan states that create orphan processes. Orphan processes are inconsistent with the actual global system state, therefore they must also be rolled back (in contrast with pessimistic logging, where failure-free processes are never rolled back). Determining which processes became orphans and computing the maximum recoverable state make the recovery process more intricate. Furthermore, a garbage collection protocol is required to reclaim old checkpoints and messages, although each process may have still to maintain multiple checkpoints on stable storage [65].

Sender-based message logging. In sender-based logging, messages are stored in the volatile memory of the sender process (of course, checkpoints must nevertheless be stored in stable storage). If a process fails, the messages needed for execution replay are available at their sender, i.e. the sender-based approach inherently tolerates a single node failure. However, in the case of multiple concurrent failure occurrences the same problem of orphan process handling arises as in optimistic receiver-based logging. There are two ways of avoiding this problem. First, the volatile message logs can be periodically written to stable storage to decrease the number of potentially lost message and so increase the maximum recoverable system state. A better solution is extending the logging protocol to tolerate several simultaneous node failures, since it eliminates the susceptibility for orphan processes completely.

There are two distinct variances of such enhanced logging protocols: family-based methods (FBL, see Section 3.4 for details), and the technique given by Strom et al. [66]. Family-based methods provide a set of logging algorithms that can be tailored to system characteristic, since there is a protocol in the set that tolerates at most k failed processes for any $1 < k \leq n$. The underlying idea is to disseminate and store the logged data among additional k nodes to obtain the necessary level of redundancy. Storm et al. followed a different approach by separating the logging of message contents from the receipt order. The content of a message is still stored at the sender, while its receipt order is maintained at the receiver. The resulting protocol tolerates an arbitrary number of failures.

The act of logging a message under sender-based logging is not atomic, since part of the message data is entered into the log when it is sent, but the corresponding RSN is only obtained after the message has reached its destination. It is thus possible for the receiver to fail while some messages do not yet have their RSNs recorded at the sender; such messages are called *partially logged messages*. The messages acknowledged by a failure-free recipient process become *fully logged*. The sender-based message logging protocols are designed so that any partially logged messages that exist for a failed process can be sent in any order after a sequence of fully logged messages have been sent to it in ascending RSN order.

3.1 Classification of message logging

Alvisi and Marzullo [4] gave the first formal specification of the consistency property "no orphan processes." From this specification they derived how the different classes of message logging protocols implement this property. Let $\text{deliver}(m_i)$ denote the event that corresponds to the delivery of message m_i with the determinant $\langle m_i \rangle = \langle P_s^i, P_r^i, SSN_s^i, RSN_r^i \rangle$ by process P_r. The set of processes whose state reflects the delivery of message m_i, in other words, they *depend* on it, is written formally as

$$\text{depend}(m_i) = \{P_j : j = r \vee \exists\, e_j, \text{deliver}(m_i) \rightarrow e_j\}$$

Denote the smallest set of fault-free processes which would cause the determinant of deliver(m_i) be lost in case they would fail as log(m_i). If processes fail independently and message logs are kept in volatile memory, then log(m_i) contains the set of processes that maintain a copy of $\angle m_i \rangle$ in their address space; note that process P_r^i is a member of log(m_i) once it delivers m_i. If the determinant of deliver(m_i) is logged to stable storage, then any subset of processes can fail without $\langle m_i \rangle$ being lost. This can be expressed as the size of log(m_i) is greater than the number of processes: $|\log(m_i)| > n$, in this case the members of log(m_i) need not be defined.

Suppose that a set of processes $\mathcal{C} \in \{P_1, \ldots, P_n\}$ fails. The determinant of a delivery event deliver(m_i) is lost when $\log(m_i) \subseteq \mathcal{C}$. The process P_r^i that originally delivered m_i is among the set of failed processes and must execute the recovery procedure. However, P_r^i may not know the order in which m_i was originally delivered, and so may re-execute a different set of instructions than before the failure, since $\langle m_i \rangle$ was last. A process P_o is an *orphan process* if its consistency depends on P_r^i being able to exactly re-execute the same set of instructions during recovery:

$$P_o \text{ is orphan} \Leftrightarrow P_o \notin \mathcal{C} \wedge \exists\, m_j, P_o \in \text{depend}(m_j) \wedge \log(m_i) \subseteq \mathcal{C} \tag{3.1}$$

A sufficient condition for there being no orphans created by the failure of a set of processes $\mathcal{C}$ can be obtained by quantifying Equation 3.1:

$$\forall i, \log(m_i) \subseteq \mathcal{C} \Rightarrow \text{depend}(m_i) \subseteq \mathcal{C} \tag{3.2}$$

The consistency requirement states that no set of failed processes results in the creation of orphans. Quantifying Equation 3.2 over all possible set of processes gives the sufficient condition for orphan-freedom:

$$\forall i, \text{depend}(m_i) \subseteq \log(m_i) \tag{3.3}$$

In many environments it is feasible (or necessary) to assume that the number of overlapping failures is bounded by a constant f. The condition for orphan-freedom expressed by Equation 3.3 needs to hold only when there can be a delivery event depend(m_i) whose determinant is not stable. The determinant $\langle m_i \rangle$ is called *stable* when it cannot become lost due to process failures. Unstable message determinants arise if the size of log(m_i) is not greater than the number of failed processes:

$$\forall i, |\log(m_i)| \leq f \Rightarrow \text{depend}(m_i) \subseteq \log(m_i) \tag{3.4}$$

Different classes of message logging protocols can be characterized with respect to Equation 3.4. *Optimistic protocols* assume that the time interval during which $|\log(m_i)| \leq f$ holds is brief, and so orphan-freedom is true nearly all the time. In the unlikely event that failures occur Equation 3.3 does not hold, the recovery protocol can diminish depend(m_i) by rolling back processes with orphan states until $\text{depend}(m_i) \subseteq \mathcal{C}$. Thus, optimistic message logging protocols implement the following property:

$$\forall i, |\log(m_i)| \leq f \Rightarrow (\log(m_i) \subseteq \mathcal{C} \Rightarrow \Diamond(\text{depend}(m_i) \subseteq \mathcal{C}))$$

where $\Diamond$ is the temporal 'eventually' operator.

Pessimistic protocols do not admit orphans and so must implement Equation 3.4. The reasoning behind the pessimistic approach comes from the same observation of optimistic protocols: the time during which $|\text{log}(m_i)| \leq f$ holds is brief. During this interval, the process P_r^i that delivered m_i is not allowed to send any application messages. In this way, pessimistic message logging protocols implement the following property:

$$\forall i, |\text{log}(m_i)| \leq f \Rightarrow |\text{depend}(m_i)| \leq 1$$

Although the blocking of processes incurs loss of performance, it is unlikely that P_r^i wishes to send an application message during the short blocking period, and the performance loss will be small.

Equation 3.4 does not bound the size of $\text{log}(m_i)$ and so it may grow arbitrarily larger than $\text{depend}(m_i)$. This allows for protocols that disseminate a large number of unnecessary copies of $\langle m_i \rangle$. These extra copies may end up wasting a significant portion of the communication bandwidth and the address spaces of the processes in the system. The minimal requirement: $\text{log}(m_i) = \text{depend}(m_i)$ is much too complicated to enforce, thus the following weaker requirement was found to be apt for implementation:

$$\begin{aligned} \forall i, j \quad : \quad & |\text{log}(m_i)| \leq f \Rightarrow \text{depend}(m_i) \subseteq \text{log}(m_i) \wedge \\ & (P_j \in \text{log}(m_i) \Rightarrow \Diamond(P_j \in \text{depend}(m_i) \vee P_j \notin \text{log}(m_i))) \end{aligned} \tag{3.5}$$

Equation 3.5 strongly connects logging with causality. It states that all processes who causally depend on $\text{deliver}(m_i)$ have logged $\langle m_i \rangle$. Also, if a process has logged a copy of $\langle m_i \rangle$ and does not leave $\text{log}(m_i)$ (e.g. by failing), the it will eventually causally depend on $\text{deliver}(m_i)$. Therefore, this class of message logging protocols is called *causal protocols*.

3.2 Pessimistic message logging

Pessimistic message logging aims at maintaining the consistency of the recorded dependency information always consistent with the actual state of the system. It therefore does not allow the applications to enter in a situation susceptible to become inconsistent. The employed logging policy is called *synchronous*, because the application must suspend communication (or in some implementations even execution) until it receives notification of the logging of the message. As the information recorded on stable storage is always consistent, the recovery procedure is relatively simple and fast. That is the reason for calling the approach "pessimistic": it prepares to handle frequent failure occurrences by improving the rollback latency on the expense of the failure-free performance.

Two important methods for the receiver-based pessimistic approach are described in literature. The first, "Publishing" uses a common bus architecture (like Ethernet), with a dedicated *recorder* unit attached to that bus. All messages that travel along the bus as well as checkpoint and recovery information are reliably stored by the recorder. The other approach, called "Auragen", is based on atomic sending of messages to the receiver and backup processes. In both cases, normal operation is suspended until the messages are fully logged.

In the method of Powell and Presotto, a recorder unit monitors all messages between the processes and stores this inter-process communication reliably and instantaneously

[57]. A link-level protocol is necessary to assure that the recorder and recipient process accept or reject the same messages, hence the message log maintained in the recorder reflects the actual message traffic. This synchronous saving of messages called "publishing" assumes frequent failure occurrences and assures that after every message, a consistent state can be reproduced using the recorded log. Periodically a local checkpoint of every process is written to stable storage (one process at a time). Previous messages received by the checkpointed process may then be discarded from the log.

After a failure, the last saved checkpoint is restored and the consistent state is reconstructed by replaying the logged messages. Duplicate messages (resent by the recovering process) are absorbed by the kernel until the consistent global state is achieved. It is possible to migrate the recovering process to different physical processor, in this case the kernel has the responsibility to redirect the messages for the relocated process.

Borg et al. propose a scheme where all inter-process messages are atomically sent both to the receiving process and to inactive backups of both sender and destination processes [13]: (1) at the recipient process the message is queued for processing, (2) at the backup of the sender process, it is queued for possible replay after rollback, and (3) at the backup of the recipient process, only the RSN counter is incremented. This complex logging action is done atomically, which implies a significant synchronization delay. The inactive backups (residing on other processors) are also regularly updated. The parts of the memory that changed since last synchronization point are stored, and the messages logged between the two previous synchronization points are discarded. A page server is used to replace the changed memory pages with those of the primary process.

After a process failed, its backup takes over from the previous synchronization point. It has all necessary messages available and re-executes the lost computation. Sending of duplicate messages is avoided using the RSN counter at the backup of the recipient process. All the processes that communicate(d) with the failed process must be notified that the process migrated to its backup. The user can fine-tune fault-tolerance characteristics: the protocol can run in a "full, half, a quarter or not" backed-up mode. The various modes are distinguished depending on that a backup is created before (full backup) or after (half backup) the failed process restarts, or no new backup is created (quarter backup) after a failure.

Several drawbacks of this technique are worth to mention. A special communication protocol is necessary for atomic delivery of the messages to the three simultaneous destinations. (At least) the double amount of hardware is necessary (due to the backup nodes). These backup nodes can possibly be used for other processes (only at synchronization-time, their CPU is used), but in case the primary node crashes, the backup node gets often overloaded so that the original processes have to be dropped.

Johnson and Zwaenepoel presented a sender-based message logging protocol [35]. Messages are stored in the volatile memory of the sender and this message log is asynchronously written to stable storage as part of the periodic local checkpoint of the sender (independently of the computation). The technique aim at minimizing the overhead associated with the logging of the messages. The sender process P_s sends the message m_i (with the assigned SSN_s^i) and stores the related partial logging data in its volatile memory. Upon the receipt of m_i, the receiver process P_r returns an acknowledgement containing the RSN_r^i (receive sequence number) to P_s. The sender process adds the obtained RSN to the entry corresponding to m_i in the log. In the meantime, the receiver suspends sending of new messages (but may continue performing other operations). After fully logging the message, the sender sends a notification to the receiver, so it can resume the sending of new messages.

The recovery protocol is the following: the failed process is restarted on available hardware from its most recent checkpoint. After restart, it broadcasts a request for logged messages. First, all fully logged messages are resent to the process, in ascending order of the RSNs. Then the partially logged messages are retransmitted (in arbitrary order). The recovering process makes up for the lost work and will thus re-send all messages since the last checkpoint. These duplicate messages can be recognized by the receivers using the table of the highest received SSNs. If the SSN_s^i of a message m_i is less than or equal to the SSN of a certain process from the table, m_i is rejected by the recipient process P_r^i as duplicate. In case of process P_r^i did not take a local checkpoint since the first receipt of m_i, it sends an acknowledgement with the same RSN_r^i as previously. Otherwise, the sender is notified that the message is already a part of a checkpoint and therefore need not be logged.

3.3 Optimistic message logging

Optimistic message logging avoids the blocking of normal computation. It lacks the synchronization phase that assures messages are always fully logged before they would be delivered by the recipient processes. Log entries are written to stable storage asynchronously, that is after a process P_s sends message m_i to process P_r, normal computation at both processes may continue without waiting for any notification. Message m_i is stored (at either the sender or recipient) in a volatile buffer and gets (eventually, either when the buffer becomes full or a time-out occurs) logged on stable storage. It might happen that process P_r receives m_i but the sender process P_s fails before the log entry corresponding to m_i is stable (i.e. it is recorded reliably on stable storage). Such a situation may cause the loss of unstable log entries and the creation of orphan processes. Therefore it is necessary to undo this receipt event, and all depending processes should roll back to a consistent recovery-line. On the other hand, if the message m_i gets logged, the communicating processes need to be informed of this, so that the dependency on lost messages can be dropped [7, 52].

A significant advantage of the optimistic to the pessimistic approach is that the asynchronous logging increases the failure-free performance of the system. This reflects an "optimistic" an in many cases more close) approximation of reality: the failure events are supposed to be rare. This enables computation, communication, and checkpointing to proceed without blocking the system; hence, time overhead is reduced. The major drawback of optimistic logging is the more complex recovery protocol and the resulting higher overhead associated with the maintenance of the inter-process dependencies.

According to [62] two approaches are possible to track the dependencies in a decentralized way:

1. The transitive dependencies of each state interval can be piggybacked to each message: this requires $O(n)$ extra information per application message and $O(n^2)$

messages necessary for recovery.

2. If only direct dependencies are kept, it is sufficient to include only the state interval number to each message (i.e. $O(1)$ extra information per message), but then $O(n^3)$ messages are necessary for recovery.

Strom and Yemini developed the first optimistic logging algorithm in [67, 68]. Their approach is based on dependency tracking. The dependency information maintained by each process P_x is in the form of a vector of message identifiers on which P_x depends. Messages are identified by their send sequence number and incarnation number appended as the tuple $\langle SSN_s^i, IN_s^i \rangle$ to each message. Any sender process P_s piggybacks his local dependency vector on every message it sends. The receiving process P_r then updates its own dependency vector comparing the entries of the local vector to the received one. In this way all direct and transitive dependencies can be tracked.

A copy of the sent message m_i is held in the volatile memory of the sender process P_s until it is logged. The recipient process P_r writes the message data supplemented with its receive sequence number RSN_r^i to the stable storage. When m_i is stable, the recipient process can remove this message from its dependency vector and notify the sender that it does not rely on the message any more. This information is broadcasted and kept in log vector to let each process know which messages are logged. This way they can update their local dependency vector and release redundant copies of stored output messages. An empty dependency vector for a certain checkpoint means this checkpoint can be committed and any preceding checkpoints and logged messages may be discarded. This so-called *garbage collection* avoids the domino-effect: a process is guaranteed to roll back only to its last committed checkpoint. This does not mean that no multiple roll-backs will occur: by tracking the dependencies a process might roll back $O(2^n)$ times in response to a single failure, in the worst case.

When a process fails, it recovers by restoring its last committed checkpointed state, replaying its input-message log and re-executing the lost computation. The incarnation number of the process is is incremented and normal operation resumes. Possible inconsistencies arising from the failure can be tracked down using the "expected" sender sequence and incarnation numbers:

1. The failed process may have lost messages which other processes believe are already sent, if these messages were not yet logged at the time of failure. This is the case if the SSN received in the message is newer than the expected SSN. The missing messages can be reclaimed from the volatile memory of their sender.

2. The recovering process may re-send messages that were already received. These duplicate messages are identified by earlier received SSN and IN numbers than expected. These messages are already located in the message log, and therefore they are rejected.

3. Some processes may have received some messages that became lost or orphaned as result of the failure. This situation occurs if the SSN of the message is earlier, but its IN is newer than the expected value. The orphan messages and states must be eliminated in the following steps:

- If the orphan was received but not yet logged, it is deleted,
- If the orphan is logged but not yet processed, it is deleted from the input queue and from the log as well. The incarnation number is incremented. Possible subsequent orphan messages are also discarded.
- If the orphan was already processed, the process is rolled back to the last checkpointed state prior to the creation of orphan. The computation is re-executed, discarding all orphan messages; then the incarnation number is incremented.

This algorithm supports the loss of unstable message log entries in the volatile storage of any number of processes. Its typical disadvantages include the increased bookkeeping (hence time and storage overhead) and the intensive communication (broadcasting log vectors, larger messages). Furthermore multiple roll-backs may occur.

Johnson and Zwaenepoel published in [35] an optimistic alternative to their sender-based logging protocol (described in Section 3.2). It makes the optimistic assumption that the logging of the majority of messages will eventually be completed (through retransmissions if necessary) before a failure occurs. If the assumption turns out to be wrong, the protocol still maintains enough extra information to be able to roll back the system and to recover a consistent global system state.

Due to the lack of synchronization of the sender and recipient processes for full logging, it is possible for a process to enter a inconsistent state during recovery after a failure. As an example, consider the situation where process P_s has received a message m_i then sent a message m_j to process P_r, and subsequently P_s failed before its sender could have added the RSN_r^i to the partial log entry corresponding to message m_i. During recovery the receipt order for m_j might not be possible to determine any longer, thus process P_s potentially can not recreate the state from which message m_j was sent. Therefore, the state of a process is called *unrecoverable* unless all messages it has received are fully logged at their senders. If a process fails in an unrecoverable state, the state is lost, and the processes and states that depend on the lost state become *orphan* processes and states, respectively.

The recovering process P_x broadcasts the value of RSN to which it was able to return, in order to determine whether its failure has cause some other processes to become orphaned. Any process that has a newer value recorded in the entry corresponding to P_x in its table of highest RSNs received concludes it is an orphan. This orphan-detection procedure must also be invoked before a process takes a local checkpoint, since orphan processes must be rolled back, and a checkpoint from before the orphaned state will be needed for recovery. After restoring the checkpointed state of the failed process, the states of any orphaned processes are rolled back by recovering their latest consistent checkpointed state and replaying their message logs in the same manner to as for failed processes. Some of the messages to be replayed might have been recorded at the failed process, but his information is recreated during the recovery of that process.

In [66] a scheme based on the above approach is presented, that proposes two enhancements. The first one permits asynchronous message logging without the restriction to single fault-tolerance, while the other avoids logging the content and receipt order information of messages to stable storage if this information can be reconstructed in other ways, so only the messages received from the environment a few additional messages must be written to stable storage.

A difference is made between logical and physical logging. Information is said to be *logically* logged if it is available on demand, should it be either on volatile or stable storage. It is *physically* logged if it resides on stable storage. The strategy the rollback-recovery protocol is to use the volatile storage of the sender as a log when the sender is failure-free, and the deterministic execution of the sender to replay its history in case of an error. All necessary messages will be either replayed (at available nodes) or recomputed (at failed nodes), thus allowing for multiple failures.

However, now it is possible that more than one checkpoint is needed to replay the execution history in case other processes require messages originated from before the last checkpoint. Therefore, the sender spools some messages to stable storage between checkpoint instances to avoid the domino-effect. The difference between spooling and logging is that external messages are not delayed until spooling is complete, and there is no notification broadcasted at end of the spooling. The appropriate set of messages should be recorded on stable storage, hence the spooling algorithm is very important. This is easily demonstrated by considering the two extremities of spooling strategies: if every message was always written to stable storage, there were no gain in comparison to pessimistic logging; and with the lack of spooling the domino-effect can occur. Extra bookkeeping has to be done to collect messages no longer needed and checkpoints that can be committed. This accounts for the following paradox: by writing data (spooling) to stable storage, space on stable storage actually *increases* (as the garbage collection procedure removes the meaningless older checkpoints).

The second enhancement is related to the logging of the message receipt order. The algorithm, called *null logging*, does not log the message determinants physically if a heuristic can predict the correct RSN (so logically the message is fully logged). The reduction of the logging overhead is strongly dependent on the characteristics of the applied heuristic. This is in turn depends on the application and the interaction between sender and receiver. Examples of heuristics can be: (1) the next message will probably arrive from the same sender process as the currently received one (when bursts of communications occur), or (2) every sender will send in turns to the same process (round robin algorithm). Note, that in this technique the domino-effect is not inherently avoided (it depends on the heuristic).

A general model for recoverability

In a distributed system using message logging and checkpointing to provide fault tolerance, there is always a unique maximum recoverable system state. The maximum recoverable system state never decreases, and if all messages are eventually logged, the domino effect cannot occur. Johnson and Zwaenepoel presented a general model for reasoning about recovery [36, 37]. They also proved the existence of the maximum recoverable system state, regardless of the message logging protocol used. The proof of this relies on the observation that the set of system states that have occurred during any single execution of a system forms a lattice, with the sets of consistent and recoverable system states as sub-lattices.

Upon each receipt of an input message, a process begins a new state interval, a deterministic sequence of execution based only on the previous state of the process and on the contents of the message. (Note, that this definition of a state interval differs from the one presented in Section 1.2, which allows for state intervals initiated by send or internal events as well.) Each state interval is identified by a unique sequential state interval index σ_x^i, which is simply the receive sequence number of input messages at process P_x. All dependencies of P_x on some other process P_y can be encoded as the maximum index of any state interval σ_y^j of P_y on which process P_x depends. All dependencies of any process P_x can, therefore, be represented by a *dependency vector*

$$\mathbf{d}_x = (d_{x,i} \equiv \sigma_x^i) = (\sigma_x^1, \ldots, \sigma_x^n)$$

Component σ_x^y of the dependency vector $\mathbf{d}_x$ gives the maximum index of any state interval of process P_y on which process P_x currently depends. Component σ_x^x is always set to the current state interval index of process P_x. If P_x has no dependency on any state interval of some process P_z, then σ_x^y is set to $\perp$, which is less than all possible state interval indices. Processes cooperate to keep their current dependency vectors up-to-date by piggybacking all messages sent with their current state interval index (i.e. RSN), and by maintaining the maximum state interval index received from each other process.

The state of the system is the composition of the states of all component processes of the system and may be represented by an $n \times n$ *dependency matrix*. Taking the dependency vector $\mathbf{d}_x$ of each process P_x in the system, the dependency matrix

$$\mathbf{D} = [d_{ij} \equiv \sigma_i^j] = \begin{bmatrix} \sigma_1^1 & \sigma_1^2 & \ldots & \sigma_1^n \\ \sigma_2^1 & \sigma_2^2 & \ldots & \sigma_2^n \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_n^1 & \sigma_n^2 & \ldots & \sigma_n^n \end{bmatrix}$$

can be formed, where row x: σ_x^i, $1 \le i \le n$, is the dependency vector for process P_x. Since component σ_x^x of the dependency vector belonging to process P_x is always its current state interval index, therefore the diagonal of the dependency matrix shows the current state interval index of each process in the system. The set of all system states that have occurred during any single execution of some system forms a partial order called the *system history relation*, in which one system state precedes another if and only if it must have occurred first during the execution. This relation can be expressed in terms of the state interval index of each process as shown in the dependency matrix:

$$\mathbf{A} \preceq \mathbf{B} \Leftrightarrow \forall i, [a_{ii} \le b_{ii}]$$

where $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$ are system states. This partial order differs from the *happened before* relation defined by Lamport [44] in that it orders the system states that result from events rather than the events themselves, and that only state intervals (started by the receipt of a message) constitute events.

A system state describes the set of messages that have been received by each process. Two system states $\mathbf{A}$ and $\mathbf{B}$ can be combined such that each process receives all of the messages that it has got in either of the two original system states. This is called the *union* of $\mathbf{A}$ and $\mathbf{B}$, and it can be expressed in terms of the dependency matrices describing these system states by choosing for each process the row that has the largest state interval index of the corresponding rows in the original matrices:

$$\mathbf{A} \cup \mathbf{B} = [c_{ij}] \Rightarrow \forall i, \left[c_{ij} = \begin{cases} a_{ij} & \text{if } a_{ii} \geq b_{ii} \\ b_{ij} & otherwise \end{cases} \right]$$

Likewise, the *intersection* of two system states **A** and **B** can be formed such that each process receives only those messages that it has got in both of the two original system states. This can be computed from the related dependency matrices by choosing for each process the row that has the smallest state interval index of the corresponding rows in the original matrices:

$$\mathbf{A} \cap \mathbf{B} = [d_{ij}] \Rightarrow \forall i, \left[d_{ij} = \begin{cases} a_{ij} & \text{if } a_{ii} \leq b_{ii} \\ b_{ij} & otherwise \end{cases} \right]$$

The set of system states that have occurred during any *single* execution of some system, ordered by the system history relation, forms the *system history lattice.* For any two system states **A** and **B** in this set, their *least upper bound* is $\mathbf{A} \cup \mathbf{B}$ and their *greatest lower bound* is $\mathbf{A} \cap \mathbf{B}$.

A system state is *consistent* if and only if all messages received by all processes have either already been sent in the current state interval of the sender process or can deterministically be sent by that process in the future. Since process execution within a state interval is deterministic, only the messages sent before the end of the current state interval can deterministically be sent. A message shown by the system state to be sent but not yet received (in-transit message) does not cause it to be inconsistent. However, a message recorded as received but not yet sent cannot deterministically be sent in the future, hence makes the system state to be inconsistent. In other words, a system state is consistent if and only if no process depends on a sender state interval of greater index than the current state interval of the sender in the dependency matrix. Expressing the consistency property formally: for each column k of the dependency matrix $[d_{ij}]$ belonging to system state **D**, no element in that column may be larger than the element on the diagonal of the matrix:

$$\mathbf{D} \text{ is consistent} \Leftrightarrow \forall i, j : [d_{ij} \leq d_{jj}] \tag{3.6}$$

The set of consistent system states forms a sub-lattice of the system history lattice.

A system state is called *recoverable* if and only if all component process states are stable and the resulting system state is consistent. A state interval of a process becomes *stable* when all messages received by the process after its latest checkpoint are logged. A message is *logged* if its data and the index of the state interval that it started in its receiver are both recorded on stable storage. Any stable state interval for a process can be recreated by restoring the process from the latest checkpoint and replaying any logged messages in the recipient state interval order. The local checkpoint includes the complete current dependency vector for the associated process. A logged message only gives the direct dependency created in the receiver by this message. The complete dependency vector for any stable state interval of some process can always be determined, however, since all messages that started state intervals since the latest checkpoint must be logged. To recover the state of the system, it must be possible to recover the states of the component processes. The recovered system state is *meaningful* if and only if there exists a failure-free execution of the system to achieve it from its initial state. The set of recoverable system states that have occurred during any single execution of some system forms a sub-lattice of the system history lattice.

In recovering after a failure, we wish to restore the most up-to-date global system state possible from the information available on stable storage, in order to minimize the amount of re-execution necessary to complete the recovery. The authors proved that there always exists such a unique maximum recoverable system state, and gave an algorithm which computes this state. Conceptually, this process could be an exhaustive search of all combinations of stable process state intervals until the maximum recoverable system state is found, but of course, such a search is too expensive in practice.

The recovery state algorithm monitors checkpoints and logged messages as they arrive on stable storage. Each time a new state interval σ_x^i of some process P_x becomes stable, the algorithm attempts to form a new maximum recovery state in which the state of process P_x is advanced to state interval σ_x^i. It does so by including any state intervals from other processes that are necessary to make this new global system state consistent. Instead of recomputing the entire current recovery state on each execution, it only examines the changed information incrementally. The check for consistency is performed by directly applying the definition of system state consistency given in Equation 3.6. The algorithm succeeds if all the included state intervals are stable, otherwise no new current recovery state is possible.

An outline of the basic recovery state algorithm is shown below. Let $\mathbf{R} = [r_{ij}]$ be the current recovery state of the system. When state interval σ_x^i of process P_x becomes stable, the following steps are taken by the algorithm:

1. If $\sigma_x^i \leq r_{xx}$, then the current recovery state is already in advance of state interval of process P_x: exit the algorithm.

2. Create a new dependency matrix $\mathbf{D} = [d_{ij}]$ from $\mathbf{R}$ with row x replaced by the dependency vector $\mathbf{d}_x$ for state interval σ_x^i of process P_x.

3. Loop until $\mathbf{D}$ becomes consistent, that is loop while there exist some i and j for which $d_{yz} > d_{zz}$, which shows that process P_y depends on a state interval of process P_z greater than the current state interval index of P_z in $\mathbf{D}$.

   Find a stable state interval $\sigma_z^i > d_{yz}$ of process P_z. If state interval d_{yz} is stable, let σ_z^i be d_{yz}, otherwise choose some later state interval if one exists:

   (a) If no such state interval exists, then exit the algorithm but remember to recheck this condition later.

   (b) Otherwise, replace row x of $\mathbf{D}$ with the dependency vector of the found state interval of process P_z.

4. The dependency matrix $\mathbf{D}$ is now consistent and recoverable, since it is composed only of stable process state intervals. Make $\mathbf{D}$ the new maximum recovery state by replacing $\mathbf{R}$.

If some state interval of a process must be rolled back to recover a consistent system state, output messages sent during re-execution may be different from those originally sent. Any processes that received such messages become orphans and must also be rolled back to a point before these messages were received. However, messages sent to the *outside world* (such as the user's display terminal) generally cannot be undone, and therefore must be delayed until the state interval from which they were sent will never need to be rolled back. At that time these messages may be committed by releasing them. If the current recovery state of the system is $\mathbf{R} = [r_{ij}]$, then any message sent by a process P_x from a state $\sigma_x^k \leq r_{xx}$ is safe to commit to the outside world.

While the system is operating, checkpoints and logged messages accumulate on stable storage in case they are needed for some future recovery. This data may be removed from stable storage whenever it does not belong to the current global recovery state $\mathbf{R} = [r_{ij}]$. For each process P_x, if σ_x^i is the state interval index of the latest checkpoint for the state interval r_{xx}, then any checkpoint for P_x with state interval index $\sigma_x^j < \sigma_x^i$ and any message that begins a state interval in P_x with index $\sigma_x^k < \sigma_x^i$ may be released from stable storage.

Using message semantics

As we have seen in the previous sections, optimistic message logging schemes with asynchronous checkpointing may suffer from multiple roll-backs due to orphan processes, and thus incur a longer recovery period. Leong and Agrawal observed [45] that certain messages involved in a distributed computation are less relevant from the recovery viewpoint. Their existence does not change the meaning and result of the overall computation. By identifying such irrelevant messages and leaving them out of consideration when the recovery actions are planned, the message dependency is curtailed. Consequently, the amount of rollback and re-computation after an error is reduced.

The authors define an *explicit state change* to be the change in the content and value of the object/process and an *implicit state change* to be the liability to send out response messages (as a result of the processed message). A message m sent by processor P_j to P_i is considered to be *insignificant*, if one of the following conditions holds,

1. m is an in-transit message. In-transit messages are assumed to be handled by the fault-tolerant communication protocol employed and the application programs, or
2. m causes no explicit or implicit state change, or
3. m does not cause explicit state change in P_i and generates only insignificant messages, or
4. it causes explicit state change, but the explicit state change is overwritten by some future message m'; furthermore m generates only insignificant messages and all messages delivered to P_i between m and m' are insignificant.

Any other message that is not insignificant by the above classification is *significant*.

The first condition is trivial, since the in-transit messages do not affect the recoverable state if their replay is guaranteed. The second condition characterizes "useless"

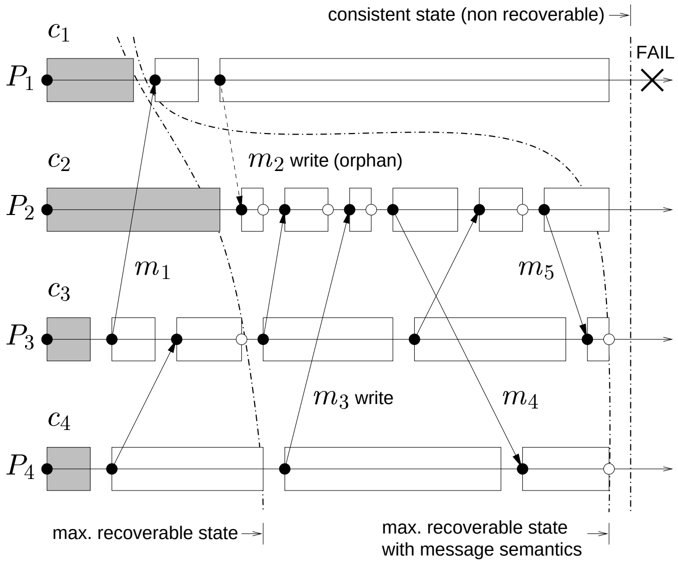

Figure 3.1: Example for the overwriting property of *write* messages

messages, which are by nature insignificant. The third condition captures the semantics of *read* messages and messages with similar property. The last condition describes the overwriting property of *write* messages. The common feature of all conditions that P_i cannot send any significant messages, because those messages may introduce transitive dependency for the receivers. Thus, the dependency relation between processes can be reformed by only taking significant messages into account, and plan the recovery operation by computing the *effective dependency* created by these messages [14].

To illustrate the improvement gained by the use of message semantics, consider the scenario shown in Figure 3.1. (There is a slight change in the notation: an empty circle at the end of a state interval denotes the logging of the corresponding message.) Assume, that P_1 is slow in logging messages and m_1 has not yet been logged when P_1 fails. The maximum recoverable state using the traditional approach is almost at the recovery line, due to the orphan message m_2. Now, if m_2 would be a significant message, then P_2 would become an orphan and transitively, m_4 and m_5 become orphan messages as well. However, note that message m_2 writes into a shared object at processor P_2 and this state change is later *overwritten* by message m_3. Therefore, m_2 is insignificant, with respect to the overall computation, and its orphanhood does not matter. Leaving this message out of consideration, the maximum recoverable state improves significantly.

Figure 3.2 is another example, this time demonstrating the semantics of the *read* messages. The traditional approach would require a complete rollback to the recovery line under the circumstances shown on the figure. Still, again we can observe that message m_2 is insignificant since it does not cause an explicit state change in processor $P_2$and generates only insignificant messages (the reply message m_4 sent back to P_1). Discarding m_2 once again leads to a better recoverable state.

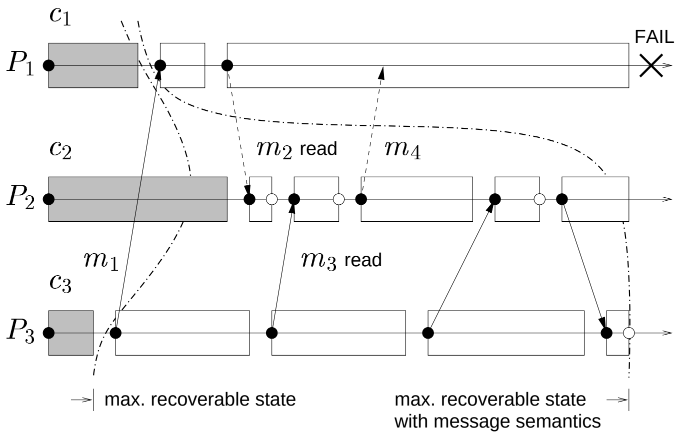

Figure 3.2: Example for *read* message semantics

3.4 Causal message logging

The message logging protocols presented so far demonstrate a pessimistic versus optimistic trade-off. Alvisi et al. were the first to show that a trade-off like this is not inherent to the problem [3, 4, 5]. *Causal protocols* have the desirable properties of both pessimistic and optimistic protocols: they prevent the creation of orphans and allow simple failure recovery, yet they do not block applications in normal operation. The message overhead of causal protocols is also small and may not grow beyond a certain limit.

The authors named their approach as a *family-based logging* (FBL) protocol. The underlying idea of FBL logging is that a message is partially logged by the time is is sent and it must be fully logged by the time it is *relevant*. To understand the meaning of relevancy in this context, consider a message m_i sent from process P_s to process P_r. The relevance of the message m_i is not determined by the state of process P_r, nor in general by the state of its parent P_s. (A process P_s is by definition a *parent* of a process P_r, and P_r is a *child* of P_s, if P_r delivers a message sent by P_s.) Rather, m_i becomes relevant when some child of P_r delivers a message that depends on m_i. If m_i is not fully logged by this time, then the message determinant $\langle m_i \rangle$ may be lost due a process failure and the messages that depend on m_i become orphans. Thus, the children of P_r determine the moment when any message delivered by P_r must be fully logged, and so it is natural to assign the responsibility of logging the receive sequence number RSN_r^i to the children processes. For this purpose, P_r will piggyback RSN_r^i on every message it sends after delivering m_i, and the recipients can log the piggybacked receive sequence number. Once P_r receives an acknowledgement from a process P_q for any message that depends on m_i, then it known that P_q has logged RSN_r^i and can cease further piggybacking of this receive sequence number.

When a process P_r fails, the messages logged at the parents and the receive sequence numbers logged at the children of P_r must be recombined. Therefore, the children of P_r are required to log more than just receive sequence numbers: there must be some means of matching the RSNs to the corresponding messages. Instead of the pure RSN_r^i

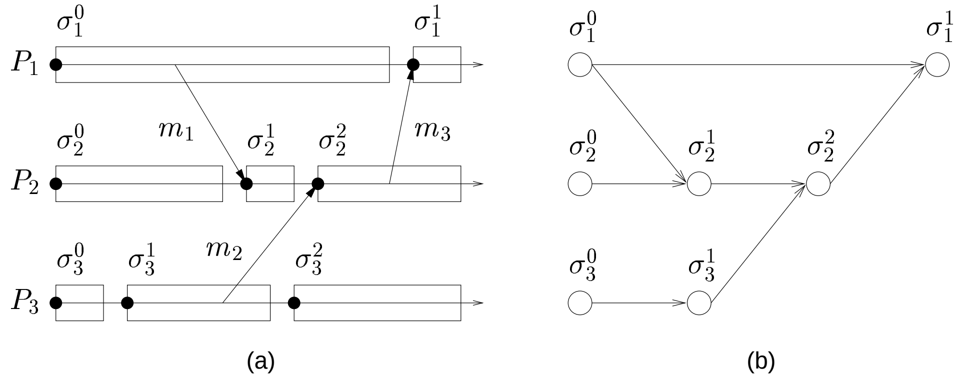

Figure 3.3: Antecedence graph: (a) Example execution, (b) Related antecedence graph, AG(σ_1^1)

the piggybacked information can be composed of triplets in the form $\langle P_s^i, SSN_s^i, RSN_r^i\rangle$. Each triplet matches and amends a unique message partially logged at P_s with send sequence number SSN_s^i; and so when P_r fails, the message determinants can be obtained by combining the matching information logged at the parents and the children of P_r.

FBL is an optimal message logging protocol in the sense that it does not send any additional messages over those needed to mask transient link failures. However, it can tolerate only one failed process at a time. On the other end of the spectrum with respect to the number of failures f that can occur before one of the processes recovers is the protocol named *Manetho* [28, 29, 30]. Manetho is an optimal message logging protocol designed for $f = n$. Each process executing the protocol maintains an *antecedence graph* (AG), which records the causal relationship between all the message delivery events of an execution. Figure 3.3 shows an example execution and the corresponding antecedence graph maintained at process P_1 in its state interval σ_1^1. For a state interval created by the receipt of a message m_i, the respective AG node has two incoming edges: one from the node representing the previous state interval σ_r^j in the recipient process P_r and one from the node representing the state interval σ_s^k from which the message was sent. A node contains the message determinant $\langle m_i\rangle$, but it does not contain a copy of the message contents.

Each process has an up-to-date copy of the AG of its current state interval in volatile memory. There is also a partial message log that contains the data and identifier of each message the process sends. The AG of the current state interval is (conceptually) piggybacked on each outgoing message. The receipt of the message starts a new state interval in the recipient process P_r, and the local AG(σ_r^i) of the previous state interval is merged with the AG(σ_s^j) piggybacked on the message. The sender need not include the complete AG of its current state interval in each message, instead, incremental piggybacking may be used, since AG(σ_r^i) is by definition a proper subgraph of AG(σ_r^{i+1}). Each process P_s that communicates with P_r includes with the message the maximum state interval index j such that the node σ_r^j is in the AG of P_s. When P_s is in the state σ_s^k and subsequent messages are sent to P_r, it needs to append only AG(σ_s^k) − AG(σ_s^j).

Periodically, each process records a local state checkpoint to stable storage. The checkpoints are taken in an uncoordinated way. At the same time of storing the checkpoint, the process also writes the volatile log and the AG of its current state interval to stable storage. Between two checkpoints, each process occasionally saves a copy of its current state interval to stable storage. The subgraph of the AG that is already stable need not be included in the information piggybacked on messages. Before sending a message to the environment, a process must again write the AG of its current state interval to stable storage (output commit). No coordination with other processes is necessary.

A recovering process P_x restores its state, message log, incarnation number and AG from stable storage. Then it initiates a fairly complex procedure to reconstruct the antecedence graph of its last stable (logged at another processes) state interval. It will also obtain a table containing the incarnation numbers of the other processes in the computation. After determining the most recent stable state interval σ_x^m, P_x proceeds to recreate the pre-failure execution up to σ_x^m by requesting messages from the logs of their senders and re-executing internal events, as necessary. The process does not send any application messages, but it stores the messages it would have sent in its volatile message log.

Later, Alvisi and Marzullo generalized the FBL protocols to handle multiple concurrent failures [5] and Elnozahy modified the recovery phase to avoid the blocking of live processes[26]. To handle f concurrent failures with $f > 1$, the information piggybacked on messages is managed slightly differently than in the original protocol. Each process P_s maintains a set DS_s containing the message determinant of all the non-stable delivery events reflected in its state. It will piggyback on every outgoing message m_i to a process P_r^i the elements of DS_s that have not been seen by P_r^i. However, when P_s delivers a new message m_j, it does not add DS_s only $\langle m_j \rangle$, but also all the non-stable determinants that were piggybacked onto m_j. As in the case of $f = 1$, P_s removes a determinant from DS_s as soon as it notices that the determinant has become stable.

FBL protocols for $f > 1$ distribute the implementation of the recovery procedure associated with a process to a larger degree than in the single failure case. That is, a process must not only log the determinants of all the messages it delivers and the content of all the messages it sends, but it may also be required to log the determinants of messages delivered by its ancestors up to degree f. Yet, FBL is a much simpler protocol than Manetho, and even for the case of $f = n$ it is more efficient in the information it piggybacks.

Chiu and Young [16, 72] describe a different approach for causal message logging. In their scheme, the maintenance of message order information is done along a *logical ring* of processes. Message processing order of a process is available at all the other processes in the ring. Transmission of application data messages initiates the circulation of special *order messages* along the logical ring. The size of order messages is small. Redundant transmission of order information is avoided, thereby reducing the communication overhead incurred during failure-free operation. Updating of the order information and reducing its amount by eliminating irrelevant data are simple operations. The approach does not require the message processing order information to be written to stable storage, since it is kept in the volatile memory of each process in the logical ring. The authors show, that the system can recover correctly as long as there is at least one surviving process. When failures occur, a live process must roll back only

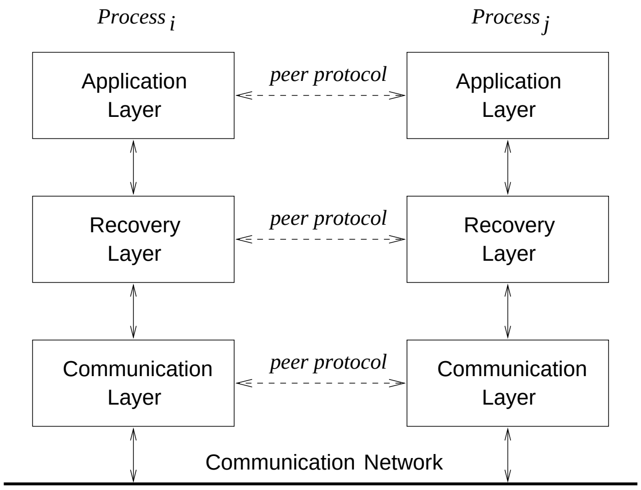

Figure 3.4: Propored layers of the software architecture

if some preceding order information is totally lost (i.e. it could not be disseminated along the logical ring quickly enough with respect to node failures), which is relatively unlikely in a high-performance interconnection network.

3.5 Software architecture for message logging

Chiu and Young propose a layered software architecture to facilitate the implementation of transparent message logging and recovery [16, 72]. Although the scheme is somewhat tailored to their recovery algorithm, it is general enough to be a good design example. The node-resident part of the operating system in the proposed scheme consists of three hierarchically arranged layers: application layer, recovery layer, and communication layer, as illustrated in Figure 3.4. The *application layer* is an interface to the user processes. Processes cooperate with each other using the *communication layer*, which implements end-to-end and broadcast communication using various protocols. However, each incoming and outgoing message must pass first through the *recovery layer*, which provides fault tolerance for the applications in a transparent manner.

While the communication layer will receive the incoming messages, the recovery layer is in charge of delivering them to the user processes. This enables the recovery layer to limit the number of processes required to roll back as a result of node failures by maintaining the processing order of messages. It can also delay the delivery of messages if necessary to avoid inconsistency (e.g. of a checkpointed global state). Another task fulfilled by the recovery layer is keeping track of process dependencies, and using the generated dependency information in the checkpointing and recovery actions.

Figure 3.5 shows the internal structure of the recovery layer in the implementation of Chiu and Young [16, 72]. For each process, there is a set of *input lists*, one per every other process in the system. An incoming data message received from another process is first stored in the input list corresponding to the sender. The *merge* function combines the contents of the various input lists by retrieving data from the head of the

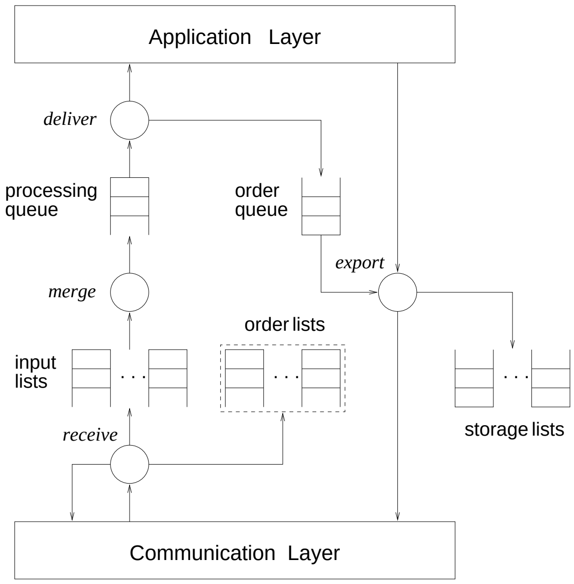

Figure 3.5: Internal structure of the recovery layer

lists and placing it into the *processing queue* in an arbitrary order. Messages sent from the same process, however, reside in the same input list and therefore are retrieved and delivered in FIFO order. Thus, the merge function expresses the same nondeterministic property of the underlying communication network. When a process makes a request for an incoming data message, the recovery layer responds by delivering the message at the head of the processing queue. At the same time, a partial logging information is appended to the tail of the *order queue*. Therefore, the order of partially logged messages in the order queue reflects the sequence in which the user process obtains the corresponding data messages. Once the data message was delivered for processing, it is no longer needed to be maintained by the recovery layer.

Data messages sent by the current process to others are first placed in a set of *storage lists*, one corresponding to each recipient process. The storage lists are utilized for retransmitting the logged messages during the recovery operation [68, 35, 66]. The OSN_s^i sequence number is piggybacked on the message to identify the order message that the data message directly depends on. Then, the *export* function passes the data message to the communication layer for transmission. At the same time, the corresponding partial logging information is assembled into an *order message packet* and passed on to the communication layer for circulation around a logical ring. (Recall, that the contents of the order queue maintain the processing order of incoming messages since the last transmission of the data message.) If the order queue is empty, then no order message need to be sent out. When a process receives an order message, the corresponding *order list* in the *order table* is updated accordingly.

Chapter 4

Comparison of recovery methods

The previous sections introduced the general techniques of backward error recovery. In this section we discuss the major advantages and disadvantages of these techniques in order to facilitate the choice for a certain application. They can be evaluated according to many different criteria (Table 4.1 summarizes of them), we examine three such criteria in particular: (1) the performance drop due to the execution of recovery protocol during the failure-free (or normal) system operation, (2) the complexity and computation-intensity of the protocol when the rollback procedure was invoked upon the detection of an error, and (3) the latency involved by the commit operation required before communicating with the environment.

Note, that Criteria 1 and 2 represent a trade-off: the more overhead is incurred by a protocol during the failure-free run, the less work is needed to restore a consistent global system state; and vice versa [33]. The pessimistic and optimistic approaches are typical examples for such a trade-off, with optimistic algorithms tending to "neglect" their duties hoping disaster will not strike frequently, yet when it strikes, they have much more things to do and result in a slower recovery than pessimistic methods. Another trade-off exists for Criterion 3: message logging with independent checkpointing requires some extra space on stable storage, but in return it offers a much shorter commit latency than consistent checkpointing. As a consequence, the best solution in general will be a combination of the above techniques tailored to the individual characteristics of the chosen application.

Failure-free performance

Optimistic message logging is a method specially developed with minimizing the added time overhead of rollback recovery during the normal operation. (It would follow then, that asynchronous logging has the best failure-free performance when the failure rate is low. Surprisingly, according to the measurement results of Elnozahy and Zwaenepoel [31] it is not so! The results are summarized in a later section.) All it has to do is partially log the messages in local memory, and eventually record the full entries of the volatile log to stable storage in one burst (which is supposed to be more efficient than a series of single write operations) in the background. The message overhead of optimistic logging is caused by the data (i.e. the recipient sequence number) circulated in order to complete the partial message log entries. However, the space overhead of

Properties	Consistent checkpointing	Pessimistic message logging	Optimistic message logging	Causal message logging
Main overhead during	failure-free	failure-free	recovery	both
Cause of overhead	checkpoint coordination	synchronization of writing log	rolling back orphans	dependency tracking
Time overhead	blocking for coordination	blocking for synchronization	consistent state recovery	execution restart, message replay
Space overhead	no	no	checkpoints, volatile log	dependency information
Message overhead	coordination	acknowledgement	logging data	dependency data
Blocks processes	checkpointing	logging	no	no
Output commit	global	no	global	local
Stored checkpoints	one	one	multiple	one
Processes rolled back	dependants / all	failed	failed + orphans	failed
Logs messages	no	before sending	independently	on relevance
Dependency information	partial (not stored)	no	partial (stored)	full (stored)
Number of failures	arbitrary	arbitrary	$1 \leq f \leq n$	$1 \leq f \leq n$

Table 4.1: Comparison of rollback recovery techniques

this approach is the most excessive there is: multiple independent checkpoints must be stored, since there is possibility for multiple roll-backs, and message logs are stored both on stable storage and in process memory. A garbage collection algorithm can (and usually must) be used to reclaim old and outdated checkpoints and log entries, thus limiting the resource requirements, but creating additional time overhead.

The major part of the time overhead by consistent checkpointing and pessimistic message logging is generated during the failure-free execution. The performance drop manifests itself in the form of blocking the running applications, which is necessary to avoid the potential for inconsistency introduced by process interactions. Causal message logging does not require such blocking, but it too has an added time overhead during normal operation as compared with optimistic logging, related to the maintenance of dependency information used in recovery. Consistent checkpointing employs a time consuming separate session using special messages for coordination among processes; but on the other hand it has the smallest space requirements, since it stores only a single set of consistent local checkpoints. Pessimistic message logging distributes the time overhead more evenly and has less excess messages (only used for acknowledgement of the receipt, which are inherent to many communication protocols anyway), but a process is blocked more frequently and the message log occupies additional space on stable storage. Causal logging in general does not involve redundant messages, then again the size of the transmitted messages is increased by the piggybacked dependency information.

Recovery protocol complexity

The "brute force" approaches: consistent checkpointing and pessimistic message logging are quite simple with regard to their operation during the recovery phase. Using consistent checkpointing the system can always be restarted in a valid global state by loading the set of local checkpoints from stable storage. Pessimistic message logging first restores a past system state (which is not necessarily consistent) using the recorded individual checkpoints, then re-executes the lost work up to the time of failure and replays the messages in the stable log; thus achieving a valid global system state. Causal logging also has all of the *relevant* messages fully logged on stable storage, however, it must recreate the dependency information it constantly maintains, adding extra time overhead. Optimistic logging has the worst performance and the most elaborate protocol in the recovery phase. Its many duties include: determining the maximum recoverable system state, finding out which processes became orphaned, rolling back the failed and orphan processes, and restoring the maximum recoverable system state in a complex procedure.

Sender- versus receiver-based logging

Sender-based message logging has many advantages over receiver-based logging methods. First and foremost, it can be implemented without relying on stable storage or other reliable media to hold the log entries. Stable storage has strict requirements which are hard and costly to realize, and usually it has a single and slow interface creating a bottleneck in a multiprocess environment. Sender-based logging is single-fault tolerant in its most basic form, but can be modified to handle an arbitrary number of failures on the expense of redundant data stored at nodes and transmitted over the communication network. (This also applies to the FBL family of causal logging protocols.) Its also outperforms receiver-based logging during the failure-free run. because the message can be logged after it is transmitted to the network, logging a message at the sender is not in the critical path of interprocess communication. Also, logging at the sender reduces the amount of data to be stored for applications involving broadcast (one-to-many) communication. In receiver-based logging the *same* data is logged at each recipient, while only a single copy is logged at the sender for sender-based methods.

Output interactions

Any output from the application process to the environment can be transmitted in a committable (i.e. consistent) system state that will surely not be rolled back. To achieve such a global state, all recovery techniques (except pessimistic message logging, which always maintains a committable state) must employ a commit protocol, which involves certain amount of delay until the output can be released to the outside world. This latency is in general much lower for the logging-based approaches than for coordinated checkpointing. Next to pessimistic logging, causal methods offer the lowest latency, because their dependency tracking algorithm allows each process to commit locally with a single stable storage I/O, thus in constant time. Optimistic message logging requires a multi-host protocol to commit the output, which entails exchanging several messages and performing several stable storage I/O operations.

Measurement results

Elnozahy and Zwaenepoel conducted several measurements [27, 31] to compare the performance of various recovery protocols on the same hardware and software platform. Their most surprising results show that causal and optimistic message logging perform *worse* than consistent checkpointing, even during normal system operation (the authors did not even implement any pessimistic logging algorithms because of their blocking nature). The cost of writing the message logs and managing the recovery information on stable storage outweigh the cost of coordinating the checkpoints during the failure-free run and the overhead of contention on the stable storage server during the global checkpoint. This result is reinforced by the fact that no garbage collection was incorporated in any of the implemented message logging protocols, therefore the measurements underestimate their actual overhead. Moreover, the performance drop due to the coordination of checkpoints was strikingly low: less than 1 percent of the total running time if at least 5 minutes passed between subsequent invocations of the protocol. With short checkpoint intervals, consistent checkpointing performed even better than independent checkpointing *without* message logging, in an application that has a tight synchronization pattern.

However, the result does not imply that consistent checkpointing is always the method of choice. The measurements show that the message logging protocols commit output much faster than consistent checkpointing. As a consequence, the authors proposed a new protocol that combines consistent checkpointing with causal message logging. This composition maintains the low output latency of message logging, but in addition it allows some important optimizations improving the failure-free performance over the traditional design [31].

Elnozahy and Zwaenepoel also highlighted some further issues related to the implementation. First, previous logging algorithms write both the message data and the dependency information to stable storage during output commit. According to the measurements, it suffices that only the dependency tracking part of the log is recorded on reliable media, resulting in reduced output latency. Second, the use of copy-on-write (see Section 5.1 for a description) in the logging process avoids blocking of user applications while storing the checkpoints. The other alternative, pre-copying introduced always more or equal overhead to that of copy-on-write, making the latter the preferred choice [27]. Finally, the results affirm that sender-based message logging has a failure-free performance superior to receiver-based logging. While qualitative arguments for sender-based logging have been made before, this was the first quantitative comparison that supports these arguments.

Application- and system-specific issues

Beside the described overhead and requirements of the algorithms, the set of applications to be run and the available system facilities also largely influence the decision upon which class of recovery methods is best suited for the purpose. Some of the determining characteristics of applications are the communication-intensity, programming paradigm, memory requirements, real-time constraints. Important facilities are the computation capacity, average load, provided communication protocols, parallel architecture, clock synchronization.

Furthermore the cost of the desired fault-tolerance properties: the overhead during normal failure-free operation (checkpointing, logging) and after an error is detected (rollback) should be taken into account. Hence the fault probability and failure semantics (behavior of the failed unit) affect the performance as well. Would a low fault probability be expected, emphasis should be on minimizing the failure-free overhead to optimize the throughput of the system; if the fault-probability is higher, the recovery overhead also becomes important. The persistence of the faults assigns the restoration strategy directly: transient faults require only rollback, where handling of permanent faults may need additional reconfiguration of the system.

The overhead associated with the checkpointing is very dependent on the checkpoint frequency. A trade-off should be made between high checkpointing frequency (which implies small loss of progress after a fault) or infrequent checkpointing (that causes further setback in time, with less overhead to store the checkpoints but possibly more memory to log the messages). Some general optimizations, as forcing a minimum number of nodes to roll back [41, 28, 40, 63] or minimizing the number of control messages using the interconnection topology [70, 47, 48] or other properties as a damage assessment phase [60] can be used in several classes of techniques. The semi-automatic checkpointing techniques have minimal overhead, but require programmer-involvement. Message logging techniques are less interesting for communication-intensive applications due to the higher (continuous) time, communication, and storage overhead associated with the logging of the messages, but are better suited otherwise, because blocking is avoided as processes individually take local checkpoints (one at a time), allocating smaller communication bandwidth for checkpointing. Consistent checkpointing techniques are suited for both communication-intensive or -sparse applications, as they have virtually no overhead during the failure-free operation; on the other hand they suspend or block the applications periodically while the global system or the interacting set of processes are checkpointed.

Recovery in checkpointing is more straightforward as no messages need to be replayed (i.e. piecewise-determinism is not a requirement), but in general more processes are involved in the rollback. In message logging techniques recovery is more complex due to dependencies tracking. A larger rollback complexity is still is tolerable, since failures in practical systems are seldom. Hybrid techniques merge advantages of the previous two groups and optimize the failure-free overhead.

Chapter 5

Massively Parallel Systems

In this section, we look at the special requirements that are necessary to develop an efficient backward error recovery scheme in massively parallel (MPP) systems. First of all, one might wonder if it is really necessary to have fault-tolerance in MPP systems at all. This argument is easily proven by the following example [24]: suppose a processing element of a 1024-node multiprocessor system has a mean time to failure (MTTF) of 10 years, then the overall MTTF of the system will be only 85.5 hours (i.e. some long lasting jobs will theoretically be impossible to compute, as occurring failures will almost surely keep forbidding its completion even if the job is restarted when an error is detected). This stresses the need for fault-tolerance in a (massively) parallel systems.

MPP systems are employed mainly in two areas: real-time applications [32] and the solution of computation-intensive problems. Backward error recovery is better suited for the latter class of applications, because they do not have stringent real-time constraints [10]. In number-crunching tasks it is important that the full available computation power should be used for solving the problem, the time overhead of fault-tolerance during normal operation should be minimized. The optimization of the system throughput can only be done at the cost of a larger rollback latency (more complex algorithm, further setback in time, etc.).

Massively parallel computers aim to provide easy expandability for growing systems; this feature is called *scalability*. The recovery component should be as scalable as possible to preserve the scalable property of the whole system. This calls the attention on the following aspects:

- Distributed algorithms are necessary, avoiding the use of central resources as much as possible (if a central device is inevitable, bottlenecks and single points of failures should be avoided, e.g. by using backup servers).

- The message overhead induced by fault tolerance must not grow exponentially or polinomially with the size of the system (i.e. it must be $O(n \log n)$, $O(n)$, or even less).

- The techniques should be as hardware-independent as possible: e.g. if the user decides to step from a single disc over to a distributed storage medium, the recovery algorithm should be easily adapted to the new hardware environment.

- If the machine is divided in separate user- or application-partitions, the recovery actions must not expanse over the boundaries of the corresponding partition, and should be independent from the others.

- Special care should be taken that the checkpoints are stored on stable storage in a way that they are still accessible after a failure occurred. Especially in the case of reconfiguration (that is, when the application is restarted on other hardware), this can be in another partition of the massively parallel system. So this issue is twofold: on one hand the destination of checkpointed data should be close to avoid communication overhead and delay; on the other the checkpoint should be stored far enough not to be affected by a failure.

The hardware can also provide special fault-tolerant features which lower the additional load associated with recovery. Fault tolerance must be integrated in the hardware from the design on. This includes redundancy, design for testability, error detection and correction, control and diagnostic tools, etc. Some fault-tolerant features are too costly to be written in software, they must be part of the hardware as well.

Several recent improvements in storage techniques may also be a good step towards fault-tolerant mass storage at high I/O-bandwidth. For instance disc stripping treats a group of discs as one storage unit, whereby data blocks are divided into sub-blocks in their separate discs [18]. Since the discs transfer their data in parallel, the time to transfer data blocks from and into main memory is drastically decreased, especially when the discs are synchronized. This idea led to the RAID (redundant array of inexpensive discs) systems, in which providing stable storage has a good price/performance ratio. Some possibilities include mirroring or shadowing, block interleaved parity, etc. Mass Storage Systems (MSS) come available and must be supported. Their typical approach is that one "logical file", with a global name is distributed over several connected MSS-nodes, according to the way that the data is mapped onto the processors for the computations.

The operating system or system software must provide primitives to facilitate the implementation of backward error recovery algorithms. They range from providing a dirty bit to allow incremental checkpointing over all kinds of reliable communication protocols to clear memory- management, etc.

5.1 Efficient checkpointing methods

In this section we describe some methods that improve the efficiency of checkpointing by avoiding certain bottlenecks in its performance. One of the major causes of overhead in checkpointing is writing the checkpoint data to stable storage. Proposed solutions to reduce this effect:

Incremental checkpointing

Incremental checkpointing is a technique to reduce the amount of information included in checkpoints [53]. This will speed up the recording of a local process state and reduces the storage requirements. Since less information is written in a shorter time interval, the probability of error occurrences during checkpointing is smaller. The underlying idea of the incremental approach is that only a part of the process state needs to be recorded if it is feasible to assume that only that part of the state variables has changed since the last checkpoint was written.

A straightforward implementation of incremental checkpointing relies on virtual memory. The memory management unit (MMU) maintains one bit, called the *dirty bit* for each page of the memory. This bit is set to '1' when the page is accessed, and it can be cleared (set to '0') by the checkpointing routine on successful recording of the global system state. Should the MMU not provide such a 'dirty bit', the same information can be generated by switching the protection of memory pages from *read-write* to *read-only*, and catching the resulting page fault exceptions. Before starting the user application, the entire initial local state is stored. Then, upon each invocation the checkpointing routine writes the modified memory pages (with the 'dirty bit' set) to the stable storage. If consistent checkpointing is employed, then the changed parts of the recorded global system state can be simply overwritten. In case of individual checkpointing changes are added to supplement the previously recorded information. However, the state restoration procedure will have to "backtrack" the stored checkpoints up to the point they form a global system state, i.e. recovery will be slower.

The advantages of the incremental checkpointing approach are obvious. Nevertheless, it has also several requirements and disadvantages that limit its applicability:

- It requires low-level hardware or compiler support for user-transparent operation. Hardware support means an MMU which is capable of virtual memory management by paging and 'dirty bit' handling [23]. Without explicit hardware support a compiler-assisted implementation can provide similar information on the program-object level [46]. During compilation some extra code is inserted to keep track of the creation, use and deallocation of program objects such as variables, dynamically allocated memory referenced by pointers, etc. Then, the checkpointing routine can obtain the "dirty bit" respective to program objects, and include only the changed data in the checkpoints. If neither hardware, nor such software support does exist, implementing the incremental state saving technique is the responsibility of the application programmer.

- Since it might be necessary to make a "backtrack" and fetch several copies of an incremental checkpoint in recreating an earlier state, multiple accesses to the—usually slow–stable storage may be necessary, resulting in a considerable delay.

Furthermore, the checkpointing process can be made more efficient if the user applications can continue to execute while their checkpoint is being stored. However, if the application modifies any of the pages belonging to the checkpoint before it was written to stable storage, the stored global state not represent the state the system had at any single point of time. In the following sections we present two well-known solutions to this problem.

Copy-on-write

The first approach to continuous application execution during checkpointing uses *copy-on-write* memory protection, supported by the memory management hardware. Before taking the incremental checkpoint, the pages to be written to the stable storage are write-protected. After each page has been successfully written, the checkpoint routine removes the write protection from them. Should a process modify one of these pages while it is still under protection, an exception of memory protection violation is generated. In this case, the exception handler copies the respective page to a temporary location, removes the write protection from the original page and allows the process to continue. The process is free to alter the original page, but it cannot access the temporary copy, which is used by the checkpoint routine to write the original contents of the page to stable storage. Then, the temporary copy is deallocated.

If the amount of memory is insufficient to hold the temporary copy, the process that attempted to modify the protected pages is suspended. It remains blocked until there is enough free memory available to allocate a page to handle the copy-on-write exception. Examples of this technique were implemented with [27] and without [49] incremental checkpointing.

Pre-copying

Pre-copying uses a different approach to maintain page consistency with the checkpointing of applications running in background. In this approach, if the number of modified pages is below a certain threshold, the pages belonging to the checkpoint are copied at once to a separate area in memory and are then written from there without interrupting the execution of application processes. Otherwise, a "pre-copying" pass is made over the application address space, saving the modified pages to stable storage. The user process continues to run and can freely write any of these and other pages during the pre-copying pass. Once the pages were written to stable storage, the number of newly modified pages in the address space is reexamined. If it is still above the threshold, additional pre-copying passes are performed. There is a defined maximum number of passes, and if that has been exceeded, the application is suspended while the remaining pages are flushed directly from its address space to stable storage.

The pre-copying method is the less complicated of the two solutions as it avoids the handling of page protection exceptions. On the other hand, it will usually need to write some pages to stable storage more than once, if they are modified during the pre-copying pass. Moreover, pre-copying may need to block the user processes in order to complete the checkpoint, if additional pages of the address space are modified too quickly by the processes during the repeated pre-copying passes. Another consequence of the pre-copying strategy that the size of a stored global state will in general be larger than it would be if incremental copy-on-write checkpointing was employed in the system.

5.2 Diskless checkpointing

Diskless checkpointing methods do not perform *writing to disk*. Instead, they assume that no more than m processors fail simultaneously in a parallel or distributed system, and describe how to recover from such failures. Each algorithm works in the case m equals to one, and some of them can be modified to handle larger values of m. The

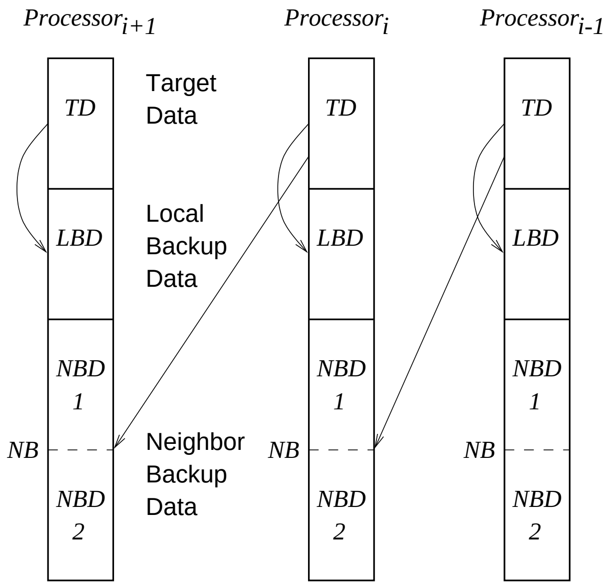

Figure 5.1: Mirror checkpointing

underlying idea is that it is possible to tolerate m simultaneous processor failures with employing one to m redundant processors (depending on the algorithm).

Mirror Checkpointing

The basic operation of mirror checkpointing is to replicate the state of every processor to the memory of its neighbor [17]. The memory of each PE is partitioned into four equal-sized sections (see Figure 5.1). The *Target Data* (TD) section contains the actual data currently in use by the user application. The *Local Backup Data* (LBD) section maintains the last valid checkpointed local state of the application. The *Neighbor First Backup Data* (NBD1) and *Neighbor Second Backup Data* (NBD2) store the two most recent checkpointed states of the neighboring processor. A pointer is used to differentiate between these two sections: it points to the most recent successfully written local state of the neighbor, called the *Neighbor Checkpoint* (NC) section, while the other is called the *Neighbor Backup* (NB) section. This double buffering scheme enables to use *shadow updating* to tolerate error occurrences during checkpointing [19, 50].

When the *checkpoint* routine is invoked, each processor first sends the contents of its TD section to the neighboring PE to be stored in the NB section, then it dumps the TD section to its LBD section as well. If the copy operation at the neighbor succeeds, the updated NB section becomes the NC section by setting the NBD pointer to point on it. When a node failure occurs during the checkpointing procedure, there are two cases. If the PE fails before the first copy operation to the NBD area is completed, the fault-free processors roll back to their checkpointed state by copying their LBD sections to the TD sections. The failed processor copies the contents of the NC area received from its neighbor. (Note, that if the first copy operation have not succeeded, then neither LBD sections nor the NBD pointer have been updated and they point to the previous valid

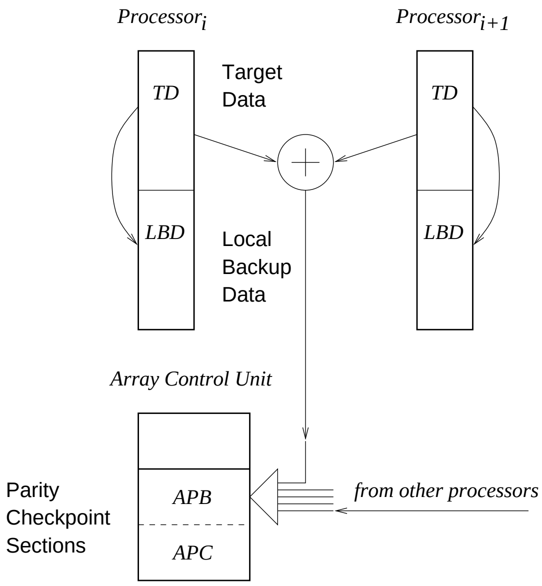

Figure 5.2: Parity checkpointing

checkpointed state.) The same steps are performed if the error occurs during normal execution and the *restore* routine is called. However, if the PE fails after the first copy operation but before the LBD section was successfully written, the rollback procedure is slightly different: the failed processor copies first the NC section obtained from the neighbor to its TD section, then every processor copies their TD sections to the LBD sections.

Parity Checkpointing

Parity checkpointing uses $n+1$ parity instead of a full replica to maintain the checkpointed global state. $n+1$ parity was first proposed by Plank [54, 56] as a way to perform diskless checkpointing, but his algorithm is non-incremental and needs each processor to maintain two in-memory copies of local checkpoints. This problem was alleviated by Chiueh and Deng [17]. In their scheme, the local memory space of each processor is divided into only two parts: the *Target Data* (TD) and the *Local Backup Data* (LBD) sections, holding the current and the last successfully checpointed local state, respectively. There are two *Array Parity* (AP) sections stored in the memory or other storage media of a reliable central server, such as the *host* computer in MIMD systems, or the *array control unit* (ACU) in SIMD systems. The section that holds the parity of the last valid stored global state is referred to as APC, the other as APB; again with the current one indicated by the value of a pointer.

When the checkpoint routine is invoked, each processor performs a collective XOR operation on their TD sections in a binary-tree manner (explained more in detail later). After these XOR operations the PEs place the resulting array checkpoint parity to the central APB section (shown in Figure 5.2). Then they copy their TD sections to the LBD sections. When a node failure occurs during normal operation and the restore

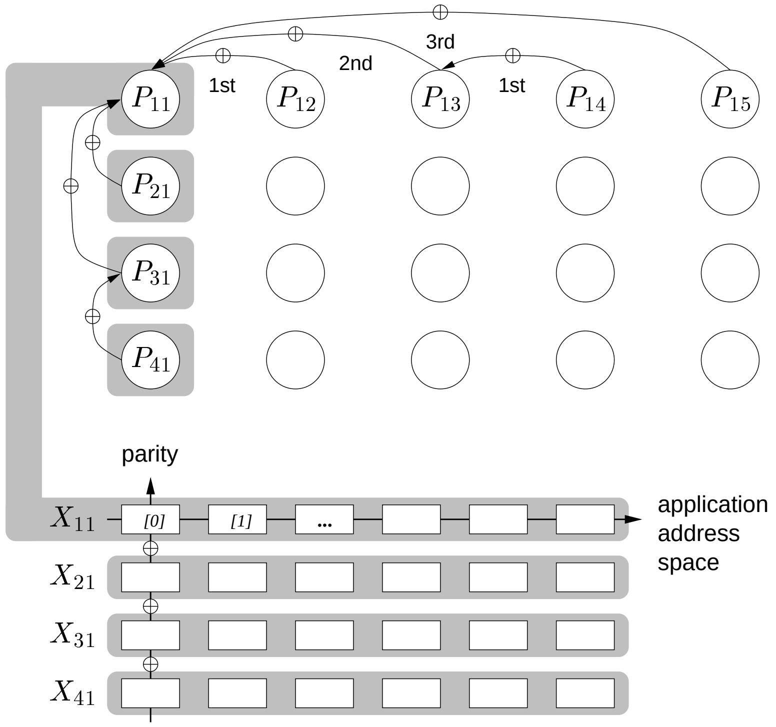

Figure 5.3: Data flow in the binary-tree global XOR operation

routine is called, every processor (including the failed one) performs a collective XOR operation based on their LBD sections. The result is then XORed again to the contents of the central APC section. This operation produces the difference between the current and previously stored parities of the LBD sections; its is zero except for the failed PE. Therefore, the failed and now recovering processor must perform an additional collective XOR using the product and its actual (and incorrect) LBD contents to cancel out the erroneous information and to retrieve the previous checkpointed state. At this point, each PE has the correct LBD section, i.e. they can use it to roll back to the last stored global state.

When an error occurs during the checkpointing procedure and it happens before the complete array parity is written to the central stable storage, the restore routine again performs the same steps as described above. Otherwise, the roles of the TD and LBD sections switch each other, that is the local TD section is restored directly using collective XOR operations and the parity obtained from the central APC section, and then is is saved to the LBD section.

An important primitive in both the parity and the partial parity checkpointing schemes is the generation of global array parity. Figure 5.3 illustrates the flow of data during this procedure. Assume n is the number of processors involved in checkpointing, and the system topology is a two-dimensional, $n = x \times y$ mesh. Instead of an $O(n)$ linear XOR operation, the global array parity can be obtained in $O(\log n)$ steps organizing the XOR operations in a binary tree structure. Each plural variable in the checkpoint data set is XORed across the PE array. Figure 5.3 shows this process in for the first

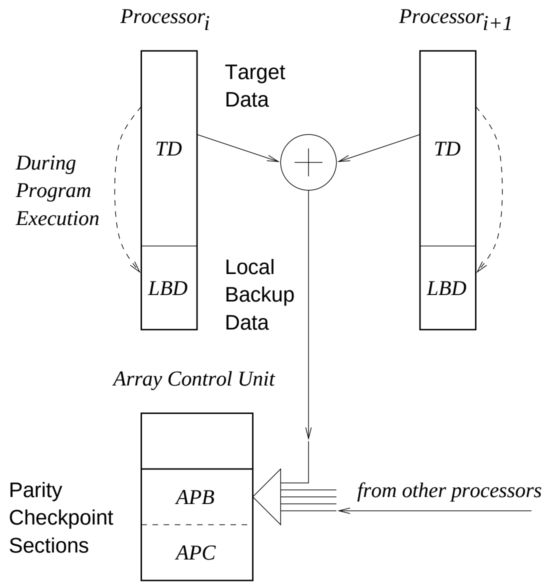

Figure 5.4: Partial parity checkpointing

column of processors in the magnified area with grey background. The resulting global array parity X_{AP} has the size of the largest local checkpoint, and is computed as

$$X_{AP} = \bigoplus_{i=0}^{x} \left(\bigoplus_{j=0}^{y} X_{ij} \right)$$

where X_{ij} denotes the application address space at processor P_{ij}. As shown on the figure, generation of the global array parity has two phases: a horizontal and a vertical phase. During the horizontal phase each row of PEs compute a linear XOR operation in parallel and collect the result at the first PE in the row. Then, during the vertical phase the PEs of the first column compute a linear XOR operation in parallel and at P_{11} the global array parity is obtained. The linear XOR operation is performed by transforming the mesh topology into a binary tree, so that the computation in the disjoint subtrees can take place simultaneously.

Partial Parity Checkpointing

Although parity checkpointing requires less local memory devoted to the sole purpose of checkpointing in each PE, there is still a one-to-one correspondence between the Target Data and Local Backup Data sections. Consequently, only the half of the memory capacity is available for user applications. Partial parity checkpointing further reduces the local memory storage overhead in each processor [55]. In this scheme, the LBD section is smaller than the TD section, and operates as a "checkpoint cache". The modified application data structures are stored in the LBD section in an incremental way. When the LBD section becomes full, a checkpoint has to be made, i.e. the system no longer has the flexibility of taking at an arbitrarily chosen point of time. The frequency of checkpointing now depends on the size of the LBD section, however, it is independent of the TD section and can be chosen according to any design criterion.

Upon the first invocation of the checkpoint routine, a binary-tree global XOR operation is performed on the TD sections of every processor, and the resulting array parity is stored in the central APC section. After this, the TD section is marked write-protected, and the occurring page protection fault exceptions are caught by the checkpoint routine. The data intended to be modified by the user process is copied to the LBD section, then the write-protection is removed and the write operation can complete. When the LBD section becomes full a checkpoint is taken, that is the number of state intervals between consecutive checkpoints cannot be greater than the size of the LBD section. If an error occurs during normal execution or in the middle of checkpointing, PEs can recover using first the *valid* entries of the LBD section to restore the TD section to its previous checkpoint state, then perform binary-tree global XOR operation and cancel out the erroneous information at the failed processor identically to the parity checkpointing approach.

Chapter 6

Conclusions

In this paper, we gave a survey on backward error recovery techniques. A brief summary of the presented techniques: semi-automatic checkpointing is a straightforward approach, but its major drawback is the lack of user-transparency, and hence programmer involvement. Coordinated checkpointing is also a relativaly simple method offering a small overhead during normal operation, but the user must suffer the loss of computation during both checkpoint synchronization and recovery. Message logging (particularly the optimistic approach) has the advantage of a non-blocking property (because independent checkpointing can be used) and a fast output commit, but a there is a constant overhead during normal computation due to the logging of messages. Hybrid techniques (like causal logging) merge advantages of both other user-transparent classes.

We described the basic concepts and main terms of these schemes. The problems related to saving and restoring the consistent global state of a system were introduced and some solutions to these problems were outlines. We discussed in detail the most general and widely used backward error recovery techniques, and presented a few particular topics and algorithms of interest. The techniques were examined from different aspects of comparison. The last part of the survey is devoted to massively parallel architectures and the means of efficient implementation of backward error recovery in this environment. We discussed how do the system characteristics determine the most suitable methods of fault tolerance. The requirements for backward error recovery techniques for massively parallel systems were examined. This work should serve as a base for our further research and for the implementation of the backward error recovery layer of the APEMille supercomputer.

Bibliography

[1] H. Al-Jaber and S. Rotenstreich. Fault tolerance of message delivery with cascading copies. In N. Rishe, S. Navathe, and D. Tal, editors, *PARBASE-90 Int. Conf. on Databases, Parallel Architectures and Their Applications*, pages 221–30, Los Alamitos, CA, USA, 1990. IEEE Comput. Soc. Press.

[2] H. Al-Jaber and S. Rotenstreich. Reliable message delivery without hard backups. *Int. Journal of Mini and Microcomputers*, 14(3):117–24, 1992.

[3] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Twenty-Third Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, pages 145–54, Los Alamitos, CA, USA, August 1993. IEEE Comput. Soc. Press.

[4] L. Alvisi and K. Marzullo. Message logging: pessimistic, optimistic, and causal. In *Proc. of the 15th Int. Conf. on Distributed Computing Systems*, pages 229–36, Los Alamitos, CA, USA, 1995. IEEE Comput. Soc. Press.

[5] L. Alvisi and K. Marzullo. Trade-offs in implementing causal message logging protocols. In *Proc. of the 15th Ann. ACM Symp. on Principles of Distributed Computing (PODC '96)*, pages 58–67, New York, USA, May 1996. ACM.

[6] T.A. Anderson and P.A. Lee. *Fault Tolerance: Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, New York, NY, USA, second revised edition, 1990. was Prentice-Hall, 1981.

[7] R. Baldoni, J. Brzezinski, J.M. Helary, A. Mostefaoui, and M. Raynal. Characterization of consistent global checkpoints in large-scale distributed systems. In *Proc. of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 314–23, Los Alamitos, CA, USA, 1995. IEEE Comput. Soc. Press.

[8] R. Baldoni, J.-M. Hélary, A. Mostefaoui, and M. Raynal. Consistent checkpointing in message passing distributed systems. Technical Report 92, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, France, May 1995.

[9] B. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems—an optimistic approach. In *Proc. 7th Symp. on Reliable Distributed Systems (SRDS-7)*, pages 3–12, Seattle, Washington, USA, 1988. IEEE Comput. Soc. Press.

[10] B. Bieker, G. Deconinck, E. Maehle, and J. Vounckx. Reconfiguration and checkpointing in massively parallel systems. In K. Echtle, D. Hammer, and D. Powell, editors, *First European Dependable Computing Conf. (EDCC-1)*, pages 353–70, Berlin, Germany, 1994. Springer-Verlag.

[11] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. Technical Report TR85-694, Cornell University, Computer Science Department, July 1985.

[12] R. V. Boppana and S. Chalasani. Fault-tolerant multicast communication for multicomputers. In *Int. Conf. on Parallel Processing*, volume 1, pages 118–125, Boca Raton, USA, August 1995. CRC Press.

[13] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proc. of the 9th ACM Symp. on Operating Systems Principles (SOSP-83)*, volume 17:5, pages 90–99, 1983.

[14] R. Bubenik and W. Zwaenepoel. Semantics of optimistic computation. In *Proc. of the 10th Int. Conf. on Distributed Computing Systems*, pages 20–7, Los Alamitos, CA, USA, 1990. IEEE Comput. Soc. Press.

[15] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[16] Ge-Ming Chiu and Cheng-Ru Young. Efficient rollback-recovery technique in distributed computing systems. *IEEE Trans. on Parallel and Distributed Systems*, 7(6):565–77, June 1996.

[17] Tzi-Cker Chiueh and Peitao Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *Proc. of the Twenty-Sixth Int. Symp. on Fault-Tolerant Computing (FTCS-26)*, pages 370–9, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.

[18] II. Courtright, W.V., G. Gibson, M. Holland, and J. Zelenka. A structured approach to redundant disk array implementation. In *Proc. IEEE Int. Computer Performance and Dependability Symp., IPDS '96*, pages 11–20, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.

[19] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[20] W.J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Trans. on Parallel and Distributed Systems*, 4(4):466–475, April 1993.

[21] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. Technical Report TR-5231-TR-86, Caltech, June 1986.

[22] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, C-36:547–553, 1987.

[23] A. Dearle and D. Hulse. On page-based optimistic process checkpointing. In L.-F. Cabrera and M. Theimer, editors, *Proc. of the Fourth Int. Workshop on Object-Orientation in Operating Systems (IWOOOS '95)*, pages 24–32, Los Alamitos, CA, USA, 1995. IEEE Comput. Soc. Press.

[24] Geert Deconinck, Johan Vounckx, Rudi Cuyvers, and Rudy Lauwereins. Survey of Checkpointing and Rollback Techniques. Technical Report F93-04, Katholieke Universiteit Leuven, Belgium, June 1993.

[25] E.W. Dijkstra, W.H. Seijen, and A.J.M.V. Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 16(5):217–219, 1983.

[26] E.N. Elnozahy. On the relevance of communication costs of rollback-recovery protocols. In *Proc. of the Fourteenth Ann. ACM Symp. on Principles of Distributed Computing*, pages 74–79, Ottawa, Ontario, Canada, August 1995.

[27] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. 11th Symp. on Reliable Distributed Systems*, pages 39–47, Los Alamitos, CA, USA, 1992. IEEE Comput. Soc. Press.

[28] E.N. Elnozahy and W. Zwaenepoel. Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. on Computers*, 41(5):526–31, May 1992.

[29] E.N. Elnozahy and W. Zwaenepoel. Replicate distributed processes in Manetho. In Dhiraj K. Pradhan, editor, *Proc. of the 22nd Ann. Int. Symp. on Fault-Tolerant Computing (FTCS-22)*, pages 18–27, Boston, MA, July 1992. IEEE Comput. Soc. Press.

[30] E.N. Elnozahy and W. Zwaenepoel. Manetho: An integrated approach to fault tolerance. In *Proc. of the Workshop on Hardware and Software Architectures for Tolerance*, May 1993.

[31] E.N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proc. of the 24th Ann. Int. Symp. on Fault-Tolerant Computing (FTCS-24)*, pages 298–309, Los Alamitos, CA, USA, June 1994. IEEE Comput. Soc. Press.

[32] A.K. Ganesh, T.J. Marlowe, A.D. Stoyenko, M.F. Younis, and J. Salinas. Architecture and language support for fault-tolerance in complex real-time systems. In *Proc. Second IEEE Int. Conf. on Engineering of Complex Computer Systems*, pages 314–22, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.

[33] F.M. Guess and J.F. Kitchin. Mean time to system recovery. *Quality and Reliability Engineering Int.*, 7(1):5–6, Jan.-Feb. 1991.

[34] B.W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems.* Addison-Wesley, Reading, MA, USA, 1989.

[35] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In *17th Int. Symp. on Fault-Tolerant Computing (FTCS-17)*, pages 14–19, Washington DC, USA, 1987. IEEE Comput. Soc. Press.

[36] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proc. of the Seventh Ann. ACM Symp. on Principles of Distributed Computing*, pages 171–81, New York, NY, USA, 1988. ACM.

[37] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–91, September 1990.

[38] Zbigniew Kalbarczyk and Jörgen Christmansson. Design principles for software fault tolerance—A survey. Technical Report 149, CTH, Dept. of Computer Engineering, Laboratory for Dependable Computing, 1993.

[39] J.L. Kim and T. Park. Efficient protocol for checkpointing recovery in distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 4(8):955–960, August 1993.

[40] K.H. Kim. Design of loosely coupled processes capable of time-bounded cooperative recovery: the ptc/sl scheme. *Computer Communications*, 16(5):305–16, May 1993.

[41] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13(1):23–31, January 1987.

[42] T. H. Lai and T. H. Yang. On distributed snapshots. Technical Report OSU-CISAC-86TR1THL, Computer and Information Science Research Center, Ohio State University, Columbus, OH, 1986.

[43] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, May 1987.

[44] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.

[45] Hong Va Leong and D. Agrawal. Using message semantics to reduce rollback in optimistic message logging recovery schemes. In *Proc. of the 14th Int. Conf. on Distributed Computing Systems*, pages 227–34, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.

[46] C.-C.J. Li and W.K. Fuchs. Catch-compiler-assisted techniques for checkpointing. In *Proc. of the 20th Int. Symp. on Fault-Tolerant Computing (FTCS-20)*, pages 74–81, Los Alamitos, CA, USA, 1990. IEEE Comput. Soc. Press.

[47] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. Technical Report TR-316-91, Princeton University, Computer Science Department, April 1991.

[48] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. In *Proc. of the 10th Symp. on Reliable Distributed Systems*, pages 1–11, October 1991.

[49] Kai Li, J.F. Naughton, and J.S. Plank. Real-time, concurrent checkpoint for parallel programs, March 1990.

[50] S. Mishra and R. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR 92-12, Dept. of Computer Science, University of Arizona, AZ, USA, 1992.

[51] C. Morgan. Global and logical time in distributed algorithms. *Information Processing Letters*, 20(4):189–194, May 1985.

[52] A. Mostefaoui and M. Raynal. Efficient message logging for uncoordinated checkpointing protocols. In A. Hlawiczka, J.G. Silva, and L. Simoncini, editors, *Proc. of the Second European Dependable Computing Conf. (EDCC-2)*, pages 353–64, Berlin, Germany, 1996. Springer-Verlag.

[53] A.C. Palaniswamy and P.A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In R. Bagrodia and D. Jefferson, editors, *Proc. of the 7th Workshop on Parallel and Distributed Simulation (PADS '93)*, pages 127–34, San Diego, CA, USA, 1993. SCS.

[54] J.S. Planck. *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University, January 1993.

[55] J. S. Plank and K. Li. Faster checkpointing with $N+1$ parity. In *Proc. of the 24th Int. Symp. on Fault-Tolerant Computing (FTCS-24)*, pages 288–297, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[56] J.S. Plank. Efficient checkpointing on MIMD architectures. Technical Report TR-406-93, Princeton University, Computer Science Department, June 1993.

[57] M.L. Powell and D.L. Presotto. PUBLISHING: A reliable broadcast communication mechanism. Technical Report CSD-83-133, University of California, Berkeley, 1983.

[58] D.K. Pradhan, editor. *Fault-Tolerant Computing, Theory and Techniques.* Prentice Hall, Englewood Cliffs, 1986.

[59] D.L. Russell. State restoration in systems of communicating processes. *IEEE Trans. of Software Engineering*, 6:183–194, 1980.

[60] L.M. Silva and J.G. Silva. Global checkpointing for distributed programs. In *Proc. of the 11th Symp. on Reliable Distributed Systems*, pages 155–162, Houston, TX, October 1992.

[61] B. Simons and A.Z. Spector. *Fault-tolerant distributed computing*, volume 448 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, USA, 1990.

[62] A. Prasad Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proc. of the Eighth Ann. ACM Symp. on Principles of Distributed Computing*, pages 223–38, New York, NY, USA, 1989. ACM.

[63] S.W. Smith and D.B. Johnson. Minimizing timestamp size for completely asynchronous optimistic recovery with minimal rollback. In *Proc. 15th Symp. on Reliable Distributed Systems*, pages 66–75, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.

[64] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *The 6th Int. Conf. on Distributed Computing Systems*, pages 382–388, Los Angeles, CA, USA, May 1986. IEEE Computer Society Press.

[65] M.V. Sreenivas and S. Bhalla. Garbage collection in message passing distributed systems. In N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. of the First Aizu Int. Symp. on Parallel Algorithms/Architecture Synthesis*, pages 213–18, Los Alamitos, CA, USA, 1995. IEEE Comput. Soc. Press.

[66] R.E. Strom, D.F. Bacon, and S.A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proc. of the 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 44–49, Washington DC, USA, June 1988. IEEE Comput. Soc. Press.

[67] R.E. Strom and S.A. Yemini. Optimistic recovery: an asynchronous approach to fault-tolerance in distributed systems. In *14th Int. Symp. on Fault-Tolerant Computing (FTCS-14)*, pages 374–379, Washington DC, USA, 1984. IEEE Comput. Soc. Press.

[68] R.E. Strom and S.A. Yemini. Optimistic recovery in distributed systems. *ACM Trans. on Computer Systems*, 3(3):204–226, August 1985.

[69] Nitin H. Vaidya. On checkpoint latency. Technical Report TR95-015, Texas A&M University, March 1995.

[70] S. Venkatesan. Message-optimal incremental snapshots. In *9th Int. Conf. on Distributed Computing Systems*, pages 53–60, Washington, DC, USA, 1989. IEEE Comput. Soc. Press.

[71] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25(5):295–304, July 1987.

[72] C.R. Young and G.M. Chiu. A crash recovery technique in distributed computing systems. In *Proc. of the 14th Int. Conf. on Distributed Computing Systems*, pages 235–243, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[73] J.W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, September 1974.

[74] A. Ziv and J. Bruck. An on-line algorithm for checkpoint placement. In *Proc. The Seventh Int. Symp. on Software Reliability Engineering*, pages 274–83, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.